

# Guidewire ClaimCenter®

## ClaimCenter Integration Guide

RELEASE 8.0.2

**Guidewire®**  
Deliver insurance your way™

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

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

All other trademarks are the property of their respective owners.

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

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

Product Name: Guidewire ClaimCenter

Product Release: 8.0.2

Document Name: ClaimCenter Integration Guide

Document Revision: 20-May-2014

# Contents

<b>About ClaimCenter Documentation .....</b>	<b>11</b>
Conventions in This Document .....	12
Support .....	12

## Part I Overview of Integration

<b>1 Integration Overview.....</b>	<b>15</b>
Overview of Integration Methods.....	15
Important Information about ClaimCenter Web Services .....	17
Preparing for Integration Development .....	18
Integration Documentation Overview .....	20
Regenerating Integration Libraries .....	22
What are Required Files for Integration Programmers?.....	23
Public IDs and Integration Code.....	27

## Part II Web Services

<b>2 Web Services Introduction.....</b>	<b>31</b>
What are Web Services?.....	31
What Happens During a Web Service Call?.....	32
Reference of All Built-in Web Services .....	33
<b>3 Publishing Web Services (WS-I) .....</b>	<b>37</b>
WS-I Web Service Publishing Overview .....	38
Publishing WS-I Web Services.....	43
Testing WS-I Web Services with wsi.local .....	47
Generating WS-I WSDL.....	48
Adding Advanced Security Layers to a WS-I Web Service.....	51
WS-I Web Services Authentication Plugin .....	56
Checking for Duplicate External Transaction IDs .....	57
Request or Response XML Structural Transformations.....	57
Reference Additional Schemas in Your Published WSDL .....	58
Validate Requests Using Additional Schemas as Parse Options .....	58
Invocation Handlers for Implementing Preexisting WSDL .....	59
Locale Support .....	62
Setting Response Serialization Options, Including Encodings.....	63
Exposing Typelists and Enumerations As String Values .....	63
Transforming a Generated Schema.....	64
Login WS-I Authentication Confirmation .....	65
Stateful WS-I Session Affinity Using Cookies.....	65
Calling a ClaimCenter WS-I Web Service from Java .....	66
<b>4 Calling WS-I Web Services from Gosu .....</b>	<b>71</b>
Consuming WS-I Web Service Overview .....	71
Adding WS-I Configuration Options .....	78
One-Way Methods .....	85
Asynchronous Methods .....	86

MTOM Attachments with Gosu as Web Service Client.....	86
<b>5 Publishing Web Services (RPCE) .....</b>	<b>89</b>
RPCE Web Service Overview .....	90
Publishing a RPCE Web Service .....	95
Writing Web Services that Use Entities .....	100
Testing Your RPCE Web Service With soap.local Namespace.....	103
Calling Your Published RPCE Web Service from Java .....	107
Calling Your RPCE Web Service from Microsoft .NET WSE 3.0 .....	110
Calling Your RPCE Web Service from Microsoft .NET WSE 2.0 .....	114
Calling Published RPCE Web Services From Other Languages .....	117
Typecodes and Web Services in RPCE Web Services.....	120
Public IDs and RPCE Web Services.....	120
Endpoint URLs and Generated WSDL in RPCE Web Services .....	122
Web Services Using ClaimCenter Clusters .....	124
SOAP Faults (Exceptions) in RPCE Web Services .....	126
Writing Command Line Tools that Call RPCE Web Services.....	129
<b>6 Calling RPCE Web Services from Gosu .....</b>	<b>135</b>
Calling External RPCE Web Services .....	135
Calling RPCE Web Service from Gosu: ICD-9 Example .....	140
<b>7 General Web Services .....</b>	<b>143</b>
Mapping Typecodes to External System Codes.....	143
Importing Administrative Data.....	145
Maintenance Web Services.....	146
System Tools Web Services .....	147
Workflow Web Services.....	149
Profiling Web Services.....	149
<b>8 Claim-related Web Services.....</b>	<b>151</b>
Claim Web Service APIs and Data Transfer Objects.....	152
Adding First Notice of Loss from External Systems .....	152
Getting a Claim from External Systems .....	153
Importing a Claim from XML from External Systems.....	153
Migrating a Claim from External Systems.....	153
Getting Information from Claims/Exposures from External Systems .....	154
Claim Bulk Validate from External Systems .....	154
Previewing Assignments from External Systems.....	155
Closing and Reopening a Claim from External Systems .....	155
Policy Refresh from External Systems .....	155
Add User Permissions on a Claim from External Systems .....	156
Archiving and Restoring Claims from External Systems.....	156
Managing Activities from External Systems .....	157
Adding a Contact from External Systems .....	158
Adding a Document from External Systems .....	159
Adding an Exposure from External Systems .....	159
Getting an Exposure from External Systems .....	160
Closing and Reopening an Exposure from External Systems .....	160
Adding to Claim History from External Systems.....	160
Creating a Note from External Systems .....	160

## Part III Plugins

<b>9 Plugin Overview .....</b>	<b>163</b>
Overview of ClaimCenter Plugins .....	164
Error Handling in Plugins .....	169
Temporarily Disabling a Plugin .....	169
Example Gosu Plugin .....	169
Special Notes For Java Plugins .....	170
Getting Plugin Parameters from the Plugins Registry Editor .....	171
Writing Plugin Templates For Plugins That Take Template Data .....	172
Plugin Registry APIs .....	174
Plugin Thread Safety .....	175
Reading System Properties in Plugins .....	180
Do Not Call Local Web Services From Plugins .....	180
Creating Unique Numbers in a Sequence .....	180
Restarting and Testing Tips for Plugin Developers .....	181
Summary of All ClaimCenter Plugins .....	181
<b>10 Authentication Integration .....</b>	<b>187</b>
Overview of User Authentication Interfaces .....	187
User Authentication Source Creator Plugin .....	189
User Authentication Service Plugin .....	191
Deploying User Authentication Plugins .....	194
Database Authentication Plugins .....	195
ABAAuthenticationPlugin for ContactManager Authentication .....	197
<b>11 Document Management .....</b>	<b>199</b>
Document Management Overview .....	199
Choices for Storing Document Content and Metadata .....	201
Document Storage Plugin Architecture .....	203
Implementing a Document Content Source for External DMS .....	204
Storing Document Metadata In an External DMS .....	206
The Built-in Document Storage Plugins .....	208
Asynchronous Document Storage .....	209
APIs to Attach Documents to Business Objects .....	210
Retrieval and Rendering of PDF or Other Input Stream Data .....	211
<b>12 Document Production .....</b>	<b>213</b>
Document Production Overview .....	213
Document Template Descriptors .....	221
Generating Documents from Gosu .....	230
Template Web Service APIs .....	233
<b>13 Geographic Data Integration .....</b>	<b>235</b>
Geocoding Plugin Integration .....	235
Steps to Deploy a Geocoding Plugin .....	237
Writing a Geocoding Plugin .....	238
Geocoding Status Codes .....	244
<b>14 Reinsurance Integration .....</b>	<b>247</b>
Reinsurance Overview .....	247
Reinsurance Plugin .....	247
<b>15 Encryption Integration .....</b>	<b>249</b>
Encryption Integration Overview .....	249
Changing Your Encryption Algorithm Later .....	254

Encryption Changes with Snapshots.....	256
Encryption Features for Staging Tables .....	257
<b>16 Management Integration.....</b>	<b>259</b>
Management Integration Overview .....	259
The Abstract Management Plugin Interface.....	260
Integrating With the Included JMX Management Plugin.....	261
<b>17 Other Plugin Interfaces.....</b>	<b>263</b>
Claim Number Generator Plugin .....	263
Approval Plugin .....	264
Automatic Address Completion and Fill-in Plugin .....	265
Phone Number Normalizer Plugin .....	265
Testing Clock Plugin (Only For Non-Production Servers) .....	265
Work Item Priority Plugin .....	267
Official IDs Mapped to Tax IDs Plugin .....	267
Preupdate Handler Plugin.....	267
Defining Base URLs for Fully-Qualified Domain Names .....	269
Exception and Escalation Plugins.....	270
<b>18 Startable Plugins.....</b>	<b>271</b>
What are Startable Plugins? .....	271
<b>19 Multi-threaded Inbound Integration.....</b>	<b>279</b>
Multi-threaded Inbound Integration Overview.....	279
Inbound Integration Configuration XML File .....	281
Inbound File Integration .....	283
Inbound JMS Integration .....	287
Custom Inbound Integrations .....	289
Understanding the Polling Interval and Throttle Interval.....	295

## Part IV Messaging

<b>20 Messaging and Events .....</b>	<b>299</b>
Messaging Overview .....	300
Message Destination Overview .....	311
Filtering Events.....	319
List of Messaging Events in ClaimCenter .....	320
Generating New Messages in Event Fired Rules .....	329
Message Ordering and Multi-Threaded Sending .....	334
Late Binding Data in Your Payload .....	339
Reporting Acknowledgements and Errors .....	340
Tracking a Specific Entity With a Message .....	343
Implementing Messaging Plugins.....	343
Resynchronizing Messages for a Primary Object.....	350
Message Payload Mapping Utility for Java Plugins.....	353
Monitoring Messages and Handling Errors .....	354
Messaging Tools Web Service .....	356
Batch Mode Integration .....	358
Included Messaging Transports .....	358

## Part V Financials

<b>21 Financials Integration.....</b>	<b>363</b>
Financial Transaction Status and Status Transitions .....	363
Claim Financials Web Service Data Transfer Objects (DTOs) .....	368
Claim Financials Web Services (ClaimFinancialsAPI) .....	369
Check Integration .....	372
Payment Transaction Integration .....	381
Recovery Reserve Transaction Integration.....	387
Recovery Transaction Integration.....	388
Reserve Transaction Integration.....	390
Bulk Invoice Integration.....	391
Deduction Plugins.....	407
Initial Reserve Initialization for Exposures .....	409
Exchange Rate Integration .....	409

## Part VI Importing Claims Data

<b>22 Importing from Database Staging Tables.....</b>	<b>415</b>
Introduction to Database Staging Table Import .....	415
Overview of a Typical Database Import.....	419
Database Import Performance and Statistics .....	424
Table Import Tools .....	424
Populating the Staging Tables .....	426
Data Integrity Checks .....	431
Table Import Tips and Troubleshooting.....	432
Staging Table Import of Encrypted Properties.....	433
<b>23 FNOL Mapper.....</b>	<b>435</b>
FNOL Mapper Overview .....	435
FNOL Mapper Detailed Flow .....	436
Structure of FNOL Mapper Classes .....	437
Example FNOL Mapper Customizations .....	442

## Part VII ISO and Metropolitan

<b>24 Insurance Services Office (ISO) Integration.....</b>	<b>447</b>
ISO Integration Overview.....	448
ISO Implementation Checklist .....	454
ISO Network Architecture .....	456
ISO Activity and Decision Timeline .....	460
ISO Authentication and Security .....	466
ISO Proxy Server Setup .....	467
ISO Validation Level .....	468
ISO Messaging Destination .....	469
ISO Receive Servlet and the ISO Reply Plugin .....	471
ISO Properties on Entities.....	471
ISO User Interface .....	472
ISO Properties File .....	473
ISO Type Code and Coverage Mapping.....	476
ISO Payload XML Customization .....	478

ISO Match Reports .....	480
ISO Exposure Type Changes .....	481
ISO Date Search Range and Resubmitting Exposures .....	482
ISO Integration Troubleshooting .....	482
ISO Formats and Feeds.....	483
<b>25 Metropolitan Reporting Bureau Integration .....</b>	<b>487</b>
Overview of ClaimCenter-Metropolitan Integration .....	487
Metropolitan Configuration .....	490
Metropolitan Report Templates and Report Types.....	492
Metropolitan Entities, Typelists, Properties, and Statuses .....	494
Metropolitan Error Handling .....	498

## Part VIII Policy and Contact Integrations

<b>26 Contact Integration.....</b>	<b>501</b>
Integrating with a Contact Management System .....	501
Configuring Contact Links .....	506
Contact Web Service APIs .....	508
<b>27 Claim and Policy Integration .....</b>	<b>511</b>
Policy System Notifications .....	512
Policy Search Plugin.....	517
Claim Search Web Service For Policy System Integration .....	522
ClaimCenter Exit Points to the Policy and Contact Management Applications.....	524
PolicyCenter Product Model Import into ClaimCenter .....	524
Catastrophe Policy Location Download .....	535
Policy Location Search Plugin .....	539
Policy Refresh Overview .....	539
Policy Refresh Plugins and Configuration Classes.....	542
Policy Refresh Steps and Associated Implementation .....	544
Determining the Extent of the Policy Graph .....	546
Policy Refresh Entity Matcher Details .....	548
Policy Refresh Relinking Details .....	552
Policy Refresh Policy Comparison Display .....	556
Policy Refresh Configuration Examples.....	565

## Part IX Other Integration Topics

<b>28 Archiving Integration .....</b>	<b>575</b>
Overview of Archiving Integration.....	575
Archiving Storage Integration Detailed Flow.....	579
Archive Retrieval Integration Detailed Flow .....	582
Archive Source Plugin Utility Methods .....	583
Upgrading the Data Model of Retrieved Data .....	585
<b>29 Custom Batch Processes.....</b>	<b>587</b>
Creating a Custom Batch Process .....	587
Custom Batch Processes and MaintenanceToolsAPI .....	594
<b>30 Free-text Search Integration .....</b>	<b>595</b>
Free-text Search Plugins Overview .....	595
Free-text Load and Index Plugin and Message Transport .....	596

Free-text Search Plugin.....	598
<b>31 Servlets .....</b>	<b>599</b>
Implementing Servlets .....	599
<b>32 Data Extraction Integration .....</b>	<b>607</b>
Why Gosu Templates are Useful for Data Extraction .....	607
Data Extraction Using Web Services .....	608
<b>33 Logging .....</b>	<b>613</b>
Logging Overview For Integration Developers .....	613
Logging Properties File .....	614
Logging APIs for Java Integration Developers .....	615
<b>34 Proxy Servers .....</b>	<b>619</b>
Proxy Server Overview.....	619
Configuring a Proxy Server with Apache HTTP Server .....	620
Certificates, Private Keys, and Passphrase Scripts.....	621
Proxy Server Integration Types for ClaimCenter.....	622
Proxy Building Blocks .....	623
<b>35 Java and OSGi Support .....</b>	<b>627</b>
Overview of Java and OSGi Support .....	627
Accessing Entity and Typecode Data in Java.....	631
Accessing Gosu Classes from Java Using Reflection .....	642
Gosu Enhancement Properties and Methods in Java .....	643
Class Loading and Delegation for non-OSGi Java.....	643
Deploying Non-OSGi Java Classes and JARs .....	644
OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor.....	645
Advanced OSGi Dependency and Settings Configuration.....	652
Updating Your OSGi Plugin Project After Product Location Changes .....	653



# About ClaimCenter Documentation

The following table lists the documents in ClaimCenter documentation.

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

## Conventions in This Document

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

## Support

For assistance 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

# Overview of Integration



# Integration Overview

You can integrate a variety of external systems with ClaimCenter by using services and APIs that link ClaimCenter with custom code and external systems. This overview provides information to help you plan integration projects for your ClaimCenter deployment and provides technical details critical to successful integration efforts.

This topic includes:

- “Overview of Integration Methods” on page 15
- “Important Information about ClaimCenter Web Services” on page 17
- “Preparing for Integration Development” on page 18
- “Integration Documentation Overview” on page 20
- “Regenerating Integration Libraries” on page 22
- “What are Required Files for Integration Programmers?” on page 23
- “Public IDs and Integration Code” on page 27

## Overview of Integration Methods

ClaimCenter addresses the following integration architecture requirements:

- **A service-oriented architecture** – Encapsulate your integration code so upgrading the core application requires few other changes. Also, a service-oriented architecture allows APIs to use different languages or platform.
- **Customize behavior with the help of external code or external systems** – For example, implement special claim validation logic, use a legacy system that generates claim numbers, or query a legacy system.
- **Send messages to external systems in a transaction-safe way** – Trigger actions after important events happen inside ClaimCenter, and notify external systems if and only if the change is successful and no exceptions occurred. For example, alert a policy database if anyone changes claim information.
- **Flexible export** – Providing different types of export minimizes data conversion logic. Simplifying the conversion logic improves performance and code maintainability for integrating with diverse and complex legacy systems.

- **Predictable error handling** – Find and handle errors cleanly and consistently for a stable integration with custom code and external systems.
- **Link business rules to custom task-oriented Gosu or Java code** – Let Gosu-based business rules in Guidewire Studio or Gosu templates call Java classes directly from Gosu.
- **Import or export data to/from external systems** – There are many methods of importing and exporting data to ClaimCenter, and you can choose which methods make the most sense for your integration project.
- **Use clearly-defined industry standard protocols for integration points** – ClaimCenter includes built-in APIs to retrieve claims, create users, manage documents, trigger events, validate records, and trigger bulk import/export. However, most legacy system integrations require additional integration points customized for each system.

To achieve these goals, the ClaimCenter integration framework includes multiple methods of integrating external code with the ClaimCenter product. The primary integration methods:

- **Web service APIs** – Web service APIs are a general-purpose set of application programming interfaces that you can use to query, add, or update Guidewire data, or trigger actions and events programmatically. Because these APIs are web services, you can call them from any language and from any operating system. A typical use of the web service APIs is to send a new FNOL (First Notice of Loss) into ClaimCenter to set up a new claim. You can use the built-in SOAP APIs, but you can also design your own SOAP APIs in Gosu and expose them for use by remote systems. Additionally, your Gosu code can easily call web services hosted on other computers to trigger actions or retrieve data.
- **Plugins** – ClaimCenter plugins are mini-programs that ClaimCenter invokes to perform an action or calculate a result. Guidewire recommends writing plugins in Gosu. You can also write plugins in Java. Note that Gosu code can call third-party Java classes and Java libraries.

General categories of plugins include:

- **Messaging plugins** – Send messages to remote systems, and receive acknowledgements of each message. ClaimCenter has a sophisticated transactional messaging system to send information to external systems in a reliable way. Any server in the cluster can create a message for any data change. The batch server, in a separate thread and database transaction, sends messages and tracks (waits for) message acknowledgments. For details, see “Messaging and Events” on page 299.
- **Authentication plugins** – Integrate custom authentication systems. For instance, define a user authentication plugin to support a corporate directory that uses the LDAP protocol. Or define a database authentication plugin to support custom authentication between ClaimCenter and its database server. For details, see “Authentication Integration” on page 187.
- **Document and form plugins** – Transfer documents to and from a document management system, and help prepare new documents from templates. Additionally, use Gosu APIs to create and attach documents. For details, see “Document Management” on page 199.
- **Inbound integration plugins** – Multi-threaded inbound integrations for high-performance data import. ClaimCenter includes built-in implementations for reading files and receiving JMS messages, but you can also write your implementations that leverage the multi-threaded framework. See “Multi-threaded Inbound Integration” on page 279.
- **Other plugins** – For example, generate claim numbers (`IClaimNumGenAdapter`), or get a policy related to a claim from a policy administration system (`IPolicySearchAdapter`). For a summary of all plugins, see “Summary of All ClaimCenter Plugins” on page 181.

#### See also

- “Plugin Overview” on page 163
- **Templates** – Generate text-based formats that contain combinations of ClaimCenter data and fixed data. Templates are ideal for name-value pair export, HTML export, text-based form letters, or simple text-based protocols. For details, see “Data Extraction Integration” on page 607.
- **Database import from staging tables** – Bulk data import into production databases using temporary loading into “staging” database tables. ClaimCenter performs data consistency checks on data before importing new data (although it does not run validation rules). Database import is faster than adding individual records one

by one. To initially load data from an external system or performing large daily imports from another system, use database staging table import. For details, see “Importing from Database Staging Tables” on page 415.

The following table compares the main integration methods.

Integration type	Description	What you write
Web services	A full API to manipulate ClaimCenter data and trigger actions externally using any programming language or platform using ClaimCenter Web services APIs, accessed using the platform-independent SOAP protocol.	<ul style="list-style-type: none"> <li>Publishing APIs: writing Gosu classes, and marking them as published services.</li> <li>Using APIs: writing Java code that calls out to the published SOAP APIs, both built-in APIs and custom-written APIs.</li> </ul>
Plugins	Small programs that ClaimCenter calls to perform tasks or calculate a result. Guidewire recommends writing plugins in Gosu. You can also write plugins in Java. Plugins run directly on each server. Java plugins run in the same Java virtual machine as ClaimCenter.	<ul style="list-style-type: none"> <li>Gosu classes that implement a Guidewire-defined interface.</li> <li>Java classes that implement a Guidewire-defined interface. Optionally use OSGi, a Java component system. See “Overview of Java and OSGi Support” on page 627</li> </ul>
Messaging code	Most messaging code is in your Event Fired rule set, or in your messaging plugin implementations. The main class you need to implement to connect to your external system is the MessageTransport plugin. There is also the MessageRequest plugin (for pre-processing your payload) and the MessageReply plugin (for asynchronous replies).	<ul style="list-style-type: none"> <li>Gosu code in the Event Fired rule set.</li> <li>Messaging plugins.</li> <li>Configure one or more messaging destinations in Studio to use your messaging plugins.</li> </ul>
Guidewire XML (GX) models	The GX model editor in Studio lets you create custom schemata (XSDs) for your data to assist in integrations. You customize which properties in an entity (or other type) to export in XML. Then, you can use this model to export XML or import XML in your integrations. For example, your messaging plugins could send XML to external systems, or your web services could take this XML format as its payload from an external system. For more information, see “The Guidewire XML (GX) Modeler” on page 302 in the <i>Gosu Reference Guide</i> .	<ul style="list-style-type: none"> <li>Using the GX model editor in Studio, you customize which properties in an entity (or other type) to export in XML.</li> <li>Various integration code that uses the XSD it creates.</li> </ul>
Templates	Several data extraction mechanisms that perform database queries and format the data as necessary. For example, send notifications as form letters and use plain text with embedded Gosu code to simplify deployment and ongoing updates. Or design template that exports HTML for easy development of web-based reports of ClaimCenter data.	<ul style="list-style-type: none"> <li>Text files that contain small amounts of Gosu code</li> </ul>
Database import	You can import large amounts of data into ClaimCenter by first populating a separate set of staging database tables. Staging tables are temporary versions of the data that exist separate from the production database tables. Next, use ClaimCenter web service APIs or command line tools to validate and import that data.	<ul style="list-style-type: none"> <li>Custom tools that take legacy data and convert them into database tables of data.</li> </ul>

## Important Information about ClaimCenter Web Services

It is critical to understand that ClaimCenter supports two types of web services:

- WS-I web services** – WS-I web services use the SOAP protocol and are compatible with the WS-I standard.

- **RPC Encoded (RPCE) web services** – RPCE web services are an older style of web service publishing APIs. Versions of ClaimCenter before version 7.0 supported only the RPCE type of web service publishing.

---

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

---

#### See also

- For a comparison of RPCE and WS-I web services, see “WS-I Web Services” on page 122 in the *New and Changed Guide*.
- For details of the WS-I standard and its use of the Document Literal encoding, see “Consuming WS-I Web Service Overview” on page 71.
- For an overview of web services, including a list of all built-in web services, see “Web Services Introduction” on page 31.

## Web Service Documentation for Each Type of Web Services

For both consuming and publishing web services, the WS-I syntax is different from the RPCE syntax. Be sure to read the correct documentation. The following table lists the documentation topics for each technology (WS-I or RPCE) and the direction of the request.

For an overview of web services, including a full reference list of each built-in web services, see “Web Services Introduction” on page 31.

Type of web service	To publish a Guidewire-hosted web service or to call one from Java or .NET...	To consume any web services from Gosu...
WS-I	See “Publishing Web Services (WS-I)” on page 37	See “Calling WS-I Web Services from Gosu” on page 71
RPCE	See “Publishing Web Services (RPCE)” on page 89	See “Calling RPCE Web Services from Gosu” on page 135

For new web services, Guidewire strongly recommends that you publish new web services using the WS-I APIs. Similarly, when consuming web services from Gosu, and the external system supports WS-I, use the WS-I APIs.

---

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

---

## Preparing for Integration Development

During integration development, edit configuration files within the hierarchy of files in the product installation directory. In most cases you modify data only through the Studio interface, which handles any SCM (Source Configuration Management) requests. The Studio interface also copies read-only files to the configuration module in the file hierarchy for your files and makes files writable as appropriate.

However, in some cases you need to add files directly to certain directories in the configuration module hierarchy, such as Java class files for Java plugin support. The configuration module hierarchy for your files is in the hierarchy:

`ClaimCenter/modules/configuration/...`

This directory contains subdirectories such as:

Directory under configuration module	Description
config/iso	Your Insurance Services Office (ISO) files. See “Insurance Services Office (ISO) Integration” on page 447
config/metro	Your Metropolitan (police report/inquiry) files. See “Metropolitan Reporting Bureau Integration” on page 487
config/web	Your web application PCF files.
config/logging	Your logging configuration files. See “Logging” on page 613.
config/templates	<p>This directory contains two types of templates relevant for integration:</p> <ul style="list-style-type: none"> <li>• <b>Plugin templates</b> – Use Gosu plugin templates for the small number of plugin interfaces that require them. These plugin templates extract important properties from entities, and they generate text that describes the results. Whenever ClaimCenter needs to call the plugin, ClaimCenter passes the results to the plugin as text-based parameters.</li> <li>• <b>Messaging templates</b> – Use optional Gosu messaging templates for your messaging code. Use messaging templates to define notification letters or other similar messages contain large amounts of text but a small amount of Gosu code.</li> </ul> <p>There are built-in versions some of these templates. To modify the built-in versions, in Studio navigate to Resources → Configuration → Other Resources → Templates.</p>
config/docmgmt	Document management files. See “Document Management” on page 199.
plugins	<p>Your Java plugin files. To add Java files for plugin support, see “Special Notes For Java Plugins” on page 170 and “Overview of Java and OSGi Support” on page 627.</p> <p>Register your plugin implementations in the Plugin registry in Studio. When you register the plugin in the Plugin registry, you can define a <i>plugin directory</i>, which is the name of a subdirectory of the plugins directory. If you do not define a subdirectory, ClaimCenter defaults the plugin directory to the shared subdirectory. Be aware that if you are using the Java API introduced in version 8.0, your classes and libraries must be in a subdirectory of the plugin directory called basic. See “Deploying Non-OSGi Java Classes and JARs” on page 644.</p> <p>For more information about registering a plugin, see “Plugin Overview” on page 163 and “Using the Plugins Registry Editor” on page 113 in the <i>Configuration Guide</i>.</p> <p>For a messaging plugin, you must register this information two different registries.</p> <ul style="list-style-type: none"> <li>• the plugin registry in the plugin editor in Studio</li> <li>• the messaging registry in the Messaging editor in Studio. See “Messaging and Events” on page 299 and “Messaging Editor” on page 137 in the <i>Configuration Guide</i>.</li> </ul>

There are some important directories other than the subdirectories for files in the configuration module.

Other directory	Description
ClaimCenter/bin	<p>Command line tools such as gwcc.bat. Use this for the following integration tasks:</p> <ul style="list-style-type: none"> <li>• Regenerating the Java API libraries and SOAP API libraries. See “Regenerating Integration Libraries” on page 22.</li> <li>• Regenerating the <i>Data Dictionary</i>. See “Data Dictionary Documentation” on page 22</li> </ul>
ClaimCenter/soap-api	SOAP API files. ClaimCenter scripts generate this directory. To regenerate these, see “Regenerating Integration Libraries” on page 22.
ClaimCenter/soap-api/wsi/wsdl	For WS-I web services, this directory contains WSDL files generated locally. To regenerate these, see “Regenerating Integration Libraries” on page 22.
ClaimCenter/soap-api/rpce/lib	For RPCE web services only (a deprecated API), this directory contains Java libraries for web service (SOAP) client development.
	<p><b>Note:</b> There is no equivalent generated libraries for WS-I web services. See “Publishing WS-I Web Services” on page 43.</p>

Other directory	Description
ClaimCenter/soap-api/rpce/doc	<p>For RPCE web services only (a deprecated API), this directory contains API documentation in Javadoc format for web service (SOAP) development.</p> <p><b>Note:</b> There is no equivalent generated documentation for WS-I web services. See “Publishing WS-I Web Services” on page 43.</p>
ClaimCenter/soap-api/rpce/wsdl	<p>For RPCE web services only (a deprecated API), this directory contains WSDL files generated locally. Typically, you do not need these. Typically it is best to use either the generated Java libraries. If you need the WSDL, typically it is best to use the dynamically generated WSDL with a full endpoint URLs from a running server.</p> <p><b>Note:</b> There is no equivalent generated on-disk for WS-I web services, although there is dynamic server-generated WSDL. See “Publishing WS-I Web Services” on page 43.</p>
ClaimCenter/admin	<p>Command line tools that control a running ClaimCenter server. Almost all of these are small command line tool wrappers for public web service APIs. You can write your own command line tools to call web service APIs, including web services that you write yourself.</p>

The final location of your code varies:

- After you regenerate a new full deployment for your web application server as a .war or .ear file, the resulting web application contain your custom code in the configuration module. Your .war or .ear file runs in the web application contain that hosts your production web application.
- Your plugins are part of the final web application and run inside the server. Some plugins might communicate with external systems.
- Your templates are part of the final web application and run inside the server.
- Your messaging plugins are part of the final web application and run inside the server. However, your plugins communicate with external systems.
- Your web service API client code uses the SOAP API library files. After you fully deploy your production system, your client code sits fully outside ClaimCenter and calls ClaimCenter web service APIs across the Internet.

## Integration Documentation Overview

Use the *Integration Guide* for important integration information. Also, consult other reference documentation during development:

- *API Reference Javadoc Documentation*
- *Data Dictionary Documentation*

### API Reference Javadoc Documentation

The easiest way to learn what interfaces are available is by reading *ClaimCenter API Reference Javadoc* documentation, which is web browser-based documentation.

There are multiple types of reference documentation.

#### Java API Reference Javadoc

The Java API Javadoc includes:

- The specification of the plugin definitions for Java plugin interfaces. These also are the specifications for plugins implemented in Gosu.

- The details of full entities, including both the entity data (get and set methods) plus additional methods called domain methods. For a high-level overview of entity differences and different entity access types, see the diagram in “What are Required Files for Integration Programmers?” on page 23. For writing Java plugins, also see “Calling Java from Gosu” on page 119 in the *Gosu Reference Guide*.
- General Java utility classes.
- After regenerating the generated files, find it in `ClaimCenter/java-api/doc/api/index.html`

**Note:** To regenerate these files, see “Regenerating Integration Libraries” on page 22.

## SOAP API Reference

### Web Service Description Language (WSDL)

On a running ClaimCenter server, you can get up-to-date WSDL from published services

- For WS-I web services, see “Getting WS-I WSDL from a Running Server” on page 48
- For RPCE web services, see “Dynamic WSDL and SOAP API Library WSDL” on page 123

After regenerating SOAP API reference files, there are additional local WSDL files on the server. To regenerate these files, see “Regenerating Integration Libraries” on page 22.

- For WS-I web services, ClaimCenter generates the WSDL at the path  
`ClaimCenter/soap-api/wsi/wsdl`
- For RPCE web services, ClaimCenter generates the WSDL at the path  
`ClaimCenter/soap-api/rpce/wsdl`

For more information about web services, see “Web Services Introduction” on page 31.

---

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

---

### Web Service Javadoc

For WS-I web services, there is no built-in mechanism to display Javadoc for SOAP APIs. That is because there are no built-in generated libraries for a specific SOAP system. Depending on which language you use and which WS-I compatible SOAP implementation you use, the method signatures could be different.

For RPCE web services, ClaimCenter generates the specification of the web services. This includes details of SOAP entities, which contain the entity data only as get and set methods. For a high-level overview of entity differences and different entity access types, see the diagram in “What are Required Files for Integration Programmers?” on page 23. This documentation also includes the RPCE-specific SOAP Java utility classes to use with the RPCE SOAP generated libraries, such as `APILocator`. After regenerating integration libraries, find the documentation at the path:

`ClaimCenter/soap-api/rpce/doc`

For a high-level overview of differences using Guidewire entities, see the diagram in “What are Required Files for Integration Programmers?” on page 23.

For more information about web services, see “Web Services Introduction” on page 31.

## Gosu Generated Documentation

Integration programmers might want to use the Gosu documentation that you can generate from the command line using the `gwcc` tool:

`gwcc regen-gosudoc`

You will then find the documentation at `ClaimCenter/build/gosudoc/index.html`.

This documentation is particularly valuable for integration programmers implementing plugins in Gosu. The information in the Javadoc-formatted files are more relevant for Gosu programmers than Java programmers due to package differences and visibility from Java.

For more information, see “Gosu Generated Documentation (‘gosudoc’)” on page 37 in the *Gosu Reference Guide*.

## Using Javadoc-formatted Documentation

Several types of generated documentation are in web-based Javadoc format. For Javadoc-formatted documentation, to look within a particular package namespace, click in the top-left pane of the documentation on the package name. The bottom left pane displays interfaces and classes listed for that package. If you click on a class, the right pane shows the methods for that class.

## Data Dictionary Documentation

Another set of documentation called the *ClaimCenter Data Dictionary* provides documentation on classes that correspond to ClaimCenter data. You must regenerate the *Data Dictionary* to use it.

To view the documentation, refer to:

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

To regenerate the dictionary, open a command window and change directory to `ClaimCenter/bin` and run the command:

```
gwcc regen-dictionary
```

The *ClaimCenter Data Dictionary* typically has more information about data-related objects than the API Reference documentation has for that class/entity. The *Data Dictionary* documents only classes corresponding to data defined in the data model. It does not document all API functions or utility classes.

The *Data Dictionary* lists some objects or properties that are part of the data model and the relevant database tables as internal properties. You must not modify internal properties within the database or use any other mechanism. Getting or setting internal data properties might provide misleading data or corrupt the integrity of application data.

For example, the `Claim` class’s `status` property.

## Regenerating Integration Libraries

You must regenerate the Java API libraries and SOAP API libraries after you make certain changes to the product configuration. Regenerate these files in the following situations:

- After you install a new ClaimCenter release
- After you make changes to the ClaimCenter data model, such as data model extensions, typelists, field validators, and abstract data types

As you work on both configuration and integration tasks, you may need to regenerate the Java API libraries and SOAP API libraries frequently. However, if you make significant configuration changes and then work on integration at a later stage, you can wait until you need the APIs updated before regenerating these files. Afterward, you can recompile integration code against the generated libraries, if necessary.

**Note:** You might read other documentation that refers to Java API libraries and SOAP API libraries and their generated documentation collectively as the *toolkit libraries* or simply *the toolkit*.

- The location for the Java generated libraries is:

```
ClaimCenter/java-api/lib
```

- The location of the generated Java library documentation:  
`ClaimCenter/java-api/doc`
- The location for the WSI web service WSDL is:  
`ClaimCenter/soap-api/wsi/wsdl`
- The location of the generated RPCE web service Java libraries is:  
`ClaimCenter/soap-api/rpce/lib`
- The location of the generated RPCE web service documentation is:  
`ClaimCenter/soap-api/rpce/doc/api`

Call the main `gwcc.bat` file to regenerate these

- For Java development, generate libraries and documentation with:  
`gwcc.bat regen-java-api`
- For web services (SOAP) development, generate libraries and documentation with:  
`gwcc.bat regen-soap-api`

For example, to generate Java API libraries and documentation:

1. In Windows, open a command window.
2. Change your working directory with the following command:  
`cd ClaimCenter/bin`
3. Type the command:  
`gwcc regen-java-api`

As part of its normal behavior while regenerating the documentation for the Java or the SOAP files, the script displays some warnings and errors.

#### See also

- “What are Required Files for Integration Programmers?” on page 23

## What are Required Files for Integration Programmers?

Depending on what kind of integrations you require, there are special files you must use. For example, libraries to compile your Java code against, to import in a project, or to use in other ways.

There are several types of integration files:

- **Web service APIs** – Some files represent files you can use with SOAP API client code.
- **Plugin interfaces** – Some files represent plugin interfaces for your code to respond to requests from ClaimCenter to calculate a value or perform a task.
- **Java entity libraries** – Some files represent *entities*, which are ClaimCenter objects defined in the data model with built-in properties and can also have data model extension properties. For example, `Claim` and `Address` are examples of Guidewire entities. To access entity data and methods from Java, you need to use Java entity libraries. See “Java and OSGi Support” on page 627.

In all cases, ClaimCenter entities such as `Claim` contain data properties that can be manipulated either directly or from some contexts using getters and setters (`get...` and `set...` methods).

Depending on the type of integration point, there may be additional methods available on the objects. These additional *domain methods* often contain valuable functionality for you. If an integration point can access both entity data and domain methods, it is said to have access to the *full entities*.

The following table summarizes the different entity integration implications for each integration type. For this table, *full entity instances* means access to all properties and domain methods.

Entity access	Description	Entities	Generated libraries
Gosu plugin implementation	Plugin interface defined in Gosu.	Full entity instances	None
Java plugin implementation	Java code that accesses an entity associated with a plugin interface parameter or return value.	Full entity instances	Java libraries
Java class called from Gosu	Java code called from Gosu that accesses an entity passed as a parameter from Gosu, or a return result to be passed back to Gosu.	Full entity instances	Java libraries
WS-I web service	WS-I web services are the newer standard for web services.	No access. For WS-I, there is no support for entity types as arguments or return values. Instead, create your own data transfer objects (DTO) as either: <ul style="list-style-type: none"> <li>• Gosu class instances that contain only the properties required for each integration point.</li> <li>• XML objects with structure defined in XSD files.</li> <li>• XML objects with structure defined with the GX modeler. The GX modeler is a tool to generate an XML model with only the desired subset of properties for each entity for each integration point. See "The Guidewire XML (GX) Modeler" on page 302 in the <i>Gosu Reference Guide</i>.</li> </ul>	None. For WS-I, there is no support for entity types as arguments or return values. However, you can use the WSDL on disk or over the network to create stubs in other languages. See "Generating WS-I WSDL" on page 48.
RPCE web service	RPCE web services are the older style of web services. <b>IMPORTANT:</b> RPCE web services are deprecated. Convert existing RPCE code to WS-I web services.	Limited SOAP entity instances. SOAP entity instances contain getters and setters for properties, but no domain methods. RPCE SOAP entities are sometimes called data-only entities.	If you use Java, use the generated RPCE web service libraries. Also, if you are using Java, the generated libraries include the Java utility JAR, which contains APILocator. The JAR file name is gw-soap-cc.jar. If you use other languages, use the WSDL on disk or over the network to generate stubs. See "Endpoint URLs and Generated WSDL in RPCE Web Services" on page 122.

## See also

- “Web Services Introduction” on page 31
- “Plugin Overview” on page 163

## New Entity Syntax Depends on Context

It is important to understand that any ClaimCenter entities that are parameters or return values of SOAP APIs always refer to SOAP entities. The SOAP entities contain data-only versions of entities such as `Claim`. This is true even if calling SOAP APIs from Java code such as Java plugins that call out to SOAP APIs.

If you write Gosu or Java code with full entities that connect to web services, be aware that SOAP entities and full Java entities are in separate package hierarchies. You may need to write code that manually copies properties or object graphs from a SOAP entity to a Gosu or Java full entity, or the other direction, or both.

The following table summarizes important differences in the way you create SOAP entities for web service integration and they way you create full entities in Java or Gosu.

Language	Entity type	Context of Use	Code to create new Guidewire entity instance
Gosu	Full	Most typical Gosu code, including writing your own web service in Gosu.	<p>Simply use the new keyword:</p> <pre>var myInstance = new Address()</pre> <p>In Gosu, some entities have multiple versions of the new constructor. In other words, sometimes you can pass certain parameters inside the parentheses, and there are multiple ways you can create the entity. Refer to the Gosu API Reference in Studio in the Help menu for details of each Guidewire entity.</p>
	SOAP WS-I	WS-I web services do not access Guidewire entities. Instead, the argument and return types are based on XML types and Gosu classes.	WS-I web services cannot access Guidewire entities.
	SOAP RPCE (deprecated API)	Creating a SOAP entity using an incoming web service not hosted by ClaimCenter. Also, in a more rare case, hosted by a different Guidewire application if you use more than one Guidewire application. In this case your code is a SOAP client.	<p>Use the regular new keyword with the correct subpackage of the soap package, using the name of the service as follows:</p> <pre>var myInstance = new soap.SERVICENAME.entity.Address()</pre> <p>For more information about this syntax, see “Calling RPCE Web Services from Gosu” on page 135. For calling web services on the same server, see that same section.</p> <p><b>WARNING</b> In general, avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. See more detailed warning after this table.</p>
Java	Full	Code in a Java plugin or from a Java class called from Gosu using the Java libraries.	<pre>Address.TYPE.newInstance(bundle)</pre> <p>See “Accessing Entity and Typecode Data in Java” on page 631 and “Creating New Entity Instances from Java” on page 640.</p>
	SOAP WS-I	WS-I web services do not access Guidewire entities. Instead, the argument and return types are based on XML types and Gosu classes.	WS-I web services cannot access Guidewire entities.
	SOAP RPCE (deprecated API)	From an external system, you can send or receive entity data. External integration code written in Java compiling against Guidewire entity libraries.	<p>Use the new keyword:</p> <pre>import com.guidewire.cc.webservices.entity.Address; ... a = new Address();</pre>

---

**WARNING** Do not call locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data. Instead, write a Gosu class that performs a similar function as the web service but that does not commit the bundle.

---

## See also

- “Web Services Introduction” on page 31
- “Java and OSGi Support” on page 627

## Public IDs and Integration Code

ClaimCenter creates its own unique ID for every entity in the system after it fully loads in the ClaimCenter system. However, this internal ID cannot be known to an external system while the external system prepares its data. Consequently, if you get or set ClaimCenter information, use unique public ID values to identify an entity from external systems connecting to a Guidewire application.

Your external systems can create this public ID based on its own internal unique identifier, based on an incrementing counter, or based on any system that can guarantee unique IDs. Each entity type must have unique public IDs within its class. For instance, two different `Address` objects cannot have the same public ID.

However, a claim and an exposure may share the same public ID because they are different entities.

If loading two related objects, the incoming request must tell ClaimCenter that they are related. However, the web service client does not know the internal ClaimCenter IDs as it prepares its request. Creating your own public IDs guarantees the web service client can explain all relationships between objects. This is true particularly if entities have complex relationships or if some of the objects already exist in the database.

Additionally, an external system can tell ClaimCenter about changes to an object even though the external system might not know the internal ID that ClaimCenter assigned to it. For example, if the external system wants to change a contact's phone number, the external system only needs to specify the public ID of the contact record.

ClaimCenter allows most objects associated with data to be tagged with a public ID. Specifically, all objects in the *Data Dictionary* that show the `keyable` attribute contain a public ID property. If your API client code does not need particular public IDs, let ClaimCenter generate public IDs by leaving the property blank. However, other non-API import mechanisms require you to define an explicit public ID, for instance database table record import.

If you choose not to define the public ID property explicitly during initial API import, later you can query ClaimCenter with other information. For example, you could pass a contact person's full name or taxpayer ID if you need to find its entity programmatically.

You can specify a new public ID for an object. From Gosu, set the `PublicID` property.

On an RPCE SOAP version of an entity, call its `setPublicID` setter method and pass a public ID string with a maximum of 20 characters. If you want to link to an already-existing object rather than create a new object, additionally set the *reference type* to `ByRef`.

**Note:** The newer WS-I style of web services do not directly manipulate Guidewire entities. Because of this, there is no need for supporting reference types on entities for WS-I web services. In contrast, the older RPCE style of web services do directly manipulate Guidewire entities.

### See also

- For more information about reference types, see “As a SOAP Client, Refer to an Entity By Reference with a Public ID” on page 121.

## Creating Your Own Public IDs

Suppose a company called ABC has two external systems, each of which contains a record with an internal ID of 2224. Each system generates public ID by using the format "`{company}:{system}:{recordID}`" to create unique public ID strings such as "abc:s1:2224" and "abc:s2:2224".

To request ClaimCenter automatically create a public ID for you rather than defining it explicitly, set the public ID to the empty string or to `null`. If a new entity's public ID is blank or `null`, ClaimCenter generates a public ID. The ID is a two-character ID, followed by a colon, followed by a server-created number. For example, "cc:1234". Guidewire reserves for itself all public IDs that start with a two-character ID and then a colon.

Public IDs that you create must never conflict with ClaimCenter-created public IDs. If your external system generates public IDs, you must use a naming convention that prevents conflict with Guidewire-reserved IDs and public IDs created by other external systems.

The prefix for auto-created public IDs is configurable using the `PublicIDPrefix` configuration parameter. If you change this setting, all explicitly-assigned public IDs must not conflict with the namespace of that prefix.

---

**IMPORTANT** Integration code must never set a public IDs to a `String` that starts with a two-character ID and then a colon. Guidewire strictly reserves all such IDs. If you use the `PublicIDPrefix` configuration parameter, integration code that sets explicit public IDs also must not conflict with that namespace. Additionally, plan your public ID naming to support large (long) record numbers. Your system must support a significant number of records over time and stay within the 20 character public ID limit.

---

**See also**

- “`PublicIDPrefix`” on page 55 in the *Configuration Guide*

---

part II

# Web Services



# Web Services Introduction

You can write web service APIs in Gosu and access them from remote systems using the standard web services protocol *SOAP*, the standard Simple Object Access Protocol. Web services provide a language-neutral, platform-neutral mechanism for invoking actions or requesting data from other applications across a network. The SOAP protocol defines request/response mechanisms for translating a function call and its response into XML-based messages, typically across the standard HTTP protocol. ClaimCenter publishes its own built-in web service APIs that you can use.

This topic includes:

- “What are Web Services?” on page 31
- “What Happens During a Web Service Call?” on page 32
- “Reference of All Built-in Web Services” on page 33

## What are Web Services?

Web services define request-and-response APIs that let you call an API on a remote computer, or even the current computer, using an abstracted well-defined interface. A data format called the Web Service Description Language (WSDL) describes available web services that other systems can call using the SOAP protocol. Many languages and third-party packages provide bindings implementations of WSDL and SOAP, including Gosu (built into ClaimCenter), Java, Perl, and other languages.

## Publish or Consume Web Services from Gosu

Gosu natively supports web services in two ways:

- **Publish your Gosu code as new web service APIs** – Write Gosu code that external systems call as a web service using the SOAP protocol. Simply add a single line of code before the definition of a Gosu class.
- **Call web service APIs that external applications publish from your Gosu code** – Write Gosu code that imports and calls web service APIs published by external systems. Gosu parses the WSDL for the service. Gosu uses the WSDL to create Gosu types that enable you to call the remote API. You can call methods on the API and access types from the WSDL, all with a natural Gosu syntax.

## Web Service Types: WS-I versus RPCE

ClaimCenter supports two types of web services:

- **WS-I web services** – WS-I web services use SOAP protocol and are compatible with the WS-I standard.
- **RPC Encoded (RPCE) web services** – RPCE web services are an older deprecated style of web service publishing APIs. Versions of ClaimCenter before version 7.0 support only RPCE web service publishing.

---

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

---

For new web services, publish them using the WS-I APIs. Similarly, when consuming web services from external systems that support RPCE and WS-I, use the WS-I APIs. ClaimCenter retains support for RPCE web services for backward compatibility. Do not create new RPCE web services nor consume RPCE web services if you can use WS-I web services instead.

## Finding the Best Web Service Documentation for Your Needs

The WS-I syntax for publishing and consuming web services differs from the RPCE syntax. You must read the correct documentation. The following table defines which documentation to read.

Type of web service	To publish a Guidewire-hosted web service or call one from Java or .NET...	To consume any web services from Gosu...
WS-I	See “Publishing Web Services (WS-I)” on page 37	See “Calling WS-I Web Services from Gosu” on page 71
RPCE	See “Publishing Web Services (RPCE)” on page 89	See “Calling RPCE Web Services from Gosu” on page 135

---

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

---

### See also

- To compare the ClaimCenter implementation of RPCE web services and WS-I web services, see “WS-I Web Services” on page 122 in the *New and Changed Guide*.
- For background about the WS-I standard and its use of the Document Literal encoding, see “Consuming WS-I Web Service Overview” on page 71.

## What Happens During a Web Service Call?

For all types of web services, ClaimCenter converts the server’s local Gosu objects to and from the flattened text-based format that the SOAP protocol requires. This process is *serialization* and *deserialization*.

For example:

- Suppose you write a web service in Gosu and publish it from ClaimCenter and call it from a remote system. Gosu must deserialize the text-based request into a local object that your Gosu code can access. If one of your web service methods returns a result, ClaimCenter serializes that local in-memory Gosu result object. ClaimCenter serializes it into a text-based reply to the remote system.
- Suppose you use Gosu to call a web service hosted by an external system. Before calling the API, Gosu automatically serializes any API parameters to convert a local object into a flattened form to send to the API. If the remote API returns a result, ClaimCenter deserializes the response into local Gosu objects for your code to examine.

Writing your own custom web service for each integration point is the best approach for maximum performance and maintainability. Guidewire strongly encourages you to write as many web services as necessary to elegantly provide APIs for each integration point.

For example, write new web services to communicate with a check printing service, a legacy financials system, reporting service, or document management system. External systems can query ClaimCenter to calculate values, trigger actions, or to change data within the ClaimCenter database.

Publishing a web service can be as simple as adding one special line of code called an *annotation* immediately before your Gosu class.

For consuming a web service, Gosu creates local objects in memory that represent the remote API. Gosu creates types for every object in the WSDL, and you can create these objects or manipulate properties on them.

#### See also

- For more details about publishing or consuming web services, see “[Finding the Best Web Service Documentation for Your Needs](#)” on page 32.

## Reference of All Built-in Web Services

The built-in web services in ClaimCenter are one of two types:

- WS-I web services
- RPCE web services

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

Web Service Name	Type	Description
<b>Application web services</b>		
IBulkInvoiceAPI	WS-I	Allows external systems to submit bulk invoices. See “ <a href="#">Bulk Invoice Integration</a> ” on page 391.
ClaimAPI	WS-I	Manipulates claim and exposure data. For example, add documents to a claim, create a new claim or FNOL (first notice of loss), re-open an exposure, and add activities to claims. See “ <a href="#">Claim-related Web Services</a> ” on page 151.
ClaimFinancialsAPI	WS-I	Performs various actions on claim financials, such as loading reserves, payments, and recoveries from external systems, and updating the status of payments. See “ <a href="#">Claim Financials Web Services (ClaimFinancialsAPI)</a> ” on page 369.
ContactAPI	WS-I	Performs created, update, and delete actions on contacts, as well as synchronizing ClaimCenter contact IDs with contact IDs in an external address book. If you configure ContactManager to connect to ClaimCenter, ContactManager uses this web service automatically. See “ <a href="#">Contact Web Service APIs</a> ” on page 508.
<b>General web services</b>		
TypeListToolsAPI	WS-I	Retrieves aliases for ClaimCenter typecodes in external systems. See “ <a href="#">Mapping Typecodes to External System Codes</a> ” on page 143.
ImportTools	WS-I	Imports administrative data from an XML file. You must use this only with administrative database tables (entities such as User). This system does not perform complete data validation tests on any other type of imported data. See “ <a href="#">Importing Administrative Data</a> ” on page 145.

Web Service Name	Type	Description
MaintenanceToolsAPI	WS-I	Starts and manages various background processes. The methods of this web service are available only when the server run level is maintenance or higher. See “Maintenance Web Services” on page 146.
SystemToolsAPI	WS-I	Performs various actions related to server run levels, schema and database consistency, module consistency, and server and schema versions. The methods of this web service are available regardless of the server run level. See “System Tools Web Services” on page 147.
WorkflowAPI	WS-I	Performs various actions on a workflow, such as suspending and resuming workflows and invoking workflow triggers. See “Workflow Web Services” on page 149.
ProfilerAPI	WS-I	Sends information to the built-in system profiler. See “Profiling Web Services” on page 149.
MessagingToolsAPI	WS-I	Manages the messaging system remotely for message acknowledgements error recovery. The methods of this web service are available only when the server run level is multiuser. See “Web Services for Handling Messaging Errors” on page 356.
DataChangeAPI	WS-I	Tool for rare cases of mission-critical data updates on running production systems. See “Data Change API Overview” on page 53 in the <i>System Administration Guide</i> .
TemplateToolsAPI	WS-I	Lists and validates Gosu templates available on the server. See “Template Web Service APIs” on page 233.
TableImportAPI	WS-I	Loads operational data from staging tables into operational tables. Typically you use this web service for large-scale data conversions, such as migrating claims from a legacy system to ClaimCenter prior to bringing ClaimCenter into production. The methods of this web service are available only when the server run level is maintenance. See “Importing from Database Staging Tables” on page 415.
ZoneImportAPI	WS-I	Imports geographic zone data from a comma separated value (CSV) file into a staging table, in preparation for loading zone data into the operational table. See “Importing from Database Staging Tables” on page 415.
ILoginAPI	RPCE	<p>Authenticates new connections only to Guidewire-hosted RPCE web services. It is not used for WS-I web services. There is a WS-I web service with a similar name LoginAPI documented later in this table.</p> <ul style="list-style-type: none"> <li>• Use ILoginAPI only to access RPCE web services from languages other than Java 1.5 or Gosu. To learn how to use ILoginAPI with .NET, see “Calling Your RPCE Web Service from Microsoft .NET WSE 3.0” on page 110.</li> <li>• Do not use ILoginAPI for Java development. Instead, use the Java utility class APILocator. See “Calling Your Published RPCE Web Service from Java” on page 107.</li> <li>• Do not use ILoginAPI from Gosu to access web services published by other Guidewire applications. Instead, use the native web services client APIs in the Gosu language. <ul style="list-style-type: none"> <li>• To access WS-I web services, see “Calling WS-I Web Services from Gosu” on page 71.</li> <li>• To access RPCE web services, see “Calling RPCE Web Services from Gosu” on page 135 and “Testing Your RPCE Web Service With soap.local Namespace” on page 103.</li> </ul> </li> </ul> <p><b>WARNING:</b> Do not confuse the RPCE web service ILoginAPI with the WS-I web service LoginAPI. They have different roles.</p>

Web Service Name	Type	Description
LoginAPI	WS-I	<p>The WS-I web service LoginAPI is not used for authentication in typical use. It is only used to test authentication or force a server session for logging. For details, see “Login WS-I Authentication Confirmation” on page 65.</p> <p><b>WARNING:</b> Do not confuse the RPCE web service ILoginAPI with the WS-I web service LoginAPI. They have different roles.</p>
<b>Restricted web services</b>		
ISREEAAuthenticationAPI	RPCE	This web service is reserved for Guidewire use only. Do not use this web service interface. This web service is part of the internal implementation that communicates between ClaimCenter and the optional reporting module for ClaimCenter.

### See also

- For detailed information on the built-in web service API interfaces in ClaimCenter, see “API Reference Javadoc Documentation” on page 20.
- For a comparison of RPCE and WS-I, see “Web Service Types: WS-I versus RPCE” on page 32.
- For information on how to convert from RPCE to WS-I, see “WS-I Web Services” on page 122 in the *New and Changed Guide*.



# Publishing Web Services (WS-I)

You can write web service APIs in Gosu and access them from remote systems using the standard web services protocol *SOAP*. The SOAP protocol defines request/response mechanisms for translating a function call and its response into XML-based messages typically sent across computer networks over the standard HTTP protocol. Web services provide a language-neutral and platform-neutral mechanism for invoking actions or requesting data from another application across a network. ClaimCenter publishes its own built-in web service APIs that you can use.

There are two ways to publish web services:

- **WS-I web services** – WS-I web services use the SOAP protocol that are compatible with the industry standard called WS-I. For new web services, Guidewire strongly recommends that you publish web services using the WS-I APIs.
- **RPC Encoded (RPCE) web services** – RPCE web services are the older style of web service publishing APIs. Versions of ClaimCenter before version 7.0 supported only the RPCE type of web service publishing. ClaimCenter retains support for RPCE web service publishing only for backward compatibility. Do not create new RPCE web services.

This topic includes:

- “WS-I Web Service Publishing Overview” on page 38
- “Publishing WS-I Web Services” on page 43
- “Testing WS-I Web Services with wsi.local” on page 47
- “Generating WS-I WSDL” on page 48
- “Adding Advanced Security Layers to a WS-I Web Service” on page 51
- “WS-I Web Services Authentication Plugin” on page 56
- “Checking for Duplicate External Transaction IDs” on page 57
- “Request or Response XML Structural Transformations” on page 57
- “Reference Additional Schemas in Your Published WSDL” on page 58
- “Validate Requests Using Additional Schemas as Parse Options” on page 58
- “Invocation Handlers for Implementing Preexisting WSDL” on page 59
- “Locale Support” on page 62

- “Setting Response Serialization Options, Including Encodings” on page 63
- “Exposing Typelists and Enumerations As String Values” on page 63
- “Transforming a Generated Schema” on page 64
- “Login WS-I Authentication Confirmation” on page 65
- “Stateful WS-I Session Affinity Using Cookies” on page 65
- “Calling a ClaimCenter WS-I Web Service from Java” on page 66

#### See also

- For the functional differences between RPCE and WS-I web services, see “What are Web Services?” on page 31.
- For the implementation differences between ClaimCenter RPCE and WS-I web services, see “WS-I Web Services” on page 122 in the *New and Changed Guide*.
- For information about the WS-I standard and its Document Literal encoding, see “Calling WS-I Web Services from Gosu” on page 71.

## WS-I Web Service Publishing Overview

Web services define request-and-response APIs that let you call an API on a remote computer, or even the current computer, using an abstracted well-defined interface. A data format called the Web Service Description Language (WSDL) describes available web services that other systems can call using the SOAP protocol. Many languages or third-party packages provide bindings implementations of WSDL and SOAP, including Gosu (built into ClaimCenter), Java, Perl, and other languages.

Gosu natively supports web services in two different ways:

- **Publish your Gosu code as new web service APIs** – Write Gosu code that external systems call as a web service using the SOAP protocol. Simply add a single line of code before the definition of a Gosu class.
- **Call web service APIs that external applications publish from your Gosu code** – Write Gosu code that imports and calls web service APIs published by external systems. Gosu parses the WSDL for the service. Gosu uses the WSDL to create Gosu types that enable you to call the remote API. You can call methods on the API and access types from the WSDL, all with a natural Gosu syntax. For more information, see “Calling WS-I Web Services from Gosu” on page 71.

## Designing Your Web Services

In both cases ClaimCenter converts the server’s local Gosu objects to and from the flattened text-based format required by the SOAP protocol. This process is *serialization* and *deserialization*.

- Suppose you write a web service in Gosu and publish it from ClaimCenter and call it from a remote system. Gosu must deserialize the text-based XML request into a local object that your Gosu code can access. If one of your web service methods returns a value, ClaimCenter serializes that local in-memory Gosu object and serializes it into a text-based XML reply for the remote system.
- Suppose you use Gosu to call a web service hosted by an external system. Before calling the API, Gosu automatically serializes any API parameters to convert a local object into a flattened text-based XML format to send to the API. If the remote API returns a result, ClaimCenter deserializes the XML response into local Gosu objects for your code to use.

Guidewire provides built-in web service APIs for common general tasks and tasks for business entities of ClaimCenter. For a full list, see “Reference of All Built-in Web Services” on page 33. However, writing your own custom web service for each integration point is the best approach for maximum performance and maintainability. Guidewire strongly encourages you to write as many web services as necessary to elegantly provide APIs for each integration point.

For example, write new web services to communicate with a check printing service, a legacy financials system, reporting service, or document management system. External systems can query ClaimCenter to calculate values, trigger actions, or to change data within the ClaimCenter database.

Publishing a web service may be as simple as added one special line of code called an annotation immediately before your Gosu class. Additional customizations may require extra annotations. See “Publishing WS-I Web Services” on page 43 for details.

There are special additional tasks or design decisions that affect how you write your web services, for example:

- **Create custom structures to send only the subset of data you need** – For large Guidewire business data objects (entities), most integration points only need to transfer a subset of the properties and object graph. Do not pass large object graphs, and be aware of any objects that might be very large in your real-world deployed production system. In such cases, you must design your web services to pass your own objects containing only your necessary properties for that integration point, rather than pass the entire entity. For example, if an integration point only needs a record’s main contact name and phone number, create a shell object containing only those properties and the standard public ID property. For details, see “Publishing WS-I Web Services” on page 43.

WS-I web services published by Guidewire applications are forbidden to use a Guidewire entity type as an argument to a method or a return type. Instead, create other kinds of objects that store the important data for that integration point.

---

**IMPORTANT** All published WS-I web services must have no arguments or return types that are Guidewire entity instances.

---

For example, you could change argument and return types to the following:

- **Gosu class instances** – Write Gosu classes that include the important properties for that integration point. For example, a check printing service might only need the amount and the mailing address, but not any associated metadata or notes about the check. To work with WS-I web services, you must add the `final` keyword on the class, and also add the `@WsiExportable` annotation on the class. For example:

```
package example.wsi.myobjects
uses gw.xml.ws.annotation.WsiExportable

@WsiExportable
final class MyCheckPrintInfo {
    var checkID : String
    var checkRecipientDisplayName : String
}
```
- **XML types created with the Guidewire XML modeler** – Use the Guidewire XML modeler tool (also called the *GX modeler*) to generate a model that contains only the desired subset of properties for each entity for each integration point. Then you can import or export XML data using that GX modeler as needed. See “The Guidewire XML (GX) Modeler” on page 302 in the *Gosu Reference Guide*.
- **Be careful of big objects** – If your data set is particularly large, it may be too big to pass over the SOAP protocol in one request. You may need to refactor your code to accommodate smaller requests. If you try to pass too much data over the SOAP protocol in either direction, there can be memory problems that you must avoid in production systems.
- **Design your web service to be testable** – For an example of testing a web service, see “Testing WS-I Web Services with wsi.local” on page 47.
- **Learn bundle and transaction APIs** – To change and commit entity data in the database, learn the bundle APIs. You do not need to use bundle APIs for SOAP APIs that simply get and return unchanged data. See “Committing Entity Data to the Database Using a Bundle” on page 40.

Guidewire provides a Java language binding for published web services. Using these bindings, you can call the published web services from Java program as easy as making a local method invocation. You can use other programming languages if it has a SOAP implementation and can access the web service over the Internet.

Because Java is the typical language for web service client code, this topic usually uses Java syntax and terminology to demonstrate APIs. In some cases, this topic uses Gosu to demonstrate examples as it relates to publishing or testing web services.

If you write APIs to integrate two or more Guidewire applications, you probably will write your web service code in Gosu using its native SOAP syntax rather than Java. See “Calling WS-I Web Services from Gosu” on page 71.

**WARNING** Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server.

## Differences Between Publishing RPCE and WS-I Web Services

There are two types of web services that ClaimCenter supports publishing and consuming:

- **WS-I web services** – WS-I web services use the SOAP protocol that also are compatible with the industry standard called WS-I. For new web services, Guidewire strongly recommends that you publish new web services using the WS-I APIs. Similarly, when consuming web services,
- **RPC Encoded (RPCE) web services** – RPCE web services are the older style of web service publishing APIs. Previous versions of ClaimCenter before version 7.0 only supported this type of web service publishing. ClaimCenter support for RPCE web services remains only for backward compatibility. Do not create new RPCE web services.

### See also

- To compare the ClaimCenter implementation of RPCE web services and WS-I web services, see:
  - “WS-I Web Services” on page 122 in the *New and Changed Guide*.
  - “Web Service Types: WS-I versus RPCE” on page 32.
- For more about the Document Literal encoding that WS-I uses, see “Calling WS-I Web Services from Gosu” on page 71.

## Committing Entity Data to the Database Using a Bundle

There is no default writable bundle for WS-I web services. Bundles are the way that Guidewire applications track changes to entity data. You must define your own new writable bundle if your web service must change any entity data.

To create a new writable bundle, use the `runWithNewBundle` API. See “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide*.

Web services that only get entity data (never modify entity data) never need to create a new bundle.

## Serializable Gosu Classes Must Be Final and Have `@WsiExportable` Annotation

WS-I web services can contain arguments and return values that contain or reference regular instances of Gosu classes, sometimes called Plain Old Gosu Objects (POGOs).

However, the Gosu class must have the following two special qualities:

- The class must be declared as `final`, which as a consequence means it has no subclasses.
- The class must have the annotation `@WsiExportable` or Gosu does not publish the service.

## WS-I Web Service Publishing Quick Reference

The following table summarizes important qualities of WS-I web services.

Feature	WS-I web service behavior
Basic publishing of a WS-I web service	Add the annotation <code>@WsiWebService</code> to the implementation class. This annotation supports one optional argument for the web service namespace. See “Declaring the Namespace for a WS-I Web Service” on page 43. If you do not declare the namespace, Gosu uses the default namespace <code>http://example.com</code> .
Does ClaimCenter automatically generate WSDL files from a running server?	Yes. See “Getting WS-I WSDL from a Running Server” on page 48.
Does ClaimCenter automatically generate WSDL files locally if you regenerate the SOAP API files?	Yes. See “Generating WS-I WSDL” on page 48.
Does ClaimCenter automatically generate JAR files for Java SOAP client code if you regenerate the SOAP API files?	No, but it is easy to generate with the Java built-in utility <code>wsimport</code> . The documentation includes examples. See “Calling a ClaimCenter WS-I Web Service from Java” on page 66.
Can serialize Gosu class instances, sometimes called POGOs: Plain Old Gosu Objects?	Yes, however the Gosu class must have two special qualities. See “Serializable Gosu Classes Must Be Final and Have <code>@WsiExportable Annotation</code> ” on page 40.
Can it serialize a Guidewire entity as an argument type or return type?	No. This is an important difference between RPCE web services and WS-I web services. If you used entity instances as arguments or return types, you must refactor your code to avoid this. Instead, return Gosu class instances that contain only the data you need. (see later in this table for “Can serialize Gosu class instances”)
Can serialize a XSD-based type	Yes, using the <code>XmLElement</code> APIs.
Can you use the Guidewire XML Modeler tool (GX Model) to generate XML types that can be WS-I arguments or return types?	Yes
Can it serialize a Java object as an argument type or return type?	No
Automatically throw <code>SOAPException</code> exceptions?	No. Declare the actual exceptions you want thrown. In general this requirement reduces typical WSDL size.  If you want identical behavior to RPCE web services, add <code>Throws (ServerStateException, PermissionException, BadIdentifierException)</code> onto your Gosu class. However, that set probably is not what you really want. Carefully consider what exceptions you really want.
Bundle handling for changing entity data?	A <i>bundle</i> is a container for entities that helps ClaimCenter track what changed in a database transaction. There is no default bundle for WS-I web services. See “Committing Entity Data to the Database Using a Bundle” on page 40.
Logging in to the server	Each WS-I request embeds necessary authentication and security information in each request. If you use Guidewire authentication, you must add the appropriate SOAP headers before making the connection.
Package name of web services from the SOAP API client perspective	The package in which you put your web service implementation class defines the package for the web service from the SOAP API client perspective. The biggest benefit from this change is reducing the chance of namespace collisions in general. In addition, the web service namespace URL helps prevent namespace collisions. See “Declaring the Namespace for a WS-I Web Service” on page 43.
Calling a local version of the web service from Gosu	Gosu creates types that use the original package name to avoid namespace collisions.  API references in the package <code>wsi.local.ORIGINAL_PACKAGE_NAME</code>  You must call the command line command to rebuild the local files: <code>ClaimCenter/bin/gwcc regen-wsi-local</code>

## WS-I Web Service Publishing Annotation Reference

The following table lists annotations related to WS-I web service declaration and configuration:

Annotation	Description	For more information
<b>Web service implementation class annotations</b>		
@WsiWebService	Declare a class as implementing a WS-I web service	"Publishing WS-I Web Services" on page 43
@WsiAvailability	Set the minimum server run level for this service	"Specifying Minimum Run Level for a WS-I Web Service" on page 44
@WsiPermissions	Set the required permissions for this service	"Specifying Required Permissions for a WS-I Web Service" on page 44
@WsiExposeEnumAsString	Instead of exposing typelist types and enumerations as enumerations in the WSDL, you can expose them as <code>String</code> values.	"Exposing Typelists and Enumerations As String Values" on page 63
@WsiWebMethod	The @WsiWebMethod annotation can do two things: <ul style="list-style-type: none"> <li>Override the web service operation name to something other than the default, which is the method name.</li> <li>Suppress a method from generating a web service operation even though the method has <code>public</code> access.</li> </ul>	"Overriding a Web Service Method Name or Visibility" on page 44
@WsiSerializationOptions	Add serialization options for web service responses, for example supporting encodings other than UTF-8.	"Setting Response Serialization Options, Including Encodings" on page 63
@WsiAdditionalSchemas	Expose additional schemas to web service clients in the WSDL. Use this to provide references to schemas that might be required but are not automatically included.	"Reference Additional Schemas in Your Published WSDL" on page 58
@WsiParseOptions	Add validation of incoming requests using additional schemas in addition to the automatically generated schemas.	"Validate Requests Using Additional Schemas as Parse Options" on page 58
@WsiRequestTransform @WsiResponseTransform	Add transformations of incoming or outgoing data as a byte stream. Typically you would use this to add advanced layers of authentication or encryption. Contrast to the annotations in the next row, which operate on XML elements.	"Data Stream Transformations" on page 51
@WsiRequestXmlTransform @WsiResponseXmlTransform	Add transformations of incoming or outgoing data as XML elements. Use this for transformations that do not require access to byte data. Contrast to the annotations in the previous row, which operate on a byte stream.	"Request or Response XML Structural Transformations" on page 57
@WsiSchemaTransform	Transform the generated schema that ClaimCenter publishes.	"Transforming a Generated Schema" on page 64
@WsiInvocationHandler	Perform advanced implementation of a web service that conforms to externally-defined standard WSDL. This is for unusual situations only. This approach limits protection against some types of common errors.	"Invocation Handlers for Implementing Preexisting WSDL" on page 59
@WsiCheckDuplicateExternalTransaction	Detect duplicate operations from external systems that change data	"Checking for Duplicate External Transaction IDs" on page 57

Annotation	Description	For more information
<b>Other annotations to support serialization</b>		
@WsiExportable	Add this annotation on a Gosu class to indicate that it supports serialization with WS-I web services. You must additionally add the <code>final</code> keyword to the class to prevent subclasses.	"Serializable Gosu Classes Must Be Final and Have @WsiExportable Annotation" on page 40

## Publishing WS-I Web Services

To publish a WS-I web service, use the `@WsiWebService` annotation on a class. Gosu publishes that class automatically when the server runs.

For example:

```
package example
uses gw.xml.ws.annotation.WsiWebService

@WsiWebService
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }

}
```

Choose your package declaration carefully. The package in which you put your web service implementation class defines the package for the web service from the SOAP API client perspective. This reduces the chance of namespace collisions.

### See also

- For the differences between RPCE and WS-I, see the table in “Differences Between Publishing RPCE and WS-I Web Services” on page 40.

## Declaring the Namespace for a WS-I Web Service

Each web service is defined within a *namespace*, which is similar to how a Gosu class or Java class is within a package. Specify the namespace as a URL in each web service declaration. The namespace is a `String` that looks like a standard HTTP URL but represents a namespace for all objects in the service’s WSDL definition. The namespace is not typically a URL for an actual downloadable resource. Instead it is an abstract identifier that disambiguates objects in this service from objects in another service.

A typical namespace specifies your company and perhaps other meaningful disambiguating or grouping information about the purpose of the service, possibly including a version number. For example:

- `"http://mycompany.com"`
- `"http://mycompany.com/xsds/messaging/v1"`

You can specify the namespace for each web service by passing a namespace as a `String` argument to the `@WsiWebService` annotation.

For example:

```
package example
uses gw.xml.ws.annotation.WsiWebService

@WsiWebService("http://mycompany.com")
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }

}
```

```
}
```

You can omit the namespace declaration entirely and provide no arguments to the annotation. If you omit the namespace declaration in the constructor, the default is "example.com".

Most tools that create stub classes for web services use the namespace to generate a package namespace for related types. For examples of how the Java `wsimport` tool uses the namespace, see "Calling a ClaimCenter WS-I Web Service from Java" on page 66.

## Specifying Minimum Run Level for a WS-I Web Service

To set the minimum run level for the service, add the annotation `@WsiAvailability` and pass the run level at which this web service is first usable. The choices include DAEMONS, MAINTENANCE, MULTIUSER, and STARTING. The default is NODAEMONS.

For example:

```
package example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.

@WsiAvailability(MAINTENANCE)
@WsiWebService
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }

}
```

## Specifying Required Permissions for a WS-I Web Service

To add required permissions, add the annotation `@WsiPermissions` and pass an array of permission types (`SystemPermissionType[ ]`). The default permission is SOAPADMIN. If you provide an explicit list of permissions, you can choose to include SOAPADMIN explicitly or omit it. If you omit SOAPADMIN from our list of permissions, your web service does not require SOAPADMIN permission. If you pass an empty list of permissions, your web service does not require authentication.

For example:

```
package example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiPermissions

@WsiPermissions({}) // A blank list means no authentication needed.
@WsiWebService

class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }

}
```

## Overriding a Web Service Method Name or Visibility

You can override the names and visibility of individual web service methods in several ways.

## Overriding Web Service Method Names

By default, the web service operation name for a method is simply the name of the method. You can override the name by adding the `@WsiWebMethod` annotation immediately before the declaration for the method on the implementation class. Pass a `String` value for the new name for the operation. For example:

```
@WsiWebMethod("newMethodName")
public function helloWorld() : String {
    return "Hello!"
}
```

## Hiding Public Web Service Methods

By default, Gosu publishes one web service operation for each method marked with `public` availability. For public methods, you can exclude the method from the web service by adding the `@WsiWebMethod` annotation immediately before the declaration for the method on the implementation class. Pass the value `true` to exclude (suppress) publishing that public method. For example:

```
@WsiWebMethod(true)
public function helloWorld() : String {
    return "Hello!"
}
```

If you pass the value `false` instead of `true` to the `@WsiWebMethod` annotation, there is no different behavior compared to omitting the annotation. If you use this annotation on a non-public method, the exclusion behavior has no effect. It never forces a non-public method to be visible on the web service.

## Overriding Method Names and Visibility with a Single `@WsiWebMethod` Annotation

You can combine both of these features of the `@WsiWebMethod` annotation by passing both arguments. For example:

```
@WsiWebMethod("newMethodName", true)
public function helloWorld() : String {
    return "Hello!"
}
```

## WS-I Web Service Invocation Context

In some cases your web service may need additional context for incoming or outgoing information. For example, to get or set HTTP or SOAP headers. You can access this information in your implementation class by adding an additional method argument to your class of type `WsiInvocationContext`. Make this new object the last argument in the argument list.

Unlike typical method arguments on your implementation class, this method parameter does not become part of the operation definition in the WSDL. Your web service code can use the `WsiInvocationContext` object to get or set important information as needed.

The following table lists properties on `WsiInvocationContext` objects and whether each property is writable.

Property	Description	Readable	Writable
<b>Request Properties</b>			
<code>HttpServletRequest</code>	A servlet request of type <code>HttpServletRequest</code>	Yes	No

Property	Description	Readable	Writable
MtomEnabled	<p>A boolean value that specifies whether to optionally support sending MTOM attachments to the WS-I web service client in any data returned from an API. The W3C Message Transmission Optimization Mechanism (MTOM) is a method of efficiently sending binary data to and from web services as attachments outside the normal response body.</p> <p>If <code>MtomEnabled</code> is <code>true</code>, ClaimCenter can send MTOM in results. Otherwise, MTOM is disabled. The default is <code>false</code>.</p> <p>This property does not affect MTOM data sent to a published web service. Incoming MTOM data is always supported.</p> <p>This property does not affect MTOM support where Gosu is the client to the web service request. See "MTOM Attachments with Gosu as Web Service Client" on page 86.</p>	Yes	Yes
RequestHttpHeaders	Request headers of type <code>HttpHeaders</code>	Yes	No
RequestEnvelope	Request envelope of type <code>XmLElement</code>	Yes	No
RequestSoapHeaders	Request SOAP headers of type <code>XmLElement</code>	Yes	No
<b>Response Properties</b>			
ResponseHttpHeaders	Response HTTP headers of type <code>HttpHeaders</code>	Yes	The property value is read-only, but you can modify the object it references
ResponseSoapHeaders	Response SOAP headers of type <code>List&lt;XmLElement&gt;</code>	Yes	The property value is read-only, but you can modify the object it references
<b>Output serialization</b>			
XmLSerializationOptions	<p>XML serialization options of type <code>XmLSerializationOptions</code>. For more information on XML serialization, including the properties of <code>XmLSerializationOptions</code> objects, see "Exporting XML Data" on page 275 in the <i>Gosu Reference Guide</i>.</p>	Yes	Yes

You can use these APIs to modify response headers from your web service and read request headers as needed.

For example, suppose you want to copy a specific request SOAP header XML object and copy to the response SOAP header object. Create a private method to get a request SOAP header XML object:

```
private function getHeader(name : QName, context : WsiInvocationContext) : XmLElement {
    for (h in context.RequestSoapHeaders.Children) {
        if (h.QName.equals(name))
            return h;
    }
    return null
}
```

From a web service operation method, declare the invocation context optional argument to your implementation class. Then you can use code with the invocation context such as the following to get the incoming request header and add it to the response headers:

```
function doWork(..., context : WsiInvocationContext) {
    var rtnHeader = getHeader(TURNAROUND_HEADER_QNAME, context)
    context.ResponseSoapHeaders.add(rtnHeader)
```

```
    // do your main work of your web service operation  
}
```

## Testing WS-I Web Services with wsi.local

For testing purposes only, you can call WS-I web services published from the same ClaimCenter server. To call a WS-I web service on the same server, you must generate WSDL files into the class file hierarchy so Gosu can access the service. This permits you to write unit tests that access the WSDL files over the SOAP protocol from Gosu.

**WARNING** Guidewire does not support calls to SOAP APIs published on the same server in a production system. If you think you need to do so, please contact Customer Support.

To regenerate the WSDL for all local web services in the class hierarchy, open a command prompt and type:

```
ClaimCenter/bin/gwcc regen-wsi-local
```

ClaimCenter generates the WSDL in the following location in the class hierarchy. The WSDL is in the `wsi.local` package, followed by fully qualified name of the web service:

```
wsi.local.FQNAME.wsdl
```

To use the class, simply prefix the text `wsi.local.` (with a final period) to the fully-qualified name of your API implementation class.

For example, suppose the web service implementation class is:

```
mycompany.ws.v100.EchoAPI
```

The `regen-wsi-local` tool generates the following WSDL file:

```
ClaimCenter/modules/configuration/gsrc/wsi/local/mycompany/ws/v100/EchoAPI.wsdl
```

Call this web service locally with the syntax:

```
var api = new wsi.local.mycompany.ws.v100.EchoAPI()
```

If you change the code for the web service and the change potentially changes the WSDL, regenerate the WSDL.

ClaimCenter includes with WSDL for some local services in the default configuration. These WSDL files are in Studio modules other than `configuration`. If ClaimCenter creates new local WSDL files from the `regen-wsi-local` tool, it creates new files in the `configuration` module.

**WARNING** The `wsi.local.*` namespace exists only to call web services from unit tests. It is unsafe to write production code that uses these `wsi.local.*` types.

To reduce the chance of accidental use of `wsi.local.*` types, Gosu prevents using these types in method signatures of published WS-I web services.

## Writing Unit Tests for Your Web Service

It is good practice to design your web services to be testable. At the time you design your web service, think about the kinds of data and commands your service handles. Consider your assumptions about the arguments to your web service, what use cases your web service handles, and which use cases you want to test. Then, write a series of GUnit tests that use the `wsi.local.*` namespace for your web service.

For example, you created the following web service.

```
package example  
  
uses gw.xml.ws.annotation.WsiWebService  
  
@WsiWebService("http://mycompany.com")  
class HelloWorldAPI {
```

```

    public function helloWorld() : String {
        return "Hello!"
    }

}

```

The following sample Gosu code is a GUnit test of the preceding `HelloWorldAPI` web service.

```

package example

uses gw.testharness.TestBase

class HelloWorldAPITest extends gw.testharness.TestBase {

    construct() {

    }

    public function testMyAPI() {
        var api = new wsi.local.example.helloworldapi.HelloWorldAPI()

        api.Config.Guidewire.Authentication.Username = "su"
        api.Config.Guidewire.Authentication.Password = "gw"
        var res = api.helloWorld();

        print("result is: " + res);

        TestBase.assertEquals("Expected 'Hello!', 'Hello!', res)
        print("we got to the end of the test without exceptions!")
    }

}

```

For more thorough testing, test your web service from integration code on an external system. To assure your web service scales adequately, test your web service with as large a data set and as many objects as potentially exist in your production system. To assure the correctness of database transactions, test your web service to exercise all bundle-related code.

## Generating WS-I WSDL

### Getting WS-I WSDL from a Running Server

Typically, you get the WSDL for a WS-I web service from a running server over the network. This approach encourages all callers of the web services to use the most current WSDL. In contrast, generating the WSDL and copying the files around risks callers of the web service using outdated WSDLs. You can get the most current WSDL for a web service from any computer on your network that publishes the web service. In a production system, code that calls a web service typically resides on computers that are separate from the computer that publishes the web service.

ClaimCenter publishes a WS-I web service WSDL at the following URL:

`SERVER_URL/WEB_APP_NAME/ws/WEB_SERVICE_PACKAGE/WEB_SERVICE_CLASS_NAME?WSDL`

A published WSDL may make references to schemas at the following URL:

`SERVER_URL/WEB_APP_NAME/ws/SCHEMA_PACKAGE/SCHEMA_FILENAME`

For example, ClaimCenter generates and publishes the WSDL for web service class `ws.eg.WebService.TestWsiService` at the location on a server with web application name cc:

`http://localhost:PORTNUM/cc/ws/eg/webservice/TestWsiService?WSDL`

Java includes a built-in command called `wsimport` that generates Java from the WSDL published on a running server. For more information, see “Calling a ClaimCenter WS-I Web Service from Java” on page 66

## WSDL and Schema Browser

If you publish web services from a Guidewire application, you can view the WSDL and a schema browser available at the following URL:

*SERVER\_URL/WEB\_APP\_NAME/ws*

For example:

<http://localhost:8580/cc/ws>

From there, you can browse for:

- Document/Literal Web Services
- Supporting Schemas
- Generated Schemas
- Supporting WSDL files

## Example WSDL

The following example demonstrates how a simple Gosu web service translates to WSDL. This example uses the following simple example web service.

```
package example

uses gw.xml.ws.annotation.WsiWebService

@WsiWebService
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }
}
```

ClaimCenter publishes the WSDL for the `HelloWorldAPI` web service at this location:

<http://localhost:PORTNUMBER/cc/ws/example>HelloWorldAPI?WSDL>

ClaimCenter generates the following WSDL for the `HelloWorldAPI` web service.

```
<?xml version="1.0"?>
<!-- Generated WSDL for example.HelloWorldAPI web service -->
<wsdl:definitions targetNamespace="http://example.com/example/HelloWorldAPI"
    name="HelloWorldAPI" xmlns="http://example.com/example/HelloWorldAPI"
    xmlns:gw="http://guidewire.com/xsd" xmlns:soap11="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

    <wsdl:types>
        <xss:schema targetNamespace="http://example.com/example/HelloWorldAPI"
            elementFormDefault="qualified" xmlns:xss="http://www.w3.org/2001/XMLSchema">
            <!-- helloWorld() : java.lang.String -->
            <xss:element name="helloWorld">
                <xss:complexType/>
            </xss:element>
            <xss:element name="helloWorldResponse">
                <xss:complexType>
                    <xss:sequence>
                        <xss:element name="return" type="xss:string" minOccurs="0"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
        </xss:schema>
    </wsdl:types>
    <wsdl:portType name="HelloWorldAPIPortType">

        <wsdl:operation name="helloWorld">
            <wsdl:input name="helloWorld" message="helloWorld"/>
            <wsdl:output name="helloWorldResponse" message="helloWorldResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldAPISoap12Binding" type="HelloWorldAPIPortType">
        <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="helloWorld">
            <soap12:operation style="document"/>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

```

<wsdl:input name="helloWorld">
  <soap12:body use="literal"/>
</wsdl:input>
<wsdl:output name="helloWorldResponse">
  <soap12:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:binding name="HelloWorldAPISoap11Binding" type="HelloWorldAPIPortType">

  <soap11:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  <wsdl:operation name="helloWorld">
    <soap11:operation style="document"/>
    <wsdl:input name="helloWorld">
      <soap11:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="helloWorldResponse">
      <soap11:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloWorldAPI">
  <wsdl:port name="HelloWorldAPISoap12Port" binding="HelloWorldAPISoap12Binding">
    <soap12:address location="http://localhost:8180/cc/ws/example/HelloWorldAPI"/>
    <gw:address location="${cc}/ws/example/HelloWorldAPI"/>
  </wsdl:port>
  <wsdl:port name="HelloWorldAPISoap11Port" binding="HelloWorldAPISoap11Binding">
    <soap11:address location="http://localhost:8180/cc/ws/example/HelloWorldAPI"/>
    <gw:address location="${cc}/ws/example/HelloWorldAPI"/>
  </wsdl:port>
</wsdl:service>
<wsdl:message name="helloWorld">
  <wsdl:part name="parameters" element="helloWorld"/>
</wsdl:message>
<wsdl:message name="helloWorldResponse">
  <wsdl:part name="parameters" element="helloWorldResponse"/>
</wsdl:message>
</wsdl:definitions>

```

The preceding generated WSDL defines multiple *ports* for this web service. A *port* in the context of a web service is unrelated to ports in the TCP/IP network transport protocol. Web service ports are alternative versions of a published web service. The preceding WSDL defines a SOAP 1.1 version and a SOAP 1.2 version of the HelloWorldAPI web service.

## Generating WS-I WSDL On Disk

Typically, callers of a WS-I web service get the WSDL from the server where the web service runs to assure they use the current WSDL, not an outdated version. Also, you can get that WSDL from anywhere on your network. In a production system, your SOAP client code typically runs on computers that are separate from the computer where your SOAP server code runs.

However, there are special situations in which you might want to generate the WSDL file locally on the server. Locally generated WSDL files requires a special annotation in your web service code and a special regeneration tool step.

## Special Annotation to Generate WSDL On Disk

To generate WSDL for a WS-I web service locally on the server where the service runs, you must add the annotation `@WsiGenInToolkit` to the web service implementation class. Most WS-I web services included with ClaimCenter do not have the annotation. If you want to generate the WSDL locally for these web services, modify the implementation file in Studio to add the `@WsiGenInToolkit` annotation.

For example, the following web service implementation class has the annotation before the class declaration:

```

package example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiGenInToolkit

```

```
@WsiWebService  
@WsiGenInToolkit  
  
class HelloWorldAPI {  
  
    public function helloWorld() : String {  
        return "Hello!"  
    }  
}
```

## Command Line tool to Generate WSDL On Disk

To generate the WSDL for all the WS-I web services with the annotation `@WsiGenInToolkit`, at a command prompt type the command:

```
ClaimCenter/bin/gwcc regen-soap-api
```

Look for the locally generated WSDL files in the directory:

```
ClaimCenter/soap-api/wsi/wsdl
```

## Generating SOAP APIs for Use with Unit Tests From Gosu

To support writing Gosu unit tests (JUnit tests), ClaimCenter can generate WSDL within the class file hierarchy for any published WS-I web services. This permits you to write unit tests that access the WSDL files over the SOAP protocol from Gosu. Other than for the purpose of unit tests, Guidewire does not support to calls from a ClaimCenter server into the same server over the SOAP protocol.

### See also

- “Testing WS-I Web Services with wsi.local” on page 47

## Adding Advanced Security Layers to a WS-I Web Service

For security options beyond simple HTTP authentication and Guidewire authentication, you can use an additional set of APIs to implement additional layers of security. For example, you might want to add additional layers of encryption, digital signatures, or other types of authentication or security. From the SOAP server side, you add advanced security layers to outgoing requests by applying transformations to the data stream of the request.

### Data Stream Transformations

#### Transformations on Data Streams

You can transform the data stream by processing it incrementally, byte by byte. For example, you can implement an encryption security layer by transforming the request data stream incrementally. Alternatively, you can transform the data stream by processing it as an entire unit at one time. For example, you must implement a digital signature authentication layer by transforming the entire request data stream at one time.

You can apply multiple types of transformations to the request data stream to add multiple security layers to your web service. The order of your transformations is important. For example, an encryption transformation followed by a digital signature transformation produces a different request data stream than a digital signature transformation followed by an encryption transformation.

If your desired transformation operates more naturally on XML elements and not a byte stream, instead consider using the APIs in “Request or Response XML Structural Transformations” on page 57

## Accessing Data Streams

To access the data stream for a request, Gosu provides an annotation (`@WsiRequestTransform`) to inspect or transform an incoming request data stream. Gosu provides another annotation (`@WsiResponseTransform`) to inspect or transform an outgoing response data stream. Both annotations take a Gosu block that takes an input stream (`java.io.InputStream`) and returns another input stream. Gosu calls the annotation block for every request or response, depending on the type of annotation.

## Example of Data Stream Request and Response Transformations

The following example implements a request transform and a response transform to apply a simple encryption security layer to a web service.

The example applies the same incremental transformation to the request and the response data streams. The transformation processes the data streams byte by byte to change the bits in each byte from a 1 to a 0 or a 0 to a 1. The example transformation code uses the Gosu bitwise exclusive OR (XOR) logical operator (`^`) to perform the bitwise changes on each byte. The XOR logical operator is a *symmetrical* operation. If you apply XOR operation to a data stream and then apply the operation again to the transformed data stream, you obtain the original data stream. Therefore, transforming the incoming request data stream by using the XOR operation encrypts the data. Conversely, transforming the outgoing response data stream by using the XOR operation decrypts the data.

```
package gw.webservice.example

uses gw.util.StreamUtil
uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiRequestTransform
uses gw.xml.ws.annotation.WsiResponseTransform
uses java.io.ByteArrayInputStream
uses java.io.InputStream

@WsiWebService

// Specify data stream transformations for web service requests and responses.
@WsiRequestTransform(WsiTransformTestService._xorTransform)
@WsiResponseTransform(WsiTransformTestService._xorTransform)

class WsiTransformTestService {

    // Declare a static variable to hold a Gosu block that encrypts and decrypts data streams.
    public static var _xorTransform(inputStream : InputStream) : InputStream = \ inputStream ->{

        // Get the input stream, and store it in a byte array.
        var bytes = StreamUtil.getContent(inputStream)

        // Encrypt the bits in each byte.
        for (b in bytes index idx) {
            bytes[idx] = (b ^ 17) as byte // XOR encryption with "17" as the encryption mask
        }

        return new ByteArrayInputStream(bytes)
    }

    function add(a : int, b : int) : int {
        return a + b
    }
}
```

In the preceding example, the request transformation and the response transformation use the same Gosu block for transformation logic because the block uses a symmetrical algorithm. In a typical production scenario however, the request transformation and the response transformation use different Gosu blocks because their transformation logic differs.

### See also

- “Using WSS4J for Encryption, Signatures, and Other Security Headers” on page 53
- “Bitwise Exclusive OR (^)” on page 67 in the *Gosu Reference Guide*

## Applying Multiple Security Layers to a WS-I Web Service

Whenever you apply multiple layers of security to your web service, the order of substeps in your request and response transformation blocks is critical. Typically, the order of substeps in your response block reverses the order of substeps in your request block. For example, if you encrypt and then add a digital signature to the response data stream, remove the digital signature before decrypting the request data stream. If you remove a security layers from your web service, assure the remaining layers preserve the correct order of substeps in the transformation blocks.

Using WSS4J for Encryption, Signatures, and Other Security Headers

The following example uses the Java utility WSS4J to implement encryption, digital cryptographic signatures, and other security elements (a timestamp). This example has three parts.

## Part 1 of the Example that Uses WSS4J

The first part of the example is a utility class called `demo.DemoCrypto` that implements an input stream encryption routine that adds a timestamp, then a digital signature, then encryption. To decrypt the input stream for a request, the utility class knows how to decrypt the input stream and then validate the digital signature.

Early in the encryption (`addCrypto`) and decryption (`removeCrypto`) methods, the code parses, or inflates, the XML request and response input streams into hierarchical DOM trees that represent the XML. The methods call the internal class method `parseDOM` to parse input streams into DOM trees.

Parsing the input streams in DOM trees is an important step. Some of the encryption information, such as time-stamps and digital signatures, must be added in a particular place in the SOAP envelope. At the end of the encryption and decryption methods, the code serializes, or flattens, the DOM trees back into XML request and response input streams. The methods call the internal class method `serializeDOM` to serialize DOM trees back into input streams.

```
package gw.webservice.example

uses gw.api.util.ConfigAccess
uses java.io.ByteArrayInputStream
uses java.io.ByteArrayOutputStream
uses java.io.InputStream
uses java.util.Vector
uses java.util.Properties
uses java.lang.RuntimeException
uses javax.xml.parsers.DocumentBuilderFactory
uses javax.xml.transform.TransformerFactory
uses javax.xml.transform.dom.DOMSource
uses javax.xml.transform.stream.StreamResult
uses org.apache.ws.security.message.*
uses org.apache.ws.security.*  
uses org.apache.ws.security.handler.*

// Demonstration input stream encryption and decryption functions.
// The layers of security are a timestamp, a digital signature, and encryption.
class DemoCrypto {

    // Encrypt an input stream.
    static function addCrypto(inputStream : InputStream) : InputStream {
        var crypto = getCrypto()

        // Parse the input stream into a DOM (Document Object Model) tree.
        var domEnvironment = parseDOM(inputStream)

        var securityHeader = new WSSecHeader()
        securityHeader.insertSecurityHeader(domEnvironment);

        var timeStamp = new WSSecTimestamp();
        timeStamp.setTimeToLive(600)
        domEnvironment = timeStamp.build(domEnvironment, securityHeader)

        var signer = new WSSecSignature();
        signer.setUserInfo("ws-client", "client-password")
        var parts = new Vector()
        parts.add(new WSEncryptionPart("Timestamp",
            "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd")
```

```

        "Element"))
parts.add(new WSEncryptionPart("Body",
    gw.xsd.w3c.soap12_envelope.Body.$QNAME.NamespaceURI, "Element"));
signer.setParts(parts);
domEnvironment = signer.build(domEnvironment, crypto, securityHeader);

var encrypt = new WSSecEncrypt()
encrypt.setUserInfo("ws-client", "client-password")
// encryptionParts=
// {Element}{http://www.w3.org/2000/09/xmldsig#}Signature;{Content}
// {http://schemas.xmlsoap.org/soap/envelope;}Body
parts = new Vector()
parts.add(new WSEncryptionPart("Signature", "http://www.w3.org/2000/09/xmldsig#", "Element"))
parts.add(new WSEncryptionPart("Body", gw.xsd.w3c.soap12_envelope.Body.$QNAME.NamespaceURI,
    "Content"));
encrypt.setParts(parts)
domEnvironment = encrypt.build(domEnvironment, crypto, securityHeader);

// Serialize the modified DOM tree back into an input stream.
return new ByteArrayInputStream(serializeDOM(domEnvironment.DocumentElement))
}

// Decrypt an input stream.
static function removeCrypto(inputStream : InputStream) : InputStream {

// Parse the input stream into a DOM (Document Object Model) tree.
var envelope = parseDOM(inputStream)

var secEngine = WSSEngine.getInstance();
var crypto = getCrypto()
var results = secEngine.processSecurityHeader(envelope, null, \ callbacks ->{
    for (callback in callbacks) {
        if (callback typeis WSPasswordCallback) {
            callback.Password = "client-password"
        }
    }
    else {
        throw new RuntimeException("Expected instance of WSPasswordCallback")
    }
}, crypto);

for (result in results) {
    var eResult = result as WSSEngineResult
    // Note: An encryption action does not have an associated principal.
    // Only Signature and UsernameToken actions return a principal
    if (eResult.Action != WSConstants.ENCR) {
        print(eResult.Principal.Name);
    }
}

// Serialize the modified DOM tree back into an input stream.
return new ByteArrayInputStream(serializeDOM(envelope.DocumentElement))
}

// Private function to create a map of WSS4J cryptographic properties.
private static function getCrypto() : org.apache.ws.security.components.crypto.Crypto {

var cryptoProps = new Properties()
cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.alias", "ws-client")
cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.password", "client-password")
cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.type", "jks")
cryptoProps.put("org.apache.ws.security.crypto.merlin.file", ConfigAccess.getConfigFile(
    "config/etc/client-keystore.jks").CanonicalPath)
cryptoProps.put("org.apache.ws.security.crypto.provider",
    "org.apache.ws.security.components.crypto.Merlin")

return org.apache.ws.security.components.crypto.CryptoFactory.getInstance(cryptoProps)
}

// Private function to parse an input stream into a hierarchical DOM tree.
private static function parseDOM(inputStream : InputStream) : org.w3c.dom.Document {

var factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);

```

```

        return factory.newDocumentBuilder().parse(inputStream);
    }

    // Private function to serialize a hierarchical DOM tree into an input stream.
    private static function serializeDOM(element : org.w3c.dom.Element) : byte[] {
        var transformerFactory = TransformerFactory.newInstance();
        var transformer = transformerFactory.newTransformer();
        var baos = new ByteArrayOutputStream();
        transformer.transform(new DOMSource(element), new StreamResult(baos));

        return baos.toByteArray();
    }

}

```

### Part 2 of the Example that Uses WSS4J

The next part of this example is a WS-I web service implementation class written in Gosu. The following sample Gosu code implements the web service in the class `demo.DemoService`.

```

package gw.webservice.example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiRequestTransform
uses gw.xml.ws.annotation.WsiResponseTransform
uses gw.xml.ws.annotation.WsiPermissions
uses gw.xml.ws.annotation.WsiAvailability

@WsiWebService
@WsiAvailability(NONE)
@WsiPermissions({})
@WsiRequestTransform(\ inputStream ->DemoCrypto.removeCrypto(inputStream))
@WsiResponseTransform(\ inputStream ->DemoCrypto.addCrypto(inputStream))

// This web service computes the sum of two integers. The web service decrypts incoming SOAP
// requests and encrypts outgoing SOAP responses.
class DemoService {

    // Compute the sum of two integers.
    function add(a : int, b : int) : int {
        return a + b
    }

}

```

Some things to notice about the preceding web service implementation class:

- The web service itself simply adds two numbers, but the service has a request and response transformation.
- The request transformation removes and confirms the cryptographic layer on the request, including the digital signature and encryption. The request transformation calls `DemoCrypto.removeCrypto(inputStream)`.
- The response transformation adds the cryptographic layer on the response. The response transformation calls `DemoCrypto.addCrypto(inputStream)`

### Part 3 of the Example that Uses WSS4J

The third and final part of this example is code to test this web service.

```

var webService = new wsi.local.demo.demoservice.DemoService()
webService.Config.RequestTransform = \ inputStream ->demo.DemoCrypto.addCrypto(inputStream)
webService.Config.ResponseTransform = \ inputStream ->demo.DemoCrypto.removeCrypto(inputStream)
print(webService.add(3, 5))

```

Paste this code into the Gosu Scratchpad and run it.

#### See also

- “Testing WS-I Web Services with `wsi.local`” on page 47

## WS-I Web Services Authentication Plugin

To handle the name/password authentication for a user connecting to WS-I web services, ClaimCenter delegates this job to the currently registered implementation of the `WebservicesAuthenticationPlugin` plugin interface. There must always be a registered version of this plugin, otherwise web services that require permissions cannot authenticate successfully.

The `WebservicesAuthenticationPlugin` plugin interface supports WS-I web service connections only. This plugin does not support authentication for the older style of RPCE web services nor for authenticating login from the application user interface.

### The Default Web Services Authentication Plugin Implementation

To authenticate web service requests, it is common to set up non-human Guidewire application users through regular ClaimCenter user administration. These non-human users never use their usernames and passwords to sign into the ClaimCenter user interface. Instead, an external system passes the username and password in the request headers of its WS-I web service requests to authenticate.

The default configuration of ClaimCenter has a registered built-in implementation of WS-I web service authentication plugin. The class is called `gw.plugin.security.DefaultWebservicesAuthenticationPlugin`. In the default configuration of ClaimCenter, this class does two things:

1. Looks at HTTP request headers for WS-I authentication information.
2. Performs authentication against the local ClaimCenter users in the database. This class calls the registered implementation of the `AuthenticationServicePlugin` plugin interface.

---

**IMPORTANT** If you write your own implementation of the `AuthenticationServicePlugin` plugin interface, be aware of this interaction with WS-I web service authentication. For example, you might want LDAP authentication for most users, but for web service authentication to authenticate against the current ClaimCenter application administrative data.

To authenticate only some user names to Guidewire application credentials, your `AuthenticationServicePlugin` code must check the user name and compare to a list of special web service user names. If the user name matches, do not use LDAP and instead authenticate with the local application administrative data. To do this in your `AuthenticationServicePlugin` implementation, use the code:

```
_handler.verifyInternalCredentials(username, password)
```

For details of authentication handlers and the `AuthenticationServicePlugin` plugin interface, see “User Authentication Service Plugin” on page 191.

### Writing Your Own Implementation of the Default Web Services Authentication Plugin

Most customers do not need to write a new implementation of the `WebservicesAuthenticationPlugin` plugin interface. Typical changes to WS-I authentication logic are instead in your `AuthenticationServicePlugin` plugin implementation. See related discussion in “The Default Web Services Authentication Plugin Implementation” on page 56.

You can change the default web services authentication behavior to get the credentials from different headers. You can write your own `WebservicesAuthenticationPlugin` plugin implementation to implement custom logic.

The `WebservicesAuthenticationPlugin` interface has a single plugin method that your implementation must implement, called `authenticate`. The `authenticate` method takes one parameter of type `gw.plugin.security.WebservicesAuthenticationContext`. The object passed to your `authenticate` method contains authentication information, such as the name and password.

### Properties on a Web Services Authentication Context

Important properties on a `WebservicesAuthenticationContext` include:

- `HttpHeaders` – HTTP headers of type `gw.xml.ws.HttpHeaders`. This includes a list of header names
- `HttpServletRequest` – the HTTP servlet request object, as the standard Java object `javax.servlet.http.HttpServletRequest`.
- `RequestSOAPHeaders` – The request SOAP headers, as an XML element (`XmlElement`)

### Values to Return from Your Default Web Services Authentication Plugin

The value that your implementation returns from its `authenticate` method depend on whether authentication succeeds:

- If authentication succeeds, return the relevant `User` object from the ClaimCenter database.
- If you cannot attempt to authenticate for some reason, such as network problems, return `null`.
- If authentication fails for other reasons, throw an exception of type `WsiAuthenticationException`.

### See also

- To authenticate RPCE web services or user interface login, see “Overview of User Authentication Interfaces” on page 187.

## Checking for Duplicate External Transaction IDs

To detect duplicate operations from external systems that change data, add the `@WsiCheckDuplicateExternalTransaction` annotation to your WS-I web service implementation class. To apply this feature for all operations on the service, add the annotation at the class level. To apply only to some operations, declare the annotation at the method level for individual operations.

If you apply this feature to an operation (or to the whole class), ClaimCenter checks for the SOAP header `<transaction_id>` in namespace `http://guidewire.com/ws/soapheaders`. If the header exists, the text data is the external transaction ID that uniquely identifies the transaction. The recommended pattern for the transaction ID is to begin with an identifier for the external system, then a colon, then an ID that is unique to that external system. The most important thing is that the transaction ID be unique across all external systems.

If the web service changes any database data, the application stores the transaction ID in an internal database table for future reference. If in the future, some code calls the web service again with the same transaction ID, the database commit fails and throws the following exception:

```
com.guidewire.pl.system.exception.DBAlreadyExecutedException
```

The caller of the web service can detect this exception to identify the request as a duplicate transaction.

Because this annotation relies on database transactions (bundles), if your web service does not change any database data, this API has no effect.

If your WS-I client code is written in Gosu, to set the SOAP header see “Setting Guidewire Transaction IDs” on page 81.

If you apply this feature to an operation (or to the whole class), and the SOAP header `<transaction_id>` is missing, ClaimCenter throws an exception.

## Request or Response XML Structural Transformations

For advanced layers of security, you probably want to use transformations that use the byte stream. See “Data Stream Transformations” on page 51.

However, there are other situations where you might want to transform either the request or response XML data at the structural level of manipulating `Xmlelement` objects.

To transform the request envelope XML before processing, add the `@WsiRequestXmlTransform` annotation. To transform the response envelope XML after generating it, add the `@WsiResponseXmlTransform` annotation.

Each annotation takes a single constructor argument which is a Gosu block. Pass a block that takes one argument, which is the envelope. The block transforms the XML element in place using the Gosu XML APIs.

The envelope reference statically has the type `Xmlelement`. However, at runtime the type is one of the following, depending on whether SOAP 1.1 or SOAP 1.2 invoked the service:

- `gw.xsd.w3c.soap11_envelope.Envelope`
- `gw.xsd.w3c.soap12_envelope.Envelope`

## Reference Additional Schemas in Your Published WSDL

If you need to expose additional schemas to the web service clients in the WSDL, you can use the `@WsiAdditionalSchemas` annotation to do them. Use this to provide references to schemas that might be required but are not automatically included.

For example, you might define an operation to take any object in a special situation, but actually accepts only one of several different elements defined in other schemas. You might throw exceptions on any other types. By using this annotation, the web service can add specific new schemas so web service client code can access them from the WSDL for the service.

The annotation takes one argument of the type `List<XmlelementAccess>`, which means a list of schema access objects. To get a reference to a schema access object, first put a XSD in your class hierarchy. Then from Gosu, determine the fully-qualified name of the XSD based on where you put the XSD. Next, get the `util` property from the schema, and on the result get the `SchemaAccess` property. To generate a list, simply surround one or more items with curly braces and comma-separate the list.

For example the following annotation adds the XSD that resides locally in the location `gw.xml.ws.wsimschema`:

```
@WsiAdditionalSchemas({ gw.xml.ws.wsimschema.util.SchemaAccess })
```

### See also

- “Reference of XSD Properties and Types” on page 282 in the *Gosu Reference Guide*
- “Referencing Additional Schemas During Parsing” on page 279 in the *Gosu Reference Guide*
- “Transforming a Generated Schema” on page 64

## Validate Requests Using Additional Schemas as Parse Options

You can validate incoming requests using additional schemas. To add an additional schema parse option, add the `@WsiParseOptions` annotation to your web service implementation class.

Before proceeding, be sure you have a reference to the XSD-based schema reference. For an XSD or WSDL, get the `SchemaAccess` property on the XSD type to get the schema reference. The argument for the annotation is an instance of type `XmlParseOptions`, which contains a property called `AdditionalSchemas`. That property must contain a list of schemas.

To add a single schema, you can use the following compact syntax:

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { YOUR_XSD_TYPE.util.SchemaAccess } })
```

For an XSD called `WS.xsd` in the source code file tree in the package `com.abc`, use the following syntax:

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { com.abc.ws.util.SchemaAccess } })
```

To include an entire WSDL file as an XSD, use the same syntax. For example, if the file is `WS.wsdl`:

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { com.abc.ws.util.SchemaAccess } })
```

**See also**

- “Introduction to the XML Element in Gosu” on page 271 in the *Gosu Reference Guide*
- “Schema Access Type” on page 301 in the *Gosu Reference Guide*
- “Reference Additional Schemas in Your Published WSDL” on page 58

## Invocation Handlers for Implementing Preexisting WSDL

The implementation of a Gosu WS-I web service is a Gosu class. Typically, each WS-I operation corresponds directly to one method on the Gosu class that implements the web service. Gosu automatically determines and generates the output WSDL for that operation. For any method arguments and return types, Gosu uses the class definition and the method signatures to determine what to export in the WSDL.

However, if necessary you can use argument types and return values defined in a separate WSDL. This might be necessary for example if you are required to implement a preexisting service definition. Perhaps ten different systems implement this service, and only one is defined by Gosu. By using a standardized WSDL, some organization can ensure that all types sent and received conform to a standard WSDL definition for the service.

### Adding an Invocation Handler for Preexisting WSDL

Only use the `@WsiInvocationHandler` annotation if you need to write a web service that conforms to externally-defined standard WSDL. Generally speaking, using this approach makes your code harder to read code and error prone because mistakes are harder to catch at compile time. For example, it is harder to catch errors in return types using this approach.

To implement preexisting WSDL, you define your web service very differently than for typical web service implementation classes.

First, on your web service implementation class add the annotation `@WsiInvocationHandler`. As an argument to this annotation, pass an invocation handler. An invocation handler has the following qualities:

- The invocation handler is an instance of a class that extends the type `gw.xml.ws.annotation.DefaultWsiInvocationHandler`.
- Implement the invocation handler as an inner class inside your web service implementation class.
- The invocation handler class overrides the `invoke` method with the following signature:  
`override function invoke(requestElement : XmlElement, context : WsiInvocationContext) : XmlElement`
- The `invoke` method does the actual dispatch work of the web service for all operations on the web service. Gosu does not call any other methods on the web service implementation. Instead, the invocation handler handles all operations that normally would be in various methods on a typical web service implementation class.
- Your `invoke` method can call its super implementation to trigger standard method calls for each operation based on the name of the operation. Use this technique to run custom code before or after standard method invocation, either for logging or special logic.

In the `invoke` method of your invocation handler, determine which operation to handle by checking the type of the `requestElement` method parameter. For each operation, perform whatever logic makes sense for your web service. Return an object of the appropriate type. Get the type of the return object from the XSD-based types created from the WSDL.

Finally, for the WSDL for the service to generate successfully, add the preexisting WSDL to your web service using the WS-I parse options annotation `@WsiParseOptions`. Pass the entire WSDL as the schema as described in that topic.

**See also**

- “Introduction to the XML Element in Gosu” on page 271 in the *Gosu Reference Guide*
- “XSD-based Properties and Types” on page 281 in the *Gosu Reference Guide*
- “Validate Requests Using Additional Schemas as Parse Options” on page 58

## Example of an Invocation Handler for Preexisting WSDL

In the following example, there is a WSDL file at the resource path in the source code tree at the path:

```
ClaimCenter/configuration/gsrc/ws/weather.wsdl
```

The schema for this file has the Gosu syntax: `ws.weather.util.SchemaAccess`. Its element types are available in the Gosu type system as objects in the package `ws.weather.elements`.

The method signature of the `invoke` method returns an object of type `XmLElement`, the base class for all XML elements. Be sure to carefully create the right subtype of `XmLElement` that appropriately corresponds to the return type for every operation. See “Introduction to the XML Element in Gosu” on page 271 in the *Gosu Reference Guide*.

The following example implements a web service that conforms to a preexisting WSDL and implements one of its operations.

```
package gw.xml.ws

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiInvocationHandler
uses gw.xml.XmLElement
uses gw.xml.ws.annotation.WsiPermissions
uses gw.xml.ws.annotation.WsiAvailability
uses gw.xml.ws.annotation.WsiParseOptions
uses gw.xml.XmlParseOptions
uses java.lang.IllegalArgumentException

@WsiWebService("http://guidewire.com/px/ws/gw/xml/ws/WsiImplementExistingWsdlTestService")
@WsiPermissions({})
@WsiAvailability(NONE)
@WsiInvocationHandler(new WsiImplementExistingWsdlTestService.Handler())
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { ws.weather.util.SchemaAccess } })

class WsiImplementExistingWsdlTestService {

    // The following line declares an INNER CLASS within the outer class.
    static class Handler extends DefaultWsiInvocationHandler {

        // Here we implement the "weather" wsdl with our own GetCityForecastByZIP implementation.
        override function invoke(requestElement : XmLElement, context : WsiInvocationContext)
            : XmLElement {

            // Check the operation name. If it is GetCityForecastByZIP, handle that operation.
            if (requestElement typeis ws.weather.elements.GetCityForecastByZIP) {
                var returnResult = new ws.weather.elements.GetCityForecastByZIPResponse() {

                    // The next line uses type inference to instantiate XML object of the correct type
                    // rather than specifying it explicitly.
                    :GetCityForecastByZIPResult = new() {
                        :Success = true,
                        :City = "Demo city name for ZIP ${requestElement.ZIP}"
                    }
                    return returnResult
                }

                // Check for additional requestElement values to handle additional operations.
                if ...

                else {
                    throw new IllegalArgumentException("Unrecognized element: " + requestElement)
                }
            }
        }
    }
}
```

First, the `invoke` method checks if the requested operation is the desired operation. An operation normally corresponding to a method name on the web service, but in this approach one method handles all operations. In this

simple example, the `invoke` method handles only the operation in the WSDL called `GetCityForecastByZip`. If the requested operation is `GetCityForecastByZip`, the code creates an instance of the `GetCityForecastByZIPResponse` XML element.

Next, the example uses Gosu object creation initialization syntax to set properties on the element as appropriate. Finally, the code returns that XML object to the caller as the result from the API call.

For additional context of the WSI request, use the `context` parameter to the `invoke` method. The `context` parameter has type `WsiInvocationContext`, which contains properties such as servlet and request headers.

#### See also

- “WS-I Web Service Invocation Context” on page 45

## Invocation Handler Profiling, Run Levels, Permissions, and Transaction IDs

If you write an invocation handler for a WS-I web service, by default you are bypassing some important WS-I features:

- The application does not enable profiling for method calls.
- The application does not check run levels even at the web service class level.
- The application does not check web service permissions, even at the web service class level.
- The application does not check for duplicate external transaction IDs if present.

However, you can support all these things in your web service even when using an invocation handler, and in typical cases it is best to do so.

#### To re-enable bypassed WS-I features from an invocation handler

1. In your web service implementation class, create separate methods for each web service operation. For each method, for the one argument and the one return value, use an `XmLElement` object. For example, the method signature:

```
static function myMethod(req : XmLElement) : XmLElement
```

2. In your invocation handler's `invoke` method, determine which method to call based on the operation name, as documented earlier.

3. Get a reference to the method info meta data for the method you want to call, using the `#` symbol to access meta data of a feature (method or property):

```
var MethodInfo = YourClassName#myMethod(XmLElement).MethodInfo
```

4. Before calling your desired method, get a reference to the `WsiInvocationContext` object that is a method argument to `invoke`. Call its `preExecute` method, passing the `requestElement` and the method info metadata as arguments. If you do not require checking method-specific annotations for run level or permissions, for the method info metadata argument you can pass `null`.

The `preExecute` method does several things:

- Enables profiling for the method you are about to call if profiling is available and configured
- Checks the SOAP headers looking for headers that set the locale. If found, sets the specified locale.
- Checks the SOAP headers for a unique transaction ID. This ID is intended to prevent duplicate requests. If this transaction has already been processed successfully (a bundle was committed with the same ID), `preExecute` throws an exception. See “Checking for Duplicate External Transaction IDs” on page 57.
- If the method info argument is non-null, `preExecute` confirms the run level for this service, checking both the class level `@wsiAvailability` annotation and any overrides for the method. As with standard WS-I implementation classes, the method level annotation supersedes the class level annotation. If the run level is not at the required level, `preExecute` throws an exception.
- If the method info argument is non-null, `preExecute` confirms user permissions for this service, checking both the class level `@wsiPermissions` annotation and any overrides for the method. As with standard WS-

I implementation classes, the method level annotation supersedes the class level annotation. If the permissions are not satisfied for the web service user, `preExecute` throws an exception.

### 5. Call your desired method from the invocation handler.

Assuming that you want to check the method-specific annotations for run level or permissions, one potential approach is to set up a map to store the method information. The map key is the operation name. The map value is the method info metadata required by the `preExecute` method.

The following example demonstrates this approach

```
@WsiInvocationHandler( new WsiImplementExistingWsdlTestService.Handler())
class WsiImplementExistingWsdlTestService {

    // The following line declares an INNER class within the outer class.
    static class Handler extends DefaultWsiInvocationHandler {

        var _map = { "myOperationName1" -> WsiImplementExistingWsdlTestService#methodA(XmlElement).MethodInfo,
                    "myOperationName2" -> WsiImplementExistingWsdlTestService#methodB(XmlElement).MethodInfo
                }

        override function invoke( requestElement : XmlElement, context : WsiInvocationContext )
            : XmlElement {

            // get the operation name from the request element
            var opName = requestElement.QName

            // get the local part (short name) from operation name, and get the method info for it
            var method = _map.get(opName.LocalPart)

            // call preExecute to enable some WS-I features otherwise disabled in an invocation handler
            context.preExecute(requestElement, method)

            // call your method using the method info and return its result
            return method.CallHandler.handleCall(null, {requestElement}) as XmlElement
        }
    }

    // After defining your invocation handler inner class, define the methods that do your work
    // as separate static methods

    // example of overriding the default permissions
    @WsiPermissions( { /* add a list of permissions here */ } )
    static function methodA(req : XmlElement) : XmlElement {
        /* do whatever you do, and return the result */ return null
    }
    static function methodB(req : XmlElement) : XmlElement {
        /* do whatever you do, and return the result */ return null
    }
}
```

This is the only supported use of a `WsiInvocationContext` object's `preExecute` method. Any use other than calling it exactly once from within an invocation handler `invoke` method is unsupported.

## Locale Support

By default, WS-I web services use the default server locale.

WS-I web service clients can override this behavior and set a locale to use while processing this web service request. To set the locale, the client can add a SOAP header element type `<gwsoap:locale>` with namespace "`http://guidewire.com/ws/soapheaders`", with the element containing the locale code.

For example, the following SOAP envelope contains a SOAP header that sets the `fr_FR` locale for French language in France:

```
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
<soap12:Header>
    <gwsoap:locale xmlns:gwsoap="http://guidewire.com/ws/soapheaders">fr_FR</gwsoap:locale>
</soap12:Header>
<soap12:Body>
    <getPaymentInstrumentsFor xmlns="http://example.com/gw/webservice/cc/cc700/PaymentAPI">
        <accountNumber>123</accountNumber>
    </getPaymentInstrumentsFor>
</soap12:Body>
</soap12:Envelope>
```

```
</getPaymentInstrumentsFor>
</soap12:Body>
</soap12:Envelope>
```

In this case, the web service would set the `fr_FR` locale before processing this web service request.

On a related topic, you can quickly configure the locale in a web service client in Gosu, such as from another Guidewire application. See “Setting Locale in a Guidewire Application” on page 84. That configuration information sets the SOAP header mentioned in this section.

## Setting Response Serialization Options, Including Encodings

You can customize how ClaimCenter serializes the XML in the web service response to the client.

The most commonly customized serialization option is changing character encoding. Outgoing web service responses by default use the UTF-8 character encoding. You might want to use another character encoding for your service to improve Asian language support or other technical reasons.

**Note:** Incoming web service requests support any valid character encodings recognized by the Java Virtual Machine. The web service client determines the encoding that it uses, not the server or its WSDL.

To support additional serializations, add the `@WsiSerializationOptions` annotation to the web service implementation class. As an argument to the annotation, pass a list of `XmlSerializationOptions` objects. The `XmlSerializationOptions` class encapsulates various options for serializing XML, and that includes setting its `Encoding` property to a character encoding of type `java.nio.charset.Charset`.

The easiest way to get the appropriate character set object is to use the `Charset.forName(ENCODING_NAME)` static method. That method returns the desired static instance of the character set object.

For example, add the following the annotation immediately before the web service implementation class to specify the Big5 Chinese character encoding

```
@WsiSerializationOptions( new() { :Encoding = Charset.forName( "Big5" ) } )
```

For more information about other XML serialization options, such as indent levels, pretty printing, line separators, and element sorting, see “Exporting XML Data” on page 275 in the *Gosu Reference Guide*.

If the web service client is a Guidewire product, you can configure the character encoding for the request in addition to the server response. See “Setting XML Serialization Options” on page 83.

## Exposing Typelists and Enumerations As String Values

For each typelist type or enumeration (Gosu or Java), the web service by default exposes this data as an enumeration value in the WSDL. This applies to both requests and responses for the service.

For example:

```
<xs:simpleType name="HouseType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="apartment"/>
    <xs:enumeration value="house"/>
    <xs:enumeration value="shack"/>
    <xs:enumeration value="mobilehome"/>
  </xs:restriction>
</xs:simpleType>
```

The published web service validates any incoming values against the set of choices and throws an exception for unknown values. Depending on the client implementation, the web service client might check if responses contain only allowed enumeration values during de-serialization. For typical cases, this approach is the desired behavior for both server and client.

For example, suppose you add new codes to a typelist or enumeration for responses. If the service returns an unexpected value in a response, it might be desirable that the client throws an exception. System integrators

would quickly detect this unexpected change. The client system can explicitly refresh the WSDL and carefully check how the system explicitly handles any new codes.

However, in some cases you might want to expose enumerations as `String` values instead:

- The WSDL for a service that references typelist or enumeration types can grow in size significantly. This is true especially if some typelists or enumerations contain a vast number of choices.
- Changing enumeration values even slightly can cause incompatible WSDL. Forcing the web service client to refresh the WSDL might exacerbate upgrade issues on some projects. Although the client can refresh the WSDL from the updated service, sometimes this is an undesirable requirement. For example, perhaps new enumeration values on the server are predictably irrelevant to an older external system.
- In some cases, your actual WS-I web service client code might be middleware that simply passes through `String` values from another system. In such cases, you may not require explicit detection of enumeration changes in the web service client code.

To expose typelist types and enumerations (from Gosu or Java) as a `String` type, add the `@WsIExposeEnumAsString` annotation before the web service implementation class. In the annotation constructor, pass the typelist or enumeration type as the argument.

You can expose multiple types as `String` values by repeating the annotation multiple types on separate lines, each providing a different type as the argument.

For example, to expose the `HouseType` enumeration as a `String`, add the following line before the web service implementation class declaration:

```
@WsIExposeEnumAsString( HouseType )
```

## Transforming a Generated Schema

ClaimCenter generates WSDL and XSDs for the web service based on the contents of your implementation class and any of its configuration-related annotations. In typical cases, the generated files are appropriate for consuming by any web service client code.

In rare cases, you might need to do some transformation. For example, if you want to specially mark certain fields as required in the XSD itself, or to add other special comment or marker information.

You can do any arbitrary transformation on the schema by adding the `@WsISchemaTransform` annotation. The one argument to the annotation constructor is a block that does the transformation. The block takes two arguments:

- a reference to the entire WSDL XML. From Gosu, this object is an `XmLElement` object and strongly typed to match the `<definitions>` element from the official WSDL XSD specification. From Gosu, this type is `gw.xsd.w3c.wsdl.Definitions`.
- A reference to the main generated schema for the operations and its types. From Gosu this object is an `XmLElement` object strongly typed to match the `<schema>` element from the official XSD metaschema. From Gosu this type is `gw.xsd.w3c.xmlschema.Schema`. There may be additional schemas in the WSDL that are unrepresented by this parameter but are accessible through the WSDL parameter.

The following example modifies a schema to force a specific field to be required. In other words, it strictly prohibits a `null` value. This example transformation finds a specific field and changes its XSD attribute `MinOccurs` to be 1 instead of 0:

```
@WsISchemaTransform( \ wsdl, schema ->{
    schema.Element.firstWhere( \ e ->e.Name == "myMethodSecondParameterIsRequired"
        ).ComplexType.Sequence.Element[1].MinOccurs = 1
} )
```

You can also change the XML associated with the WSDL outside the schema element.

## Login WS-I Authentication Confirmation

In typical cases, WS-I web service client code sets up authentication and calls desired web services, relying on catching any exceptions if authentication fails. You do not need to call a specific WS-I web service as a precondition for login authentication. In effect, WS-I authentication happens with each API call.

However, if your web service client code wants to explicitly test specific authentication credentials, ClaimCenter publishes the built-in `Login` web service. Call this web service's `login` method, which takes a user name as a `String` and a password as a `String`. If authentication fails, the API throws an exception.

If the authentication succeeds, that server creates a persistent server session for that user ID and returns the session ID as a `String`. The session persists after the call completes. In contrast, a normal WS-I API call creates a server session for that user ID but clears the session as soon as the API call completes.

If you call the `login` method, you must call the matching `logout` method to clear the session, passing the session ID as an argument. If you were merely trying to confirm authentication, call `logout` immediately.

However, in some rare cases you might want to leave the session open for logging purposes to track the owner of multiple API calls from one external system. After you complete your multiple API calls, finally call `logout` with the original session ID.

If you fail to call `logout`, all application servers are configured to time out the session eventually.

## Stateful WS-I Session Affinity Using Cookies

By default, WS-I web services do not ask the application server to generate and return session cookies. However, ClaimCenter supports cookie-based load-balancing options for web services. This is sometimes referred to as *session affinity*.

The WS-I web services layer can generate a cookie for a series of API calls. You can configure load balancing routers to send consecutive SOAP calls in the same conversation to the same server in the cluster. This feature simplifies things like customer portal integration. Repeated page requests by the same user (assuming they successfully reused the same cookie) go to the same node in the cluster.

By using session affinity, you can improve performance by ensuring that caches for that node in the cluster tend to already contain recently used objects and any session-specific data.

To create cookies for the session, append the text "`?stateful`" (with no quotes) to the WS-I API URL.

From Gosu client code, you can use code similar to following to append text to the URL:

```
uses gw.xml.ws.WsdlConfig
uses java.net.URI
uses wsi.remote.gw.webservice.ab.ab800.abcontactapi.ABContactAPI

// get a reference to an API object
var api = new ABContactAPI()

// get URL and override URL to force creation of local cookies to save session for load balancing
api.Config.ServerOverrideUrl = new URI(ABContactAPI.ADDRESS + "?stateful")

// call whatever API method you need as you normally would...
api.getReplacementAddress("12345")
```

The code might look very different depending on your choice of web service client programming language and SOAP library.

## Calling a ClaimCenter WS-I Web Service from Java

For WS-I web services, there are no automatically generated libraries to generate stub classes for use in Java. You must use your own procedures for converting WSDL for the web service into APIs in your preferred language. There are several ways to create Java classes from the WSDL:

- Java 6 (Java 1.6) includes a built-in utility to generate Java-compatible stubs using the `wsimport` tool.
- The CXF open source tool.
- The Axis2 open source tool.

**Note:** In contrast, Guidewire applications generate Java stubs using an embedded version of the Axis tool for RPCE web services. (The `ClaimCenter/bin/gwcc regen-soap-api` tool regenerates these libraries.)

### Calling Web Services using Java 1.6 and `wsimport`

Java 6 (Java 1.6) includes a built-in utility that generates Java-compatible stubs from WSDL. This tool is called `wsimport` tool. This documentation uses this built-in Java tool and its output to demonstrate creating client connections to the ClaimCenter web services.

You can get the WSDL for the WS-I web service from one of two sources:

- “Getting WS-I WSDL from a Running Server” on page 48
- “Generating WS-I WSDL On Disk” on page 50

The `wsimport` tool supports getting WSDL over the Internet published directly from a running application. This is the approach that this documentation uses to demonstrate how to use `wsimport`.

**Note:** The `wsimport` tool also supports using local WSDL files. Refer to the `wsimport` tool documentation for details.

#### To generate Java classes that make SOAP client calls to a SOAP API

1. Launch the server that publishes the WS-I web services.
2. On the computer from which you will run your SOAP client code, open a command prompt.
3. Change the working directory to a place on your local disk where you want to generate Java source files and `.class` files.
4. Decide the name of the subdirectory of the current directory where you want to place the Java source files. For example, you might choose the folder name `src`. Assure the subdirectory already exists before you run the `wsimport` tool in the next step. If the directory does not already exist, create the directory now. This topic calls this the `SUBDIRECTORY_NAME` directory.
5. Type the following command:

```
wsimport WSDL_LOCATION_URL -s SUBDIRECTORY_NAME
```

For `WSDL_LOCATION_URL`, type the HTTP path to the WSDL. See “Getting WS-I WSDL from a Running Server” on page 48.

For example:

```
wsimport http://localhost:PORTNUMBER/cc/ws/example/HelloWorldAPI?WSDL -s src
```

---

**IMPORTANT** You must assure the `SUBDIRECTORY_NAME` directory already exists before running the command. If the directory does not already exist, the `wsimport` action fails with the error “directory not found”.

---

6. The tool generates Java source files and compiled class files. Depending on what you are doing, you probably need only the class files or the source files, but not both.

- The .java files are in the *SUBDIRECTORY\_NAME* subdirectory. To use these, add this directory to your Java project's class path, or copy the files to your Java project's *src* folder. The location of the files represent the hierarchical structure of the web service namespace (in reverse order) followed by the fully qualified name.

The namespace is a URL that each published web service specifies in its declaration. It represents a namespace for all the objects in the WSDL. A typical namespace would specify your company domain name and perhaps other meaningful disambiguating or grouping information about the purpose of the service. For example, "http://mycompany.com". You can specify the namespace for each web service by passing a namespace as a *String* argument to the @WebService annotation. If you do not override the namespace when you declare the web service, the default is "http://example.com". For more details, see "Declaring the Namespace for a WS-I Web Service" on page 43.

The path to the Java source file has the following structure:

*CURRENT\_DIRECTORY/SUBDIRECTORY\_NAME/REVERSED\_NAMESPACE/FULLY\_QUALIFIED\_NAME.java*

For example, suppose your web service fully qualified name is *com.mycompany.HelloWorld* and the namespace is the default "http://example.com". Suppose you use the *SUBDIRECTORY\_NAME* value "src".

The *wsimport* tool generates the .java file at the following location:

*CURRENT\_DIRECTORY/src/com/example/com/mycompany/HelloWorld.java*

- The compiled .class files are placed in a hierarchy (by package name) with the same basic naming convention as the .java files but with no *SUBDIRECTORY\_NAME*. In other words, the location is:  
*CURRENT\_DIRECTORY/SUBDIRECTORY\_NAME/REVERSED\_NAMESPACE/FULLY\_QUALIFIED\_NAME*
- Continuing our example, the *wsimport* tool generates the .class files at the following location:  
*CURRENT\_DIRECTORY/com/example/com/mycompany/HelloWorld.class*
- To use the Java class files, add this directory to your Java project's class path, or copy the files to your Java project's *src* folder.

7. The next step depends on whether you want just the .class files to compile against, or whether you want to use the generated Java files.

- To use the .java files, copy the *SUBDIRECTORY\_NAME* subdirectory into your Java project's *src* folder.
- To use the .class files, copy the files to your Java project's *src* folder or add that directory to your project's class path.

8. Write Java SOAP API client code that compiles against these new generated classes.

To get a reference to the API itself, access the WSDL port (the service type) with the syntax:

```
new API_INTERFACE_NAME().getAPI_INTERFACE_NAMESoap11Port();
```

For example, for the API interface name *HelloWorldAPI*, the Java code looks like:

```
HelloWorldAPIService port = new HelloWorldAPI().getHelloWorldAPISoap11Port();
```

Once you have that port reference, you can call your web service API methods directly on it.

You can publish a web service that does not require authentication by overriding the set of permissions necessary for the web service. See "Specifying Required Permissions for a WS-I Web Service" on page 44. The following is a simple example to show calling the web service without worrying about the authentication-specific code.

```
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIService;

public class WsiTestNoAuth {

    public static void main(String[] args) throws Exception {
        // get a reference to the SOAP 1.1 port for this web service
        HelloWorldAPIService port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

        // call API methods on the WS-I web service
        String res = port.helloWorld();

        // print result
        System.out.println("Web service result = " + res);
    }
}
```

```

    }
}

}

```

**See also**

- “Adding HTTP Basic Authentication in Java” on page 68
- “Adding SOAP Header Authentication in Java” on page 68

## Adding HTTP Basic Authentication in Java

In previous topics the examples showed how to connect to a service without authentication. The following code shows how to add HTTP Basic authentication to your WS-I client request. HTTP Basic authentication is the easiest type of authentication to code to connect to a ClaimCenter WS-I web service.

```

import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIPortType;
import com.sun.xml.internal.ws.api.message.Headers;
import javax.xml.ws.BindingProvider;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.ws.BindingProvider;
import java.util.Map;

public class WsITest02 {

    public static void main(String[] args) throws Exception {
        System.out.println("Starting the web service client test...");

        // get a reference to the SOAP 1.1 port for this web service
        HelloWorldAPIPortType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

        // cast to BindingProvider so the following lines are easier to understand
        BindingProvider bp = (BindingProvider) port;

        // "HTTP Basic" authentication
        Map<String, Object> requestContext = bp.getRequestContext();
        requestContext.put(BindingProvider.USERNAME_PROPERTY, "su");
        requestContext.put(BindingProvider.PASSWORD_PROPERTY, "gw");

        System.out.println("Calling the service now...");
        String res = port.helloWorld();
        System.out.println("Web service result = " + res);
    }
}

```

## Adding SOAP Header Authentication in Java

Although HTTP authentication is the easiest to code for most integration programmers, ClaimCenter also supports optionally authenticating WS-I web services using custom SOAP headers.

For Guidewire applications, the structure of the required SOAP header is:

- An element `<authentication>` with the namespace `http://guidewire.com/ws/soapheaders`.  
That element contains two elements:
  - `<username>` – Contains the username text
  - `<password>` – Contains the password text

This SOAP header authentication option is also known as Guidewire authentication.

The Guidewire authentication XML element looks like the following:

```

<?xml version="1.0" encoding="UTF-16"?>
<authentication xmlns="http://guidewire.com/ws/soapheaders"><username>su</username><password>gw
</password></authentication>

```

The following code shows how to use Java client code to access a web service with the fully-qualified name example.helloworldapi.HelloWorld using a custom SOAP header.

After authenticating, the example calls the `helloWorld` SOAP API method.

```
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIService;
import com.sun.org.apache.xml.internal.serialize.DOMSerializerImpl;
import com.sun.xml.internal.messaging.saaj.soap.ver1_1.Header1_1Impl;
import com.sun.xml.internal.messaging.saaj.soap.ver1_1.HeaderElement1_1Impl;
import com.sun.xml.internal.ws.developer.WSBindingProvider;
import com.sun.xml.internal.ws.api.message.Headers;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilderFactory;

public class WsTest01 {

    private static final QName AUTH = new QName("http://guidewire.com/ws/soapheaders",
        "authentication");
    private static final QName USERNAME = new QName("http://guidewire.com/ws/soapheaders", "username");
    private static final QName PASSWORD = new QName("http://guidewire.com/ws/soapheaders", "password");

    public static void main(String[] args) throws Exception {
        System.out.println("Starting the web service client test...");

        // Get a reference to the SOAP 1.1 port for this web service.
        HelloWorldAPIService port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

        // cast to WSBindingProvider so the following lines are easier to understand
        WSBindingProvider bp = (WSBindingProvider) port;

        // Create XML for special SOAP headers for Guidewire authentication of a user & password.
        Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element authElement = doc.createElementNS(AUTH.getNamespaceURI(), AUTH.getLocalPart());
        Element usernameElement = doc.createElementNS(USERNAME.getNamespaceURI(),
            USERNAME.getLocalPart());
        Element passwordElement = doc.createElementNS(PASSWORD.getNamespaceURI(),
            PASSWORD.getLocalPart());

        // Set the username and password, which are content within the username and password elements.
        usernameElement.setTextContent("su");
        passwordElement.setTextContent("gw");

        // Add the username and password elements to the "Authentication" element.
        authElement.appendChild(usernameElement);
        authElement.appendChild(passwordElement);

        // Uncomment the following lines to see the XML for the authentication header.
        DOMSerializerImpl ser = new DOMSerializerImpl();
        System.out.println(ser.writeToString(authElement));

        // Add our authentication element to the list of SOAP headers.
        bp.setOutboundHeaders(Headers.create(authElement));

        System.out.println("Calling the service now...");
        String res = port.helloWorld();
        System.out.println("Web service result = " + res);
    }
}
```



# Calling WS-I Web Services from Gosu

Gosu code can import web services (SOAP APIs) from external systems and call these services as a SOAP client (an API consumer). The Gosu language handles all aspects of object serialization, object deserialization, basic authentication, and SOAP fault handling.

This topic only relates to WS-I web services, not RPCE web services. For discussion about the differences, see “Web Services Introduction” on page 31.

This topic includes:

- “Consuming WS-I Web Service Overview” on page 71
- “Adding WS-I Configuration Options” on page 78
- “One-Way Methods” on page 85
- “Asynchronous Methods” on page 86
- “MTOM Attachments with Gosu as Web Service Client” on page 86

## Consuming WS-I Web Service Overview

Gosu supports calling WS-I compliant web services. WS-I is an open industry organization that promotes industry-wide best practices for web services interoperability among diverse systems. The organization provides several different profiles and standards. The WS-I Basic Profile is the baseline for interoperable web services and more consistent, reliable, and testable APIs.

Gosu offers native WS-I web service client code with the following features:

- Call web service methods with natural Gosu syntax for method calls
- Call web services optionally asynchronously. See “Asynchronous Methods” on page 86.
- Support one-way web service methods. See “One-Way Methods” on page 85.
- Separately encrypt requests and responses. See “Adding WS-I Configuration Options” on page 78.
- Process attachments that use the multi-step MTOM protocol. See “MTOM Attachments with Gosu as Web Service Client” on page 86.

- Sign incoming responses with digital signatures. See “[Implementing Advanced Web Service Security with WSS4J](#)” on page 84.

One of the big differences between WS-I and older styles of web services is how the client and server encodes API parameters and return results.

An older style of web services is called *Remote Procedure Call encoded* (RPCE) web services. The bulk of the incoming and outgoing data are encoded in a special way that does not conform to XSD files. Many older systems use RPCE web services, but there are major downsides with this approach. Most notably, the encoding is specific to remote procedure calls, so it is difficult to validate XML data in RPC encoded responses. It would be more convenient to use standard XML validators which rely on XSDs to define the structure of the main content.

When you use the WS-I standards, you can use the alternative encoding called Document Literal encoding (`document/literal`). The document-literal-encoded SOAP message contains a complete XML document for each method argument and return value. The schema for each of these documents is an industry-standard XSD file. The WSDL that describes how to talk to the published WS-I service includes a complete XSD describing the format of the embedded XML document. The outer message is very simple, and the inner XML document contains all of the complexity. Anything that an XSD can define becomes a valid payload or return value.

The WS-I standard supports a mode called RPC Literal (`RPC/literal`) instead of Document Literal. Despite the similarity in name, WS-I RPC Literal mode is not closely related to RPC encoding. Gosu supports this WS-I RPC Literal mode for Gosu web service client code. However, it does so by automatically and transparently converting any WSDL for RPC Literal mode into WSDL for Document Literal mode. The focus of the Gosu documentation for WS-I web services is the support for Document Literal encoding.

## [Loading WS-I WSDL Locally Using Studio Web Service Collections](#)

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

---

**IMPORTANT** To create a web service collection in Studio, see “[Using the WS-I Web Service Editor](#)” on page 122 in the *Configuration Guide* for details.

---

For adding configuration options to the connection, such as authentication and security settings, there are multiple approaches. See “[Adding WS-I Configuration Options](#)” on page 78.

Web service collection (`.wsc`) files encapsulate the set of resources necessary to connect to a web service on an external system. If you view a web service collection in Studio and click the **Fetch Updates** button, Studio retrieves WSDL and XSD files from the servers that publish those web services. You can trigger the **Fetch Updates** process from a command line tool called `regen-from-wsc`:

- First, be sure that your `suite-config.xml` file correctly refers to your server names and port numbers for your Guidewire applications. See “[Suite Configuration File Overrides URLs to Guidewire Applications](#)” on page 80.
- Next, from a command prompt in the `ClaimCenter/bin` directory, type the following:  
`gwcc regen-from-wsc`

The tool refreshes all of your `.wsc` files in your Studio configuration.

## [Loading WS-I WSDL Directly into the File System](#)

To consume an external web service, you must load the WSDL and XML schema files (XSDs) for the web service. You must fetch copies of WSDL files, as well as related WSDL and XSD files, from the web services server. Fetch the copies into an appropriate place in the Gosu class hierarchy on your local system. Place them in the directory that corresponds to the package in which you want new types to appear. For example, place the

WSDL and XSD files next to the Gosu class files that call the web service, organized in package hierarchies just like class files.

#### To automatically fetch WSDLs and XSDs from a web service server

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

#### Web Service Collections in the File System

When you fetch WSDL resources, Studio retrieves the WSDL from the web service and stores it locally at the path location in Studio where you defined your web service collection. Studio downloads the WSDL and any related XSDs into a subdirectory of that location. The subdirectory has the same name as your web service collection.

The path of the WSDL is `PACKAGE_NAME/web_service_name.wsdl`. Studio creates the value for `web_service_name` from the last part of the web service endpoint URL, not from the WSDL. For example, the URL is `MY_URL/info.wsdl` or `MY_URL/info.wso?wsdl`. Gosu creates all the types for the web service in the namespace `PACKAGE.info`.

#### Sample Code to Manually Fetch WSDLs and XSDs from a Web Service Server

The following sample Gosu code shows how to manually fetch web service WSDLs for test purposes or for command-line use from a web service server.

```
uses gw.xml.ws.*  
uses java.net.URL  
uses java.io.File  
  
// -- set the web service endpoint URL for the web service WSDL --  
var urlStr = "http://www.aGreatWebService.com/GreatWebService?wsdl"  
  
// -- set the location in your file system for the web service WSDL --  
var loc = "/wsi/remote/GreatWebService"  
  
// -- load the web service WSDL into Gosu --  
Wsdl2Gosu.fetch(new URL(urlStr), new File(loc))
```

The first long string (`urlStr`) is the URL to the web service WSDL. The second long string (`loc`) is the path on your file system where the fetched WSDL is stored. You can run your version of the preceding sample Gosu code in the Gosu Scratchpad.

#### Security and Authentication

The WS-I basic profile requires support for some types of security standards for web services, such as encryption and digital signatures (cryptographically signed messages). See “Adding WS-I Configuration Options” on page 78.

## Types of WS-I Client Connections

From Gosu, there are three types of WS-I web service client connections:

- Standard round trip methods (synchronous request and response)
- Asynchronous round trip methods (send the request and immediately return to the caller, and check later to see if the request finished). See “Asynchronous Methods” on page 86.
- One-way methods, which indicate a method defined to have no SOAP response at all. See “One-Way Methods” on page 85.

## How Does Gosu Process WSDL?

As mentioned before, Studio adds a WSDL file to the class hierarchy automatically for each registered web service in the Web Services editor in Studio. Gosu creates all the types for your web service in the namespace `ws.web_service_name`. For this example, assume that the name of your web service is `MyService`. Gosu creates all the types for your web service in the namespace `gw.config.webservices.MyService`.

Suppose you add a Web Service in Studio and you name the web service `MyService`. Gosu creates all the types for your web service in the namespace:

```
ws.myservice.*
```

Suppose you add a WSDL file directly to your class hierarchy called `MyService.wsdl` in the package `example.pl.gs.wsic`. Gosu creates all the types for your web service in the namespace:

```
example.pl.gs.wsic.myservice.*
```

The name of `MyService` becomes lowercase `myservice` in the package hierarchy for the XML objects because the Gosu convention for package names is lowercase. There are other name transformations as Gosu imports types from XSDs.

For details, see “Normalization of Gosu Generated XSD-based Names” on page 284 in the *Gosu Reference Guide*.

The structure of a WSDL comprises the following:

- One or more services
- For each service, one or more ports

A port represents a protocol or other context that might change how the WSDL defines that service. For example, methods might be defined differently for different versions of SOAP, or an additional method might be added for some ports. WSDL might define one port for SOAP 1.1 clients, one port for SOAP 1.2 clients, one port for HTTP non-SOAP access, and so on. See discussion later in this topic for what happens if multiple ports exist in the WSDL.

- For each port, one or more methods

A method, also called an operation or action, performs a task and optionally returns a result. The WSDL includes XML schemas (XSDs), or it imports other WSDL or XSD files. Their purposes are to describe the data for each method argument type and each method return type.

Suppose the WSDL looks like the following:

```
<wsdl>
  <types>
    <schemas>
      <import schemaLocation="yourschema.xsd"/>
      <!-- now define various operations (API methods) in the WSDL ... -->
```

The details of the web service APIs are omitted in this example WSDL. Assume the web service contains exactly one service called `SayHello`, and that service contains one method called `helloWorld`. Let us assume for this first example that the method takes no arguments, returns no value, and is published with no authentication or security requirements.

In Gosu, you can call the remote service represented by the WSDL using code such as:

```
// get a reference to the web service API object in the namespace of the WSDL
```

```
// warning: this object is not thread-safe. Do not save in a static variable or singleton instance var.  
var service = new .myservice.SayHello()  
  
// call a method on the service  
service.helloWorld()
```

Of course, real APIs need to transfer data also. In our example WSDL, notice that the WSDL refers a secondary schema called `yourschema.xsd`.

Studio adds any attached XSDs into the `web_service_name.wsdl.resources` subdirectory.

Let us suppose the contents of your `yourschema.xsd` file looks like the following:

```
<schema>  
  <element name="Root" type="xsd:int"/>  
</schema>
```

Note that the element name is "root" and it contains a simple type (`int`). This XSD represents the format of an element for this web service. The web service could declare a `<root>` element as a method argument or return type.

Now let us suppose there is another method in the `SayHello` service called `doAction` and this method takes one argument that is a `<root>` element.

In Gosu, you can call the remote service represented by the WSDL using code similar to the following:

```
// get a Gosu reference to the web service API object  
// warning: this object is not thread-safe. Do not save in a static variable or singleton instance var.  
var service = new ws.myservice.SayHello()  
  
// create an XML document from the WSDL using the Gosu XML API  
var x = new ws.myservice.Root()  
  
// call a method that the web service defines  
var ret = service.doAction( x )
```

The package names are different if you place your WSDL file in a different part of the package hierarchy.

**Note:** If you use Guidewire Studio, you do not need to manipulate the WSDL file manually. Studio auto-mates getting the WSDL and saving it when you add a Web Service in the user interface.

For each web service API call, Gosu first evaluates the method parameters. Internally, Gosu serializes the root `XmLElement` instance and its child elements into XML raw data using the associated XSD data from the WSDL. Next, Gosu sends the resulting XML document to the web service. In the SOAP response, the main data is an XML document, whose schema is contained in (or referenced by) the WSDL.

Be sure to read the warnings in the section “WS-I API Objects Are Not Thread Safe” on page 75.

## WS-I API Objects Are Not Thread Safe

To create a WS-I API object, you use a `new` expression to instantiate the right type:

```
var service = new .myservice.SayHello()
```

Be warned that the WS-I API object is not thread-safe in all cases.

---

**WARNING** It is dangerous to modify any configuration when another thread might have a connection open. Also, some APIs may directly or indirectly modify the state on the API object itself.

---

For example, the `initializeExternalTransactionIdForNextUse` method saves information that is specific to one request, and then resets the transaction ID after one use. See “Setting Guidewire Transaction IDs” on page 81.

It is safest to follow these rules:

- Instantiate the WS-I API object each time it is needed. This careful approach allows you to modify the configuration and use API without concerns for thread-safety.
- Do not save API objects in static variables.

- Do not save API objects in instance variables of classes that are singletons, such as plugin implementations. Each member field of the plugin instance becomes a singleton and needs to be shared across multiple threads.

## Learning Gosu XML APIs

All WS-I method argument types and return types are defined from schemas (the XSD embedded in the WSDL). From Gosu, all these objects are instances of subclasses of `XmlElement`, with the specific subclass defined by the schemas in the WSDL. From Gosu, working with WS-I web service data requires that you understand Gosu XML APIs.

In many cases, Gosu hides much of the complexity of XML so you do not need to worry about it. For example, for XSD-types, in Gosu you do not have to directly manipulate XML as bytes or text. Gosu handles common types like number, date, or Base64 binary data. You can directly get or set values (such as a `Date` object rather than a serialized `xsd:date` object). The `XmlElement` class, which represents an XML element hide much of the complexity.

Other notable tips to working with XML in Gosu:

- When using a schema (an XSD, or a WSDL that references an XSD), Gosu exposes shortcuts for referring to child elements using the name of the element. See “XSD-based Properties and Types” on page 281 in the *Gosu Reference Guide*.
- When setting properties in an XML document, Gosu creates intermediate XML element nodes in the graph automatically. Use this feature to significantly improve the readability of your XML-related Gosu code. For details, see “Automatic Creation of Intermediary Elements” on page 295 in the *Gosu Reference Guide*.
- For properties that represent child elements that can appear more than once, Gosu exposes that property as a list. For list-based types like this, there is a special shortcut to be aware of. If you assign to the list index equal to the size of the list, then Gosu treats the index assignment as an insertion. This is also true if the size of the list is zero: use the `[0]` array/list index notation and set the property. This inserts the value into the list, which is equivalent to adding an element to the list. However, you do not have to worry about whether the list exists yet. If you are creating XML objects in Gosu, by default the lists do not yet exist.

In other words, use the syntax:

```
element.PropertyName[0] = childElement
```

If the list does not exist yet for a list property at all, Gosu creates the list upon the first insertion. In other words, suppose an element contains child elements that represent an address and the child element has the name `Address`. If the XSD declares the element could exist more than once, the `element.Address` property is a list of addresses. The following code creates a new `Address`:

```
element.Address[0] = new my.package.xsdname.elements.Address()
```

### See also

- “Automatic Insertion into Lists” on page 286 in the *Gosu Reference Guide*
- “XSD-based Properties and Types” on page 281 in the *Gosu Reference Guide*
- “Gosu and XML” on page 267 in the *Gosu Reference Guide*

## What Gosu Creates from Your WSDL

Within the namespace of the package of the WSDL, Gosu creates some new types.

For each service in the web service, Gosu creates a service by name. For example, if the external service has a service called `GeocodeService` and the WSDL is in the package `examples.gosu.wsdl`, then the service has the fully-qualified type `examples.gosu.wsdl.GeocodeService`. Create a new instance of this type, and you then you can call methods on it for each method.

For each operation in the web service, generally speaking Gosu creates two local methods:

- One method with the method name in its natural form, for example suppose a method is called `doAction`

- One method with the method name with the `async_` prefix, for example `async_doAction`. This version of the method handles asynchronous API calls. For details, see “Asynchronous Methods” on page 86.

## Special Behavior For Multiple Ports

Gosu automatically processes ports from the WSDL identified as either SOAP 1.1 or SOAP 1.2. If both are available for any service, Gosu ignores the SOAP 1.1 ports. In some cases, the WSDL might define more than one available port (such as two SOAP 1.2 ports with different names).

For example, suppose you add a WSDL file to your class hierarchy called `MyService.wsdl` in the package `example.pl.gs.wsic`. Gosu chooses a default port to use and creates types for the web service at the following path:

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.NORMALIZED_SERVICE_NAME
```

The `NORMALIZED_SERVICE_NAME` name of the package is the name of the service as defined by the WSDL, with capitalization and conflict resolution as necessary. For example, if there are two services in the WSDL named `Report` and `Echo`, then the API types are in the location

```
example.pl.gs.wsic.myservice.Report  
example.pl.gs.wsic.myservice.Echo
```

Gosu chooses a default port for each service. If there is a SOAP 1.2 version, Gosu prefers that version.

Additionally, Gosu provides the ability to explicitly choose a port. For example, if there is a SOAP 1.1 port and a SOAP 1.2 port, you could explicitly reference one of those choices. Gosu creates all the types for your web service ports within the `ports` subpackage, with types based on the name of each port in the WSDL:

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.ports.SERVICE_AND_PORT_NAME
```

The `SERVICE_AND_PORT_NAME` is the service name, followed by an underscore, followed by the port name.

For example, suppose the ports are called `p1` and `p2` and the service is called `Report`. Gosu creates types within the following packages:

```
example.pl.gs.wsic.myservice.ports.Report_p1  
example.pl.gs.wsic.myservice.ports.Report_p2
```

Additionally, if the port name happens to begin with the service name, Gosu removes the duplicate service name before constructing the Gosu type. For example, if the ports are called `ReportP1`, and `HelloP2`, Gosu creates types within the following packages:

```
example.pl.gs.wsic.myservice.ports.Report_P1 // NOTE: it is not Report_ReportP1  
example.pl.gs.wsic.myservice.ports.Report_helloP2 // not a duplicate, so Gosu does not remove "Hello"
```

Each one of those hierarchies would include method names for that port for that service.

## A Real Example: Weather

There is a public free web service that provides the weather. You can get the WSDL for this web service at the URL <http://wsf.cdyne.com/WeatherWS/Weather.asmx?wsdl>. This web service does not require authentication or encryption.

In Studio, navigate to the package `ws.weather` in the `Classes` hierarchy. Choose **New → Web Service Collection**. Enter the previously mentioned URL in the top field.

---

**IMPORTANT** There are other fields in the Studio user interface for configuring the web service collection in Studio. Refer to “Using the WS-I Web Service Editor” on page 122 in the *Configuration Guide* for details.

---

The following Gosu code gets the weather in San Francisco:

```
var ws = new ws.weather.Weather()  
var r = ws.GetCityWeatherByZIP(94114)  
print( r.Description )
```

Depending on the weather, your result might be something like:

Mostly Sunny

## Request XML Complexity Affects Appearance of Method Arguments

A WS-I operation defines a request element. If the request element is simply a sequence of elements, Gosu converts these elements into multiple method arguments for the operation. For example, if the request XML has a sequence of five elements, Gosu exposes this operation as a method with five arguments.

If the request XML definition uses complex XML features into the operation definition itself, Gosu does not extract individual arguments. Instead Gosu treats the entire request XML as a single XML element based on an XSD-based type.

For example, if the WSDL defines the operation request XML with restrictions or extensions, Gosu exposes that operation in Gosu as a method with a single argument. That argument contains one XML element with a type defined from the XSD.

Use the regular Gosu XML APIs to navigate that XML document from the XSD types in the WSDL. See “Introduction to the XML Element in Gosu” on page 271 in the *Gosu Reference Guide*.

## Adding WS-I Configuration Options

If a web service does not need encryption, authentication, or digital signatures, you can just instantiate the service object and call methods on it:

```
// get a reference to the service in the package namespace of the WSDL
var api = new example.gosu.ws.I.myService.SayHello()

// call a method on the service
api.helloWorld()
```

If you need to add encryption, authentication, or digital signatures, there are two approaches

- **Configuration objects** – Set the configuration options directly on the configuration object for the service instance. The service instance is the newly-instantiated object that represents the service. In the previous example, the `api` variable holds a reference for the service instance. That object has a `Config` property that contains the configuration object. For details, see “Directly Modifying the WSDL Configuration Object for a Service” on page 78.
- **Configuration providers** – Add one or more WS-I web service configuration providers in the Studio web service collection **Settings** tab. A web service configuration provider centralizes and encapsulates the steps required to add encryption, authentication, or digital signatures to your web service client code. For example, instead of adding encryption and authentication in your code each time that you call out to a single service, you can centralize that code. This has the side effect of making your web service client code look cleaner. This approach separates the code that requests the web service call from the transport-layer authentication and security configuration.

### Directly Modifying the WSDL Configuration Object for a Service

To add authentication or security settings to a web service you can do so by modifying the options on the service object. To access the options from the API object (in the previous example, the object in the variable called `api`), use the syntax `api.Config`. That property contains the API options object, which has the type `gw.xml.ws.WsdlConfig`.

The WSDL configuration object has properties that add or change authentication and security settings. The `WsdlConfig` object itself is not an XML object (it is not based on `Xmlelement`), but some of its subobjects are defined as XML objects. Fortunately, in typical code you do not need to really think about that difference.

Instead, simply use a straightforward syntax to set authentication and security parameters. The following subtopics describe `WsdlConfig` object properties that you can set on the WSDL configuration object.

**Note:** For XSD-generated types, if you set a property several levels down in the hierarchy, Gosu adds any intermediate XML elements if they did not already exist. This makes your XML-related code look concise. See also “Automatic Creation of Intermediary Elements” on page 295 in the *Gosu Reference Guide*.

## Adding WS-I Configuration Provider Classes (To Centralize Your WSDL Configuration)

You can add one or more WS-I web service configuration providers in the Studio web service collection **Settings** tab. A web service configuration provider centralizes and encapsulates the steps required to add encryption, authentication, or digital signatures to your web service client code.

For example, instead of adding encryption and authentication in your code each time that you call out to a single service, you can centralize that code. Using a configuration provider has the side effect of making your web service client code look cleaner. This approach separates the code that requests the web service call from the transport-layer authentication and security configuration.

A configuration provider is a class that implements the interface

`gw.xml.ws.IWsiWebserviceConfigurationProvider`. This interface defines a single method with signature:

```
function configure( serviceName : QName, portName : QName, config : WsdlConfig )
```

The arguments are as follows:

- The service name, as a `QName`. This is the service as defined in the WSDL for this service.
- The port name, as a `QName`. Note that this is not a TCP/IP port. This is a port in the sense of the WSDL specification.
- A WSDL configuration object, which is the `WsdlConfig` object that an API service object contains each time you instantiate the service. For more details, see “Directly Modifying the WSDL Configuration Object for a Service” on page 78.

You can write zero, one, or more than one configuration providers and attach them to a web service collection. This means that for each new connection to one of those services, each configuration provider has an opportunity to (optionally) add configuration options to the `WsdlConfig` object.

For example, you could write one configuration provider that adds all configuration options for web services in the collection, or write multiple configuration providers that configure different kinds of options. For an example of multiple configuration providers, you could:

- Add one configuration provider that knows how to add authentication
- Add a second configuration provider that knows how to add digital signatures
- Add a third configuration provider that knows how to add an encryption layer

Separating out these different types of configuration could be advantageous if you have some web services that share some configuration options but not others. For example, perhaps all your web service collections use digital signatures and encryption, but the authentication configuration provider class might be different for different web service collections.

The list of configuration provider classes in the Studio editor is an ordered list. For typical systems, the order is very important. For example, performing encryption and then a digital signature results in different output than adding a digital signature and then adding encryption. You can change the order in the list by clicking a row and clicking **Move Up** or **Move Down**.

---

**WARNING** The list of configuration providers in the Studio editor is an ordered list. If you use more than one configuration provider, carefully consider the order you want to specify them.

---

The following is an example of a configuration provider:

```
package wsi.remote.gw.webservice.ab
```

```

uses javax.xml.namespace.QName
uses gw.xml.ws.Wsd1Config
uses gw.xml.ws.IWsiWebserviceConfigurationProvider

class ABConfigurationProvider implements IWsiWebserviceConfigurationProvider {

    override function configure(serviceName : QName, portName : QName, config : Wsd1Config) {
        config.Guidewire.Authentication.Username = "su"
        config.Guidewire.Authentication.Password = "gw"
    }
}

```

In this example, the configuration provider adds Guidewire authentication to every connection that uses that configuration provider. Any web service client code for that web service collection does not need to bother with adding these options with each use of the service. (For more information about Guidewire authentication, see “[Guidewire Authentication](#)” on page 80.)

## HTTP Authentication

To add simple HTTP authentication to an API request, use the basic HTTP authentication object at the path as follows. Suppose *api* is a reference to a SOAP API that you have already instantiated with the new operator. The properties on which to set the name and password are on the object:

```
api.Config.Http.Authentication.Basic
```

That object has a **Username** property for the user name and a **Password** property for the password. Set those two values with the desired user name and password.

For example:

```

// Get a reference to the service in the package namespace of the WSDL.
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.Http.Authentication.Basic.Username = "jms"
service.Config.Http.Authentication.Basic.Password = "b5"

// Call a method on the service.
service.helloWorld()

```

## Guidewire Authentication

To add Guidewire application authentication to API request, use the Guidewire authentication object at the path as follows. In this example, *api* is a reference to a SOAP API that you have already instantiated with the new operator: *api*.Config.Guidewire.Authentication.

That object has a **Username** property for the user name and a **Password** property for the password. Set those two values with the desired user name and password.

For example:

```

// Get a reference to the service in the package namespace of the WSDL.
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.Guidewire.Authentication.Username = "jms"
service.Config.Guidewire.Authentication.Password = "b5"

// Call a method on the service.
service.helloWorld()

```

## Suite Configuration File Overrides URLs to Guidewire Applications

ClaimCenter now has a suite configuration file for web service configuration. This file applies only to inter-application integrations among Guidewire applications. This file provides a central place to configure (override) the URLs for other Guidewire products in the suite.

For inter-application Guidewire integrations that use WS-I web services, always use the suite configuration file to set the URLs. The suite configuration file is called `suite-config.xml`. You can find it in Studio under Other Resources, or use CNTL+N and type the file name.

By default, all subelements are commented out. If you have multiple Guidewire products, the file might look like the following:

```
<suite-config xmlns="http://guidewire.com/config/suite/suite-config">
  <product name="cc" url="http://localhost:8080/cc"/>
  <product name="pc" url="http://localhost:8180/pc"/>
  <product name="ab" url="http://localhost:8280/ab"/>
  <product name="bc" url="http://localhost:8580/bc"/>
</suite-config>
```

When importing WS-I WSDL, ClaimCenter does the following:

1. Checks to see whether there is a WSDL port of element type `<gwwsdl:address>`. If this is found, ClaimCenter ignores any other ports on the WSDL for this service.
2. If so, it looks for shortcuts of the syntax  `${PRODUCT_NAME_SHORTCUT}`. For example:  `${cc}`
3. If that product name shortcut is in the `suite-config.xml` file, ClaimCenter substitutes the URL from the XML file to replace the text  `${PRODUCT_NAME_SHORTCUT}`. If the product name shortcut is not found, Gosu throws an exception during WSDL parsing.

For web services that Guidewire applications publish, all WSDL documents have the `<gwwsdl:address>` port in the WSDL. The Guidewire application automatically specifies the application that published it using the standard two-letter application shortcut. For example, for ClaimCenter the abbreviation is `cc`. For example:

```
<wsdl:port name="TestServiceSoap11Port" binding="TestServiceSoap11Binding">
  <soap11:address location="http://172.24.11.41:8480/cc/ws/gw/test/TestService/soap11" />
  <gwwsdl:address location="${cc}/ws/gw/test/TestService/soap11" />
</wsdl:port>
```

Always use the `suite-config.xml` file to override the URL including the port number to connect to other Guidewire applications using this shortcut system.

The suite configuration file is used only for WS-I web services, not RPCE web services.

## Accessing the Suite Configuration File Using APIs

ClaimCenter exposes the suite configuration file as APIs that you can access from Gosu. The `gw.api.suite.GuidewireSuiteUtil` class can look up entries in the file by the product code. This class has a static method called `getProductInfo` that takes a `String` that represents the product. Pass the `String` that is the `<product>` element's name attribute. The method returns the associated URL from the `suite-config.xml` file.

For example:

```
uses gw.api.suite.GuidewireSuiteUtil
var x = GuidewireSuiteUtil.getProductInfo("cc")
print(x.Url)
```

For the earlier `suite-config.xml` example file, this code prints:

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

## Setting Guidewire Transaction IDs

If the web service you are calling is hosted by a Guidewire application, there is an optional feature to detect duplicate operations from external systems that change data. The service must add the `@WsCheckDuplicateExternalTransaction` annotation. For implementation details, see “Checking for Duplicate External Transaction IDs” on page 57.

If this feature is enabled for an operation, ClaimCenter checks for the SOAP header `<transaction_id>` in namespace `http://guidewire.com/ws/soapheaders`.

To set this SOAP header, call the `initializeExternalTransactionIdForNextUse` method on the API object and pass the transaction ID as a `String` value.

---

**IMPORTANT** ClaimCenter sends the transaction ID with the next method call and applies only to that one method call. If you require subsequent method calls with a transaction ID, call `initializeExternalTransactionIdForNextUse` again before each external API that requires a transaction ID. If your next call to an web service external operation does not require a transaction ID, there is no need for you to call the `initializeExternalTransactionIdForNextUse` method.

---

For example:

```
uses gw.xsd.guidewire.soapheaders.TransactionID
uses gw.xml.ws.WsdlConfig
uses java.util.Date
uses wsi.local.gw.services.wsidbupdateservice.faults.DBAlreadyExecutedException

function callMyWebService {

    // Get a reference to the service in the package namespace of the WSDL.
    var service = new example.gosu.wsi.myservice.SayHello()

    service.Config.Guidewire.Authentication.Username = "su"
    service.Config.Guidewire.Authentication.Password = "gw"

    // create a transaction ID that has an external system prefix and then a guaranteed unique ID
    // If you are using Guidewire messaging, you may want to use the Message.ID property in your ID.
    transactionIDString = "MyExtSys1:" + getUniqueTransactionID() // somehow create your own unique ID

    // Set the transaction ID for the next method call (and only the next method call) to this service
    service.initializeExternalTransactionIdForNextUse(transactionIDString)

    // Call a method on the service -- a transaction ID is set only for the next operation
    service.helloWorld()

    // Call a method on the service -- NO transaction ID is set for this operation!
    service.helloWorldMethod2()

    transactionIDString = "MyExtSys1:" + getUniqueTransactionID() // somehow create your own unique ID

    // Call a method on the service -- a transaction ID is set only for the next operation
    service.helloWorldMethod3()

}
```

## Setting a Timeout

To set the timeout value (in milliseconds), set the `CallTimeout` property on the `WsdlConfig` object for that API reference.

For example:

```
// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.CallTimeout = 30000 // 30 seconds

// call a method on the service
service.helloWorld()
```

## Custom SOAP Headers

SOAP HTTP headers are essentially XML elements attached to the SOAP envelope for the web service request or its response. Your code might need to send additional SOAP headers to the external system, such as custom headers for authentication or digital signatures. You also might want to read additional SOAP headers on the response from the external system.

To add SOAP HTTP headers to a request that you initiate, first construct an XML element using the Gosu XML APIs (`XmlElement`). Next, add that `XmlElement` object to the list in the location

`api.Config.RequestSoapHeaders`. That property contains a list of `XmLElement` objects, which in generics notation is the type `java.util.ArrayList<XmLElement>`.

To read (get) SOAP HTTP headers from a response, it only works if you use the asynchronous SOAP request APIs described in “Asynchronous Methods” on page 86. There is no equivalent API to get just the SOAP headers on the response, but you can get the response envelope, and access the headers through that. You can access the response envelope from the result of the asynchronous API call. This is an `gw.xml.ws.AsyncResponse` object. On this object, get the `ResponseEnvelope` property. For SOAP 1.2 envelopes, the type of that response is type `gw.xsd.w3c.soap12_envelope.Envelope`. For SOAP 1.1, the type is the same except with "soap11" instead of "soap12" in the name.

From that object, get the headers in the `Header` property. That property contains a list of XML objects that represent all the headers.

## Server Override URL

To override the server URL, for example for a test-only configuration, set the `ServerOverrideUrl` property on the `WsdlConfig` object for your API reference. It takes a `java.net.URI` object for the URL.

For example:

```
// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.ServerOverrideUrl = new URI("http://testingserver/xx")

// call a method on the service
service.helloWorld()
```

## Setting XML Serialization Options

To send a SOAP request to the SOAP server, ClaimCenter takes an internal representation of XML and serializes the data to actual XML data as bytes. For typical use, the default XML serialization settings are sufficient. If you need to customize these settings, you can do so.

The most common serialization option to set is changing the character encoding to something other than the default, which is UTF-8.

You can change serialization settings by getting the `XmlSerializationOptions` property on the `WsdlConfig` object, which has type `gw.xml.XmlSerializationOptions`. Modify properties on that object to set various serialization settings.

For full information about XML serialization options, such as encoding, indent levels, pretty printing, line separators, and element sorting, see “Exporting XML Data” on page 275 in the *Gosu Reference Guide*.

The easiest way to get the appropriate character set object for the encoding is to use the `Charset.forName(ENCODING_NAME)` static method. That method returns the desired static instance of the character set object.

For example, to change the encoding to the Chinese encoding Big5:

```
uses java.nio.charset.Charset

// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.XmlSerializationOptions.Encoding = Charset.forName( "Big5" )

// call a method on the service
service.helloWorld()
```

This API sets the encoding on the outgoing request only. The SOAP server is not obligated to return the response XML in the same character encoding.

If the web service is published from a Guidewire product, you can configure the character encoding for the response. See “Setting Response Serialization Options, Including Encodings” on page 63.

## Setting Locale in a Guidewire Application

WS-I web services published on a Guidewire application support setting a specific international locale to use (to override) while processing this web services request.

For example:

```
// get a reference to the service in the package namespace of the WSDL  
var service = new example.gosu.ws.IHello()  
  
// set the locale to the French language in the France region  
service.Config.Guidewire.Locale = "fr_FR"  
  
// call a method on the service  
service.helloWorld()
```

This creates a SOAP header similar to the following:

```
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">  
  <soap12:Header>  
    <gwsoap:locale xmlns:gwsoap="http://guidewire.com/ws/soapheaders">fr_FR</gwsoap:locale>  
  </soap12:Header>  
  <soap12:Body>  
    <getPaymentInstrumentsFor xmlns="http://example.com/gw/webservice/bc/bc700/PaymentAPI">  
      <accountNumber>123</accountNumber>  
    </getPaymentInstrumentsFor>  
  </soap12:Body>  
</soap12:Envelope>
```

## Implementing Advanced Web Service Security with WSS4J

For security options beyond HTTP Basic authentication and optional SOAP header authentication, you can use an additional set of APIs to implement whatever additional security layers.

For example, you might want to add additional layers of encryption, digital signatures, or other types of authentication or security.

From the SOAP client side, the way to add advanced security layers to outgoing requests is to apply transformations of the stream of data for the request. You can transform the data stream incrementally as you process bytes in the stream. For example, you might implement encryption this way. Alternatively, some transformations might require getting all the bytes in the stream before you can begin to output any transformed bytes. Digital signatures would be an example of this approach. You may use multiple types of transformations. Remember that the order of them is important. For example, an encryption layer followed by a digital signature is a different output stream of bytes than applying the digital signature and then the encryption.

Similarly, getting a response from a SOAP client request might require transformations to understand the response. If the external system added a digital signature and then encrypted the XML response, you need to first decrypt the response, and then validate the digital signature with your keystore.

The standard approach for implementing these additional security layers is the Java utility WSS4J, but you can use other utilities as needed. The WSS4J utility includes support for the WSS security standard.

### Outbound Security

To add a transformation to your outgoing request, set the `RequestTransform` property on the `WsdlConfig` object for your API reference. The value of this property is a Gosu block that takes an input stream (`InputStream`) as an argument and returns another input stream. Your block can do anything it needs to do to transform the data.

Similarly, to transform the response, set the `ResponseTransform` property on the `WsdlConfig` object for your API reference.

The following simple example shows you could implement a transform of the byte stream. The transform is in both outgoing request and the incoming response. In this example, the transform is an XOR (exclusive OR) transformation on each byte. In this simple example, simply running the transformation again decodes the request.

The following code implements a service that applies the transform to any input stream. The code that actually implements the transform is as follows. This is a web service that you can use to test this request.

The class defines a static variable that contains a field called `_xorTransform` that does the transformation.

```
package gw.xml.ws

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiRequestTransform
uses java.io.ByteArrayInputStream
uses gw.util.StreamUtil
uses gw.xml.ws.annotation.WsiResponseTransform
uses gw.xml.ws.annotation.WsiAvailability
uses gw.xml.ws.annotation.WsiPermissions
uses java.io.InputStream

@WsiWebService
@WsiAvailability( NONE )
@WsiPermissions( {} )
@WsiRequestTransform( WsiTransformTestService._xorTransform )
@WsiResponseTransform( WsiTransformTestService._xorTransform )
class WsiTransformTestService {

    // THE FOLLOWING DECLARES A GOSU BLOCK THAT IMPLEMENTS THE TRANSFORM
    public static var _xorTransform( is : InputStream ) : InputStream = \ is ->{
        var bytes = StreamUtil.getContent( is )
        for ( b in bytes index idx ) {
            bytes[ idx ] = ( b ^ 17 ) as byte // xor encryption
        }
        return new ByteArrayInputStream( bytes )
    }

    function add( a : int, b : int ) : int {
        return a + b
    }
}
```

The following code connects to the web service and applies this transform on outgoing requests and the reply.

```
package gw.xml.ws

uses gw.testharness.TestBase
uses gw.testharness.RunLevel
uses org.xml.sax.SAXParseException

@RunLevel( NONE )
class WsiTransformTest extends TestBase {

    function testTransform() {
        var ws = new wsi.local.gw.xml.ws.wsitransformtestservice.WsiTransformTestService()
        ws.Config.RequestTransform = WsiTransformTestService._xorTransform
        ws.Config.ResponseTransform = WsiTransformTestService._xorTransform
        ws.add( 3, 5 )
    }
}
```

## One-Way Methods

A typical WS-I method invocation has two parts: the SOAP request, and the SOAP response. Additionally, WS-I supports a concept called *one-way methods*. A one-way method is a method defined in the WSDL to provide no SOAP response at all. The transport layer (HTTP) may send a response back to the client, however, but it contains no SOAP response.

Gosu fully supports calling one-way methods. Your web service client code does not have to do anything special to handle one-way methods. Gosu handles them automatically if the WSDL specifies a method this way.

---

**IMPORTANT** Be careful not to confuse one-way methods with asynchronous methods. For more information about asynchronous methods, see “Asynchronous Methods” on page 86.

---

## Asynchronous Methods

Gosu supports optional asynchronous calls to web services. Gosu exposes alternate web service methods signatures on the service with the `async_` prefix. Gosu does not generate the additional method signature if the method is a one-way method. The asynchronous method variants return an `AsyncResponse` object. Use that object with a polling design pattern (check regularly whether it is done) to choose to get results later (synchronously in relation to the calling code).

See the introductory comments in “Consuming WS-I Web Service Overview” on page 71 for related information about the basic types of connections for a method.

**IMPORTANT** Be careful not to confuse one-way methods with asynchronous methods. For more information about one-way methods, see “One-Way Methods” on page 85.

The `AsyncResponse` object contains the following properties and methods:

- `start` method – initiates the web service request but does not wait for the response
- `get` method – gets the results of the web service, waiting (blocking) until complete if necessary
- `RequestEnvelope` – a read-only property that contains the request XML
- `ResponseEnvelope` – a read-only property that contains the response XML, if the web service responded
- `RequestTransform` – a block (an in-line function) that Gosu calls to transform the request into another form. For example, this block might add encryption and then add a digital signature.
- `ResponseTransform` – a block (an in-line function) that Gosu calls to transform the response into another form. For example, this block might validate a digital signature and then decrypt the data.

The following is an example of calling the a synchronous version of a method contrasted to using the asynchronous variant of it.

```
var ws = new ws.weather.Weather()

// Call the REGULAR version of the method.
var r = ws.GetCityWeatherByZIP("94114")
print( "The weather is " + r.Description )

// Call the **asynchronous** version of the same method
// -- Note the "async_" prefix to the method
var a = ws.async_GetCityWeatherByZIP("94114")

// By default, the async request does NOT start automatically.
// You must start it with the start() method.
a.start()

print("the XML of the request for debugging... " + a.RequestEnvelope)
print("")

print ("in a real program, you would check the result possibly MUCH later...")
// Get the result data of this asynchronous call, waiting if necessary.
var laterResult = a.get()
print("asynchronous reply to our earlier request = " + laterResult.Description)

print("response XML = " + a.ResponseEnvelope.asUTFString())
```

## MTOM Attachments with Gosu as Web Service Client

The W3C Message Transmission Optimization Mechanism (MTOM) is a method of efficiently sending binary data to and from web services as attachments outside the normal response body.

The main response contains placeholder references for the attachments. The entire SOAP message envelope for MTOM contains multiple parts. The raw binary data is in other parts of the request than the normal SOAP

request or response. Other parts can have different MIME encoding. For example, it could use the MIME encoding for the separate binary data. This allows more efficient transfer of large binary data.

When Gosu is the web service client, MTOM is not supported in the initial request. However, if an external web service uses MTOM in its response, Gosu automatically receives the attachment data. There is no additional step that you need to perform to support MTOM attachments.

On a related topic, you can support MTOM support when publishing a web service. See “WS-I Web Service Invocation Context” on page 45.



# Publishing Web Services (RPCE)

You can write web service APIs in Gosu and access them from remote systems using the standard web services protocol *SOAP*. The SOAP protocol defines request/response mechanisms for translating a function call and its response into XML-based messages typically sent across computer networks over the standard HTTP protocol. Web services provide a language-neutral and platform-neutral mechanism for invoking actions or requesting data from another application across a network. ClaimCenter publishes its own built-in web service APIs that you can use.

---

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

---

This topic includes:

- “RPCE Web Service Overview” on page 90
- “Publishing a RPCE Web Service” on page 95
- “Writing Web Services that Use Entities” on page 100
- “Testing Your RPCE Web Service With soap.local Namespace” on page 103
- “Calling Your Published RPCE Web Service from Java” on page 107
- “Calling Your RPCE Web Service from Microsoft .NET WSE 3.0” on page 110
- “Calling Your RPCE Web Service from Microsoft .NET WSE 2.0” on page 114
- “Calling Published RPCE Web Services From Other Languages” on page 117
- “Typecodes and Web Services in RPCE Web Services” on page 120
- “Public IDs and RPCE Web Services” on page 120
- “Endpoint URLs and Generated WSDL in RPCE Web Services” on page 122
- “Web Services Using ClaimCenter Clusters” on page 124
- “SOAP Faults (Exceptions) in RPCE Web Services” on page 126
- “Writing Command Line Tools that Call RPCE Web Services” on page 129

## RPCE Web Service Overview

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

Web services define request-and-response APIs that let you call an API on a remote computer, or even the current computer, using an abstracted well-defined interface. A data format called the Web Service Description Language (WSDL) describes available web services that other systems can call using the SOAP protocol. Many languages or third-party packages provide bindings implementations of WSDL and SOAP, including Gosu (built-into ClaimCenter), Java, Perl, and other languages. For more information about WSDL, see “Endpoint URLs and Generated WSDL in RPCE Web Services” on page 122.

Gosu natively supports web services in two different ways:

- **Publish your Gosu code as new web service APIs** – Write Gosu code that external systems call as a web service using the SOAP protocol. Simply add a single line of code before the definition of a Gosu class. For more information, see “Publishing a RPCE Web Service” on page 95 and “Writing Web Services that Use Entities” on page 100.
- **Call web service APIs that external applications publish from your Gosu code** – You can write code that easily imports web service APIs from external systems. After registering the external web service by its WSDL URL in Guidewire Studio and giving the service a name, you can easily call it from Gosu. Gosu parses the WSDL and allows you to use the remote API with a natural Gosu syntax. For more information, see “Calling RPCE Web Services from Gosu” on page 135.

In both cases, ClaimCenter converts the server’s local Gosu objects to and from the flattened, text-based format required by the SOAP protocol. This process is *serialization* and *deserialization*.

- Suppose you write a web service in Gosu and publish it from ClaimCenter and call it from a remote system. Gosu must deserialize the text-based request into a local object that your Gosu code can access. If one of your web service methods returns a value, ClaimCenter serializes that local in-memory Gosu object and serializes it into a text-based reply for the remote system.
- Suppose you use Gosu to call a web service hosted by an external system. Before calling the API, Gosu automatically serializes any API parameters to convert a local object into a flattened form to send to the API. If the remote API returns a result, ClaimCenter deserializes the response into local Gosu objects for your code to examine.

Guidewire provides built-in web service APIs for common general tasks and tasks for business entities of ClaimCenter. For a full list, see “Reference of All Built-in Web Services” on page 33.

However, writing your own web service for each integration point is simple. Guidewire strongly encourages you to write as many web services as necessary to elegantly provide APIs for each integration point.

For example, write new APIs to communicate with a check printing service, a legacy financials system, reporting service, or document management system. Systems can query ClaimCenter to calculate values, trigger actions, or to change data within the ClaimCenter database.

Depending on the complexity and data size for one request, publishing a web service may be as simple as writing one special line of code before your Gosu class. See “Publishing a RPCE Web Service” on page 95.

There are special additional tasks or design decisions that affect how you write your web services, for example:

- **Learn bundle and transaction APIs** – To change and commit entity data in the database, use special APIs discussed in “Writing Web Services that Use Entities” on page 100. You do not need to use these APIs if you simply get (return) data in your web service and do not change entity data.
- **Be careful of big objects** – If your data set is particularly large, it may be too big to pass over the SOAP protocol in one request. You may need to refactor your code to accommodate smaller requests. If you try to pass too much data over the SOAP protocol in either direction, there can be memory problems that you must avoid in production systems.

- **Create custom structures to send only the subset of data you need** – For large Guidewire business data objects (entities), most integration points only need to transfer a subset of the properties and object graph. Do not pass large object graphs, and be aware of any objects that might be very large in your real-world deployed production system. In such cases, you must design your web services to pass your own objects containing only your necessary properties for that integration point, rather than pass the entire entity. For example, if an integration point only needs a record's main contact name and phone number, create a shell object containing only those properties and the standard public ID property. For details, see “Publishing a RPCE Web Service” on page 95.
- **Web service client code can reference existing entities by reference** – For web services that take parameters, web service clients can reference an existing entity in the system without sending the entire entity across the network. Instead, the web service client can send entities by reference. Simply set the entity’s ByRef property and the public ID property. See “As a SOAP Client, Refer to an Entity By Reference with a Public ID” on page 121. This approach does not apply to new entities (to add to the ClaimCenter database) or to entities returned from the web service. Even better than this approach is to use custom structures containing only the subset of data you need (see previous paragraph).
- **Design your web service to be testable** – For a detailed example of changing entity data, see “Testing Your RPCE Web Service With soap.local Namespace” on page 103.

Guidewire recommends Java for code on external systems that connect to web services published by ClaimCenter. Guidewire provides a Java language binding for published web services. Using these bindings, you can call the published web services from Java program as easy as making a local method invocation. However, you can use other programming languages if they have access to a SOAP implementation and can access the ClaimCenter server over the Internet or intranet. Guidewire recommends Java for web service client code, so this topic usually uses Java syntax and terminology to demonstrate APIs. In some cases, this topic uses Gosu to demonstrate examples as it relates to publishing or testing web services.

**Note:** If you use or write APIs to integrate two or more Guidewire applications, write all your web service code in Gosu using its native SOAP interface.

---

**WARNING** Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance. If you have questions about how to convert some particular use locally-hosted SOAP APIs from plugins or rules, contact Customer Support. This is true for all types of local loopback SOAP calls to the same server. This includes `soap.local.*` objects, the SOAP client Java classes from Java plugins, and Studio-registered web services that call the same server as the client.

---

## Differences Between Publishing RPCE and WS-I Web Services

There are two ways to publish web services:

- **WS-I web services** – WS-I web services use the SOAP protocol that are compatible with the industry standard called WS-I.

---

**IMPORTANT** For new web services, Guidewire strongly recommends that you publish web services using the WS-I APIs.

- **RPC Encoded (RPCE) web services** – RPCE web services are the older style of web service publishing APIs. Versions of ClaimCenter before version 7.0 supported only the RPCE type of web service publishing. ClaimCenter retains support for RPCE web service publishing only for backward compatibility. Do not create new RPCE web services.

**See also**

- For the functional differences between RPCE and WS-I web services, see “Web Services Introduction” on page 31.
- For the implementation differences between the ClaimCenter implementation of RPCE web services and WS-I web services, see “WS-I Web Services” on page 122 in the *New and Changed Guide*.

## Designing RPC Encoded (RPCE) Web Services

The following information applies only to RPCE web services, not necessarily also for WS-I web services.

### Entity Access for RPCE Web Services

If you must call a SOAP API from an external system, or even from another Guidewire application, you must specially handle Guidewire entities. Entities represent business data such as claims, users, addresses, and notes. These objects have names that describe their data, such as `Claim`, `User`, `Address`, and `Note`.

From an external system, you can access ClaimCenter entities that correspond to the native full entities in Gosu. However, they are simple versions of the entities that contain data only. You cannot remotely call domain methods on these entities. Domain methods are functions on the entities that you can access from Gosu. Domain methods trigger complex business logic or calculate values.

In contrast, the only methods on the entities as viewed from the external system are getter and setter methods that get and set properties on local versions of data-only entities. These methods have names that start with "get" and "set". To call attention to the differences between the full entities and these data-only entities viewed through the SOAP interface, Guidewire documentation sometimes calls the data-only entities *SOAP entities*.

To import new records into ClaimCenter, your web services client code would create a new SOAP entity such as an instance of the `Address` class. Populate all relevant properties, taking care to populate all required properties. Determine required properties by referring to the application *Data Dictionary*. Finally, your code can then pass the object as an argument to a web service API call.

If you call SOAP APIs, the namespace hierarchies (packages) from your web service client code of entities differ from namespaces of entities as accessed from Gosu code. See “New Entity Syntax Depends on Context” on page 25.

## Which Objects and Properties Work With RPCE Web Services?

You must be careful what types you expose to SOAP as method parameters or types of return values from methods. Not all types work across the SOAP layer.

### Exposing Entities to the SOAP Layer

You can expose Guidewire entities to the SOAP layer, although as a general design principle you might consider avoiding it, especially for large entities and large graphs. Instead, you can create Gosu classes containing the fields that you want to expose to some particular integration point. This approach can simplify logic relating to bundles. For large entities and large graphs, it also reduces the memory and network impact of any SOAP request that uses the entities. Remember that ClaimCenter must serialize or deserialize objects that travel across SOAP.

If you use entities, note that the SOAP layer exposes data model extension properties. For more information about data model extensions, see “Modifying the Base Data Model” on page 209 in the *Configuration Guide*. For example, if you add an extension property named `MyNewField` to `Claim`, then the `Claim` data object includes a new first-class property `MyNewField`. In Gosu, access it with `Claim.MyNewField`. From Java SOAP client code, access it with the accessor method `Claim.getMyNewField()`.

If you add entirely new custom entities, they are available as SOAP entities if and only if they their `exportable` property has the value `true`. To read and write an entity using the SOAP interface, in the data model set the

`exportable` attribute for the property to `true`. If you make changes and you use the Java generated libraries, be sure to regenerate the integration libraries after data model updates.

In some cases, the data model defines some properties as read-only or write-only. Keep this in mind as you design APIs that use these properties.

### Avoiding Locally-Hosted SOAP APIs

Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data.

Instead, refactor your code to avoid making calls to locally-hosted SOAP APIs. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance.

If you have questions about how to refactor code that calls locally-hosted SOAP APIs from plugins or rules, contact Guidewire Customer Support. Contact Customer Support for all types of local loopback SOAP calls to the same server. This includes `soap.local.*` objects, the SOAP client Java classes from Java plugins, and Studio-registered web services that call the same server as the client.

**IMPORTANT** Avoid calling locally hosted SOAP APIs in production systems.

### Limitations on What Kinds of Types Work with SOAP

Some Java-based classes or language features do not transparently convert over the SOAP protocol. The following limits exist for publishing Gosu classes as web services:

- **No lists** – You can pass arrays over the SOAP protocol, but you cannot pass lists based on `java.util.List` over the SOAP protocol as parameters or return results. Convert any lists to arrays before returning results from your APIs. For parameters, convert array parameters to lists to use with list-based APIs. For list and array conversion methods, see “Collections” on page 251 in the *Gosu Reference Guide*.
- **No maps or other collections** – Similar to limitations on `List`, web services cannot pass map instances (objects based on `java.util.Map`) or other collections across the SOAP protocol.
- **No overloaded methods with same argument numbers** – You may not have two or more methods with the same method name and the same number of arguments. This is unsupported and has undefined results.
- **No types that lack a no-argument constructor** – To be exportable to web services, objects must provide a simple constructor with no arguments. You optionally can provide other constructors, but Gosu does not use them for web services. Typically this is not problematic because if you do not define a constructor at all in a class, Gosu implicitly creates a trivial no-argument constructor. If you create a constructor with arguments, you must also explicitly create a constructor with no arguments. Gosu relies on the no-argument constructor to create a new object during deserialization of SOAP objects.
- **No generic types or parameterized types** – You cannot pass objects with types that use Gosu generics features across the SOAP protocol. For more information about these features, “Gosu Generics” on page 239 in the *Gosu Reference Guide*.
- **No Java types outside the `gw.*` namespace** – Certain built-in Java types can be passed across the SOAP protocol. However, you cannot pass Java-based types outside the `gw.*` namespace. Do not attempt to do this. For example, if you use your own Java libraries, Gosu cannot serialize objects in that library as arguments or return values anywhere in an object graph. However, you can create your own Gosu classes that contain the same properties. You can pass your custom Gosu class to and from your web service.
- **No annotations** – You cannot serialize or deserialize classes that implement annotations classes. For more information about these features, see “Annotations” on page 219 in the *Gosu Reference Guide*.

- **No abstract types** – You cannot serialize or deserialize classes that are abstract because they effectively have no data to pass across. For more information about abstract types, see “Modifiers” on page 198 in the *Gosu Reference Guide*.
- **No passing across SOAP APIs or already-SOAP-specific types** – You cannot write web services that attempt to serialize or deserialize classes that implement SOAP APIs or types already defined in Gosu within the package hierarchy `soap.*`. This restriction prevents certain types of recursive compiler issues.

#### See also

- “Integration Documentation Overview” on page 20.
- For full specifications of entity classes, object properties, and typelist values, see the *API Reference* documentation and the *ClaimCenter Data Dictionary*.

## Web Service Types Must Have Unique Names

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

For example, suppose you had two web services:

- Web service MyAPI1 exposes an entity `entity.Policy` as a return type.
- Web service MyAPI2 exposes a Gosu type `mypackage.integration.Policy` as a method parameter

If you try to access locally-published SOAP types, both evaluate to `soap.local.entity.Policy`. Similarly, there is ambiguity in the WSDL and generated Java libraries.

If you attempt to use two different types with the same name (even if different package), Gosu flags it as a compile error.

To make development due to this problem easier, ClaimCenter has a configuration parameter called `AllowSoapWebServiceReferenceNamespaceCollisions`. If set to `true`, these error messages become warnings instead. Use this for development and debugging until you have time to rename your classes to fix the namespace collision. This value is `false` by default.

---

**WARNING** It is unsafe to set `AllowSoapWebServiceReferenceNamespaceCollisions` to `true` for production servers. If you find it difficult to rename classes to avoid namespace collisions, please contact Guidewire Customer Support.

---

## SOAP Entity Terminology Notes

Some terminology notes about *objects* and *entities*:

- The WSDL/SOAP specification describes all objects passed across SOAP as what those protocol specifications call *entities*. The SOAP sense of the term *entities* differs greatly from the Guidewire sense of the term. Except for this topic about SOAP, all other Guidewire documentation uses the term *entities* to mean business data objects defined in the data model configuration files.
- Guidewire Integration documentation sometimes refers to SOAP entities with the general term *objects*. The Java and Gosu implementations of SOAP make what SOAP calls entities appear as objects in these programming languages. SOAP implementations for languages other than Java and Gosu might not represent SOAP entities as objects.

The following table summarizes publishing a web service as a RPCE web service.

Feature	RPCE web service behavior
The basic annotation on a class to publish a web service	@RPCWebService <b>IMPORTANT:</b> This is just a new name for the annotation @WebService. The old name still works and refers to RPCE web services, but is deprecated.
Can serialize a Java object?	Yes, in limited situations, such as some Java-based types inside the gw.* package namespace. For details, see “Which Objects and Properties Work With RPCE Web Services?” on page 92.
Can serialize a Guidewire entity and its subobjects?	Yes
Can serialize a XSD-based type	No
Can serialize Gosu class instances, sometimes called POGOs (Plain Old Gosu Objects)?	Yes
How to specify the minimum run level	As an argument to the basic web service publishing annotation
How to specify the required permissions to use the service?	As an argument to the basic web service publishing annotation
Automatically throw SOAPException exceptions?	Yes
Bundle handling	RPCE web services put any entity changes into a default bundle, which you can manually commit as needed.
<b>Note:</b> a <i>bundle</i> is a container for entities that helps ClaimCenter track what changed in a database transaction. See “Bundles and Database Transactions” on page 331 in the <i>Gosu Reference Guide</i> .	
Logging in to the server	There are two ways to log into a server. <ul style="list-style-type: none"> <li>For Java clients that used the generated libraries, the standard approach was to use the APILocator utility class to manage the connection and handling connection pooling.</li> <li>Other client languages (or Java if desired) can connect to the ILoginAPI web service interface.</li> </ul>
Package name of web services from the SOAP API client perspective	The namespaces are flattened so that all web services appear in the same package.
Calling a local version of the web service	API references in the package soap.local.api.APINAME All the references are in that package, independent of the package of the implementation class. You must call the command line command to rebuild the local files: ClaimCenter/bin/gwcc regen-soap-local

**IMPORTANT** For new web services, Guidewire strongly recommends that you use WS-I web services. Guidewire retains RPCE web services for backward compatibility only.

#### See also

For a comparison for RPCE and WS-I web services, see “WS-I Web Services” on page 122 in the *New and Changed Guide*.

## Publishing a RPCE Web Service

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

You can write Gosu code that external systems can call. For any Gosu function you want to publish, simply make it a method in a new Gosu class and publish the class as a web service. This is often as easy as adding a single *Gosu annotation* immediately before a Gosu class. A Gosu annotation is a special line of text before the class definition.

External systems can easily call this method from external code. External systems can use ClaimCenter-published XML data called WSDL (Web Services Description Language) to formulate a request and its results. The WSDL describes the service, all its methods, what parameters they take, what kinds of data they return, and every detail about parameter types and return types.

To publish a web service, simply add the following line before a Gosu class definition:

```
@WebService
```

For example, this simple class publishes a simple web service with one API method:

```
@WebService
class MyServiceAPI {

    function echoInputArgs(p1 : String) : String {
        return "You said " + p1
    }

}
```

Note the naming convention is to name a web service implementation class ending with the string “API”.

By default, the application publishes this web service in the ClaimCenter SOAP API libraries. If you regenerate the SOAP API libraries, ClaimCenter generates Java libraries you can use to connect to the web service from Java code. You can customize this behavior with the optional Boolean parameter `generateInToolkit`, as discussed further in “Optional Web Service Publishing Options” on page 97.

As mentioned in “RPCE Web Service Overview” on page 90, for large Guidewire business data objects (entities), most integrations need only to transfer particular parts of the graph. Do not pass large object graphs, and be aware of any objects that might be very large in your real-world deployed production system. In such cases, you must design your web services to pass your own objects containing only your necessary properties for that integration point, rather than pass the entire entity.

**Note:** Web service methods that take parameters do not need to send the entire entity across the SOAP protocol. Instead, the web service client can send some or all entities by reference. See “As a SOAP Client, Refer to an Entity By Reference with a Public ID” on page 121.

For example, if an integration point gets a name and phone number, create a Gosu class containing only those properties and the standard public ID property.

Suppose you just want a couple properties from a `User` entity. Create a simple Gosu class such as the following example:

```
package example

class UserPhones {

    private var _publicID : String as PublicID
    private var _h : String as HomePhone
    private var _w: String as WorkPhone
}
```

You can then write a web service to get a user by the public ID and return one of your custom structures:

```
package example

@Service
class MyServiceAPI {

    function getUserPhones(userPublicID : String) : UserPhones {
        var u : User = User(userPublicID)

        if (u==null) {
            throw "No such user"
        }
}
```

```
// Create an new instance of your special integration point structure.  
var up = new UserPhones()  
up.PublicID = u.PublicID  
up.HomePhone = u.Contact.HomePhone  
up.WorkPhone = u.Contact.WorkPhone  
  
    return up  
}  
}
```

**IMPORTANT** Write web services that output only the properties you need, and accept only the properties you need rather than transferring entire entities. Write different web services for each integration point to contain different subsets of entity data. Create custom entities or Gosu classes as necessary to move objects to and from the external system.

**IMPORTANT** To access local SOAP APIs for RPCE web services, you must call the `gwcc regen-soap-local` script to regenerate local files. Also, after you add, modify, or delete an RPCE web service, call `regen-soap-local` again.

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

#### See also

- “Optional Web Service Publishing Options” on page 97.
- “Writing Web Services that Use Entities” on page 100.
- “Local RPCE SOAP Endpoints in Gosu” on page 137.
- “Avoiding Locally-Hosted SOAP APIs” on page 93.
- “Regenerating Integration Libraries” on page 22.
- For a detailed example of testing an web service that reads entity data and commits entity changes, see “Testing Your RPCE Web Service With `soap.local` Namespace” on page 103.

## Optional Web Service Publishing Options

The `@WebService` annotation has optional arguments with which you can specify authentication rules and restrictions to be enforced for the entire class. These authentication rules may, however, be overridden on a per-method basis using the `@WebServiceMethod` annotation. In cases in which authentication rules differ between a web service class and a method, the method-specific annotation always takes precedence.

The following optional authentication restrictions can be specified for a published web service and optionally overridden on for any subset of methods on the class:

- Generate in toolkit** – Creates generated Java libraries to connect to this web service in the `soap-api` subdirectory of the application. (The SOAP API and Java API libraries are collectively called the *toolkit*, hence the name). Also generates Javadoc for the web service and WSDL for the web service. By default, this is set to `true`, meaning generate web service materials in the SOAP API libraries. You can set it to `false` to suppress this behavior. Suppressing this behavior is useful if you plan to connect to the web service that uses WSDL generated and published dynamically from ClaimCenter, rather than Java library files. For example, Microsoft .NET web service clients connect to the dynamic WSDL. Microsoft .NET does not use ClaimCenter generated SOAP API Java libraries nor the raw WSDL in the SOAP API subdirectories. If

another Guidewire application is the only client for your web service, use this parameter to suppress publishing the web service.

---

**WARNING** Merely disabling generating the web service in the toolkit does not prevent outside code from calling a web service. If Gosu publishes the web service, the server accepts connections if it has proper authentication credentials even if this parameter is set to `false`. To disable the web service, comment out the `@WebService` annotation entirely using the `//` symbols before that line. If you comment out this line, the class is unavailable to clients. The application refuses SOAP client connections to this service.

---

- **Minimum run level** – Specifies a minimum run level of the server before publishing the SOAP API. Specify the run level as a `WSRunLevel` enumeration.
  - `SHUTDOWN` – The system is active, but the product is not.
  - `NODAEMONS` – The system and product run but no background processes run. Background processes are also known as *daemon processes*. This is the default value if you omit this parameter.
  - `DAEMONS` – The system and product run and background processes run.
  - `MULTIUSER` – The system and product are live with multiple users
- **Required permissions** – Specifies a list of required permissions that a user must have to connect to use this API. Specify the required permissions as an array of `SystemPermissionType` typekeys. The default value if you omit this parameter is the one-element array with the SOAP Administrator permission. In other words, the default is `{ SystemPermissionType.TC_SOAPADMIN }`. To specify that you do not require any particular user permissions to access this web service, specify the empty array: `{ }`

The complete set of possible combinations of parameters for the `@WebService` annotation are:

```
@WebService(WSRunlevel, SystemPermissionType[], Boolean /*generateInToolkit*/)
@WebService(WSRunlevel, SystemPermissionType[])
@WebService(SystemPermissionType[], Boolean /*generateInToolkit*/)
@WebService(SystemPermissionType[])
@WebService(WSRunlevel, Boolean /*generateInToolkit*/)
@WebService(WSRunlevel)
@WebService(Boolean /*generateInToolkit*/)
```

You can also specify no parameters on the annotation to use all default parameter values:

```
@WebService() // no parameters
@WebService() // no parameters, with parentheses omitted
```

For example, to require a minimum run level of `DAEMONS` and require the `NOTECREATE` permission, and to use the default generate in toolkit option, use the following code:

```
@WebService(DAEMONS, { SystemPermissionType.TC_NOTECREATE } )
```

To set the generate in toolkit parameter to `false`, use the code:

```
@WebService(false /*GenerateInToolkit*/)
```

---

**IMPORTANT** Disabling toolkit generation does not disable the service. See the warning in this section.

---

## Method-level Authentication Overrides

Use the `@WebServiceMethod` annotation to override authentication restrictions on a method-by-method basis. If you do not specify the authentication features explicitly on the class, Gosu uses the default values as described in the previous section. Authentication settings that you do not explicitly set on the method defaults to the class's authentication levels. If the class annotation does not specify authentication values, the application uses the default values listed in the previous section. If authentication rules differ between a web service class and a method, the method annotation always takes precedence.

The possible combinations of parameters for the `@WebService` parameter are:

```
WebServiceMethod(WSRunlevel, SystemPermissionType[])
WebServiceMethod(SystemPermissionType[])
WebServiceMethod(WSRunlevel)
```

The default values for arguments are the same as for the class-level authentication settings: the NODAEMONS for the run level and SOAP\_Admin permission for system permission.

**IMPORTANT** Use the @WebServiceMethod annotation to strategically set permission requirements and run level restrictions for particular web service actions.

## Method-level Visibility Overrides

The @DoNotPublish annotation is a method-level annotation for Gosu classes. Use this annotation on methods of web service implementation classes to omit methods from the external web service. Use this to annotate public methods only, which means only those that use the `public` modifier. This has no affect on non-public methods. Gosu never publishes non-public methods, which are methods with the modifier `private`, `package`, or `protected`.

## Declaring Exceptions

Problems might occur during a web service API call because of invalid requests or other reasons. Your web service must check for unusual conditions and handle exceptions appropriately. If a web service implementation throws an exception, Gosu returns the error to the SOAP client. Declare exceptions that you expect to throw so that Gosu can publish those exception types in the WSDL. If your method throws an exception that you did not declare, Gosu returns a general error that might not contain enough details for effective debugging and error handling.

To declare the exception, use the @Throws annotation. For example:

```
package example
uses gw.api.webservice.exception.SOAPException
uses java.util.ArrayList

@WebService
class ExampleAPI {

    // Get all address public IDs that match a certain query (in this case, by postal code)
    @Throws(SOAPException, "If too many addresses match the given postal code")
    public function getAddressPublicIDs(posCode : String) : String[] {

        var q = find( a in Address where a.PostalCode == posCode)

        // You must do error checking for large data sets so you do not
        // try to send too much data across the network or try to serialize
        // too much data on the server.
        if (q.getCount() > 1000) {
            throw new SOAPException("Integration exception: too many addresses to return.")
        }
    ...
}
```

For more information about exceptions, including the hierarchy of exception types, see “SOAP Faults (Exceptions) in RPCE Web Services” on page 126.

## Configuring When To Initialize RPCE Web Service Implementation Classes

There is a `config.xml` parameter called `LoadSoapServicesOnStartup`, which affects RPCE web service publishing:

- If present and set to `true`, for RPCE web services, instantiation of all the implementation classes and registering of the WSDL happens on server startup.
- Otherwise, initialization of all implementation classes happens on the first request to any service.

For customers who use clusters of servers, you can set this differently on each server. For example, if you have only one server handling web service requests, if you set this only on that one server, startup time on the other servers is unaffected.

This parameter affects only RPCE web services, not WS-I web services. WS-I web services construct objects only when after the first request for that specific web service.

## Writing Web Services that Use Entities

---

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

---

This section assumes the reader understands public IDs and their role in the ClaimCenter data model. For details, see “Public IDs and RPCE Web Services” on page 120. That topic also covers information regarding special issues with writing SOAP client code using public IDs.

### Querying for Entity Data in Your Web Service

Real integration code must perform complex tasks like querying the database for Guidewire entities in the database. For example, you might want to perform tasks like:

- An API to get public IDs for certain entities that match a certain query.
- An API to get a single entity by its public ID

The following example has two methods:

- One method finds all Address entities matching a certain postal code and returns a list of public ID for the matching data.
- One method gets an Address entity by its public ID and returns the entire entity. Address entities are small. For some entities, full entity is very large including all subobjects.

Always test if your database queries potentially return too much data. If they do, throw exceptions or return partial data and set a flag, whichever is more appropriate for your integration point.

---

**WARNING** Guidewire strongly recommends error checking, bounds checking, and checking for situations where too many results create memory errors. Follow the example of error checking in the example, which checks for too many results.

---

Add this example class in the example package in Guidewire Studio to use it:

```
package example
uses gw.api.webservice.exception.SOAPException
uses java.util.ArrayList

@WebService
class ExampleAPI {

    // Get all address public IDs that match a certain query (in this case, by postal code)
    @Throws(SOAPException, "If too many addresses match the given postal code")
    public function getAddressPublicIDs(posCode : String) : String[] {

        var q = find( a in Address where a.PostalCode == posCode)

        // You must do error checking for large data sets so you do not
        // try to send too much data across the network or try to serialize
        // too much data on the server.
        if (q.getCount() > 1000) {
            throw new SOAPException("Integration exception: too many addresses to return.")
        }

        var newList = new ArrayList<String>();
        for( oneAddress in q ) {
            var summary = (oneAddress as String)
            print("Adding to SOAP results.. " + summary)
            newList.add( oneAddress.PublicID )
        }

        return newList
    }
}
```

```
// Get an entire Address by its public ID.  
// The return value has the static type = Address.  
public function getAddress(publicID : String) : Address {  
    return Address(publicID)  
}  
  
}
```

For a detailed example of testing a web service that commits entity changes, see “Testing Your RPCE Web Service With soap.local Namespace” on page 103.

**WARNING** Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance. If you have questions about how to convert some particular use locally-hosted SOAP APIs from plugins or rules, contact Customer Support. This is true for all types of local loopback SOAP calls to the same server. This includes `soap.local.*` objects, the SOAP client Java classes from Java plugins, and Studio-registered web services that call the same server as the client.

## Committing Entity Data in Your Web Service

**IMPORTANT** The following information applies only to RPCE web services, not WS-I web services

You may need to use database transactions APIs if you design RPCE web services that take parameters that you must later commit to the database. You may need to use other APIs if your RPCE web service looks up entity data in the database and make database changes. The full set of information is in “Bundles and Database Transactions” on page 331 in the *Gosu Reference Guide*.

A *bundle* is a group of Guidewire entities, grouped together so they save to the database together. You can get the current transaction if there is one by calling `gw.transaction.Transaction.getCurrent()`.

The most important things to know about changing entity data in your RPCE web service as follows:

- When a web service client calls your RPCE web service, the system sets up a bundle automatically.
- Get the current bundle by calling `gw.transaction.Transaction.getCurrent()`. The current bundle is the bundle setup for your code. You can add entities to this writable bundle. For example, add read-only entities to this bundle.
- If you have a reference to an entity instance, get its bundle as the `object.bundle` property. However, generally speaking use `gw.transaction.Transaction.getCurrent()`.
- Any objects serialized into the RPCE web service as arguments to your web service are already in the current bundle. You do not need to add them to the current transaction.
- No data is automatically committed to the database after your RPCE web service completes. If you want the existing transaction (including serialized data) to commit to the database you must do so manually by calling the `commit` method on the bundle.
- If you query the database, entity instance query results are initially read-only. You must add them to the current writable transaction to modify the entities. Add them by calling the following code:  
`e = gw.transaction.Transaction.getCurrent().add(e)`

You must save the return result of the `add` method, which is a clone of the object. It is best to reuse the variable if possible to avoid referencing the original entity instance reference. See “Adding Entity Instances to Bundles” on page 334 in the *Gosu Reference Guide*.

- You can create an entirely new bundle to encapsulate a task in a Gosu block. A block is an in-line Gosu function contained in other Gosu code. See “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide*.

New entity instances by default are in the new bundle because the current bundle is set to the new bundle within your Gosu block. You can use the no-argument constructor, for example:

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

However, for database query results, you must add found entity instances to the new writable bundle, as mentioned previously. You must save the return result of the add method, which is a clone of the object. It is best to reuse the variable if possible to avoid referencing the original entity instance reference. See “Adding Entity Instances to Bundles” on page 334 in the *Gosu Reference Guide*.

- There may be cases where RPCE web service method parameters include entities that you do not want to commit but you make other data changes that must commit. If so, create an entirely new bundle and explicitly add entities to that group. See “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide*. Simply let the application commit the new bundle created in `runWithNewBundle`. If you use `runWithNewBundle`, do not commit the default (current) bundle for the RPCE web service.
- The `config.xml` configuration parameter `UseSafeBundleForWebServiceOperations` changes the behavior of bundle commits in RPC-Encoded web services published on this server. The default value is `true`. If set to `false`, the application ignores bean version conflicts as it commits a bundle. If set to `true`, the application detects (and does not ignore) bean version conflicts. If you set this parameter to `true`, it is possible for Gosu to throw a `ConcurrentDataChangeException` exception. This exception can happen if another thread or cluster node modified this entity as it was loaded from the database. If this error condition occurs, then the SOAP client receives a `SOAPRetryableException`. Guidewire strongly recommends that web service clients catch all retryable exceptions such as this and retry the SOAP API call.

---

**WARNING** Carefully consider how to set the configuration parameter `UseSafeBundleForWebServiceOperations`.

---

The following simple example RPCE web service takes an entity instance:

```
uses gw.transaction.Transaction

@WebService
class AddressExampleAPI {

    public function insertAddress(newAddress : Address) : String {
        // commit all objects in the current bundle, which by default includes RPCE operation arguments
        gw.transaction.Transaction.getCurrent().commit()

        // note: for SOAP API implementations, bundle includes incoming entities
        //       deserialized with the request.

        return newAddress.PublicID
    }
}
```

---

**WARNING** Be extremely careful only to commit entity instance changes at appropriate times or serious data integrity errors occur. If an entity instance’s changes commit to the database, changes can no longer roll back if errors happen in related code. Typically, errors must undo all related database changes. For example, Gosu code in rule sets must never commit data explicitly since the application automatically commits the bundle automatically only after all rule sets and validation successfully runs.

---

The following example RPCE web service gets a user and adds roles to it. Finally, it commits the result:

```
/**
 * Adds roles to a User.
 * If a role already belongs to the user, it is ignored.
 *
 * @param userID The ID of the user
 * @param roleIDs The public IDs of roles to be added.
 */
```

```
@Throws(DataConversionException, "if the userID or roleID does not exist")
@Throws(SOAPException, "")
public function addRolesToUser(userID : String, roleIDs : String[]) : String {

    var user = User(userID) // This syntax finds the user with a specific ID.
    if (user == null){
        throw new DataConversionException("No User exists with PublicID: " + userID);
    }

    // Add the user to current bundle, the current database transaction.
    // This is always strictly required if you got the entity instance from a find() query!
    // We get the return value of add() and it has a DIFFERENT value than before.
    // It is a copy in the new bundle!
    user = Transaction.getCurrent().add(user);

    if (roleIDs == null || roleIDs.length == 0){
        throw new RequiredFieldException("Roles");
    }

    // Add in all the roles for the user.
    for (var roleId in roleIDs) {
        var role = loadRoleByIdOrThrow(roleId);

        // Pass an argument when creating a new entity instance with "new" to use a specific bundle.
        var userRole = new UserRole();

        userRole.Role = role
        user.addRoles(userRole);
    }

    // Commit our bundle
    Transaction.getCurrent().commit();
    return userID;
}
```

For more bundle and transaction API information, see “Bundles and Database Transactions” on page 331 in the *Gosu Reference Guide*. This topic does not duplicate that topic’s information.

---

**IMPORTANT** For a detailed example of testing a web service that commits entity changes, see “Testing Your RPCE Web Service With soap.local Namespace” on page 103.

---

**WARNING** Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance. If you have questions about how to convert some particular use locally-hosted SOAP APIs from plugins or rules, contact Customer Support. This is true for all types of local loopback SOAP calls to the same server. This includes `soap.local.*` objects, the SOAP client Java classes from Java plugins, and Studio-registered web services that call the same server as the client.

---

## Testing Your RPCE Web Service With soap.local Namespace

---

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

---

If you write new web service API, you must carefully test it to confirm that it works.

To test only the basic logic of your code or to use your API from other places in Gosu, do not need to connect over SOAP. Instead, just call it directly from Gosu using the class name, such as:

```
var res = MyServiceAPI.MyMethod()
```

However, that does not connect over the SOAP protocol. A better test is to use the SOAP interface to test for issues that might arise due to the SOAP implementation. For example, test that Gosu appropriately serializes and deserializes objects. This approach also tests database transaction management (bundles) more realistically.

For information about calling RPCE web services from Gosu, see “Calling RPCE Web Services from Gosu” on page 135.

You can call your RPCE SOAP API by using the special `soap.local.api.*` namespace to find your API on the same server. This lets you actually call your API over the SOAP protocol, rather than just directly calling the implementation classes. Calling your own server using SOAP is good for testing, but unsupported for production code. See the warning later in this topic.

---

**IMPORTANT** To access local SOAP APIs for RPCE web services, you must call the `gwcc regen-soap-local` script to regenerate local files. Also, after you add, modify, or delete an RPCE web service, call `regen-soap-local` again.

---

The namespace hierarchy of all the web service implementation classes are flattened. The name of your service is not part of the package hierarchy in which you originally defined your class with the `@WebService` annotation. This means that all local web services share the same namespace for all objects related to all SOAP APIs as argument types or return types.

For example, if your web service is `MyServiceAPI`, for testing purposes you can call the API on the same server as a local SOAP endpoint:

```
var myAPI = soap.local.api.MyServiceAPI
```

For example, suppose you publish the following simple web service:

```
@WebService
class MyServiceAPI {
    function echoInputArgs(p1 : String) : String {
        return "You said " + p1
    }
}
```

Write a GUnit test to test this using the SOAP interface. You must add `@SOAPLocalTest` annotation to tell GUnit that your test requires the `soap.local.*` namespace (otherwise GUnit does not set it up).

```
package example
uses gw.api.soap.GWAuthenticationHandler

@SOAPLocalTest
@gw.testharness.ServerTest
class MyTestClassTest extends gw.testharness.TestBase {

    public function test1() {

        // Get the API reference, but do not call it yet.
        var myAPI = new soap.local.api.MyServiceAPI()
        myAPI.addHandler( new GWAuthenticationHandler("su", "gw") )

        // Now, call the API.
        var r = myAPI.echoInputArgs( "San Francisco" )

        // What answer did the API give?
        print(r)

        // Check the answer and throw an exception if it is wrong!
        if (r != "You said San Francisco") {
            throw "Wrong answer!"
        }
    }
}
```

In Studio, if you select **Class → Run Test Class**, the following eventually prints in the GUnit console window:

```
22:28:08,536 INFO ***** MyTestClassTest ***** SUITE SETUP OK
You said San Francisco
22:28:09,017 INFO ***** MyTestClassTest ***** SUITE TEARDOWN
```

If you want even simpler and easy-to-read code, use built-in utility functions that compare values and throw exceptions and display user-readable errors. For example:

```
TestBase.assertEquals("My message", r, "You said San Francisco")
```

For the most complete testing, test from external system integration code to confirm your integration code works as expected. It is important to test with large data sets and objects as large as potentially exist in your production system database. Connecting from an external system is also important to test assumptions and interactions regarding database transactions. Be sure to test all your bundle-related code in the web service.

---

**WARNING** Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance. If you have questions about how to convert some particular use locally-hosted SOAP APIs from plugins or rules, contact Customer Support. This is true for all types of local loopback SOAP calls to the same server. This includes `soap.local.*` objects, the SOAP client Java classes from Java plugins, and Studio-registered web services that call the same server as the client.

---

## Extended Web Service Testing

For large Guidewire business data objects (entities), most integration points only need to transfer a subset of the properties and graph. You must not pass large object graphs. Be aware of any objects that might be very large in your real-world deployed production system. Generally speaking, design custom web services to pass your own Gosu classes containing only your necessary properties for that integration point. Do not pass the entire entity. For example, if an integration point only needs a contact name and phone number, create a Gosu class containing only those properties and the standard public ID property.

This example uses this approach to get or set information in the system.

Suppose you have a simple class to encapsulate user phone numbers:

```
package example

class UserPhones {
    private var _publicID : String as PublicID
    private var _h : String as HomePhone
    private var _w: String as WorkPhone
}
```

You can write this web service with getter and setter methods:

```
package example
uses example.UserPhones
uses gw.api.webservice.exception.DataConversionException;

@WebService
class MyServiceAPI {

    @Throws(DataConversionException, "Throws if no such user")
    function setUserPhones(up : UserPhones) {

        // Find user by public ID and load into writable bundle.
        var u = gw.transaction.Transaction.getCurrent().loadByPublicID( User, up.PublicID) as User

        if (u == null) {
            throw new DataConversionException("No user exists with PublicID: " + up.PublicID);
        }

        // get associated contact
        var c = u.Contact

        if (c == null) {
            throw new DataConversionException("User has no valid contact set up yet.");
        }
    }
}
```

```

        c.HomePhone = up.HomePhone
        c.WorkPhone = up.WorkPhone
        gw.transaction.Transaction.getCurrent().commit()

    }

function getUserPhones(userPublicID : String) : UserPhones {
    // Get a user -- note this is in a READ ONLY bundle. If you need to modify the user,
    // instead use gw.transaction.Transaction.getCurrent().loadByPublicId(...)
    var u = find ( u in User where u.PublicID == userPublicID).getAtMostOneRow()
    if (u == null) {
        throw new DataConversionException("no such user");
    }

    // Create an new instance of your special integration point structure
    var up = new UserPhones()
    up.PublicID = u.PublicID
    up.HomePhone = u.Contact.HomePhone
    up.WorkPhone = u.Contact.WorkPhone

    return up
}

```

Notice that the web service has two main methods. You must probably test all of the methods at least once. You might need more if you have different types of tasks or different data types to test.

The following example shows how you might test this class. It does the following:

1. Get a user public ID. This example simply uses the su user.
2. Sets a user's home phone and work phone numbers.
3. Checks a user's home phone and work phone numbers.
4. Sets a user's home phone and work phone numbers for a second value.
5. Checks a user's home phone and work phone numbers for a second value.

To access local SOAP APIs for RPCE web services, you must call the `gwcc regen-soap-local` script to regenerate local files. Also, after you add, modify, or delete an RPCE web service, call `regen-soap-local` again.

Refer to the following code example to test this class or to test in Studio:

```

package example
uses gw.api.soap.GWAuthenticationHandler
uses soap.local.entity.UserPhones
uses gw.testharness.TestBase

@SOAPLocalTest
@gw.testharness.ServerTest
class MyTestClassTest extends gw.testharness.TestBase {

    public function testMyServiceAPI() {
        var r : UserPhones
        var outgoingPhones : UserPhones

        // Get a User, in this case the "su" user.
        var userQuery = find (u in User where exists(creds in User.Credential
            where creds.UserName == "su"))

        var userPublicID = userQuery.getAtMostOneRow().PublicID
        print("Public ID of the record is " + userPublicID)

        var myAPI = new soap.local.api.MyServiceAPI()
        myAPI.addHandler(new GWAuthenticationHandler("su", "gw"))

        // Set the data for phone #1.
        outgoingPhones = new UserPhones()
        outgoingPhones.PublicID = userPublicID
        outgoingPhones.HomePhone = "home1"
        outgoingPhones.WorkPhone = "work1"
        myAPI.setUserPhones(outgoingPhones)

        // Check values for phone #1 numbers.
        r = myAPI.getUserPhones(userPublicID)
        print("before HomePhone:" + r.HomePhone)
    }
}

```

```
print("before WorkPhone:" + r.WorkPhone)
TestBase.assertEquals(r.HomePhone, "home1")
TestBase.assertEquals(r.WorkPhone, "work1")

// Set the data phone #2.
outgoingPhones = new UserPhones()
outgoingPhones.PublicID = userPublicID
outgoingPhones.HomePhone = "home2"
outgoingPhones.WorkPhone = "work2"
myAPI.setUserPhones(outgoingPhones)

// Check values for phone #2 numbers.
r = myAPI.getUserPhones( userPublicID )
print("after HomePhone:" + r.HomePhone)
print("after WorkPhone:" + r.WorkPhone)
TestBase.assertEquals(r.HomePhone, "home2")
TestBase.assertEquals(r.WorkPhone, "work2")
}
}
```

In Studio, if you select **Class → Run Test Class**, the following eventually prints in the GUnit console window:

```
19:37:44,725 INFO ***** MyTestClassTest ***** SUITE SETUP OK
Public ID of the record is default_data:1
before HomePhone:home1
before WorkPhone:work1
after HomePhone:home2
after WorkPhone:work2
19:37:47,369 INFO ***** MyTestClassTest ***** SUITE TEARDOWN
```

## Calling Your Published RPCE Web Service from Java

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

Guidewire recommends the Java language to integrate with ClaimCenter from external systems. Guidewire provides a Java class that improves performance for Java web service clients by transparently caching authenticated proxies. Cached proxies avoid unnecessary time and resources to get new proxies every time client code requests one. Developers who use the standard ClaimCenter Java libraries can closely follow the instructions and code snippets in this section for basic web service connectivity and authentication.

To connect to the ClaimCenter web service APIs using Java, follow these high-level steps:

- 1. Get references to API interfaces that you want** – Typical applications include `IClaimAPI` but may also use custom web service interfaces exported in the package `com.guidewire.cc.webservices.api`.  
If you create a custom SOAP API called `MyServiceAPI`, reference the service as `com.guidewire.cc.webservices.api.MyServiceAPI`.

**IMPORTANT** From the perspective of Java code connecting to SOAP APIs through SOAP API generated libraries, the API class is always in the package `com.guidewire.cc.webservices.api.*`. This is true independent of the original package of the Gosu class that implements the web service as defined within Studio.

- 2. Call those interface's methods.** Call API interface methods to query ClaimCenter or to change its data.
- 3. If you are finished, log out.**

If you use Java, get references to API interfaces with the Java-specific `APILocator` class. The `APILocator` class and related classes are special in the SOAP API generated libraries because they are specifically for Java. In contrast, ClaimCenter generates almost all SOAP API classes from WSDL and thus do not vary by SOAP client language.

Use the following steps to use APILocator:

1. Get a proxy. Call the APILocator method `getAuthenticatedProxy`. Pass it the API name, the server URL, the user name, and that user's password. This method obtains and returns a SOAP API proxy object. Do not use the ClaimCenter `ILoginAPI` interface from Java because `APIHandler` handles authentication for you.
2. Use the API interface by calling methods of the interface.
3. Before your application exits, log out by calling `APILocator.releaseAndLogoutAllProxies()`.

It is important to understand that with this approach there is no call with `ILoginAPI` to log into the server. By calling `APILocator.getAuthenticatedProxy(...)`, your code effectively logs into the server to get the proxy.

However, before your program exits, you must log out of the server. You must log out from ClaimCenter by calling `APILocator.releaseAndLogoutAllProxies()` after your code completes. Failure to call the `releaseAndLogoutAllProxies` method results in lingering server sessions and potential memory leaks on the client.

The following example in Java uses APILocator:

```
// Import your web service in the external toolkit namespace. It is always in the following package:  
package com.example;  
import javax.xml.rpc.ServiceException;  
  
import com.guidewire.cc.webservices.api.*;  
import com.guidewire.cc.webservices.entity.*;  
import com.guidewire.util.webservices.APILocator;  
  
public class SoapTest1 {  
  
    public static void main(String args[]) throws ServiceException {  
  
        IUserAPI userAPI = null;  
        User oneUser;  
        String userName = "sys";  
        String url = "http://localhost:8080/cc"; // NOTE: change server or port as appropriate  
  
        // Get API interface proxies here.  
        System.out.println("About to get the proxy -- doesn't call the API yet!");  
        userAPI = (IUserAPI) APILocator.getAuthenticatedProxy(IUserAPI.class, url, "su", "gw");  
  
        try {  
            // use the API proxies...  
  
            // First, get the public ID of a User based on its user name string.  
            String oneUserPublicID = userAPI.findPublicIdByName(userName);  
  
            // Next, use that public ID to get the SOAP version of the entity  
            oneUser = userAPI.getUser(oneUserPublicID);  
  
            if (oneUser == null) {  
                throw new Exception("That user name was not found!!!!");  
            }  
  
            System.out.println("DETAILS OF USER NAME '" + userName + "' ");  
            System.out.println("Public ID = " + oneUser.getPublicID());  
            System.out.println("First Name = " + oneUser.getContact().getFirstName());  
            System.out.println("Last Name = " + oneUser.getContact().getLastName());  
        }  
  
        catch (Exception e2) {  
            System.out.println("EXCEPTION DETECTED!! Details:" + e2.getMessage());  
            e2.printStackTrace();  
        }  
  
        // Before the application finally exits, always release all the proxies!  
        APILocator.releaseAndLogoutAllProxies();  
    }  
}
```

If you use a different server name, port, web application, or if you use secure sockets (SSL/HTTPS), you might pass a slightly different connection URL to the `getAuthenticatedProxy` method.

Each program that calls ClaimCenter must authenticate as a ClaimCenter user. Create some special ClaimCenter users that represent systems rather than actual people. This helps you track the original source of data in your

database. Use different user IDs for different systems to help debug problems. You can track which code module imported which data. Remember to give this user identity sufficient roles and permissions to make the desired API calls. All API calls except for logging in require the SOAPADMIN permission. Some APIs specify additional permissions that the user must have.

ClaimCenter supports plugins that can optionally manage user authentication. For instance, to integrate ClaimCenter user sign-on with a corporate LDAP authentication directory. If you register a user authentication plugin, it handles the user authentication automatically. There is no difference for login or logout code to support authentication plugins. For information about writing a user authentication plugin, see “Deploying User Authentication Plugins” on page 194.

On a related note, you can design custom handing of authentication for certain users. For example, use ClaimCenter internal authentication for the SOAP API special users, and then use LDAP authentication for other users. For more details, see “SOAP API User Permissions and Special-Casing Users” on page 194.

The SOAP protocol defines several approaches to server-client communication. ClaimCenter supports SOAP *conversational mode* and *non-conversational mode*. See “Conversational and Non-Conversational SOAP Modes” on page 124.

## More Java Web Service API Client Examples

The previous section shows basic web service API connectivity and authentication. This section contains examples using the Java language interface that demonstrate basic usage of other ClaimCenter APIs.

This example shows how to use APIs to:

1. Create a new note
2. Find a claim and a user programmatically
3. Add a note to an existing claim.

Use Java code similar to the following:

```
// Log in and get references to IUserAPI and IClaimAPI API interfaces.  
// (See previous sections of this topic.)  
  
// Make new Note object.  
Note note = new Note();  
note.setSubject("A sample note");  
note.setBody("Sample note text");  
note.setTopic(NoteTopicType.TC_general);  
note.setConfidential(new Boolean(false));  
  
try {  
    // Get user from sample data (note : real code must catch exceptions!)  
    User user = userAPI.getUser(userAPI.findPublicIdByName("aappleage"));  
  
    // set the note's author  
    note.setAuthor(user.getPublicID());  
  
    // find claim publicID by claim # (note: real code must handle exceptions)  
    String claimPublicID = claimAPI.findPublicIdByClaimNumber("235-53-365870");  
  
    // Make the method call to add the note, and catch exceptions  
    String notePublicID;  
    notePublicID = claimAPI.addNote(claimPublicID, note);  
}  
  
catch (Exception e) {/* Examine exceptions thrown from ClaimCenter */}  
  
// If connecting from a language other than Java, be sure to  
// log out from the ClaimCenter server if you are done (see previous sections)  
// For the Java language... log out is automatic if you use APILocator.  
// However, BEFORE THE APPLICATION FINALLY EXITS, ALWAYS RELEASE ALL THE PROXIES:  
//     APILocator.releaseAndLogoutAllProxies();
```

If need to refer to entities by reference, see “As a SOAP Client, Refer to an Entity By Reference with a Public ID” on page 121.

## Setting Timeout with Java API Locator

To override the default message timeout when using `APILocator`, you can specify a new timeout in milliseconds.

From your web service client code, use the following code to set the timeout to 12345 milliseconds:

```
com.guidewire.util.webservices.SOAPOutboundHandler.READ_TIMEOUT.set(new Integer(12345));
```

If you do not override the timeout, the `APILocator` class uses the Apache Axis default, which is 600000 milliseconds. That default is the Axis configuration constant `org.apache.axis.Constants.DEFAULT_MESSAGE_TIMEOUT`.

## Calling Your RPCE Web Service from Microsoft .NET WSE 3.0

---

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

---

You can call your web service from Microsoft .NET with WSE 3.0 releases of .NET.

---

**IMPORTANT** This section is about WSE 3.0. For connection from older WSE 2.0 systems, see the next section “Calling Your RPCE Web Service from Microsoft .NET WSE 2.0” on page 114.

---

The syntax of different .NET languages varies widely, so there is no single syntax this documentation can use for .NET examples. This section uses Microsoft Visual Basic examples.

### To create a .NET connection to web services

1. **Decide what interfaces you want** – You need `ILoginAPI` to log in. Choose other API interfaces based on what tasks you require.
2. **Get a reference to interface objects** – This varies by language and SOAP implementation. Guidewire calls this a handle, or proxy, to the API.
3. **Create a username token** – Create a WSS security token for your session. The details of this varies by language and SOAP implementation. See later in this topic for more detailed steps for Visual Basic .NET connections.
4. **Log in.** Authenticate and log in with the `ILoginAPI` interface `WSSLogin` method. Pass it a name and password.
5. **Associate the token with your interface proxies** – To permit each API interface to authenticate properly, associate each interface with your `UsernameToken`.
6. **Call your desired API functions** – Perform various functions to control, query, or add data to ClaimCenter using additional API interfaces. For example call methods on a web service interface you wrote in Gosu and exposed as a web service.
7. **Log out** – Call the `ILoginAPI` interface `WSSLogout` method.

Guidewire strongly recommends you write error checking code. Be careful that your API client code always logs out if your program finishes communicating with ClaimCenter. To log out, call the `WSSLogout` method on the `ILoginAPI` interface.

---

**IMPORTANT** Your code must always log out of ClaimCenter after it is done. Catch all errors and exceptions. Carefully check that your code successfully logged in, it logs out before fully exiting.

---

Because each program that calls ClaimCenter needs to log in to the server, create special ClaimCenter users that represent systems rather than actual people. This helps you track of where data came from. If you use different

user IDs for different code, you can help debug problems because you know which code module imported the data. Remember to give this user identity sufficient roles and permissions to make the desired API calls. All API calls (except for logging in) require the SOAPADMIN permission. Some APIs specify additional required permissions.

ClaimCenter supports plugins that can optionally manage *user authentication*. For details, see “Deploying User Authentication Plugins” on page 194.

Before you write code to call ClaimCenter APIs, set up your .NET development environment to work with the SOAP interfaces. Import the WSDL for the ClaimCenter SOAP service into your .NET development environment. It makes a local representation of the data objects required by each ClaimCenter API.

After setting up authentication and login, most web service API usage is straightforward. Guidewire provides an alternative security approach based on the WS-Security standard, which Microsoft supports in .NET.

### Connecting to Web Services from .NET with WSE 3.0

Guidewire provides Visual Basic .NET sample code within the `ClaimCenter/soap-api/examples` directory. To use the included `DotNetAPISample` sample code:

1. First, confirm that Microsoft Web Services Enhancements 3.0 (WSE 3.0) is on your computer. If not, install it and restart Visual Studio.NET
2. Open the project in Visual Studio.NET
3. Right-click on the name of the `DotNetAPISample` project to open the contextual menu for the project. It displays in bold text on the right side of the window.
4. From the contextual menu, choose **WSE 3.0 Settings....**
5. Check the box next to **Enable this project for Web Services Enhancements**
6. Click **OK**.
7. Confirm your ClaimCenter server is running. Test it in your web browser.
8. Delete `ILoginAPI` and `ISystemToolsAPI` under **Web References** from the right pane called **Solution Explorer**.
9. Right-click on the `DotNetAPISample` project (the bold text) on the right side of the window to open the contextual menu.
10. From the contextual menu, click **Add Web Reference...**
11. Enter the URL to the login interface WSDL. It looks something like:  
`http://localhost:8080/cc/soap/ILoginAPI?wsdl`
12. For the **Web Reference name**, type `ILoginAPI` and then click the **Add Reference** button.
13. Check for a Microsoft WSE Service Proxy bug that sometimes manifests at this step. Visual Studio has a known bug that sometimes manifests to prevent proper creation of the WSE service proxy. If the bug manifests, then only the traditional service generates and not the service with the `WseService` extension.
  - a. Double-click on the web reference for the new API, in this case `ILoginAPI`.
  - b. In the Object Browser pane on the left, view the list of elements of the definition of your new interface you imported.
  - c. Find the service, which is the API name followed by the string `Service`. In this case, search for `ILoginAPIService`.
  - d. Double-click on that service.
  - e. In the source code that appears called `reference.cs`, find the line:  
`public partial class IUserAPIService : System.Web.Services.Protocols.SoapHttpClientProtocol {`

- f. Check if this line already contains the string `Microsoft.Web.Services3.WebServicesClientProtocol`. If it does not, change it to extend the service from `Microsoft.Web.Services3.WebServicesClientProtocol` instead.  
For example, change this line to:  

```
public partial class IUserAPIService : Microsoft.Web.Services3.WebServicesClientProtocol {
```
- g. Save the file.

**IMPORTANT** If you ever update the web reference in Studio, you must repeat this workaround. For example, suppose your server name or port changes. If you return to the Solutions Explorer pane and update the web reference URL for the WSDL, the .NET environment downloads the WSDL again and regenerates the service. This new download removes your changes, so you must repeat this procedure.

14. Right-click on the `DotNetAPISample` project, which is the bold text on the right side of the window. It opens the contextual menu.
15. Click **Add Web Reference...**
16. Enter the URL to another SOAP API interface, such as the system tools interface WSDL:  
`http://localhost:8080/cc/soap/ISystemToolsAPI?wsdl`
17. For the web reference name, type `ISystemToolsAPI`.
18. Click the **Add Reference** button.
19. Check for the WSE Service Proxy bug (see step 13).
20. If you use WSE3 and your code assumes WSE 2.0, you must update `imports` statements in your .NET code to reference version 3 libraries, not version 2 libraries. For example, the included sample code in ClaimCenter requires the following changes at the top of the file to change the number 2 to the number 3.  
Find each line such as:  

```
Imports Microsoft.Web.Services2
```

Change them to:  

```
Imports Microsoft.Web.Services3
```

In the example, there are three such lines.
21. If you use WSE3 and your code assumes WSE 2.0, you must ensure that the name of the service does not have the `Wse` suffix. For example, if you use the included example, change the line:  

```
Dim loginAPI As New ILoginAPI.ILoginAPIServiceWse
```

To the following:  

```
Dim loginAPI As New ILoginAPI.ILoginAPIService
```
22. Set up a Username Token. Your web service client passes the `UsernameToken` as an authentication credential to the server with every SOAP call. You only need to do this once. After that, use the authentication credential, which is typically more secure, instead of a plaintext password. For example:  

```
Dim credentials As New UsernameToken("su", "gw", PasswordOption.SendHashed)
```
23. Write your code to connect with the .NET authentication using the `ILoginAPI` interface's `WSSLogin` method. The `login` method `WSSLogin` is part of the Guidewire implementation of the `UsernameToken` profile of the Web Services Security specification. Support for this specification is part of the WSE package from Microsoft. The next step might look like:  

```
LoginAPI.WSSLogin("su", "gw")
```

**24.** In Studio.NET, set up what Microsoft calls a *security policy*.

**IMPORTANT** What Microsoft .NET calls a *security policy* is unrelated to a policy in the context of the Guidewire data model. Be careful not to confuse these two meanings of the term. A Microsoft .NET security policy represents an abstraction of the security you use to connect to a service. In the case of Guidewire applications, the security policy is simple, and uses the name/password authentication embedded in the UsernameToken object.

- a. Go to the Solution Explorer pane and right-click on the project name.
- a. From the contextual menu, choose **WSE 3.0 Settings...**
- b. Select the **Policy** tab.
- c. Click **Add...**
- d. In the dialog that says **Add or Modify Policy Friendly Name**, enter the name **ClientPolicy**.
- e. Click **OK**.
- f. In the **WSE Security Settings Wizard** that appears, for the top question in the wizard, click **Secure a service application**.
- g. For the authentication method, click **Username**.
- h. Click **Next**.
- i. In the next screen, leave the checkbox **Specify Username Token in Code** checked.
- j. Click **Next**.
- k. In the next screen, check **Enable WS-Security 1.1 Extensions**.
- l. Click the protection order **None**
- m. Click **Finish**.

**25.** In your code, set your client credential with .NET code using the security policy name that you defined earlier. Do this for every API interface that you plan to use, including **ILoginAPI**. You use **ILoginAPI** later for the log out process. The following example demonstrates setting the .NET security policy:

```
' set client credential for ILoginAPI...
loginAPI.SetClientCredential(userToken)
loginAPI.SetPolicy("ClientPolicy")

' use additional API interfaces to do your actual work...
Dim systoolsAPI As New ISystoolsAPI.ISystoolsAPIService
systoolsAPI.SetClientCredential(userToken)
systoolsAPI.SetPolicy("ClientPolicy")
```

**26.** Call any desired API methods. You do not need to worry SOAP protocol details. For example, to get the ClaimCenter server's current run level, use code such as:

```
Dim runlevel As ISystoolsAPI.SystemRunlevel
runlevel = systoolsAPI.getRunlevel()
```

**27.** Log out after you are done:

```
loginAPI.WSSLogout("su")
```

Be aware that an authentication credential is not as secure as SSL-encrypted authentication or transport-level strong encryption. For additional security, make your initial connection request the username token over a secure

sockets layer (SSL) connection using an `https` URL instead of an `http` URL. For more information about configuring secure sockets layer and its performance implications, see the *ClaimCenter Configuration Guide*.

**Note:** After the initial connection, you can choose to login and connect to the web service APIs over regular (non-SSL) connections. Although the data transport is not encrypted in either direction over a non-SSL connection, your password is safe from interception. Your password is safe because the only time the password was sent to the server, the password was hashed and passed over a secure socket layer. For maximum security, connect to the ClaimCenter server over full SSL connections.

Note the following things about using web service objects from .NET:

- The package hierarchy appears such that objects appear in their interface's namespace. For example, the Javadoc may show an entity with the package symbol as `com.guidewire.cc.webservices.entity.Note`. However, if an API interface called `IMyAPI` uses this object as a return result or parameter, it appears as `IMyAPI.Note` from within .NET.
- If another API interface uses that object, it appears as a separate object within that API proxy's scope. Be careful to use the variant of the object in the correct namespace for the API that you call.
- The API Reference Javadoc documentation was produced from tools that process the Java version of the ClaimCenter APIs. Consequently, some information is Java-centric and it appears different if using .NET.

Refer to the included `DotNetAPISample` sample for a working Visual Basic .NET example.

---

**IMPORTANT** The included `DotNetAPISample` requires minor modifications to work with WSE 3.0. Carefully read the procedure earlier in the section and note the items that mention differences between WSE 2.0 and WSE 3.0.

---

#### Multi-Dimensional Arrays and .NET SOAP Access

Microsoft .NET does not properly import WSDL describing multi-dimensional arrays. As a consequence, there is an additional step necessary to use multi-dimensional arrays.

For example, suppose a SOAP API returns a two-dimensional array of `String` objects, an object of type `String[][]`. As Microsoft .NET imports the WSDL and generates the method, it incorrectly converts the return type to a single-dimensional array, a `String[]` instead of a `String[][]`.

To work around this issue, first import the WSDL into Visual Studio. Then, update the generated method signature to access the proper multi-dimensional object, such as `String[][]` instead of `String[]`.

## Calling Your RPCE Web Service from Microsoft .NET WSE 2.0

---

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

---

You can call your web service from Microsoft .NET with older WSE 2.0 releases of .NET, rather than WSE 3.0.

---

**IMPORTANT** This section is about WSE 2.0. For WSE 3.0, see the previous section “Calling Your RPCE Web Service from Microsoft .NET WSE 3.0” on page 110.

---

The syntax of different .NET languages varies widely, so there is no single syntax this documentation can use for .NET examples. This section uses Microsoft Visual Basic examples.

Before writing .NET code to call SOAP APIs, set up your .NET development environment. You must import the WSDL for the ClaimCenter SOAP service into your .NET development environment. It makes a local representation of the data objects needed by each ClaimCenter API.

After that, most API usage is straightforward. However, the authentication system for initial connection from Java does not easily work with .NET. Guidewire provides an alternative security approach based on the WS-Security (WSS) standard, which Microsoft supports in .NET.

### Connecting to Web Services from .NET with WSE 2.0

Guidewire provides Visual Basic .NET sample code within the `ClaimCenter/soap-api/examples` directory. The following are detailed instructions for using the included `DotNetAPISample` sample code:

1. First, make sure that Microsoft Web Services Enhancements (WSE) 2.0 is installed. If not, install it and restart Visual Studio.NET
2. Open the project in Visual Studio.NET
3. Right click on the name of the `DotNetAPISample` project, shown in bold text, on the right side of the window to open the contextual menu for the project.
4. From the contextual menu, choose **WSE 2.0 Settings...**, then check the box next to **Enable this project for Web Services Enhancements** then click **OK**.
5. Ensure your ClaimCenter server is up and running. Test it in your web browser.
6. Delete `ILoginAPI` and `ISystemToolsAPI` under **Web References** from the right side.
7. Right click on the `DotNetAPISample` project (the bold text) on the right side of the window to open the contextual menu.
8. From the contextual menu, choose **Add Web Reference...**
9. Enter the URL to the Login interface WSDL, in the form:  
`http://localhost:8080/cc/soap/ILoginAPI?wsdl`
10. Enter “`ILoginAPI`” as the web reference name and then click the **Add Reference** button.
11. Check for the WSE Service Proxy bug that sometimes manifests at this step. Visual Studio has a known bug that sometimes manifests to prevent proper creation of the WSE service proxy. In this case, only the traditional service generates, and not the service with the `WseService` extension.
  - a. Double click on the web reference for the new API, in this case `ILoginAPI`.
  - b. Look in the Object Browser pane to view the list of elements of the definition of the new interface.
  - c. Find the service, which is the API name followed by the string `Service`. In this case, search for `ILoginAPIService`.
  - d. Double-click on that service.
  - e. In the source code that appears called `reference.cs`, look at the line that looks like:  
`public partial class IUserAPIService : System.Web.Services.Protocols.SoapTtpClientProtocol {`
  - f. If this line does not already contain the string `Microsoft.Web.Services2.WebServicesClientProtocol`, change this line to extend the service from `Microsoft.Web.Services2.WebServicesClientProtocol` instead. For example, change this line to:  
`public partial class IUserAPIService : Microsoft.Web.Services2.WebServicesClientProtocol {`
  - g. Save the file.
  - h. If you ever update the web reference, you must repeat this workaround. For example, suppose your server name or port changes and you go back to the Solutions Explorer pane and you try to update the web reference URL for the WSDL. The .NET environment downloads the WSDL again and regenerates the service, thus removing your changes. In such cases, you must repeat this procedure.
12. Right-click on the `DotNetAPISample` project (the bold text) on the right side of the window to open the contextual menu, and choose **Add Web Reference...**

13. Enter the URL to another SOAP API interface, such as the system tools interface WSDL:

```
http://localhost:8080/cc/soap/ISystemToolsAPI?wsdl
```

14. Enter “ISystemToolsAPI” as the web reference name and then click the Add Reference button.

15. Check for the WSE Service Proxy bug (see step 13).

16. Write your code to connect with the .NET authentication using the ILoginAPI interface’s WSSLogin method.

The following example code connects to the server and authenticates your SOAP client. The first task to is login to the server. The login method WSSLogin is part of the Guidewire implementation of the UsernameToken profile of the Web Services Security specification. Support for this specification is provided directly by the WSE package from Microsoft:

```
Dim loginAPI As New ILoginAPI.ILoginAPIServiceWse

' tell the proxy where to connect. Adjust accordingly.
loginAPI.Url = "http://localhost:8080/cc/soap/ILoginAPI"
loginAPI.WSSLogin("su", "gw")

' The server knows about the session now. A SOAP fault happens
' given if the username/password combination was wrong for some
' reason
```

Next, this example sets up a UsernameToken to pass as an authentication credential to the server with every SOAP call. You only need to do this once. After initial connection, you can use the (typically more secure) authentication credential instead of a plaintext password.

```
Dim credentials As New UsernameToken("su", "gw", PasswordOption.SendHashed)

' Now set up the proxy to the sysadmin api.
Dim sysadminAPI As New ISystemToolsAPI.ISystemToolsAPIServiceWse
sysadminAPI.Url = "http://localhost:8080/cc/soap/ISystemToolsAPI"

' Now attach credentials to the SOAP context. The credentials
' are now inside of a WSS SOAP header
Dim context As SoapContext = sysadminAPI.RequestSoapContext
context.Security.Tokens.Add(credentials)
```

Be aware that an authentication credential is not as secure as SSL-encrypted authentication or transport-level strong encryption. For additional security, make your initial connection request the username token over a secure sockets layer (SSL) connection using an https URL instead of an http URL. For more information about configuring secure sockets layer and its performance implications, see the *ClaimCenter Configuration Guide*.

**Note:** After the initial connection, you can choose to login and connect to the web service APIs over regular (non-SSL) connections. Although the data transport is not encrypted in either direction over a non-SSL connection, your password is safe from interception. Your password is safe because the only time the password was sent to the server, the password was hashed and passed over a secure socket layer. For maximum security, connect to the ClaimCenter server over full SSL connections.

Now you can make any calls to the ClaimCenter APIs without worrying about details of the SOAP protocol. For example, to get the ClaimCenter server’s current run level, you would write:

```
' use other ClaimCenter API interfaces
Dim runlevel As ISystemToolsAPI.SystemRunlevel
runlevel = sysadminAPI.getRunlevel()

' Then logout
loginAPI.WSSLogout("su")
```

Note the following things about using web service objects from .NET:

- The package hierarchy appears such that objects appear in their interface’s namespace. For example, the Javadoc may show an entity with the package symbol as com.guidewire.cc.webservices.entity.Note. However, if an API interface called IMyAPI uses this object as a return result or parameter, it appears as IMyAPI.Note from within .NET.
- If another API interface uses that object, it appears as a separate object within that API proxy’s scope. Be careful to use the variant of the object in the correct namespace for the API that you call.

- The API Reference Javadoc documentation was produced from tools that process the Java version of the ClaimCenter APIs. Consequently, some information is Java-centric and it appears different if using .NET.

Refer to the included `DotNetAPISample` sample for a working Visual Basic .NET example.

#### Multi-Dimensional Arrays and .NET SOAP Access

Microsoft .NET does not properly import WSDLs that describe multi-dimensional arrays. As a consequence, you must perform an additional to use multi-dimensional arrays.

For example, suppose a SOAP API returns a two-dimensional array of `String` objects, an object of type `String[][]`. As Microsoft .NET imports the WSDL and generates the method signature, it incorrectly converts the return type to a single-dimensional array, a `String[]` instead of a `String[][]`.

To work around this issue, first import the WSDL into Visual Studio. Then, update the generated method signature to access the proper multi-dimensional object, such as `String[][]` instead of `String[]`.

## Calling Published RPCE Web Services From Other Languages

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

This topic discusses accessing Guidewire-hosted RPCE web services from alternative languages. Previous sections discuss connecting to web services from Java 1.5 (using the included Java-specific `APILocator` utility) and from Microsoft .NET (using WSS security and .NET example code). If you must connect from another system, such as Java 1.4 or other languages entirely, you must use a different procedure for authentication.

There are two major differences connecting from Java 1.4 or other non-.NET languages or other tools:

- The built-in generated Java libraries for connecting to Guidewire libraries require Java version 1.5. You must generate your Java classes on your own from the WSDL files using the Axis utility for Java. The WSDL files are generated at the path:  
`ClaimCenter/soap-api/wsdl/api/`
- The SOAP API utility class `APILocator` is the recommended way to log on to web services from remote Java code. It requires Java version 1.5. Although you can use Java 1.4 with Guidewire web services, you cannot use `APILocator`. This class works with Java 1.5 to easily connect to web services, pool proxies for higher performance, and reduces memory issues from uncaught exceptions that skip logout.

Because of these differences, the login procedure to authenticate with web services is different from other versions of Java (such as Java 1.4) or other non-.NET languages or systems. Most importantly, you must manually set a header in the HTTP transaction in whatever way that is done in that language or the set of tools that implement SOAP.

This topic discusses Java but most of it applies for languages that are not Java 1.5 and not Microsoft .NET languages. The important thing to understand is how the published web service expects authentication information SOAP transaction headers.

The exact header to use varies depending on whether you want to use conversational or non-conversational SOAP modes (see “Conversational and Non-Conversational SOAP Modes” on page 124). Conversational mode is the typical connection method for connecting to ClaimCenter web services.

---

**IMPORTANT** The most important thing to know about connecting to web services from other languages is that you must add SOAP transaction headers. The header names and values vary depending on conversational or non-conversational SOAP mode. The exact mechanism for adding SOAP headers to a SOAP transaction varies greatly by language. The examples in this section use Java 1.4, but you must consult your documentation for your language and tools for the syntax of how to add SOAP headers.

---

## Calling SOAP in Conversational Mode from Other Languages

To authenticate with conversational SOAP mode, first call the `ILoginAPI` web service method `login` (however that is done in your language of choice) to generate a session ID. Next, set the `gw_auth_prop` SOAP header in the transaction to include the session ID (however that is done in your language of choice).

---

**IMPORTANT** The syntax of setting SOAP headers varies by language. Check your documentation for your language and tools for how to do this.

---

For example, the following Java 1.4 code connects to the server and executes a method on the `ISystemToolsAPI` web services interface. It assumes you used Apache Axis to generate Java 1.4 libraries, which allow you to use generated stubs using the `org.apache.axis.client.Stub` class.

The stub class lets you set a SOAP header for the Guidewire authentication to contain the session ID. You obtain the session ID by calling the `ILoginAPI` interface method `login`. Pass it the user name and password, and it returns the session ID.

For example:

```
String sessionId = loginAPI.login(userName, pw);
```

Next, use this session ID and set it in the SOAP header `gw_auth_prop`:

```
systoolsStub.setHeader("http://www.guidewire.com/soap", "gw_auth_prop", sessionId);
```

The following full example in Java 1.4 shows how you would do this using the Axis-generated libraries using Java 1.4:

```
// Set up constants.
String loginUrlString = "http://localhost:8080/cc/soap/ILoginAPI?wsdl";
String systoolsUrlString = "http://localhost:8080/cc/soap/ISystemToolsAPI?wsdl";
String userName = "su";
String pw = "gw";

// Locate ILoginAPI and call login method with name and password.
URL loginURL = new URL(loginUrlString);
loginAPI = loginLocator.getILoginAPI(loginURL);
String sessionId = loginAPI.login(userName, pw);
System.out.println("Successful login to web service (url: " + loginUrlString + ")");

// Locate ISystemToolsAPI.
URL systoolsURL = new URL(systoolsUrlString);
ISystemToolsAPIServiceLocator systoolsLocator = new ISystemToolsAPIServiceLocator();
systoolsAPI = systoolsLocator.getISystemToolsAPI(systoolsURL);

// Set ISystemToolsAPI and set the special authentication header with the session ID.
Stub systoolsStub = (Stub) systoolsAPI;
systoolsStub.setHeader("http://www.guidewire.com/soap", "gw_auth_prop", sessionId);

// Use the API method as desired.
runlevel = systoolsAPI.getRunLevel();

// Do other actions here.
// ...
```

```
// Log out.  
loginAPI.logout(sessionId);
```

This section just discusses authentication differences from other systems. For more information about using web services, see the section “Calling Your Published RPCE Web Service from Java” on page 107. However, ignore the APILocator discussion since that applies only to Java 1.5.

## Calling SOAP in Non-conversational Mode from Other Languages

To authenticate with non-conversational SOAP mode, first manually set the header in the HTTP transaction for the user name and password in each SOAP request. Use the SOAP headers `gw_auth_user_prop` for the name and `gw_auth_password_prop` for the password. The exact syntax of setting SOAP headers varies based on your language and tools of choice, check your tool’s documentation for details. Unlike conversational mode, you must not call the `ILoginAPI` web service method `login` for non-conversational mode before calling your desired API calls on other SOAP interfaces.

**IMPORTANT** The syntax of setting SOAP headers varies by language. Check your documentation for your language and tools for how to do this.

For example:

```
systoolsStub.setHeader("http://www.guidewire.com/soap", "gw_auth_user_prop", userName);  
systoolsStub.setHeader("http://www.guidewire.com/soap", "gw_auth_password_prop", pw);
```

The following full example in Java 1.4 shows how you would do this using the Axis-generated libraries using Java 1.4:

```
// Set up constants.  
String systoolsUrlString = "http://localhost:8080/cc/soap/ISystemToolsAPI?wsdl";  
String userName = "su";  
String pw = "gw";  
  
// Locate ISystemToolsAPI.  
URL systoolsURL = new URL(systoolsUrlString);  
ISystemToolsAPIServiceLocator systoolsLocator = new ISystemToolsAPIServiceLocator();  
systoolsAPI = systoolsLocator.getISystemToolsAPI(systoolsURL);  
  
// Set ISystemToolsAPI authentication headers for NON-conversational mode.  
Stub systoolsStub = (Stub) systoolsAPI;  
systoolsStub.setHeader("http://www.guidewire.com/soap", "gw_auth_user_prop", userName);  
systoolsStub.setHeader("http://www.guidewire.com/soap", "gw_auth_password_prop", pw);  
  
// Use the API method as desired.  
runlevel = systoolsAPI.getRunLevel();  
  
// Do other actions here.  
// ...
```

## How SOAP Headers Appear in the SOAP Envelope

The following example shows how your SOAP header might appear within the SOAP envelope sent to the published web service:

```
<?xml version="1.0" encoding="UTF-8"?>  
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"  
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
    <soapenv:Header>  
        <ns1:gw_auth_user_prop soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"  
            soapenv:mustUnderstand="0" xmlns:ns1="http://www.guidewire.com/soap">user</ns1:gw_auth_user_prop>  
        <ns2:gw_auth_password_prop soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"  
            soapenv:mustUnderstand="0" xmlns:ns2="http://www.guidewire.com/soap">password  
        </ns2:gw_auth_password_prop>  
    </soapenv:Header>  
    <soapenv:Body>  
        ...  
    </soapenv:Body>  
</soapenv:Envelope>  
...
```

## Typecodes and Web Services in RPCE Web Services

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

A typelist is an object property with a discrete set of possible values for the property. If you look for typelist values, you can refer either to the API Reference Javadoc or the *ClaimCenter Data Dictionary*. You find the values defined as enumeration constants defined within a class whose name describes their contents. For simple typelist value names, the typelist value's name is the official name defined in the data model, prefixed by "TC\_".

For example, a claim object's loss type (`LossType`) property contains a value that indicates whether the claim is an auto claim, a worker's compensation claim, or a few other choices. For example, the loss type on a claim object is represented within the typecode enumeration class `LossType`. In Java syntax, describe the loss types from your web service client code as `LossType.TC_AUTO` for auto claims and `LossType.TC_WC` for worker's compensation claims.

If the typecode contains characters other than ASCII letters and numbers, Gosu exposes the typecode property name differently across SOAP. For example, if the typecode contains space characters, symbols, or any non-English language characters, the typecode name changes. This change maximizes compatibility with a wide variety of programming languages for web service client code. The modified encoding surrounds the Unicode value of the character with \$ characters. For example, suppose a typecode contains a hyphen character, such as `married-joint`. Across the SOAP interface, it becomes `TC_married$45$joint` because 45 is the numeric value of the dash character.

**Note:** Typelist values are static properties, so do not use "get" or "set" methods to access their values. Simply use code such as `LossType.TC_WC`, not `LossType.getTC_WC(...)`.

To convert an enumerated typelist value into a string, use the `toString` method of the typelist value instance to which you have a reference. Similarly, to create a typelist value from a typecode string, pass the `fromString` or `fromValue` method a string containing full typecode values with the TC\_ prefix. Be aware that the `fromString` or `fromValue` methods perform the same function despite their different names.

For example, call `LossType.TC_AUTO.toString()` to return a text representation. Pass the value "TC\_AUTO" to `fromString` to get the typelist value.

At development time, you can determine the available set of typelist codes from the *Data Dictionary*. This documentation includes typelist codes that you create after you run the dictionary regeneration scripts. In addition, code completion in Guidewire Studio provides easy access in Gosu code to the list of typecodes for a typelist. Alternatively, the Java integrated development environments IntelliJ and Eclipse provide code completion for easy access in Java code to the list of typecodes for a typelist. At run time, use the `ITypelistToolsAPI` function `getTypelistValues` to get the list of typelist codes for a typelist, if for example you need to verify that a code is valid.

### See also

- "Data Dictionary Documentation" on page 22
- "Mapping Typecodes to External System Codes" on page 143

## Public IDs and RPCE Web Services

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

ClaimCenter creates its own unique ID for every entity in the system after it fully loads in the ClaimCenter system. However, this internal ID cannot be known to an external system while the external system prepares its

data. Consequently, if you get or set ClaimCenter information, use unique public ID values to identify an entity from external systems connecting to a Guidewire application.

For important information about working with (and generating) public IDs in integration code, see “Creating Your Own Public IDs” on page 27

## Using Public IDs With API Functions

If you simply want to get an entity based on its public ID, get a reference to the object using the web service APIs using `get...` functions in the interface for that type. For instance, get a `User` data object with the `IUserAPI.getUser` method. It takes a public ID and returns a `User` object.

Sometimes an API requires you to refer to an object by public ID but you may not know that value. For example, you may want add a note to a claim and you have the claim number but not the claim’s public ID. For this type of situation, various API functions can return a public ID based on a claim number, a user name, or a group name. If the built-in APIs do not provide the functionality you want, write your own web service that does.

Refer to the following table for a list of built-in public ID search methods.

To find a...	By...	Use...
Claim	claim number	<code>IClaimAPI.findPublicIdByClaimNumber</code>
User	user name	<code>IUserAPI.findPublicIdByName</code>
Group	group name	<code>IGroupAPI.findPublicIdByName</code>

Write many custom web services as needed for your integration points. In almost all cases, link to an object by its public ID except in cases in which there is another high-level ID that already exists. For example, claim numbers.

**IMPORTANT** In general, use public IDs to refer to entities unless there is a high-level ID that already exists such as claim numbers that make more sense for that integration point.

## As a SOAP Client, Refer to an Entity By Reference with a Public ID

Sometimes you might want to add a new data object that references a previously added object. For example, if you add financial information, you might provide a list of checks (a data object of class `CheckSet`) whose checks reference payees mailing addresses. In this case, typically you would not want to add new addresses to the ClaimCenter address table. The payee’s address already corresponds to an address already existing in the system.

In this case, simply reference the object that already exists. However, if you fill in all the properties in the `Address` object, ClaimCenter creates a new address record in the database.

Fortunately, there is a way to reference pre-existing objects. First, set the public ID for the object. Next, tell ClaimCenter that want to refer to it by reference (`ByRef`). Call the object’s `setRefType` method and pass it the reference type enumeration value `RefTypeEnum.GW_ByRef`.

If an object’s reference type (`refType`) is `GW_ByRef` and its public ID matches an entity in the database or in the current request, the server uses the existing object. Do not add additional properties other than public ID and the reference type because they not copy to the database.

Consider the `Address` data object as an example. If you add a check, refer to an existing address in the ClaimCenter database:

```
Address addr = new Address();
addr.setPublicID("ABC:SYSTEM01:2224"); // existing public ID!
addr.setRefType(RefTypeEnum.GW_ByRef);    // set the ref type!

// leave all other properties blank -- they are ignored anyway
```

Or, associate the check with a new address to add to the ClaimCenter database:

```
Address addr = new Address();
addr.setAddressLine1("100 Main St.");
// Set other address properties here.

// Then, either
//   (1) leave Public ID empty to let the server create it
// or (2) set an explicit new Public ID for the new record.
addr.setPublicID("ABC:SYSTEM01:99"); // NEW record & public ID!

// Then, assign this object as appropriate.
myContainerObject.setAddress(addr);
```

The two choices for reference type are `GW_ByRef` and `GW_NotByRef`, and the default value is `GW_NotByRef`. If for some reason, you need to get the reference type of a data object, call its `getRefType` method.

## Identification Exceptions, Particularly During Entity Add

If you try to refer to an object by reference but the server cannot find the object, the server throws a `BadIdentifierException` exception. This exception extends from the `DataConversionException` exception.

In contrast, if you refer to an object not by reference and you specify a non-empty public ID in the object, you must ensure the public ID is unique. Take care not to accidentally add the object twice to your ClaimCenter database. If you try to add the object twice, the request fails with a SOAP fault (an exception) due to the duplicate public ID. For more information about error handling, see “SOAP Faults (Exceptions) in RPCE Web Services” on page 126.

## Endpoint URLs and Generated WSDL in RPCE Web Services

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

Web services enable remote procedure calls from one system to another without assuming that the technology on both ends is the same. ClaimCenter exposes APIs using the platform-neutral and language-neutral web services *SOAP* protocol. If a programming language has a SOAP implementation, connecting to web service APIs is easy. The SOAP implementation hides the complexity underlying the SOAP protocol.

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

Guidewire also provides a Java language binding for ClaimCenter APIs. Simply compile against the ClaimCenter generated libraries and use the included interfaces and classes that describe each interface.

However, if you use languages other than Java or you implement a more complex system, you may need to know more about how SOAP works.

This topic provides only a rough overview of web service standards and is not a full specification of web services standards. ClaimCenter uses the following web service standards and technologies:

- **XML (Extensible Markup Language)** – A standard that describes complex structured data in a text-based format with strict syntax for easy data interchange. For more information, refer to the World Wide Web Consortium (W3C) web page <http://www.w3.org/XML>.
- **SOAP** – The web services request-and-response protocol based on XML, which is often implemented over the HTTP protocol. SOAP supports remote APIs in a platform-neutral and language-neutral way. For more information, refer to the World Wide Web Consortium (W3C) web page <http://www.w3.org/TR/soap>.
- **WSDL (Web Services Description Language)** – The web services API description language. The language describes API interfaces for web services, including interface names, method names, function arguments, relevant entities (classes), and return values of functions. Any system can use a web service if it knows how

to follow the definition provided in the WSDL file for each API. This API description language relates closely to the XML standard and the SOAP protocol. For more information, refer to the World Wide Web Consortium (W3C) web page <http://www.w3.org/TR/wsdl>.

- **Axis (Apache eXtensible Interaction System)** – A free implementation of the SOAP protocol, which ClaimCenter uses behind the scenes to generate WSDL for ClaimCenter APIs. Also, Axis generates Java and Gosu language bindings (interface libraries) for programs that connect to ClaimCenter SOAP APIs. For more information, refer to the Apache Axis web page <http://axis.apache.org/>.

ClaimCenter exposes APIs as web services using WSDL and SOAP so that code using any language or operating system can use them. The Axis tools included in ClaimCenter automatically regenerate new WSDL and Java bindings each time you run regenerate SOAP API library files (see “Regenerating Integration Libraries” on page 22). These Java bindings are not the native internal implementation of the APIs. ClaimCenter generates these bindings from the Web Service Description Language code it generates for the services.

The `APILocator` class is special in the SOAP API libraries because it is not a class generated from the data model. The `APILocator` class makes Java development easier and makes SOAP API proxy management safe from some memory leaks errors by pooling server connections and managing disposal of proxies.

## Dynamic WSDL and SOAP API Library WSDL

If your code remotely calls ClaimCenter web service APIs, ClaimCenter is the SOAP server for the transaction and your code is the SOAP sender.

With the server running, you can get the Web Services Description Language (WSDL) for an interface by requesting it as dynamically generated WSDL URL:

`http://appserver:port/appname/soap/interfaceName?wsdl`

For example, if your ClaimCenter server is `ccserver`, get `IClaimAPI` WSDL using:

`http://ccserver:8080/cc/soap/IClaimAPI?wsdl`

The exact URL might vary. For example, you might use a different server name, application name, port number, or use secure sockets layer (HTTPS). See the *ClaimCenter System Administration Guide* for more details about using SSL with ClaimCenter.

Alternatively, use the WSDL in the following directory (generated by the `gwcc regen-soap-api` command):

`ClaimCenter/soap-api/wsdl/api/`

However, that WSDL only gets updates after you regenerate the SOAP API files using the `regen-soap-api` target

To view the documentation for the web services, refer to:

`ClaimCenter/soap-api/doc/api/`

For more information about documentation, see “Integration Documentation Overview” on page 20.

To use WSDL files with Microsoft .NET, see “Calling Your RPCE Web Service from Microsoft .NET WSE 3.0” on page 110. Use dynamic WSDL for the latest WSDL without having to regenerate the SOAP API library files.

## Apache Axis Customization

In most cases, you do not need to customize Apache Axis, the SOAP implementation that ClaimCenter uses. ClaimCenter uses it to generate Java interfaces from SOAP/WSDL services. These translation processes typically happen inside ClaimCenter scripts, such as the `regen-soap-api` script that regenerates the SOAP API library.

The version of Axis in the ClaimCenter `gw-axis.jar` file is slightly different from the Axis version 1.2.1 that you can download from the main Apache site. Guidewire customizes Axis to customize the generated Java code slightly and also fixes a known bug in Axis with respect to serializing simple types.

The generated Java code in SOAP API libraries is slightly different. All of these changes relate to the Javadoc that Axis produces. The original version of the Axis JavaWriter classes produce suboptimal Javadoc documentation and places the Javadoc documentation on incorrect classes. These changes are not to the Axis library directly. Instead, they are standard extensions to Java. Specifically, Guidewire overrides the generator factory in the wsdl2java tool.

The second change to the Axis library is a fix for a known bug in Axis. Learn more about the bug at the following web page:

<http://issues.apache.org/jira/browse/AXIS-2077>

The patch ClaimCenter applies is similar to the patch mentioned on that page.

There are a few custom Axis parameters that can be set in `server-config.wsdd`. For details about these parameters, refer to the Apache Axis documentation at the web page:

<http://ws.apache.org/axis/java/user-guide.html>

## Web Services Using ClaimCenter Clusters

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

If you write integration code that connects to a ClaimCenter cluster, be aware of important considerations about choosing which server to connect.

### Optimizing for Load Balancing

The SOAP protocol does not natively support session affinity, so some load-balancers are not able to reliably direct all calls to the same server every time. There are two similar but different concepts that relate to load balancing:

- **Conversational versus non-conversational mode** – This is primarily an issue of reducing re-authentication to a server. This would allow you to load balance with good performance without requiring session (cookie) support. For more information, see “Conversational and Non-Conversational SOAP Modes” on page 124.
- **Cookie support for certain types of load-balancing routers** – To ensure that a load balancing router that supports cookie tracking, ClaimCenter supports session affinity using cookies. For more information, see “Statefullness of Connection Using Cookies” on page 125.

### Conversational and Non-Conversational SOAP Modes

The SOAP protocol defines several approaches to server-client communication. ClaimCenter supports SOAP *conversational mode* and *non-conversational mode*. If you use *conversational mode* with SOAP APIs, you connect directly to a specific server rather than letting a load-balancer direct the call to any server in the cluster.

Improperly load-balanced API calls in conversational mode might authenticate to one server but following requests go to another server in the cluster. If this happens, the original session ID is invalid.

There are several approaches to avoid this issue with conversational mode:

- Direct all calls directly to a specific server by name (`myserver1`, `myserver2`) or by IP number.
- Use a modern load balancer that supports *IP stickiness*. IP stickiness means you can configure it to consistently direct all requests from one incoming IP address to one specific server.
- Stop using conversational mode and instead switch to non-conversational mode for SOAP API access. Perhaps switch only for certain types of communications. Remember to reconfigure your load balancer accordingly.

It is common to implement a user authentication plugin that controls user authentication using an external service such as LDAP or other single-sign on systems. In non-conversational mode, ClaimCenter calls the user authenti-

cation plugin to authenticate with every SOAP request. ClaimCenter has little control over the performance and resources required by that code, so performance can be affected. To learn about the authentication plugin, see “Authentication Integration” on page 187.

In theory, conversational mode can improve performance, particularly for large numbers of small API requests.

In real-world implementations, customers typically choose to write authentication plugins that cache their results from their authentication provider for at least a few minutes. Because of this, the performance advantage of SOAP conversational mode is low for some customers. Guidewire encourages real-world testing to determine the ideal configuration for your authentication plugin design (especially caching) and the frequency of API calls. On a related note, it is typically better to design customer SOAP APIs for appropriately-targeted data updates. (For example, one API method for each logical transaction, not one API method for every property change.)

If you use the `APILocator` method `getAuthenticatedProxy` as documented, your API client uses conversational mode. If you use ClaimCenter clusters, you must configure your load balancers to ensure session affinity. Session affinity means that each request from a client must go to the same server.

To use non-conversational mode, use `APILocator.getUnauthenticatedProxy(...)` instead of `APILocator.getAuthenticatedProxy(...)`. Although this is not generally recommended, this allows you to use ClaimCenter clusters without requiring session affinity. Each time the client code connects over SOAP, it authenticates with the server for that API call only. Additional API calls reauthenticate with the server.

To learn more about conversational mode of SOAP interactions, refer to the World Wide Web Consortium site:

<http://www.w3.org/TR/xmlp-scenarios>

## Statefullness of Connection Using Cookies

By default, the Guidewire web services implementation does not ask the application server to generate and return session cookies. However, ClaimCenter supports cookie-based load-balancing options for web services. The web services layer can hold session state by generating a cookie. This helps load balance consecutive SOAP calls done in the same conversation. This feature simplifies things like portal integration. If your load balancing router supports this feature, further requests by the same user (assuming they reused the same cookie) are redirected to the same node in the cluster.

You can enable stateful conversational SOAP call load-balancing in several ways:

- For simple HTTP requests or from Gosu web service APIs, append the text “?stateful” (with no quotes) to the URL. You can use this approach for ClaimCenter to ContactManager integration to load balance the connection to ContactManager.
- From web service client code programmatically through Axis, add the following method given a reference to the Guidewire web service interface:  
`((org.apache.axis.client.Service) webServiceReference).setMaintainSession( true );`

## Built-in Web Services With Special Cluster Behaviors

There is only one batch server in any cluster. Some API requests only work with the batch server because they make use of services that only run on the batch server. The batch-server-only API requests include:

- The messaging services such as `IMessagingToolsAPI` run only on the batch server. In other words, all `IMessagingToolsAPI` interface methods require that you make the SOAP request only to the batch server.
- Background process actions run only on the batch server. You must call batch process APIs only to the batch server. These APIs start background processes, check on background processes, or terminate background processes. For more about background processes, see “Batch Processes and Work Queues” on page 123 in the *System Administration Guide*. You can start background processes using the built-in scheduler or using the SOAP API `IMaintenanceToolsAPI.startBatchProcess()`.

Background processes include activity escalation, claim exceptions, financials escalation (for sending scheduled checks), and generating database statistics.

**IMPORTANT** Purging claims can only occur on the batch server. To do this, use the API call `IMaintenanceToolsAPI.startBatchProcess("purge")` with the batch server.

- Table import actions can occur only on the batch server. In other words, all `ITableImportAPI` interface methods require that you make the SOAP request directly to the batch server computer.

Some APIs only affect the server to which the call is made, in contrast to most APIs that add or modify entity data. Typical APIs force stale copies of entity data in other ClaimCenter servers to purge from the server in-memory cache. You can send these API calls to any server and all server update with the new information. However, some APIs change run time attributes of the server itself and only change the server to which you send the request:

- `ISystemToolsAPI.setRunlevel`
- `ISystemToolsAPI.updateLogLevel`
- `ISystemToolsAPI.getActiveSessionData`
- `ILoginAPI` – for setting up or releasing a session with a specific server

## SOAP Faults (Exceptions) in RPCE Web Services

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

Problems might occur during a web service API call, and your code must handle exceptions appropriately. Web service calls always use the SOAP protocol, even if your SOAP implementation hides most protocol details. If a web service generates an error, the error returns to the SOAP client.

**Note:** Technically speaking, an error over the SOAP protocol is called a *SOAP fault*. However, because the Gosu and Java language represent SOAP faults as exception classes, Guidewire documentation generally refers to SOAP faults as exceptions.

ClaimCenter defines three basic types of SOAP exceptions:

- SOAP server exceptions** – SOAP server exceptions indicate a problem with the SOAP server, or the server's basic functioning. SOAP server exceptions can also include database server errors like connection and table corruption issues. However, these never include uniqueness and key value issues because those are SOAP sender exceptions. Generally speaking, SOAP server exceptions indicate a problem in the server that is likely to succeed if tried again at some future time. For example, an exception to indicate the database is temporarily down uses the exception symbol `SOAPServerException`.
- SOAP sender exceptions** – SOAP sender exceptions indicate problems with data sent by the code that implements the web service API function. This usually means that the caller misencoded or omitted required data. SOAP sender exceptions also indicate other problems such as inserting two objects of the same type with the same public ID. For example, trying to insert two `Claim` entities with the same public ID. Generally speaking, SOAP sender errors indicate an error that the client code must correct before trying again. This exception is identified with the exception symbol `SOAPSenderException`.
- SOAP timeout exceptions** – If the connection to the server times out, plan for a timeout exception. This exception is identified with the exception symbol `SOAPTimeoutException`. This exception is not currently thrown by the server, but future versions might, so your external code must prepare for this exception. For the ClaimCenter-generated Java libraries files do not raise this exception type. However, other alternative SOAP interfaces used by SOAP clients may timeout due to network problems and throw this exception.

If you write a web service in Gosu, for each method declare exceptions it can throw using the `@Throws` annotation. Throw the most specific exception class that is appropriate for the error. In addition to the built-in exception

classes, you can also define your own exception classes that extend existing SOAP exception classes. If your web services declares it can throw that exception, the WSDL for that method includes your custom exception.

#### See also

“Declaring Exceptions” on page 99.

## Programming Context and Package Hierarchy for SOAP Exceptions

The package hierarchy for SOAP exceptions differ depending on the programming context. The following table compares the programming context and package hierarchy for SOAP exceptions:

Context	Language	Root class for SOAP exceptions
Implementing a custom web service in Gosu	Gosu	<code>gw.api.webservice.exception.SOAPException</code>
Connecting as a SOAP client to a web service published by a Guidewire application.	Java (using the generated libraries)	<code>com.guidewire.cc.webservices.fault.SOAPException</code>
	Gosu	<code>soap.SERVICENAME.fault.SOAPException</code> , where <i>SERVICENAME</i> is the name of your web service as registered in Studio
	Other languages or SOAP implementations	In other languages or SOAP implementations the SOAP fault syntax might vary. However, typically the hierarchy will look similar to <code>com.guidewire.cc.webservices.fault.SOAPException</code> .
Calling from Gosu as a client to a SOAP client of a web service published by an external system.	Gosu	<code>soap.SERVICENAME.fault.SOAPException</code> , where <i>SERVICENAME</i> is the name of your web service as registered in Studio

More detailed exceptions that indicate specific types of problems subclass from the three exception types related to servers, senders and timeouts.

The following table lists ClaimCenter API SOAP exceptions as a hierarchy. The three top-level items in this list extend from the `SOAPException` base class. If you create any custom exception classes, extend from either `SOAPServerException` or `SOAPSenderException`, not from `SOAPException`.

Exception name (indenting shows subclass)	Description
<b>SOAP Server Exceptions</b>	
<code>SOAPServerException</code>	The root exception for all server exceptions. See discussion earlier in this section.
→ <code>BatchProcessException</code>	An exception specific to batch process operations
→ <code>ServerStateException</code>	Indicates that the requested action is disallowed because of the current state of the server as a whole. For example, certain actions are forbidden if the server is at the multi-user run level.
<b>SOAP Sender Exceptions</b>	
<code>SOAPSenderException</code>	Root exception for all sender exceptions. See discussion earlier in this section.
→ <code>EntityStateException</code>	An action is disallowed because of the status of the business data it affects.  For example, only certain types of activities can be added to a closed claim, so attempting to add a forbidden type to a closed claim would throw this exception.

Exception name (indenting shows subclass)	Description
→ AlreadyExecutedException	The server throws this exception if the web service client attempts to perform a SOAP operation that has the same transaction ID as a previous SOAP operation in the database. You must never throw this exception.
→ PermissionException	The calling system did not have adequate permission to request this action. Try logging in as a different user or give the user the missing permissions
→ LoginException	A special permission exception for login authentication.
→ DataConversionException	ClaimCenter encountered a problem in provided data.
→ BadIdentifierException	Invalid identifier. For example, if a method had a public ID as an argument, and it references an non-existing record, the server throws this exception.
→ DuplicateKeyException	The server detected a duplicate key.
→ FieldConversionException	The server could not convert a property
→ FieldFormatException	A property is in the wrong format.
→ RequiredFieldException	A required property was omitted.
→ UnknownTypeKeyException	Some data to convert has an unknown type
<b>SOAP Timeout Exceptions</b>	
SOAPTimeoutException	The SOAP transaction timed out.

## Remote and Client-side SOAP Exceptions

In addition to the SOAP exceptions mentioned earlier, a web service in very rare cases could throw a remote exception. This exception indicates an unexpected error somewhere within SOAP processing itself. For example, this error could occurs within code that translates SOAP calls to Java calls or the other way around. All such exceptions derive from the class `java.rmi.RemoteException`.

You must prepare for errors specific to a client-side SOAP implementation. For generated Java libraries, you could get Apache Axis errors. For example, the generated Java SOAP libraries generate Apache Axis errors if there is a network failure that prevents communication with the SOAP server for 10 minutes. The libraries also generate errors if they detect problems in XML serialization or deserialization code.

## Web Service Client SOAP Exception Troubleshooting

From client code connecting to web services published by ClaimCenter, during debugging you might see unexpected or confusing SOAP exceptions. Check ClaimCenter error messages for hints about what may be wrong with the data. Use logging to identify the exact problem.

If you log errors while loading large numbers of entities, Guidewire strongly recommends that you include all public IDs (and any other relevant information) in log messages. ClaimCenter error messages do not automatically include public IDs.

Be careful of data conversion problems. In rare cases, the errors might not throw the exception symbol that you expect. This is sometimes due to differences in how the web service client views the object compared to the code that implements it.

For example, suppose a web service method contains a `String` argument. That API might store that value in the database as a 255 character string (`varchar(255)`). If the `String` that the client provides is too long, the web service might detect overflow problems only as it attempts to insert into the database. In this cases, you might see the error message “`inserted value too large for column`”. You might see other errors depending on the database driver. If you see this type of error, look for problems with the lengths of strings.

# Writing Command Line Tools that Call RPCE Web Services

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

Guidewire provides a Java class to compile your code into a command line tool. The tool that the Java class generates supports command line arguments. Use command line tools to encapsulate web services API client code for legacy systems that work best with command line tools.

During development, you might also want to write command line tools for commonly used actions:

- Fix a claim with errors that seems not properly synchronized with an external system by triggering a claim resync remotely
- Generate custom reports to evaluate whether an action was successful
- Start a validation batch process
- Start other batch processes
- Search for claims based on custom criteria

This topic discusses APIs for writing Java classes that run outside ClaimCenter call into it by using RPCE web services.

## See also

To write Gosu programs instead of Java classes that act like command line tools and call RPCE web services, see “Command Line Arguments” on page 379 in the *Gosu Reference Guide*.

## Base Java Class for Command Line Tools that Call RPCE Web Services

The Java class `CmdLineToolBase` in the package `com.guidewire.util.webservices` is an abstract implementation of a command line tool that provides:

- Built-in support for calling a web service API interface
- Parsing of command line arguments
- Establishing a connection to the ClaimCenter server providing request web services
- An interface to output help information from the command line tool.

The `CmdLineToolBase` class works in conjunction with standard open source Java classes from the Apache Foundation: `Option`, `Options`, and `OptionBuilder`. These classes provide a standard mechanism for command line option definitions, pattern definitions that define command line option validity, command line parsing, and the help information for that option.

The following table lists built-in command line arguments defined by the `CmdLineToolBase` class:

Built-in option	Meaning	Details
<code>-help</code>	Display option help	Display the available options and default values for this command line tool.
<code>-d</code>	Set a system property with <code>name=value</code> syntax	Set a system property as shown in this example: <code>-D javax.net.ssl.trustStore=C:/cmd-tools/lib/keystore</code>
<code>-server</code>	ClaimCenter server	The ClaimCenter server that is publishing web services. The Java class can override the method <code>getDefaultValue</code> to define the default URL.
<code>-user</code>	ClaimCenter user name	The ClaimCenter user name on this server. The Java class can override the method <code>getDefaultValue</code> to define the default URL.
<code>-password</code>	ClaimCenter password	The ClaimCenter password for this user.

To create custom command line arguments, create new Option objects and pass them in the constructor. The constructor includes an array of Option objects. In Java, create new objects as static class objects using the following syntax:

```
@SuppressWarnings("static-access")
private static final Option MYOPTIONNAME = OptionBuilder
    .hasArg()
    .withDescription("My description....")
    .create("theoptionname");

// A list of options
private static final Option[] OPTIONS = {MYOPTIONNAME};
```

### Important Methods in the Java Classes of Command Line Tools that Call RPCE Web Services

The most important method for the tool for you to override is the execute method. It takes an CommandLine object (in package org.apache.commons.cli), a server URL, and a web service proxy. Cast the web service proxy to the specific interface subclass you are calling, such as IUserAPI or your custom web service API class. Get the proper URL from the web service endpoint using the SOAP binding stub that contains your web service name with the SoapBindingStub suffix. Refer to the example for details. Look for the following line in the constructor and change the bold text to your web service name:

```
super(OPTIONS, MyServiceAPISoapBindingStub.ENDPOINT_ADDRESS_PROPERTY, MyServiceAPI.class);
```

There are other required methods, such as getProductId. You must implement this method to return the product code ("cc", "pc", or "bc"). For ClaimCenter, write a method that looks like the following:

```
protected String getProductId() {
    return "cc";
}
```

**IMPORTANT** If you do not implement getProductId properly, your tool throws exceptions during login.

You must also implement the getDefaultServerUrl method. It must return the server URL of the ClaimCenter application, such as "<http://localhost:8080/cc>".

### Calling More than One RPCE Web Service from Command Line Tools that Call RPCE Web Services

The CmdLineToolBase class makes it simple to code tools that need to communicate with only one web service interface. You can communicate with more than one ClaimCenter web service interface in your tool if necessary. From within your execute method, get an additional SOAP API proxy by calling getProxy() on the tool itself.

The getProxy method is implemented in the superclass CmdLineToolBase. Do not attempt to use the APILocator class, which is the standard way to use Java to connect to Guidewire RPCE web services. Command line tools based on CmdLineToolBase do not support using APILocator.

**IMPORTANT** Do not use the Java class APILocator within the code of command line tools that subclass CmdLineToolBase.

### Example Command Line Tool that Calls RPCE Web Services

The following example command line tool, TestCLI, calls the web service, MyServiceAPI, defined in the following sections:

- “Publishing a RPCE Web Service” on page 95
- “Testing Your RPCE Web Service With soap.local Namespace” on page 103

The web service MyServiceAPI returns a custom Gosu class called UserPhones that represents properties in a User entity. For this example, the command line tool defines one option called getuserphones. This option takes a public ID of a user.

Since the password is also a required parameter, the required arguments would look like:

```
-password PASSWORDHERE -getuserphones PUBLICID
```

For example:

```
-password gw -getuserphones default_data:1
```

To use and test this example, first create the web service as shown in “Publishing a RPCE Web Service” on page 95. Then, regenerate the integration libraries as discussed in “Regenerating Integration Libraries” on page 22. The generated libraries include your new web service MyServiceAPI in the Java JAR files.

## The Java Program for Example Command Line Tool

Create a new project directory. Relative to that project directory, create the following file at the location:

```
src/example/TestCLI.java
```

In that file, insert the following code:

```
package example;

import java.io.IOException;
import java.net.MalformedURLException;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.ParseException;
import org.xml.sax.SAXException;
import com.guidewire.util.webservices.CmdLineToolBase;

// MY IMPORTS
import com.guidewire.cc.webservices.api.MyServiceAPI;
import com.guidewire.cc.webservices.api.MyServiceAPISoapBindingStub;
import com.guidewire.cc.webservices.entity.UserPhones;

public class TestCLI extends CmdLineToolBase {

    // my command line tool options...
    @SuppressWarnings("static-access")
    private static final Option PUBLICID = OptionBuilder
        .hasArg()
        .withDescription("My custom parameter: Get user phones. Supply the PublicID of a USER entity!")
        .create("getuserphones");

    // a list of options...
    private static final Option[] OPTIONS = {PUBLICID};
    private static MyServiceAPI myAPI;

    // Default Constructor
    public TestCLI() {
        super(OPTIONS, MyServiceAPISoapBindingStub.ENDPOINT_ADDRESS_PROPERTY, MyServiceAPI.class);
    }

    // 3-option constructor
    public TestCLI(Option[] listofoptions, String s, Class aClass) {
        super(listofoptions, s, aClass);
    }

    public void execute(CommandLine arguments, String arg1, Object apiObject)
        throws IOException, ParseException, RemoteException,
        MalformedURLException, ServiceException, SAXException {

        System.out.println("Beginning command line tool test....");

        // Cast the inbound object to the ImportToolsAPI web services interface.
        myAPI = (MyServiceAPI) apiObject;

        String publicID = arguments.getOptionValue("getuserphones");
        System.out.println("Call the web service with public ID " + publicID);

        // Call the web service method!
        UserPhones up = myAPI.getUserPhones(publicID);

        System.out.println("The home phone for this user is: " + up.getHomePhone());
    }
}
```

```

        System.out.println("The work phone for this user is: " + up.getWorkPhone());
        System.out.println("Ending command line tool test.");
    }

    protected String getDefaultServerUrl() {
        return "http://localhost:8080/cc";
    }

    protected String getDefaultUserName() {
        return "su"; // Usually, you create a NEW user just for your SOAP access for easy tracking.
    }

    public Option[] getOptionsOneRequired() {
        return OPTIONS; // return list of options -- caller must use ONE of them
    }

    // WARNING: You must implement getProductCode and return the product code ("cc", "pc", or "bc").
    protected String getProductCode() {
        return "cc";
    }

    public static void main(String[] args) {
        System.out.println("Starting main() method");
        try {
            new TestCLI().execute(args); // This triggers the main execute() method.
        }

        catch (RemoteException e) {
            e.printStackTrace();
        }

        System.out.println("Ending main() method");
    }

}

```

Given the command line arguments mentioned earlier, the tool outputs something like:

```

Starting main() method
Beginning command line tool test....
Call the web service with public ID default_data:1
The home phone for this user is: 415-555-1212
The work phone for this user is: 415-555-1213
Ending command line tool test.
Ending main() method

```

## Run the Example Command Line Tool from the Command Line

You can use the following Ant file to test your command line tool. This example assumes that the Java source file is in the following directory relative to this Ant file at the path:

```
src/example/TestCLI.java
```

Create the following `build.xml` file:

```

<project name="Command Line Tool Test" default = "run">

    <property name="dir.build" value="classes"/>
    <property name="dir.src" value="src"/>
    <property name="build.toolkit" value="C:/ClaimCenter/soap-api/libs"/>

    <path id="project.class.path">
        <fileset dir="${build.toolkit}/lib">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${dir.build}"/>
    </target>

    <target name="init">
        <mkdir dir="${dir.build}"/>
    </target>

```

```
<target name="compile" depends="init">
<depend srcdir="${dir.src}" destdir="${dir.build}" closure="true">
  <classpath>
    <fileset dir="${build.toolkit}/lib">
      <include name="gw-*--cc.*jar"/>
    </fileset>
  </classpath>
</depend>

<javac srcdir="${dir.src}" destdir="${dir.build}" fork="true" failonerror="false">
  <classpath refid="project.class.path"/>
</javac>
</target>

<target name="run" depends="compile">
<java classname="example.TestCLI">
  <arg line=${build.commandlineargs}>
  <classpath>
    <pathelement location="${dir.build}"/>
    <pathelement path="${java.class.path}"/>
    <fileset dir="${build.toolkit}/lib">
      <include name="*.jar"/>
    </fileset>
  </classpath>
</java>
</target>

</project>
```

Compile and run this tool. Open a command line window and change your working directory to the same directory that contains the Ant file and typing the command:

```
ant -Dbuild.commandlineargs="-password gw -getuserphones default_data:1"
```

### Run the Example Command Line Tool from a Command Line Script

For a better interface to your command line tool, create a wrapper for it. For example, for Windows create a command line batch script (.bat file) file in that directory called userphones.bat with contents:

```
ant -Dbuild.commandlineargs="%*"
```

Next, compile and run this tool. Open a command line window and change your working directory to the same directory that contains the Ant file and the getuserphones.bat file. Next, type the command:

```
getuserphones -password gw -getuserphones default_data:1
```

This outputs something similar to the following:

```
Buildfile: build.xml

init:

compile:
[javac] Compiling 1 source file to C:\Documents and Settings\MyLogin\workspace\cli\classes

run:
[java] Starting main() method
[java] Beginning command line tool test....
[java] Call the web service with public ID default_data:1
[java] The home phone for this user is: 415-555-1212
[java] The work phone for this user is: 415-555-1213
[java] Ending command line tool test.
[java] Ending main() method

BUILD SUCCESSFUL
Total time: 21 seconds
```

You can call your class as a command line tool without Ant to reduce the amount of text that outputs to standard out. Simply run the example.TestCLI class directly with the java command line tool. Remember to include all the necessary JAR files using the -classpath argument.



# Calling RPCE Web Services from Gosu

Gosu code can import web services (SOAP APIs) from external systems and call these services as a SOAP client (an API consumer). The Gosu language handles all aspects of object serialization, object deserialization, basic authentication, and SOAP fault handling.

---

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

---

This topic only relates to RPCE web services, not WS-I web services. For discussion about the differences, see “Web Services Introduction” on page 31

This topic includes:

- “Calling External RPCE Web Services” on page 135
- “Calling RPCE Web Service from Gosu: ICD-9 Example” on page 140

## Calling External RPCE Web Services

---

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

---

This topic describes legacy support for SOAP using the Remote Procedure Call encoding (RPCE). If you can use WS-I web services instead of RPCE web services, use the WS-I web services. See “Web Service Types: WS-I versus RPCE” on page 32.

You can write code that easily imports RPCE SOAP APIs from external systems. By specifying the external web service in Guidewire Studio, you can easily call external web services from Gosu. The Gosu language handles all aspects of object serialization, object deserialization, basic authentication, and SOAP fault handling.

In Guidewire Studio, define a new web service endpoint in the **Web Services** folder in the left-most column of Guidewire Studio. Right-click on the **Web Services** folder. Click **New → Web Service**. Carefully review “Working with Web Services” on page 119 in the *Configuration Guide* for details on specifying a URL for your web service. That URL must start with the string “`http://`” and must return a WSDL format document.

You can configure the URL to have different values with different values of the different server (`server`) or environment (`env`) variables. For example, if using the development environment, directly connect to the service. If using the production environment, connect through a proxy server.

In the web service configuration screen, note the name to enter in the `Name` field. The name that you enter in that field is important because the web services client code uses that name explicitly. Choose the name carefully.

In the web service configuration screen, provide the timeout value. If the external system exceeds that timeout without a response, Gosu returns an error as a Gosu exception. A timeout of zero specifies no timeout, which means the system waits infinitely for a response. However, other factors might timeout the request if it takes too long, such as the destination's web application container if it has a timeout.

Write code that calls the SOAP endpoint and accesses its methods and entities. The name that you define for the SOAP endpoint affects how you call the service from Gosu. The name also affects the namespaces of objects related to the endpoint. For example, if you define the web service with the name `MyServiceAPI`, all related Gosu objects are under the following package hierarchy:

```
soap.MyServiceAPI.*
```

The following table displays the subpackage hierarchies for any web service accessed from Gosu.

Package hierarchy	Contains
<code>soap.SERVICENAME.*</code>	Root package hierarchy for the service
<code>soap.SERVICENAME.api.*</code>	The API classes provided by the remote service and specified in its WSDL file. Using SOAP terminology, these corresponds to <i>SOAP ports</i> defined by the service. Objects within this hierarchy are what you must instantiate to use the API.
<code>soap.SERVICENAME.entity.*</code>	The entity classes for this web service. This includes all objects, but does not include enumerations.
<code>soap.SERVICENAME.enum.*</code>	Enumerations for this web service.
<code>soap.SERVICENAME.fault.*</code>	SOAP faults, which are analogous to Gosu or Java exceptions.

From the perspective of Gosu code connecting to an external inbound SOAP API, the API class is always in the package `soap.SERVICENAME.entity.EntityName`. This is true independent of the original package of the class that implements the web service in the remote system.

To set up the remote API, use the Gosu keyword `new` with the SOAP API object.

For example, if the remote service provides a `FindServiceSoap` API, instantiate it with code similar to:

```
var api = new soap.MyServiceAPI.api.FindServiceSoap()
```

Then, API methods from this service can then just be called as normal method calls, for example:

```
var addr = api.FindAddress( spec )
```

The following example uses a hypothetical geocoding service, named `AddressID`.

The following example in Gosu uses this service:

```
uses soap.AddressID.entity.FindOptions
uses soap.AddressID.entity.FindAddressSpecification
uses soap.AddressID.entity.Address
uses soap.AddressID.entity.FindRange

// insantiate the SOAP endpoint (the SOAP port)
var api = new soap.AddressID.api.FindServiceSoap()

// set the logging output to SHOW YOU the *full* SOAP outgoing call and SOAP response!
api.setLoggingOutputStream( java.lang.System.out )

// In this example, the API provides a class called FindAddressSpecification
// and an API method called FindAddress

// set up a FindAddressSpecification entitiy
var spec = new FindAddressSpecification()
var address = new Address()
address.PostalCode = "94062"
```

```
address.CountryRegion = "USA"
address.Subdivision="CA"
spec.InputAddress = address
spec.DataSourceName = ".NA"
spec.Options = new FindOptions()
spec.Options.Range = new FindRange()
spec.Options.Range.Count = 50
spec.Options.ResultMask = {}

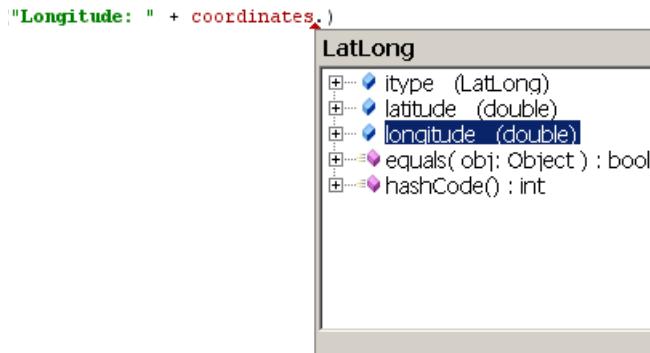
// Actually send the request. This actually does the request
var addr = api.FindAddress(spec)
var coordinates = addr.Results.FindResult.FoundLocation[0].LatLong

print("Latitude: " + coordinates.Latitude)
print("Longitude: " + coordinates.Longitude)
```

Note the beginning of this example has uses statements that define the namespaces of the destination service:

```
uses soap.AddressID.entity.FindOptions
uses soap.AddressID.entity.FindAddressSpecification
uses soap.AddressID.entity.Address
uses soap.AddressID.entity.FindRange
```

You must understand the namespaces of your desired objects and add lines like these for Gosu to know which objects and types you require. The *dot completion* (auto-completion) features in Guidewire Studio let you access entity properties. Studio displays your choices automatically based on the imported WSDL for your web service and the types of your variables:



### Local RPCE SOAP Endpoints in Gosu

The instructions listed in the previous section describe how to access external web service APIs. However, you can also call web services published on the same server as an outgoing (published) SOAP endpoint. This is particularly valuable for testing your SOAP APIs or calling ClaimCenter SOAP APIs that only exist as SOAP APIs.

To import a web service published on the same server, you do not need to set up the web service using the **Web Service** configuration dialog in Studio. You can simply access the class directly, although there is another important difference. All web service APIs publish to the same server from Gosu in the `soap.local.api.*` namespace in Guidewire Studio. In contrast, do not use the package hierarchy `soap.SERVICENAME.*` to connect to APIs published from the same server.

Before you use this syntax, you must regenerate the local files related to RPCE services. At a command prompt, change the working directory to `ClaimCenter/bin` and type the following command:

```
gwcc regen-soap-local
```

To access local SOAP APIs for RPCE web services, you must call the `gwcc regen-soap-local` script to regenerate local files. Also, after you add, modify, or delete an RPCE web service, call `regen-soap-local` again.

The service name is not part of the package hierarchy. All local web services share the same namespace for their APIs objects and their entities.

The section “Testing Your RPCE Web Service With soap.local Namespace” on page 103 shows how to use this feature to test your RPCE web service. It also shows how to authenticate to the server using handlers, a feature discussed in the following section.

---

**WARNING** Do not make calls to locally-hosted SOAP APIs from within the same server, such as from a plugin implementation or the rules engine.

---

If the SOAP API hosted locally modified entity data and commits the bundle, the current transaction does not always detect and reload local data. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance. If you have questions about how to convert some particular use locally-hosted SOAP APIs from plugins or rules, contact Customer Support. This is true for all types of local loopback SOAP calls to the same server. This includes `soap.local.*` objects, the SOAP client Java classes from Java plugins, and Studio-registered web services that call the same server as the client.

### Connecting to Microsoft.NET Web Services

Gosu automatically handles Windows Integrated authentication (NTLM authentication) exposed by web services published by Microsoft.NET. For NTLM authentication, you must specify the Windows domain in the user field separated by a backslash. For example, if the Windows domain is `mygroup` and the username is `jsmith`, set authentication to HTTP authentication and set the username in Studio for the web service to:

```
mygroup\jsmith
```

## Dynamic SOAP Authentication Handlers

If you need to connect to a SOAP API that does not require authentication, your code can simply connect to the service as discussed in previous sections. Similarly, to connect to a web service set up in Studio with HTTP authentication, connect to it directly using the code specified in previous sections. To set up authentication in Studio, see “Working with Web Services” on page 119 in the *Configuration Guide*. You can set up various combinations of username and password, varying by the system server and env variables.

However, if you need to dynamically generate name and/or password or other authentication criteria, you can send your authentication information with a SOAP request. At an implementation level, this typically involves adding *headers* to the SOAP request. You can add SOAP headers with *API handlers*, which are blocks of code associated with a web service interface. In a typical case, the block of code embeds authentication information in the web service API call by adding SOAP headers.

The authentication handlers customize authentication at the SOAP header level, which is in the HTTP *request content*. Authentication handlers cannot dynamically generate HTTP *request header* authentication information. You can vary SOAP user names and passwords by server environments or server IDs in a cluster. Studio has built-in configuration options to create setting groups for this purpose. See “Working with Web Services” on page 119 in the *Configuration Guide*. However, in Studio you cannot dynamically set the username and password for HTTP authentication other than varying based on server ID and environment variables. To do more complex dynamic generation of authentication information, use an API handler.

A handler operation can trigger before the SOAP call, after the SOAP call returns, or both. The primary use of API handlers is to add SOAP headers to the outgoing request SOAP request. The ability to trigger an action before a method call, after a method call, or both, is similar to Gosu *interceptors*. See “Annotations” on page 219 in the *Gosu Reference Guide* for more information.

After you choose a handler type, instantiate a handler. In the constructor, pass the appropriate authentication credentials. Next, add it to an API by call the `addHandler` method on the web service interface.

The following sections show how use each built-in type of handler as well how to create your own handler.

## Dynamic Transaction Authentication

Some web services require transaction identifiers (transaction IDs) in the GW\_TRANSACTION\_ID\_PROPERTY header of each request.

Use the handler called `GWTransactionIDHandler` to pass a transaction ID to the external API. The constructor takes one argument, the transaction ID.

For example:

```
var api = new soap.MyRemoteWebService.api.ServiceName()
api.addHandler(new GWTransactionIDHandler("123451"))
api.myMethodName()
```

## Dynamic Guidewire Authentication

Suppose you have more than one Guidewire application and want to write custom APIs for them to communicate. For typical cases, simply set up the authentication in the standard `Web Service` editor in Studio. However, if you did not specify Guidewire authentication in the connection setup configuration, you can authenticate dynamically. The built-in handler called `GWAuthenticationHandler` lets you authenticate dynamically. Similarly, if you store this authentication information in a hidden or generate it dynamically in some other way, authenticate the connection using `GWAuthenticationHandler`.

You can use this handler to connect to a web service on this server using the local endpoint system. See “Local RPCE SOAP Endpoints in Gosu” on page 137. That section includes more information and an example that uses this authentication system.

Use `GWAuthenticationHandler` to authenticate the external API with Guidewire authentication. All methods on the API object authenticate with the given credentials. It takes two arguments, the username and the password, as shown in the example in this section:

```
var api = new soap.MyRemoteWebService.api.ServiceName()
api.addHandler(new GWAuthenticationHandler("su", "gw"))
api.Methodname()
```

Your web service calls to this `api` object include the `GW_AUTHENTICATION_USERNAME_PROPERTY` and `GW_AUTHENTICATION_PASSWORD_PROPERTY` headers.

## Writing Your Own Handlers

In some cases you might need to access a web service that requires other headers. Insert arbitrary headers by making your own handler. To do this, write your own implementation of the `Handler` interface and insert whatever headers you need.

### To create a custom handler

1. **Learn what custom headers you need** – Sometimes this is a fixed header name. Sometimes this is dynamic value. Check the WSDL data to learn the headers. For example, suppose you see something like this in the WSDL for the service:

```
<soap:header message="tns:GetClientTokenUserInfoHeader" part="UserInfoHeader" use="literal"/>
```

This indicates that you need the header `tns:GetClientTokenUserInfoHeader`.

2. **Learn what custom data you need to insert in that header** – Sometimes this is a fixed value. Sometimes this is dynamic, perhaps encoded within the WSDL data. Sometimes this is simple text, such as a user ID, but it might be an entire complex structure.

3. **Create a custom CallHandler implementation** – Create a new call handler that inserts custom headers into an outgoing web services request. If you call a method on the web service interface, Gosu calls the handler. Define your own custom constructors that take authentication information or other important information. To embed static information into the request, encode authentication information directly into a handler class whose constructor takes no arguments.

**4. Add the handler to a web service API object** – Attach the handler to the web service interface:

```
var api = new soap.MyRemoteWebService.api.ServiceName()
api.addHandler(new MyHandlerClass("su", "gw"))
```

**5. Call methods on the web service API object** – Simply use the API by calling methods on the service object:

```
api.theMethodname()
```

There are two required methods on the `CallHandler` interface. The most important method is the `beforeRequest` method. Its last argument is a list of headers. You can call `headers.setHeader(...)` to append new outgoing headers to each request. One special requirement of this method is that the header name is not simply a `String` value but a two-part value called a qualified name (`QName`) object. A `QName` has two parts, the namespace and the name. A `QName` concatenates these two parts into one `String` separated by a colon in the resulting XML. Create one of these by passing to the `QName` constructor the two parts of the qualified name.

For example, to add the header with name `tns:GetClientTokenUserInfoHeader`, you can use the code:

```
var qn = new QName("tns", "GetClientTokenUserInfoHeader")
```

Next, you can add a header using this object and a value to add. For example:

```
headers.setHeader(qn, "my custom header value")
```

For the header argument to this method, you can pass a complex object that maps to a SOAP entity rather than a simple `String` value. The WSDL for the service, define all valid SOAP entities for that service.

Trigger this code in the `beforeRequest` method of your call handler with the following method signature:

```
beforeRequest(opName: String, args: Object[], headers : OutboundHeaders)
```

You can use a compact Gosu syntax to create a subclass of `CallHandler` without naming the subclass. This uses the Gosu feature anonymous subclasses, which means in-line subclasses without names. Use anonymous subclasses to create and use your handler with code such as the following example:

```
var api = new soap.MyRemoteWebService.api.ServiceName()

var userHeader = "my userHeader" // this example assumes a string value for the header

var myHandler = new CallHandler(){
    function beforeRequest(opName: String, args: Object[], headers : OutboundHeaders){
        headers.setHeader( new QName("tns", "GetClientTokenUserInfoHeader"), userHeader)
    }
    function afterRequest(opName: String, args: Object, headers : InboundHeaders){}
}
api.addHandler(myHandler)

// use it...
api.Methodname()
```

For more information, see “Inner Classes” on page 205 in the *Gosu Reference Guide*.

## Calling RPCE Web Service from Gosu: ICD-9 Example

Suppose your application wants to look up special codes. For example, ICD-9 codes represent International Statistical Classification of Diseases and Related Health Problems. One way to look up codes is to put all information directly into your application, such as a large look-up table. However, in some cases it might be necessary to let an external system look up the code, especially if the information changes frequently. This example looks up codes using an external web service. It creates a utility class in Gosu to help connect to this service and prepare its arguments.

---

**IMPORTANT** Guidewire does not certify the web service mentioned in the example to integrate with ICD-9 codes. This example simply demonstrates how web services might be used in application logic.

---

Set up the external web service in Studio:

1. Launch studio

2. Create a new web service
3. Name the web service ICD9
4. Click the **Update** button to update the WSDL location
5. Type the web service URL: <http://www.webservicemart.com/icd9code.asmx?WSDL>
6. Save the web service definition.

**Note:** Carefully note the created service name. The .NET service class has the original name of the class plus Soap at the end. Because the original name for the service is ICD9Code, Gosu creates this class as ICD9CodeSoap and you must use that name to connect to it.

To create a class to represent a single ICD-9 code, add the following code in the package `example.webservice.icd9` in Studio:

```
package example.webservice.icd9

/**
 * A single ICD9 code containing a code and a description.
 */
class ICD9Code
{
    private var _code : String as Code
    private var _desc : String as Desc
    construct()
    {
    }

    construct(aCode : String, aDesc : String){
        _code = aCode;
        _desc = aDesc;
    }
}
```

Next use a utility class to talk to this web service and to prepare data to send over the web service. In this case, the service requires a list of string lengths (code lengths) to be provided as a list. For example, to look for 3-to-6-character codes, the service expects the string "(3,4,5,6)". The `lookupICD9CodesByPrefix` method provides an easy-to-use API to provide a minimum and maximum length argument, and the method generates the list automatically

Add this class in the package `example.webservice.icd9`:

```
package example.webservice.icd9
uses java.lang.Integer
uses java.lang.IllegalArgumentException
uses java.util.ArrayList
uses gw.api.xml.XMLNode
uses java.lang.RuntimeException

class ICD9Util
{
    private var _api : soap.ICD9.api.ICD9CodeSoap

    construct()
    {
        _api = new soap.ICD9.api.ICD9CodeSoap()
    }

    private function getICD9Codes(code : String, len : String) : String {
        return _api.ICD9Codes( code, len )
    }

    static function getICD9Codes2(code : String, len : String) : String {
        var util = new ICD9Util()
        return util.getICD9Codes( code, len )
    }

    /**
     * @codePrefix the prefix of any code that to look up.
     * @minLen the minimum number of characters the code must contain. This may be null.
     * @maxLen the maximum number of characters the code must contain. This may be null.
     * If both the minLen and maxLen, then
    */
}
```

```

*/
static function lookupICD9CodesByPrefix(codePrefix : String, minLen : Integer,
    maxlen : Integer) : List<ICD9Code>{
    //no-op
    var pre = codePrefix + "*"
    var lenStr = createLengthFilter( minLen, maxlen )
    var result = getICD9Codes2( pre, lenStr )
    return parseResultIntoCodes( result )
}

private static function parseResultIntoCodes(result : String) : List<ICD9Code> {
    var codes = new ArrayList<ICD9Code>()
    var itemsNode = XMLNode.parse( result )
    if (itemsNode.ElementName != "items"){
        throw new RuntimeException("Could not parse the result into ICD9 Codes.")
    }
    itemsNode.Children.each( \ item -> {
        var code = item.Attributes.get( "code")
        var desc = item.Attributes.get( "description")
        if (item.ElementName != "item" || code == null){
            throw new RuntimeException("Could not parse the result into ICD9 Codes.")
        }
        codes.add(new ICD9Code(code, desc))
    })
    return codes
}

private static function createLengthFilter(minLen : Integer, maxlen : Integer) : String{
    if ((minLen != null && minLen < 0) || (maxLen != null && maxLen < 0)){
        throw new IllegalArgumentException("Cannot have a minimum length or maximum
            length that is negative: Min:" + minLen + ", Max:" + maxLen)
    }
    if (minLen != null && maxlen != null && minLen > maxlen){
        throw new IllegalArgumentException("Maximum length is less than the minimum length: "
            + minLen + " > " + maxlen)
    }
    var min = minLen == null ? 0 : minLen
    var max = maxlen == null ? 15 : maxlen //assume that no ICD9 code is more than 15 characters long
    var diff = (max + 1) - min
    var range : int[] = new int[diff]
    for (i in diff){
        range[i] = i + min
    }
    var rangeStr = range.join( "," )
    return "(" + rangeStr + ")"
}

```

### Test the ICD-9 Web Service from Gosu Scratchpad

Launch the Gosu Scratchpad and invoke the web service to test it. You must create an instance of the web service that has the class:

```
soap.WEBSERVICE.api.SERVICE
```

In this case, the fully qualified name is `soap.ICD9.api.ICD9CodeSoap`.

For a simple test, paste the following code in the Gosu Scratchpad:

```

var api = new soap.ICD9.api.ICD9CodeSoap()

var code = "E*"
var len = "(2,3,4)"
var result = api.ICD9Codes( code, len )

print(result) // print the final answer

```

# General Web Services

This topic describes general-purpose web services, such as mapping typecodes general system tools.

**Note:** This topic relies on you to understand what web services are and how to call them from remote systems. Before reading this topic, read “What are Web Services?” on page 31.

This topic includes:

- “Mapping Typecodes to External System Codes” on page 143
- “Importing Administrative Data” on page 145
- “Maintenance Web Services” on page 146
- “System Tools Web Services” on page 147
- “Workflow Web Services” on page 149
- “Profiling Web Services” on page 149

## Mapping Typecodes to External System Codes

If possible, configure the ClaimCenter typelists to include typecode values that match those already used in external systems. If you can do that, you do not need to map codes between systems.

However, in many installations this is infeasible and you must map between internal codes and external system codes. For example, you might have multiple external legacy systems and they do not match each other, so ClaimCenter can only match one system at maximum.

ClaimCenter provides a built-in utility to support simple typecode mappings. The first step in using the utility is to define the mappings. The mappings go into the `ClaimCenter/modules/configuration/config/typelists/mapping/typecodemapping.xml` file. The following is a simple example:

```
<?xml version="1.0"?>
<typecodemapping>
  <namespacelist>
    <namespace name="ns1" />
    <namespace name="ns2" />
  </namespacelist>

  <typelist name="LossType">
```

```

<mapping typecode="PR" namespace="ns1" alias="Prop" />
<mapping typecode="PR" namespace="ns2" alias="CPL" />
</typelist>
</typecodemapping>

```

The first section of the mapping file lists the set of *namespaces*. These namespaces correspond to the different external systems you need to map. For example, if the mappings are different between two external systems, then each has its own namespace.

The rest of the mapping file contains sections for each typelist that requires mapping. Within the typelist, add elements for mapping entries. Each mapping entry contains:

- **Typecode code** – The ClaimCenter typecode to map.
- **Namespace** – The namespace is the name of the external system in the XML file and used by any tool that translates type codes. You may wish to define your namespace strings with your company name and a system name. For example, for the company name ABC for a check printing service, you might use "ABC:checkprint".
- **Typecode alias** – The *alias* is the value for this typecode in the external system.

There can be multiple mapping entries for the same typecode and namespace. This supports situations in which multiple external codes map to the same ClaimCenter code during data import.

## Using Web Services to Translate Typecodes

After you define the mappings, you can translate between internal and external codes using the following `ITypelistToolsAPI` web service interface methods:

- `getAliasByInternalCode` – Finds the alias, if any, for a given typelist, code, and namespace. If there is no alias for that code and namespace, returns `null`. The `null` return value indicates that the internal and external codes are the same.
- `getTypeKeyByAlias` – Finds the internal typecode, if any, for a given typelist, alias, and namespace. This returns a data object that contains the typecode's code, name, and description properties. If no mapping is found, there are two possible meanings. It might indicate that the external and internal codes are the same. However, it might indicate that the mapping is missing from your mapping code. Use the `getTypelistValues` method to verify that the external code is a valid internal code in this situation.
- `getAliasesByInternalCode` – During exports use this to get an array of `String` values that represent external aliases to internal typecodes given a typelist, a namespace, and an internal code.
- `getTypeKeysByAlias` – During imports, use this to get an array of `TypeKeyData` objects given a typelist, a namespace, and an alias.
- `getTypelistValues` – Given the name of a typelist, returns an array of all the typekey instances contained within that typelist.

For example, suppose you want to translate a typecode from the `Contact` typelist from SOAP. Use the following Java web service client code to translate the code `ATTORNEY` for the external system `ABC:SYSTEM1`:

```

t1API = (ITypelistToolsAPI) APILocator.getAuthenticatedProxy(ITypelistToolsAPI.class, url, "su", "gw");
...
typekeyString = t1API.getAliasByInternalCode("Contact", "ABC:system1", "ATTORNEY");

```

## Using Gosu or Java to Translate Typecodes

Typecode translation may occur very frequently in Gosu or Java plugin code, or in Gosu templates used for data extraction by external systems. Calling the SOAP API to translate typecodes lowers server performance. For high performance from Gosu and Java, always use the `TypecodeMapperUtil` utility class. It provides methods that are similar to the SOAP API, such as `getInternalCodeByAlias`.

From Java, translate a typecode by using `TypecodeMapperUtil` with code such as:

```
TypeKey tk = TypecodeMapperUtil.getInternalCodeByAlias("Contact", "ABC:system1", "ATTORNEY");
```

From Gosu, translate a by using TypecodeMapperUtil with code such as:

```
var mapper = gw.api.util.TypecodeMapperUtil.getTypecodeMapper()  
var mycode = mapper.getInternalCodeByAlias( "Contact", "ABC:system1", "ATTORNEY" )
```

#### See also

- For details of other plugin utilities, see “Useful Java Plugin APIs” on page 171.

## Importing Administrative Data

ClaimCenter provides tools for importing and exporting data in XML using the import tools API (`IIImportToolsAPI`) interface. The easiest way to export data suitable for import is to use the built-in user interface in the application. Log into ClaimCenter with a user with administrative privileges. Then, in the left sidebar, click **Import/Export Data**. In the right pane, click **Export**.

For related topics, see:

- “Importing Administrative Data from the Command Line” on page 112 in the *System Administration Guide*
- “Importing and Exporting Administrative Data from ClaimCenter” on page 115 in the *System Administration Guide*

To prepare data for XML import, use the generated *XML Schema Definition* (XSD) files that define the XML data formats:

```
ClaimCenter/build/xsd/cc_import.xsd  
ClaimCenter/build/xsd/cc_entities.xsd  
ClaimCenter/build/xsd/cc_typelists.xsd
```

Regenerate these files by calling the `regen-xsd` command using the `gwcc` command line tool. See “Importing and Exporting Administrative Data from ClaimCenter” on page 115 in the *System Administration Guide*.

The `IIImportToolsAPI` web service interface’s `importXmlData` method imports administrative data from an XML file. Only use this for administrative database tables. Any other use is unsafe. This API does not perform integrity checks (as done during staging-table import) or data validation on imported data. Administrative tables include `User` and `Group` and their related entities.

**IMPORTANT** `IIImportToolsAPI` interface’s `importXmlData` is **only** supported for administrative data due to the lack of validation for this type of import.

The main XML import routine is `importXmlData` and it takes a single `String` argument containing XML data:

```
importToolsAPI.importXmlData(myXMLData);
```

## CSV Import and Conversion

There are several other methods in this interface related to importing and converting to and from comma-separated value data (CSV data). For example, import data from simple sample data files and convert them to a format suitable for XML import, or import them directly. See the Javadoc in the implementation file for more information about these CSV-related methods:

- `importCsvData` – import CSV data
- `csvToXml` – convert CSV data to XML data
- `xmlToCsv` – convert XML data to CSV data

## Advanced Import or Export

If you must import data in other formats or export administrative data programmatically, write a new web service to export administrative information one record at a time. For both import and export, if you write your own web

service, be careful never to pass too much data across the network in any API call. If you send too much data, memory errors occur. Do not try to import or export all administrative data in a dataset at once.

## Maintenance Web Services

The maintenance tools (`MaintenanceToolsAPI`) web service provides a set of tools available only if the system is at the `maintenance` run level or higher.

### Starting or Querying Batch Processes Using Web Services

One of the most important methods in the `MaintenanceToolsAPI` web service is `startBatchProcess`, which starts a background *batch process*. The API notifies the caller that the request is received. The caller must poll the server later to see if the process failed or completed successfully. For server clusters, batch processes **only** occur on the batch server. However, you can make the API request to any of the servers in the cluster. If the receiving server is not the batch server, the request automatically forwards to the batch server.

For example, start a batch process and get the process ID of the batch process:

```
processID = maintenanceTools.startBatchProcess("memorymonitor");
```

Terminate a batch process by process name or ID, for example:

```
maintenanceTools.terminateBatchProcessByName("memorymonitor");
maintenanceTools.terminateBatchProcessByID(processID);
```

Check the status of a batch process, for example:

```
maintenanceTools.batchProcessStatusByName("memorymonitor");
maintenanceTools.batchProcessStatusByID(processID);
```

Remember that for these APIs, the batch processes apply only to the current product, not any additional Guidewire applications that you have that might be integrated. In particular, if requesting this on a ClaimCenter server, it applies only to ClaimCenter servers not ContactManager servers.

If you use the `MaintenanceToolsAPI` web service to start a batch process, you can identify a batch process either the pre-defined strings as commands. If you defined any custom batch processes, you can also pass a `BatchProcessType` code value. This requires your custom `BatchProcessType` typecode to have the category `UIRunnable` or `APIRunnable`.

### Starting or Stopping Startable Plugins

Startable plugins are a special type of code that you can write that typically runs in the background. Startable plugins typically start on startup, rather than at certain points in the application logic to perform an action or return a value, like a typical plugin operates. For more information about startable plugins, see “[What are Startable Plugins?](#)” on page 271.

You can start or stop a startable plugin using methods on the `MaintenanceToolsAPI` web service interface.

To start a startable plugin, call the `startPlugin` method. To stop a startable plugin, call the `stopPlugin` method. Each method takes a single argument, which is the plugin name as a String. The plugin name is defined as you register the plugin in Studio. For details of the plugins editor in Studio, see “[Using the Plugins Registry Editor](#)” on page 113 in the *Configuration Guide*.

Before using these APIs, be sure that you understand the `@Distributed` annotation, discussed in the section “[Configuring Startable Plugins to Run on All Servers](#)” on page 274.

### Manipulating Work Queues Using Web Services

Similar to a batch process, a work queue represents a pool of work items that can be processed in a distributed way across multiple threads or even multiple servers. Several SOAP APIs query or modify the existing work

queue configuration. For example, APIs can get the number of instances of the workers on this server, and the sleep time after each worker finishes a work item (the *throttle interval*).

Wake up all workers for the specified queue across the entire cluster:

```
maintenanceTools.notifyQueueWorkers("ActivityEsc");
```

Get the work queue names for this product:

```
stringArray = maintenanceTools.getWorkQueueNames();
```

Get the number of instances and throttle interval for a work queue:

```
WorkQueueConfig wqConfig = maintenanceTools.getWorkQueueConfig("ActivityEsc");
numInstances = wqConfig.getInstances();
```

Set the number of instances and throttle interval for a work queue:

```
WorkQueueConfig wqConfig = new WorkQueueConfig();
wqConfig.setInstances(1);
wqConfig.setThrottleInterval(999);
WorkQueueConfig wqConfig = maintenanceTools.setWorkQueueConfig("ActivityEsc", wqConfig);
```

Any currently running worker instances stop after the current work item completes. The server creates and starts new worker instances as specified by the configuration object that you pass to the method.

The changes made using the batch process web service API are temporary. If the server starts (or restarts) at a later time, the server rereads the values from config.xml to define how to create and start workers.

For these APIs, the term *product* and *cluster* apply to the current Guidewire product only as determined by the SOAP API server requested.

If you use ContactManager and you use these APIs on a ClaimCenter server, it applies only to ClaimCenter not ContactManager. Similarly, if you use these APIs on a ContactManager server, it applies only to ContactManager not ClaimCenter.

## Marking Claims for Archive

To mark claims for archive, on the MaintenanceToolsAPI web service call the `scheduleForArchive` method. Internally, this just calls the `scheduleForArchive` on the ClaimAPI web service. It is provided on this web service as a convenience. This method is also triggered using the `maintenance_tools` command line tool as the `-scheduleforarchive` option.

For usage and warnings about this API, see “Archiving and Restoring Claims from External Systems” on page 156.

## Marking Claims for Purging Using Web Services

Mark claims for potential purging using the method `markForPurge` with an array of claim numbers, not claim public IDs. For example:

```
claimNumberArray[0] = "claim1234";
maintenanceTools.markForPurge(claimNumberArray);
```

You can also get the set of SQL statements required to update database statistics using the `getUpdateTableStatisticsData` method. It returns an `UpdateTableStatisticsData` object, which encapsulates a list of `String` objects which are SQL statements.

If you want the date that the current statistics were calculated, call the `whenStatsCalculated` method.

## System Tools Web Services

The system tools API (SystemToolsAPI) interface provides a set of tools that are always available, even if the server is set to dbmaintenance run level. For servers in clusters, system tools API methods execute **only** on the

server that receives the request. For the complete set of methods in the system tools API, refer to the Gosu implementation class in Studio.

## Getting and Setting the Run Level

The most important usage of the `SystemToolsAPI` interface is to set the system run level:

```
systemTools.setRunLevel(SystemRunlevel.GW_MAINTENANCE);
```

Or, to get the system run level:

```
runlevelString = systemTools.getRunLevel().getValue();
```

Sometimes you may want a more lightweight way of determining the run level of the server from another computer on the network than to use SOAP APIs. You might want to informally use your web browser during development to check the run level. This avoids details of SOAP authentication, regenerating the SOAP API libraries, or recompiling tools with the latest SOAP API libraries.

To check the run level, simply call the ping URL on a ClaimCenter server:

```
http://server:port/cc/ping
```

For example:

```
http://ClaimCenter.mycompany.com:8080/cc/ping
```

Assuming that the server is running, this returns an extremely short result as an HTML document containing a text encoding of the server run level in a special format.

If you are just checking whether the server is up, you do not care about the return result from this ping URL. Typically in such cases, if it returns a result at all it proves your server is running. In contrast, if there is an HTTP error or browser error, the server is not running to respond to the ping request.

If you want the actual run level, check the contents of the HTTP result. However, the value is not simply a standard text encoding of the public run level enumeration. It represents the ASCII character with decimal value of an integer that represents the internal system run level (30, 40, 50).

The following table lists the correlation between the run level and the value return by the ping URL:

Run level	Ping URL result character
<i>Server not running</i>	<i>No response to the HTTP request</i>
DB_MAINTENANCE	ASCII character 30, which is the Record Separator character
MAINTENANCE	ASCII character 40, which is the character "("
MULTIUSER	ASCII character 50, which is the character "2"
GW_STARTING	ASCII character 0. A null character result might not be returnable for some combinations of HTTP servers and clients.

## Getting Server and Schema Versions

You can use the `SystemToolsAPI` interface to get the current server and schema versions.

The following example code in Java demonstrates how to get this information:

```
versionInfo = systemTools.getVersion();
appVersion = versionInfo.getAppVersion();
schemaVersion = versionInfo.getSchemaVersion();
configVersion = versionInfo.getConfigVersion();
configVersionModified = versionInfo.getConfigVersionModified();
```

## Workflow Web Services

The web service API interface `WorkflowAPI` allows you to control ClaimCenter workflows from external client API code, including ClaimCenter plugins that use the web service APIs. In addition to being called by remote systems, the built-in `workflow_tools` command-line tools use these methods internally.

### Workflow Basics

You can invoke a workflow trigger remotely from an external system using the `invokeTrigger` method. To check whether you can invoke that trigger, call the `isTriggerAvailable` method, described later in the section.

Be aware that any time the application detects a workflow error, the workflow sets itself to the state `TC_ERROR`. If this happens, you can remotely resume the workflow using these APIs.

Refer to the following table for workflow actions you can request from remote systems:

Action	WorkflowAPI method	Description
Invoke a workflow trigger	<code>invokeTrigger</code>	Invokes a trigger key on the current step of the specified workflow, causing the workflow to advance to the next step. This method takes a workflow public ID and a String value that represents the workflow trigger key from the <code>WorkflowTriggerKey</code> typelist. To check whether you can call this workflow trigger, use the <code>isTriggerAvailable</code> method in this interface (see later in this table). This method returns nothing.
Check whether a trigger is available	<code>isTriggerAvailable</code>	Check if a trigger is available in the workflow. If a trigger is available, it means that it is acceptable to pass the trigger ID to the <code>invokeTrigger</code> method in this web services interface. This method takes a workflow public ID and a String value that represents the workflow trigger key from the <code>WorkflowTriggerKey</code> typelist. It returns true or false.
Resume a single workflow	<code>resumeWorkflow</code>	Restarts one workflow specified by its public ID. This method sets the state of the workflow to <code>TC_ACTIVE</code> . This method returns nothing.
Resume all workflows	<code>resumeAllWorkflows</code>	Restarts all workflows that are in the error state. It is important to understand that this only affects workflows currently in the error state <code>TC_ERROR</code> or <code>TC_SUSPENDED</code> . The workflow engine subsequently attempts to advance these workflows to their next steps and set their state to <code>TC_ACTIVE</code> . For each one, if an error occurs again, the application logs the error sets the workflow state back to <code>TC_ERROR</code> . This method takes no arguments and returns nothing.
Suspend a workflow	<code>suspend</code>	Sets the state of the workflow to <code>TC_SUSPENDED</code> . If you must restart this workflow later, use the <code>resumeWorkflow</code> method or the <code>resumeAllWorkflows</code> method.
Complete a workflow	<code>complete</code>	Sets the state of a workflow (specified by its public ID) to <code>TC_COMPLETED</code> . This method returns nothing.

## Profiling Web Services

From remote systems you can enable or disable the ClaimCenter profiler system using the `ProfilerAPI` web service. The methods use some or all of these common arguments:

- a boolean value that indicates whether to enable (`true`) or disable (`false`) the profiler
- a process type (a typecode in the `BatchProcessType` typelist)
- a boolean value that controls whether to use high resolution clock for timing (`true`) or not (`false`). This only has an affect on the Windows operating system.

- a boolean value that controls whether to enable stack traces (`true`) or not (`false`). This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- a boolean value that controls whether to enable query optimizer tracing (`true`) or not (`false`). This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- a boolean value that controls whether to allow *extended* query tracing. This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- an integer (`int`) value that threshold for how long (in milliseconds) a database operation can take before generating a report using dbms counters. Set the value 0 to disable dbms counters.

Each method takes a boolean value as the `enable` parameter. Set that parameter to `true` to enable the profiler. Set that parameter to `false` to disable the profiler for that component of the system:

#### **Batch Processes**

To enable the profiler for batch processes of a specific type, call the `setEnableProfilerForBatchProcess` method. Refer to the SOAP API Javadoc for API details.

#### **Batch Processes and Work Queues**

To enable the profiler for batch processes and work queues, call the `setEnableProfilerForBatchProcessAndWorkQueue` method. Refer to the SOAP API Javadoc for API details.

#### **Messaging Destinations**

To enable the profiler for messaging destinations, call the `setEnableProfilerForMessageDestination` method. Refer to the SOAP API Javadoc for API details.

#### **Startable Plugins**

To enable the profiler for startable plugins, call the `setEnableProfilerForStartablePlugin` method. Refer to the SOAP API Javadoc for API details.

#### **Subsequent Web Sessions**

To enable the profiler for subsequent web sessions, call the `setEnableProfilerForSubsequentWebSessions` method. Refer to the SOAP API Javadoc for API details.

#### **Web Services**

To enable the profiler for web services, call the `setEnableProfilerForWebService` method. Refer to the SOAP API Javadoc for API details.

#### **Work Queues**

To enable the profiler for work queues, call the `setEnableProfilerForWorkQueue` method. Refer to the SOAP API Javadoc for API details.

# Claim-related Web Services

This topic includes:

- “Claim Web Service APIs and Data Transfer Objects” on page 152
- “Adding First Notice of Loss from External Systems” on page 152
- “Getting a Claim from External Systems” on page 153
- “Importing a Claim from XML from External Systems” on page 153
- “Migrating a Claim from External Systems” on page 153
- “Getting Information from Claims/Exposures from External Systems” on page 154
- “Claim Bulk Validate from External Systems” on page 154
- “Previewing Assignments from External Systems” on page 155
- “Closing and Reopening a Claim from External Systems” on page 155
- “Policy Refresh from External Systems” on page 155
- “Add User Permissions on a Claim from External Systems” on page 156
- “Archiving and Restoring Claims from External Systems” on page 156
- “Managing Activities from External Systems” on page 157
- “Adding a Contact from External Systems” on page 158
- “Adding a Document from External Systems” on page 159
- “Adding an Exposure from External Systems” on page 159
- “Getting an Exposure from External Systems” on page 160
- “Closing and Reopening an Exposure from External Systems” on page 160
- “Adding to Claim History from External Systems” on page 160
- “Creating a Note from External Systems” on page 160

**See also**

- “Web Services Introduction” on page 31

## Claim Web Service APIs and Data Transfer Objects

To manipulate claims and exposures from external systems in general ways, use the `ClaimAPI` WS-I web services. If you need to manipulate claim financials, instead use the separate `ClaimFinancialsAPI` web service. See “Claim Financials Web Services (`ClaimFinancialsAPI`)” on page 369.

The ClaimCenter WS-I web services do not support entity data directly as method arguments or return values. Therefore, any web service APIs use a separate *data transfer object* (DTO) to encapsulate entity data.

For example, instead of passing an `Address` entity directly as a method argument, the API might pass a Gosu class called `AddressDTO`, which acts as the DTO. The DTO class contains only the properties in an `Address` entity instance that are necessary for the web service.

---

**IMPORTANT** If you extend the entity data model with properties that must exist in the web service for sending or receiving data, you must also extend the data transfer object. Add to the set of properties in the corresponding Gosu class with the DTO suffix. For example, if you add an important property to the `Address` entity, also add the property to the Gosu class `AddressDTO`. Before editing these files, carefully review the introductory comments at the top of each DTO file

---

These methods are for active claims only. If an external system posts any new updates on an archived claim, `ClaimAPI` methods throw the exception `EntityStateException`.

## Adding First Notice of Loss from External Systems

You can add a first notice of loss (FNOL) from an external system using the `ClaimAPI` method `addFNOL`.

---

**WARNING** Never add new claims or exposures to ClaimCenter with the web service APIs while the Financials Calculations batch process is running. This applies to `ClaimAPI` web service methods `addFNOL`, `migrateClaim`, `addExposure`, and `addExposures`. The batch process runs only while the server is in maintenance mode. For more information see “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide*.

---

**Note:** This method uses data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

If you add a first notice of loss (FNOL) with `addFNOL`, ClaimCenter does the following steps:

1. Runs the Loaded ruleset.
2. Runs all the normal rule sets, initial reserves, access control lists as for new claims in the New Claim Wizard.

The `addFNOL` method takes exactly two arguments:

- a claim DTO (`claimDTO`) object, which includes exposures
- a policy DTO (`policyDTO`) object

The method returns the public ID (as a `String`) of the newly-created claim.

## Getting a Claim from External Systems

An external system can get the populated data transfer object (DTO) for a claim if the external system knows the public ID for the claim. Call the `ClaimAPI` web service method `getDtoForClaim` to get the populated data transfer object (DTO) for that claim.

**Note:** This method uses data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

For example, getting the loss cause for a claim from Java:

```
ClaimDTO c = claimAPI.getDtoForClaim("cc:1234");
LossCauseDTO lossCause = c.getLossCause();
```

## Importing a Claim from XML from External Systems

There are two `ClaimAPI` methods to import a claim from XML data from an external system:

- If the format is the ACORD format, use the `importAcordClaimFromXML` method.
- If the format is not the ACORD format, use the `importClaimFromXML` method.

For details about this feature and how to customize ClaimCenter behavior, see “FNOL Mapper” on page 435.

## Migrating a Claim from External Systems

You can migrate an existing claim from another system with the `ClaimAPI` method `migrateClaim`. If you migrate a claim, some normal processing steps for an entirely new claim are unnecessary.

**WARNING** Never add new claims or exposures to ClaimCenter with the web service APIs while the Financials Calculations batch process is running. This applies to `ClaimAPI` web service methods `addFNOL`, `migrateClaim`, `addExposure`, and `addExposures`. The batch process runs only while the server is in maintenance mode. For more information see “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide*.

**Note:** This method uses data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

For example, it is unnecessary to pick a claim number and assign to an adjuster to migrate a claim. ClaimCenter assumes that the regular load rules or setup that ClaimCenter runs for a first notice of loss (FNOL) are unnecessary. Contrast this with the `addFNOL` method. See “Adding First Notice of Loss from External Systems” on page 152.

To migrate a claim, ClaimCenter does the following steps:

1. Set the claim state to open
2. Assign the claim to the appropriate user and group
3. Validate the claim at the `loadsSave` level
4. Commit the claim’s data to the database, including all exposures, activities, and other objects creating during the request
5. Commit the data to the database triggers Pre-update rules, double checks validation, and run any appropriate Event Fired rule sets for new entities.

The method takes exactly one argument, which is a claim DTO object (`ClaimDTO` instance). The method returns the Public ID (as a String) of the newly-created claim.

## Getting Information from Claims/Exposures from External Systems

The ClaimAPI web service provides various straightforward methods to get information from a claims, for example determining if the claim exists, is valid, is flagged, or is open.

The following table summarizes some various simple methods that you can use from external systems.

To do this task	Use this ClaimAPI method	Description
Check if a claim exists	doesExist	Takes a public ID for a claim and determines if the claim exists.
	claimsExist	Similar to doesExist, but takes a list of claim public ID String values, not a single public ID.
Check if a claim reached a specified validation level	checkValid	Takes a public ID for a claim and a minimum validation level. Returns true if the claim reached that validation level. Otherwise, returns false.
Check if a claim is flagged	isFlagged	Takes a public ID for a claim, and returns true if the <code>Claim.Flagged</code> value has the value <code>TC_ISFLAGGED</code> .
Get claim state	getClaimState	Returns the code of the claim state (a <code>ClaimState</code> typecode) converted to a string. If the claim state for that claim is undefined, returns <code>null</code> .
Find claim public ID from a claim number	findPublicIDByClaimNumber	Takes a claim number and returns the public ID of the corresponding claim.
Get the claim info for a claim, which corresponds to the <code>ClaimInfo</code> entity.	getClaimInfo	Takes a public ID for a claim, and returns a data transfer object for the <code>ClaimInfo</code> object ( <code>ClaimInfoDTO</code> ). See “Claim Web Service APIs and Data Transfer Objects” on page 152.
Get exposure state	getExposureState	Takes an exposure public ID and returns the exposure state as an <code>ExposureState</code> enumeration, or <code>null</code> if undefined.

## Claim Bulk Validate from External Systems

After importing many claims, you can schedule the newly-imported claims for validation, including associated policies and exposures. To use this API, you must know the *load command ID* from the import request, which is sometimes referred to as a `loadCommandID`.

**Note:** This method uses data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

Bulk validation uses distributed work queues to spread the work of validating large numbers of claims across multiple threads/nodes. This method does not perform the validation synchronously. This API creates a distributed work item to validate each claim, and starts a batch process.

**IMPORTANT** Do not expect the work items to complete before the method returns.

The method returns a *process ID* value, which has type `long`. Use this process ID value to query or manipulate the status of the batch process using the methods on `MaintenanceToolsAPI`. For example, to query the batch process, call the `MaintenanceToolsAPI` method `batchProcessStatusByID`. To cancel the batch process, call the `MaintenanceToolsAPI` method `terminateBatchProcessByID`.

By default, no worker instances are configured to run for this process. To perform the actual validation, use the `MaintenanceToolsAPI` web service method `setWorkQueueConfig` to dynamically allocate worker instances.

### To bulk load and validate the claims from an external system

1. Load claims with `TableImportAPI.integrityCheckStagingTableContentsAndLoadSourceTables()`

2. From the returned `TableImportResult` object, extract the `loadCommandID`.
3. Call `ClaimAPI.bulkValidate(loadCommandID)` and get the return value process ID value.
4. If you have not yet configured workers, as is the case in the default configuration, configure workers using `MaintenanceToolsAPI` web service. Call the method `setWorkQueueConfig` with work queue name `ClaimValidation`.
5. Notify workers using `MaintenanceToolsAPI` web service method `notifyQueueWorkers` with work queue name `ClaimValidation`.
6. Regularly call the `MaintenanceToolsAPI` method `batchProcessStatusByID` using the returned process ID until the batch process completes.

## Previewing Assignments from External Systems

From an external system, you can preview the assignment choices that ClaimCenter would make for a claim. Call the `ClaimAPI` web service method `previewAssignment`. The method takes a claim public ID and returns an assignment response (`AssignmentResponse`) object that encapsulates the results.

The `AssignmentResponse` object includes properties such as `ReviewAssignment` (a boolean), `UserID`, `GroupID`, `GroupType`, `QueueID`, and `ReviewerID`.

---

**IMPORTANT** The `previewAssignment` does not commit any changes to the claim to the database.

---

## Closing and Reopening a Claim from External Systems

From an external system, you can close or reopen a claim using methods on the `ClaimAPI` web service.

To close a claim, call the `closeClaim` method, and pass it the following:

- public ID for a claim
- an outcome type, in an `ClaimClosedOutcomeType` enumeration
- a reason, which is an optional `String` value that describes the reason for closing the claim

The method returns no value.

ClaimCenter uses the same logic that governs the Close Claim screen user interface.

To reopen a claim, call the `reopenClaim` method.

- public ID for a claim.
- the reason type for re-opening the claim, in an `ClaimReopenedReason` enumeration.
- a reason, which is an optional `String` value that describes the reason for re-opening the claim.

## Policy Refresh from External Systems

There are several methods on the `ClaimAPI` web service that manipulate policy data on a claim.

**Note:** These methods use data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

To refresh the policy on the claim with the latest information from the external policy administration system, call the `refreshPolicy` method. It takes a public ID for a claim and returns no result. The `claim.LossDate` field must be non-null for that claim, or this method throws an `RequiredFieldException` exception.

In some cases, you may want to refresh the policy based on the subset of information in a policy summary object, which corresponds to the `PolicySummary` entity type. If this is the case, instead use the `ClaimAPI` web service method `setPolicy`. It takes a data transfer object for a policy summary encapsulated in a `PolicySummaryDTO` object. Add any necessary properties to the `PolicySummaryDTO` parameter, which ClaimCenter uses to populate a request to your `IPolicySearchAdapter` plugin implementation. For related information, “Policy Search Plugin” on page 517. The set of required fields is only the set of fields that `IPolicySearchAdapter` needs. This set of fields may be fewer than the set of fields necessary to display the summary in the user interface. The method returns no result.

---

**WARNING** Changing the policy can have many side effects, especially if the old and new policies do not contain the same set of subobjects (such as vehicles). Use this method with caution. Also see “Policy Refresh Overview” on page 539.

---

## Add User Permissions on a Claim from External Systems

To grant a user permissions on a claim from an external system, call the `ClaimAPI` method `giveUserPermissionsOnClaim`. It takes the following:

- a claim public ID
- a user public ID
- an array of access permission types to set, as the type `ClaimAccessType[]`

The method returns nothing.

The new permissions are additive, any existing permissions remain if already existing. The method never removes permissions. If you pass an access permission type that the user already has, there is no effect for that permission and the method does not throw an exception.

## Archiving and Restoring Claims from External Systems

To schedule a claim for archiving, there are two methods in the `ClaimAPI` web service.

- To schedule by claim number, call `scheduleForArchive`. It takes an array of claim number `String` values.
- To schedule by public ID, call `scheduleForArchiveByPublicId`. It takes an array of public IDs for claims.

In both cases, the claim does not archive immediately. A background process archives the claim eventually.

For each claim, ClaimCenter confirms it is closed, it schedules it for archive by creating a high priority work item that the archiving work queue processes.

---

**WARNING** The archiving work queue is asynchronous. Do not assume claims are archived after this API call returns.

---

There is a race condition that can affect these calls. If a claim to be archived references a newly created administrative object, such as a new user, there is a chance the archiving of the claim fails. This is because the new admin object was not yet copied to the archiving database. This is a rare edge case because most claims to be archived are old, closed, claims which have been unaltered for a long time. The chances of hitting this race condition can be minimized by explicitly running the archive batch process before calling this method. However, this work-around is resource-intensive and not recommended as a general practice.

The methods throws the `SOAPException` exception if claims cannot be scheduled for archive because they cannot be found, are closed, or because an archive work item could not be created. If any of the claims is not found or not closed, then the call fails before attempting to archive any other claims. However, if all claims are present

and closed it is possible, though very unlikely, for ClaimCenter to create some work items successfully and other work items to fail.

The following Gosu web service client example shows the variant that uses public IDs:

```
claimPublicIDs = { "abc:1234", "abc:5678" }  
claimAPI.scheduleForArchiveByPublicID(claimPublicIDs)
```

These methods return no value.

### Restoring Archived Claims

To restore archived claims, call the `ClaimAPI` web service method `restoreClaims`. It takes as arguments:

- An array of the public ID values for the claims, as the type `String[]`
- A comment for restoring the claims, as a `String` object

It returns the public IDs of the claims that were restored.

**IMPORTANT** The `restoreClaims` method is unusual in the `ClaimAPI` web service because it does not contain the `@WsiCheckDuplicateExternalTransaction` annotation. Because of this, the method does not use or enforce transaction IDs. The method restores each claim in a separate bundle commit.

Multiple bundle commits with the same transaction ID are incompatible with the `@WsiCheckDuplicateExternalTransaction` annotation. See “Checking for Duplicate External Transaction IDs” on page 57.

For example:

```
String[] restored;  
String[] claimPublicIDs = { "abc:1234", "abc:5678" };  
// restore the claims:  
restored = claimAPI.restoreClaim(claimPublicIDs, "Policy system needs to restore claim to view it");
```

## Managing Activities from External Systems

In the `ClaimAPI` web service interface, there are several activity-related methods.

### Completing or Skipping Activities

To complete an activity, call the `ClaimAPI` web service method `completeActivity`. It takes a public ID for the activity, and returns no result.

To skip an activity, call the `skipActivity` method. It takes a public ID for the activity and returns no result.

Both methods require the claim to have the permissions `VIEW_CLAIM`, `CREATE_ACTIVITY`.

Both methods run the standard rule sets for skipped or completed activities, and set the activity properties `CloseDate` and `CloseUser`.

### Adding Activities

#### Adding Activities from DTO

From an external system, you can create a new activity and enter all properties in the data transfer object (DTO). Set the properties on the `ActivityDTO` object and pass it as an argument to the `createActivity` method:

```
var claimPublicID = claimAPI.createActivity(activityDTO)
```

The following properties on the `ActivityDTO` object must be non-null: `ClaimID`, `ActivityPatternID`.

The method returns public ID of the newly-created activity

### Adding Activities from Activity Pattern

To create an activity from an activity pattern, call the `addActivityFromPattern` method. The method takes the following two parameters:

- The claim public ID, which must be non-null
- The activity pattern public ID, which must be non-null

The activity pattern must be from the list of activity patterns for the given claim that meet the following criteria:

- If the claim is closed, then the activity pattern must be available to closed claims.
- The loss type on the activity pattern must be `null` or must match the loss type on the claim.

If the activity pattern does not match the above criteria, the API throws an `EntityStateException` exception.

The new activity is initialized with the following fields from the activity pattern:

- `Pattern`, `Type`, `Subject`, `Description`, `Mandatory`, `Priority`, `Recurring`, and `Command`

ClaimCenter calculates the activity target using the following fields in the pattern:

- `targetStartPoint`, `TargetDays`, `TargetHours`, and `TargetIncludeDays`

The escalation date of the activity is calculated using the following fields in the pattern:

- `escalationStartPoint`, `EscalationDays`, `EscalationHours`, and `EscalationIncludeDays`

If those fields are not included in the activity pattern, then the ClaimCenter does not set the target and/or escalation date. If the target date is calculated to be after the escalation date, then the target date is set to be the same as the escalation date.

The claim public ID of the activity is set to the given claim public ID, and the exposure public ID is set to `null`. The previous user public ID property is updated with the current web service authenticated user.

ClaimCenter assigns the newly-created activity to a group and/or user using the assignment engine. Finally, ClaimCenter persists the activity in the database. The method returns public ID of the newly-created activity.

For example, from Java:

```
claimAPI.addActivityFromPattern("abc:123", "abc:emailnote1");
```

### Get Activity Patterns

To get the data transfer object (DTO) for an activity pattern object from its activity pattern code, call the `ClaimAPI` method `getActivityPatternDataForCode`. It takes the code as a `String` value and returns the `ActivityPatternDTO` instance.

To get the data transfer object (DTO) for an activity pattern object from its public ID, call the `ClaimAPI` method `getActivityPatternData`. It takes the public ID as a `String` value and returns the `ActivityPatternDTO` instance.

## Adding a Contact from External Systems

To add a contact to the ClaimCenter database from an external system, call the `createContact` method.

**Note:** This method uses data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

The method takes a data transfer object for a contact (`ContactDTO`) and returns the public ID of the newly-created `Contact` entity instance.

## Adding a Document from External Systems

To add a document to the ClaimCenter database from an external system, call the `createDocument` method.

**Note:** This method uses data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

The method takes a data transfer object for a document (`DocumentDTO`) and returns the public ID of the newly-created `Document` entity instance.

The `createDocument` method strictly requires that `DocumentDTO.ClaimID` refers to a valid claim. Any other requirements are imposed (and enforced) by the data model configuration files. Refer to the data dictionary for the list of related fields.

A document is always related to a `Claim` object, and can be optionally related to at most one subobject on that same claim: a `ClaimContact`, `Exposure`, or `Matter` object. The properties on `DocumentDTO` reference these objects as `ClaimContactID`, `ExposureID`, and `MatterID`.

## Adding an Exposure from External Systems

The `ClaimAPI` web service contains several methods to create exposures on a claim.

**Note:** These methods use data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

The `ClaimAPI` method `addExposure` creates one new exposure and associates it with a claim. If the state of the claim is `open`, then the method runs the standard save and setup processes on the new exposure. If the state of the claim is `draft`, this method merely sets the exposure order on the claim. The `addExposure` method takes only one argument, which is the data transfer object for the exposure, as an `ExposureDTO` object. The `ExposureDTO.ClaimID` property must contain the public ID of the associated claim. The method returns a public ID `String` that contains the identifier of the newly created exposure.

To add multiple exposures to the same claim in one method call, use the `addExposures` method instead of `addExposure`. It takes two arguments:

- The claim public ID
- An array of exposures, as `ExposureDTO` objects. The `ExposureDTO.ClaimID` must either be `null` or match the claim public ID passed into the method. It is unsupported to add exposures to multiple claims in the same call to `addExposure`.

The `addExposures` method returns an array of `String` objects, which are the public IDs for the newly created `Exposure` entity instances.

---

**WARNING** Never add new claims or exposures to ClaimCenter with the web service APIs while the Financials Calculations batch process is running. This applies to `ClaimAPI` web service methods `addFNOL`, `migrateClaim`, `addExposure`, and `addExposures`. The batch process runs only while the server is in maintenance mode. For more information see “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide*.

---

The save and setup procedure performs the same set of initialization, rule sets, initial reserves, and bundle commits as adding exposures from the user interface.

## Getting an Exposure from External Systems

To get an exposure from an external system, call the `ClaimAPI` web service method `getDtoForExposure`.

**Note:** This method uses data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

The method takes a public ID for the exposure and returns an instance of the corresponding data transfer object `ExposureDTO`.

## Closing and Reopening an Exposure from External Systems

The `ClaimAPI` web service contains methods to close and reopen an exposure.

To close an exposure, call the `closeExposure` method. It takes three arguments:

- An exposure public ID
- An outcome enumeration from the `ExposureClosedOutcomeType` typelist
- A reason, which is a `String` that describes why you are closing the exposure.

The method returns no result.

To reopen an exposure, call the `reopenExposure` method. It takes three arguments:

- An exposure public ID
- An outcome enumeration from the `ExposureReopenedReason` typelist
- A reason, which is a `String` that describes why you are reopening the exposure.

The method returns no result.

## Adding to Claim History from External Systems

The `ClaimAPI` web service defines several methods for adding claim history events.

To add a custom history event with a specified `CustomHistoryType` but a blank description, call the `createCustomHistory` method. It takes a claim public ID, the history type (a `CustomHistoryType` enumeration). The method returns the public ID of the newly created history event.

To add a human-readable description, use the `createCustomHistoryWithDesc` method instead. It takes an additional argument at the end, which is a `String` value with a description.

## Creating a Note from External Systems

To create a note on an exposure from an external system, call the `ClaimAPI` web service method `createNote`.

**Note:** This method uses data transfer objects (DTOs) to represent data from entity types. See “Claim Web Service APIs and Data Transfer Objects” on page 152.

The method takes a note data transfer object (`NoteDTO`) as an argument. The `NoteDTO.ExposureID` and `NoteDTO.ClaimID` properties identify the exposure and claim for the new note.

The only required property is `ClaimID`.

The method returns the public ID of the newly-created note.

---

part III

# Plugins



# Plugin Overview

ClaimCenter plugins are software modules that ClaimCenter calls to perform an action or calculate a result. ClaimCenter defines a set of plugin interfaces. You can write your own implementations of plugins in Gosu or Java.

This topic includes:

- “Overview of ClaimCenter Plugins” on page 164
- “Error Handling in Plugins” on page 169
- “Temporarily Disabling a Plugin” on page 169
- “Example Gosu Plugin” on page 169
- “Special Notes For Java Plugins” on page 170
- “Getting Plugin Parameters from the Plugins Registry Editor” on page 171
- “Writing Plugin Templates For Plugins That Take Template Data” on page 172
- “Plugin Registry APIs” on page 174
- “Plugin Thread Safety” on page 175
- “Reading System Properties in Plugins” on page 180
- “Do Not Call Local Web Services From Plugins” on page 180
- “Creating Unique Numbers in a Sequence” on page 180
- “Restarting and Testing Tips for Plugin Developers” on page 181
- “Summary of All ClaimCenter Plugins” on page 181

## See also

- To help with writing Java plugins, see “Calling Java from Gosu” on page 119 in the *Gosu Reference Guide*.
- To help understand plugin interfaces, see “Interfaces” on page 211 in the *Gosu Reference Guide*.
- For information about messaging plugins, see “Messaging and Events” on page 299.
- For information about authentication plugins, see “Authentication Integration” on page 187.
- For information about document and form plugins, see “Document Management” on page 199.
- For information about other plugins, see “Other Plugin Interfaces” on page 263.

## Overview of ClaimCenter Plugins

ClaimCenter plugins are classes that ClaimCenter invokes to perform an action or calculate a result at a specific time in its business logic. ClaimCenter defines plugin interfaces for various purposes:

- Perform calculations
- Provide configuration points for your own business logic at clearly defined places in application operation, such as validation or assignment
- Generate new data for other application logic, such as generating a new claim number
- Define how the application interacts with other Guidewire InsuranceSuite applications for important actions relating to claims, policies, billing, and contacts.
- Define how the application interacts with other third-party external systems such as a document management system, third-party claim systems, third-party policy systems, or third-party billing systems.
- Define how ClaimCenter sends messages to external systems.

You can implement a plugin in the programming languages Gosu or Java. In many cases, it is easiest to implement a plugin with a Gosu class. If you use Java, you must use a separate IDE other than Studio, and you must regenerate Java libraries after any data model changes.

If you implement a plugin in Java, optionally you can write your code as an *OSGi* bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java libraries. Guidewire recommends OSGi for all new Java plugin development. For extended documentation on Java and OSGi usage, see “Java and OSGi Support” on page 627.

From a technical perspective, ClaimCenter defines a plugin as an *interface*, which is a set of functions (also known as methods) that are necessary for a specific task. Each plugin interface is a strict contract of interaction and expectation between the application and the plugin implementation. Some other set of code that implements the interface must perform the task and return any appropriate result values. For conceptual information about interfaces, see “Interfaces” on page 211 in the *Gosu Reference Guide*.

Conceptually, there are two main steps to implement a plugin:

- 1. Write a class (in Gosu or Java) that implements a plugin interface** – See “Implementing Plugin Interfaces” on page 164.
- 2. Register your plugin implementation class** – See “Registering a Plugin Implementation Class” on page 166

For most plugin interfaces, you can only register a single plugin implementation for that interface. However, some plugin interfaces support multiple implementations for the same interface, such as messaging plugins and startable plugins. For the maximum supported implementations for each interface, see the table in “Summary of All ClaimCenter Plugins” on page 181.

The plugin itself might do most of the work or it might consult with other external systems. Many plugins typically run while users wait for responses from the application user interface. Guidewire strongly recommends that you carefully consider response time, including network response time, as you write your plugin implementations.

## Implementing Plugin Interfaces

### Choose a Plugin Implementation Type

There are several ways to implement a ClaimCenter plugin interface:

- **Gosu plugin** – A Gosu class. Because you write and debug your code directly in ClaimCenter Studio, in many cases it is easiest to implement a plugin interface as a Gosu class. The Gosu language has powerful features including type inference, easy access to web service and XML, and language enhancements such as Gosu blocks and collection enhancements.

- **Java plugin** – A Java class. You must use an IDE other than Studio. If you write your plugin in Java, you must regularly regenerate the Java API libraries after changes to data model configuration. You can use any Java IDE. You can choose to use the included application called IntelliJ IDEA with OSGi Editor for your Java plugin development even if you do not choose to use OSGi.
- **OSGi plugin** – A Java class encapsulated in an *OSGi* bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java libraries. To simplify OSGi configuration, ClaimCenter includes an application called IntelliJ IDEA with OSGi Editor. For more information about OSGi, see “Overview of Java and OSGi Support” on page 627.

**IMPORTANT** Guidewire recommends OSGi for all new Java plugin development.

The following table compares the types of plugin implementations you can create:

Features of each plugin implementation type	Gosu plugin	Java plugin (no OSGi)	OSGi plugin (Java with OSGi)
<b>Choice of development environment</b>			
You can use ClaimCenter Studio to write and debug code	●		
You can use the included application IntelliJ IDEA with OSGi editor to write code		●	●
<b>Usability</b>			
Native access to Gosu blocks, collection enhancements, Gosu classes	●		
Entity and typecode APIs are the same as for Rules code and PCF code	●		
Requires regenerating Java libraries after data model changes		●	●
<b>Third-party Java libraries</b>			
Your plugin code can use third-party Java libraries	●	●	●
You can embed third-party Java libraries within a OSGi bundle to reduce conflicts with other plugin code or ClaimCenter itself.			●
Dependencies on specific third-party packages and classes are explicit in manifest files and validated at startup time.			●

## Writing Your Plugin Implementation Class

First you must decide which plugin implementation type you want to use and launch the appropriate IDE based on whether you are using Gosu or Java. To learn how the plugin type affects what IDE you must use, see “Implementing Plugin Interfaces” on page 164.

In your IDE, create the class within your own package hierarchy such as `mycompany.plugins`. Do not create your own classes in the `gw.*` or `com.guidewire.*` package hierarchies.

In Gosu and Java, create a class with a declaration containing the `implements` keyword, such as:

```
public class MyContactSystemClass implements IContactSystemPlugin {
```

If your class does not properly implement the plugin interface, your IDE shows compilation errors and can offer to add methods to your class that the interface requires. You must fix any issues before you code compiles.

Implement all public methods of the interface. Create as many other private methods and related classes as you need to provide internal logic for your code. However, ClaimCenter only calls the public methods in your main implementation class.

**IMPORTANT** If you use OSGi, you must perform additional configuration. See “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 645.

### For Gosu Plugins, the Method Signatures May Be Different than Java

In ClaimCenter Studio or in your Java IDE, the compiler can detect that a class that implements an interface does not yet implement all the methods in the interface. The editor provides a tool that creates new stub versions of any unimplemented methods.

Note that the Gosu versions of the methods look different from the Java versions in many cases. The most common compilation issue is that if an interface's method looks like properties, you must implement the interface as a Gosu property.

If the interface contains a method starting with the substring `get` or `is` and takes no parameters, define the method using property syntax. Do not simply implement it as simple method with the name as defined in the interface. For example, if plugin `ExamplePlugin` declared a method `getMyVar()`, your Gosu plugin implementation of this interface must not include a `getMyVar` method. Instead, it must look similar to the following:

```
package mycompany.plugins

class MyClass implements gw.plugin.IExamplePlugin {

    property get MyVar() : String {
        ...
    }
}
```

Similarly, methods that begin with `set` and take exactly one argument become property setters.

See “Java get/set/is Methods Convert to Gosu Properties” on page 121 in the *Gosu Reference Guide*.

### Plugin Templates (Only for Some Plugin Interfaces)

A small number of plugin interface methods provide method arguments that specify data as `String` values that contain data extracted from potentially large object graphs. This `String` data is called *template data*. Template data is the output of a Gosu template called a *plugin template*. A Gosu template is a short Gosu program that generates some output. In this case, the extracted data is an intermediary data format between rather than passing references to the original entity data objects directly to the plugin implementation.

If you see a plugin method with an argument called `templateData`, configure an appropriate plugin template that generates the information that your plugin implementation needs. See “Writing Plugin Templates For Plugins That Take Template Data” on page 172.

## Built-in Plugin Implementation Classes

For some plugin interfaces, ClaimCenter provides a built-in plugin implementation that you can use instead of writing your own version. Some plugin implementation classes are pre-registered in the default configuration.

Some plugin implementations are for demonstration only. Check the documentation for each plugin interface or contact Customer Support if you are not sure whether an included plugin implementation is supported for production.

ClaimCenter includes plugin implementations to connect with other Guidewire applications in the Guidewire InsuranceSuite. The plugin implementations for InsuranceSuite integration are located in packages that include the intended target application and version number. This helps ensure smooth migration and backwards compatibility among Guidewire applications. Carefully confirm package names for any plugin implementations you want to use. The package may include the Guidewire application two-digit abbreviation, followed by the application version number with no periods. For example, the shortened version of ClaimCenter 8.0.1 is `cc801`. Be sure to choose the plugin implementation class to match the version of the *other* external application, not the current application.

## Registering a Plugin Implementation Class

In most cases, the first step to implementing a plugin interface is to create your implementation class. See “Implementing Plugin Interfaces” on page 164.

In other cases, you might be using a built-in plugin implementation class. See “Built-in Plugin Implementation Classes” on page 166.

In either case, you must also *register* the plugin implementation class so the application knows about it. The registry configures which implementation class is responsible for which plugin interface. If you correctly register your plugin implementation, ClaimCenter calls the plugin at the appropriate times in the application logic. The plugin implementation performs some action or computation and in some cases returns results back to ClaimCenter.

To use the Plugins registry, you must know all of the following:

- The plugin interface name. You must choose a supported ClaimCenter plugin interface as defined in “Summary of All ClaimCenter Plugins” on page 181. You cannot create your own plugin interfaces.
- The fully-qualified class name of your concrete plugin implementation class.
- Any initialization parameters, also called *plugin parameters*. See “Plugin Parameters” on page 167.

In nearly all cases, you must only have one active enabled implementation for each plugin interface. However, there are exceptions, such as messaging plugins, encryption plugins, and startable plugins. The table of all plugin interfaces has a column that specifies whether the plugin interface supports one or many implementations. See “Summary of All ClaimCenter Plugins” on page 181. If the plugin interface only supports one implementation, be sure to remove any existing registry entries before adding new ones.

As you register plugins in Studio, the interface prompts you for a *plugin name*. If the interface accepts only one implementation, the plugin name is arbitrary. However, it is the best practice to set the plugin name to match the plugin interface name and omit the package. For example, enter the name `IContactSystemPlugin`.

If the interface accepts more than one implementation, the plugin name may be important. For example, for messaging plugins, enter the plugin name when you configure the messaging destination in the separate Messaging editor in Studio. See “Messaging Editor” on page 137 in the *Configuration Guide*. For encryption plugins, if you ever change your encryption algorithm, the plugin name is the unique identifier for each encryption algorithm.

To register a plugin, in Studio in the Project window, navigate to **configuration** → **config** → **Plugins** → **registry**. Right-click on **registry**, and choose **New** → **Plugin**. For more instructions about options in the Plugins registry editor, see “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.

In the dialog box that appears, it will ask for the interface. If the plugin name you used is the interface name (for example, `IContactSystemPlugin`), it is best to enter the interface name to be explicit. However, you can optionally leave the interface field blank.

---

**IMPORTANT** If the interface field is blank, ClaimCenter assumes the interface name matches the plugin name. You might notice that some of the plugin implementations that are pre-registered in the default configuration have that field blank for this reason.

---

## Plugin Parameters

In the Plugins registry editor, you can add a list of parameters as name-value pairs. The plugin implementation can use these values. For example, you might pass a server name, a port number, or other configuration information to your plugin implementation code using a parameter. Using a plugin parameter in many cases is an alternative to using hard-coded values in implementation code.

To use plugin parameters, a plugin implementation must implement the `InitializablePlugin` interface in addition to the main plugin interface. If ClaimCenter detects that your plugin implementation implements `InitializablePlugin`, ClaimCenter calls your plugin’s `setParameters` method. That method must have exactly one argument, which is a `java.util.Map` object. In the map, the parameters names are keys in the map. See “Getting Plugin Parameters from the Plugins Registry Editor” on page 171.

By default, all plugin parameters have the same name-value pairs for all values of the servers and environment system variables. However, the Plugins registry allows optional configuration by server, by environment, or both. See “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*. For example, you could set a server name differently depending on whether you are running a development or production configuration.

### For Java Plugins (Without OSGi), Define a Plugin Directory

For Java plugin implementations that do not use OSGi, the Plugins registry editor has a field for a *plugin directory*. A *plugin directory* is where you put non-OSGI Java classes and library files. In this field, enter the name of a subdirectory in the `ClaimCenter/modules/configuration/plugins` directory.

To reduce the chance of conflicts in Java classes and libraries between plugin implementations, define a unique name for a plugin directory for each plugin implementation. For details, see “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*. If you do not specify a plugin directory, the default is shared.

---

**IMPORTANT** OSGi plugin implementations automatically isolate code for your plugin with any necessary third-party libraries in one OSGi bundle. Therefore, for OSGi plugin implementations, you do not configure a plugin directory in the Plugins registry editor in Studio. To deploy OSGi bundles and third-party libraries in OSGi plugins, see “Java and OSGi Support” on page 627

---

For details of where to deploy your Java files for non-OSGi use, see “Deploying Non-OSGi Java Classes and JARs” on page 644.

### Deploying Java Files (Including Java Code Called From Gosu Plugins)

How to deploy any Java files or third-party Java libraries varies based on your plugin type:

- If your Gosu plugin implements the plugin interface but accesses third-party Java classes or libraries, you must put these files in the right places in the configuration environment. See “Deploying Non-OSGi Java Classes and JARs” on page 644. It is important to note that the plugin directory setting discussed in that section has the value `Gosu` for code called from `Gosu`.
- For Java plugins that do not use OSGi, first ensure you define a plugin directory. See “For Java Plugins (Without OSGi), Define a Plugin Directory” on page 168. For details of where to put your Java files, see “Deploying Non-OSGi Java Classes and JARs” on page 644.
- For OSGi plugins (Java classes deployed as OSGi bundles), deployment of your plugin files and third party libraries is very different from deploying non-OSGi Java code. See “Java and OSGi Support” on page 627 and “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 645.

---

**IMPORTANT** ClaimCenter supports OSGi bundles only to implement a ClaimCenter plugin interface and any of your related third-party libraries. It is unsupported to install OSGi bundles for any other purpose.

---

## Additional Information By Plugin Type

The additional information by plugin type, see the following sections.

Plugin type	For more information
Gosu	<ul style="list-style-type: none"><li>“Example Gosu Plugin” on page 169</li></ul>
Java (no OSGi)	<ul style="list-style-type: none"><li>“Special Notes For Java Plugins” on page 170</li><li>“Useful Java Plugin APIs” on page 171</li><li>“Java and OSGi Support” on page 627</li><li>“Calling Java from Gosu” on page 119 in the <i>Gosu Reference Guide</i></li></ul>
OSGi (Java with OSGi)	<ul style="list-style-type: none"><li>“Java and OSGi Support” on page 627</li><li>“Accessing Entity and Typecode Data in Java” on page 631</li><li>“OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 645</li></ul>

## Error Handling in Plugins

Where possible, ClaimCenter tries to respond appropriately to errors during a plugin call and performs a default action in some cases.

However, in cases such as claim number generation (`IClaimNumGenAdapter`), there is no meaningful default behavior, so the action that triggered the plugin fails and displays an error message.

There is not one standard way of handling errors in plugins. It is highly dependent on the plugin interface and the context. Check the method signatures to see what exceptions are expected. In some cases, it might be appropriate to return an empty set or error value if the response return value supports it. Contact the Guidewire Customer Technical Support group if you have questions about specific plugin interfaces and methods.

The ClaimCenter server catches any runtime exception and rolls back any related database transaction, and then displays an error in the application user interface.

## Temporarily Disabling a Plugin

By default, ClaimCenter calls Java plugins in the Plugins registry at the appropriate time in the application logic. To disable a plugin in the registry temporarily or permanently, navigate to the Plugins registry editor in Studio for your plugin implementation. To disable the plugin implementation, deselect the **Enabled** checkbox. To enable the plugin again, select the **Enabled** checkbox.

## Example Gosu Plugin

The most important thing to remember about implementing a plugin interface is Gosu is that Gosu versions of the interface methods look different from the Java versions in many cases. The most common problem is that if an interface’s method looks like properties, you must implement the interface as a Gosu property. See “Writing Your Plugin Implementation Class” on page 165.

This topic shows a basic Gosu plugin implementation that uses parameters. For more information about plugin parameters, see “Getting Plugin Parameters from the Plugins Registry Editor” on page 171.

The following Gosu example is a simple messaging plugin that uses parameters:

```
uses java.util.Map;
uses java.plugin;

class MyTransport implements MessageTransport, InitializablePlugin {
    private var _servername : String
```

```

// note the empty constructor. If you do provide an empty constructor, the application
// calls it as the plugin instantiates, which is before application calls setParameters
construct() {
}

override function setParameters(parameters: Map<String, String>) {
    // access values in the MAP to get parameters defined in Plugins registry in Studio
    _servername = parameters["ServerName"] as String
}

override function suspend() {}

override function shutdown() {}

override function setDestinationID(id:int) {}

override function resume() {}

override function send(message:entity.Message, transformedPayload:String) {
    print("MESSAGE SEND ===== ${message.Payload} --> ${transformedPayload}")
    message.reportAck()
}
}

```

#### If Your Gosu Plugin Needs Java Classes and Library Files

If your Gosu class implements the plugin interface but needs to access Java classes or libraries, you must put these files in the right places in the configuration environment. See “Deploying Non-OSGi Java Classes and JARs” on page 644. Note that the plugin directory setting discussed in that section can have the value `Gosu` for code called from Gosu. Alternatively, you can put your classes in the plugin directory called `shared`.

## Special Notes For Java Plugins

If you write your plugin in Java, you must regularly regenerate the Java API libraries to compile against the latest generated Java libraries. For a comparison of Gosu and Java for plugin development, see “Overview of ClaimCenter Plugins” on page 164.

It is important to understand some of the special considerations for writing Java code to use within ClaimCenter. For example:

- The way you access and use Guidewire business data entity instances is different between Gosu and Java. They are in different packages. Also, the way you create new entity instances is different in Java compared to Gosu.
- For plugin interface methods, remember that what Gosu exposes as properties (the `PropertyName` property) appear in Java as getter and setter methods (for example, the `getMyPropertyName` method).

For important information about writing Java in ClaimCenter, see “Java and OSGi Support” on page 627. Note that deployment of OSGi plugins is very different from non-OSGi plugin deployment.

## Which Libraries to Compile Java Code Against

After changes to the data model, it might be necessary to regenerate Java libraries. See “Regenerating Integration Libraries” on page 22.

ClaimCenter creates the library JAR files at the path:

`ClaimCenter/java-api/lib/...`

For important information about deploying Java classes and libraries, see “Java and OSGi Support” on page 627 and “Deploying Non-OSGi Java Classes and JARs” on page 644.

For more information about working with Guidewire entities from your Java code, see “Accessing Entity and Typecode Data in Java” on page 631.

## Where To Put Java Class and Library Files Needed By Java Plugins

For Java plugins without OSGi, see “Deploying Non-OSGi Java Classes and JARs” on page 644.

For OSGi plugins (Java with OSGi), see “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 645.

## Useful Java Plugin APIs

### Entity Data from Java

For important information about working with entity data, see “Accessing Entity and Typecode Data in Java” on page 631.

### Getting Current User from a Java Plugin

Sometimes it is useful to determine which user triggered a user interface action that triggered a Java plugin method call. Similarly, it is sometimes useful to determine which application user called a SOAP API triggered a Java plugin method call. In both cases, the Java plugin can use the `CurrentUserUtil` utility class.

To get the current user from a Java plugin, use the following code:

```
myCurrentUser = CurrentUserUtil.getCurrentUser().getUser();
```

### Translating Typecodes

Typical ClaimCenter implications need integration code that interfaces with external systems with different typecodes values than in ClaimCenter. ClaimCenter provides a typecode translation system that you can configure with name/value pairs.

Configure the values using an XML file. To modify the typelist conversion configuration file or to convert typecodes from Gosu or Java, see “Mapping Typecodes to External System Codes” on page 143.

Because typecode translation may occur very frequently in Java plugin code, Java plugins can use a utility class called `TypecodeMapperUtil`.

From external systems, the web service `ITypelistAPI` translates typecodes. It has similar methods to the Java API, such as `getInternalCodeByAlias`.

For example, to translate a typecode with `TypecodeMapperUtil`, use code such as:

```
TypeKey tk = TypecodeMapperUtil.getInternalCodeByAlias("Contact", "ABC:system1", "ATTORNEY");
```

## Getting Plugin Parameters from the Plugins Registry Editor

In the Studio Plugins registry editor, you can add one or more optional parameters to pass to your plugin during initialization. For example, you could use the editor to pass server names, port numbers, timeout values, or other settings to your plugin code. The parameters are pairs of `String` values, also known as name/value pairs. ClaimCenter treats all plugin parameters as text values, even if they represent numbers or other objects.

To use the plugin parameters in your plugin implementation, your plugin must implement the `InitializablePlugin` interface in addition to the main plugin interface.

If you do this, ClaimCenter calls your plugin’s `setParameters` method. That method must have exactly one argument, which is a `java.util.Map` object. In the map, the parameters names are keys in the map, and they map to the values from Studio.

The following Gosu example demonstrates how to define an actual plugin that uses parameters:

```
uses java.util.Map;
uses java.plugin;
```

```

class MyTransport implements MessageTransport, InitializablePlugin {
    private var _servername : String

    // note the empty constructor. If you do provide an empty constructor, the application
    // calls it as the plugin instantiates, which is before application calls setParameters
    construct() {
    }

    override function setParameters(parameters: Map<String, String>) {
        // access values in the MAP to get parameters defined in Plugins registry in Studio
        _servername = parameters["MyServerName"] as String
    }

    // NEXT, define all your other methods required by the MAIN interface you are implementing...
    override function suspend() {}

    override function shutdown() {}

    override function setDestinationID(id:int) {}

    override function resume() {}

    override function send(message:entity.Message, transformedPayload:String) {
        print("MESSAGE SEND ===== ${message.Payload} --> ${transformedPayload}")
        message.reportAck()
    }
}

```

#### Getting the Local File Path of the Root Directory

For Gosu plugins and Java plugins, you can access the plugin root directory path by getting a special built-in property from the Map. For the key name for root directory, use the name in the static variable `InitializablePlugin.ROOT_DIR`. This parameter is unavailable from OSGi plugins.

#### Getting the Local File Path of the Temp Directory

For Gosu plugins and Java plugins, you can access the plugin temporary directory path by getting a special built-in property from the Map. For the key name to get the root directory, use the name in the static variable `InitializablePlugin.TEMP_DIR`. This parameter is unavailable from OSGi plugins.

## Writing Plugin Templates For Plugins That Take Template Data

Some plugin interface methods have parameters that directly specify their data as simple objects such as `String` objects. Some plugin methods have parameters of structured data such as a `java.util.Map` objects. Some methods take entities such as `Claim`. Some objects might link to other objects, resulting in a potentially large object graph.

However, some plugin interface methods take a single `String` that it must parse to access important parameters. This text data is *template data* specified as plugin method parameters called `templateData`. Template data is the output of running a Gosu template called a *plugin template*. Plugin templates always have the suffix `.gsm`.

---

**IMPORTANT** For plugin templates, the file suffix must be `.gsm`. Do not use `.gst`, which is the normal Gosu template extension (see “Template Overview” on page 347 in the *Gosu Reference Guide*).

---

For example, the claim number generator plugin (`IClaimNumGenAdapter`) uses template data as a parameter in its methods.

ClaimCenter passes this potentially-large `String` as a parameter to the plugin for the subset of plugins that use template data. This approach lets ClaimCenter pass the plugin all necessary properties in a large data graph but with minimal data transfer.

For example, plugins or the external systems that they represent often need to analyze a claim's properties to make assignment, segmentation, validation, or other decisions. Frequently, these properties are not just part of the claim itself. Instead, they may be part of the entire graph of objects connected to the claim. For example, properties from the policy (existence of coverage on the policy type) or from the insured's contact record (age) could be needed to make decisions.

In this case, a claim's object graph can be very large, but the claim number generator plugin might only need a small subset of data in simple *fieldname=value* pairs. The corresponding plugin template might generate simple text like the following:

```
ClaimLossDate=03/30/2006
PolicyNumber=HO-3234598765
PolicyType=TC_homeowners
PolicyEffectiveDate=1/01/2005
...
...
```

Then, a plugin method can simply parse the text to access the properties. For plugins written in Java, it is easy to use the standard Java class called `Properties`. It can parse a `String` in that format into name/value pairs from which you can extract information using code such as: `propertiesObject.getProperty(fieldname)`.

For example, this Java code takes the `templateData` parameter encoded in the simple format described earlier, and then extracts the value of the `AddressID` property from it:

```
// Create a Java Properties object
Properties claimProperties = new Properties();

try {
    // extract the template data string and load it to the Properties object
    claimProperties.load(new ByteArrayInputStream(templateData.Bytes));
} catch (java.io.IOException IOE) {
    System.out.println("MyPluginName: bad template data");
}

// Extract properties from the Properties object
String myAddressID = claimProperties.getProperty("AddressID");
```

This Gosu code does a similar thing for a Gosu plugin as a private method within the Gosu class:

```
private function loadPropertiesFromTemplateData(templateData : String) : Properties
{
    var props = new Properties();
    try{
        props.load(new ByteArrayInputStream(templateData.getBytes()));
        _logger.info("The properties are : " + props);
    }
    catch (e) {
        e.printStackTrace();
        return null;
    }
    return props;
}
```

You can design any text-based data format you want to pass to the plugin in the `templateData` string. If your data is not very structured, Guidewire recommends the simple *fieldname=value* format demonstrated earlier. In some cases, it may be convenient to generate XML formatted data, which permits hierarchical structure despite being a text format. This is especially useful for communicating to external systems that require XML-formatted data. Whatever text-based format you choose to use, you can modify the associated plugin template to generate the desired XML format.

For each plugin method call that takes a `templateData` parameter (not all methods do), ClaimCenter has a Gosu template file in `ClaimCenter/modules/configuration/config/templates/plugins/....` ClaimCenter selects the correct plugin using a naming convention:

```
{interface name}_{entity name}.gsm
```

For example, for claim assignment, the template in that directory would be named `IAssignmentAdapter_Claim.gsm`. It might look like:

```
ClaimNumber=<%= Claim.ClaimNumber %>
LossType=<%= Claim.LossType %>
```

```
LossCause=<%= Claim.LossCause %>
LossDate=<%= Claim.LossDate %>
```

In the case of claim assignment, for example, the root object passed to the Gosu template is `Claim`. Each plugin that requires template date for some parameters has a different template for each combination of plugin and entity type. In all cases, it is possible to access the associated claim object from within the template. For example, if an `Activity` is the root object to a template that handles activity assignment, the template looks up the associated claim as `Activity.Claim`.

After the template runs, it generates template data that looks like the following:

```
ClaimNumber=HO-2983472-01
LossType=TC_PR
LossCause=TC_burglary
LossDate=2007-02-01
```

After the Gosu engine generates a response using the designated Gosu template, the resulting `String` passes to the plugin as the `templateData` parameter to the plugin method. Again, this is only for plugin interface methods that take a `templateData` parameter.

## Plugin Registry APIs

### Getting References to Plugins from Gosu

To ask the application for the currently-implemented instance of a plugin, call the `Plugins.get(INTERFACENAME)` static method and pass the plugin interface name as an argument. It returns a reference to the plugin implementation. The return result is properly statically typed so you can directly call methods on the result.

For example:

```
uses gw.plugin.Plugins

// Gosu uses type inference to know the type of the variable is IContactSystemPlugin
var plugin = Plugins.get( IContactSystemPlugin )

try{
    plugin.retrieveContact("abc:123" )
}catch(e){
    e.printStackTrace()
    throw new DisplayableException(e.Message)
}
```

Alternatively, you can request a plugin by the plugin name in the `Plugins` registry. This is important if there is more than one plugin implementation of the interface. For example, this is common in messaging plugins. To do this, use an alternative method signature of the `get` method that takes a `String` for the plugin name as defined in the Studio plugin configuration.

For example, if your plugin was called `MyPluginName` and the interface name is `IStartablePlugin`:

```
uses gw.plugin.Plugins

// you must downcast to the plugin interface
var contactSystem = Plugins.get( "MyContactPluginRegistryName" ) as IContactSystemPlugin

try{
    contactSystem.retrieveContact("abc:123")
}catch(e){
    e.printStackTrace()
    throw new DisplayableException(e.Message)
}
```

### Check Is Enabled

From Gosu you can use the `Plugins` class to determine if a plugin is enabled in Studio. Call the `isEnabled` method on the `Plugins` class and pass either of the following as a `String` value:

- the interface name type (with no package)
- the plugin implementation name in the `Plugins` registry

For example:

```
uses gw.plugin.Plugins  
var contactSystemEnabled = Plugins.isEnabled( IContactSystemPlugin )
```

### Getting References to Java Plugins from Java

From Java you can get references to other installed plugins either by class or by the string representation of the class. Do this using methods on the `PluginRegistry` class within the `guidewire.pl.plugin` package. Get the plugin by class by calling the `getPlugin(Class)` method. Get the plugin by name calling the `getPluginByName(String)` method.

This is useful if you want to access one plugin from another plugin. For example, a messaging-related plugin that you write might need a reference to another messaging-related plugin to communicate or share common code.

Also, this allows you to access plugin interfaces that provide services to plugins or other Java code. For example, the `IScriptHost` plugin can evaluate Gosu expressions. To use it, get a reference to the currently-installed `IScriptHost` plugin. Next, call its methods. Call the `putSymbol` method to make a Gosu context symbol such `claim` to evaluate to a specific `Claim` object reference. Call the `evaluate` method to evaluate a `String` containing Gosu code.

---

**IMPORTANT** Do not use this API from Gosu. From Gosu, call the `Plugins.get(INTERFACENAME)` static method and pass the plugin interface name or plugin name as an argument. See earlier in this topic for details.

---

## Plugin Thread Safety

If you register a Java plugin or a Gosu plugin, exactly one instance of that plugin exists in the Java virtual machine on that server, generally speaking. For example, if you register a document production plugin, exactly one instance of that plugin instantiates on each server.

The rules are different for messaging plugins in ClaimCenter server clusters. Messaging plugins instantiate only on the batch server. The other non-batch servers have zero instances of message request, message transport, and message reply plugins. For more information, see “Messaging Flow Details” on page 305. Messaging plugins must be especially careful about thread safety because messaging supports a large number of simultaneous threads, configured in Studio.

However, one server instance of the Java plugin or Gosu plugin must service multiple user sessions. Because multiple user sessions use multiple process threads, follow these rules to avoid thread problems:

- Your plugin must support multiple simultaneous calls to the same plugin method from different threads. You must ensure multiple calls to the plugin never access the same shared data. Alternatively, protect access to shared resources so that two threads do not access it simultaneously.
- Your plugin must support multiple simultaneous calls to the plugin in general. For example, ClaimCenter might call two different plugin methods at the same time. You must ensure multiple method calls to the plugin never access the same shared data. Alternatively, protect access to shared resources so that two threads do not access it simultaneously.
- Your plugin implementation must support multiple user sessions. Generally speaking, do not assume shared data or temporary storage is unique to one user request (one HTTP request of a single user).

Collectively, these requirements describe thread safety. You must ensure your implementation is thread safe.

---

**IMPORTANT** For important information about concurrency, see “Concurrency” on page 369 in the *Gosu Reference Guide*.

---

The most important way to avoid thread safety problems in plugin implementations is to avoid variables stored once per class, referred to as *static variables*. Static variables are a feature of both the Java language and the Gosu language. Static variables let a class store a value once per class, initialized only once. In contrast, object *instance variables* exist once per instance of the class.

Static variables can be extremely dangerous in a multi-threaded environment. Using static variables in a plugin can cause serious problems in a production deployment without taking great care to avoid problems. Be aware that such problems, if they occur, are extremely difficult to diagnose and debug. Timing in a multi-user multi-threaded environment is difficult, if not impossible, to control in a testing environment.

Because plugins could be called from multiple threads, there is sometimes no obvious place to store temporary data that stores state information. Where possible and appropriate, replace static variables with other mechanisms, such as setting properties on the relevant data passed as parameters. For example, in some cases perhaps use a data model extension property on a `Claim` or other relevant entity (including custom entities) to store state-specific data for the plugin. Be aware that storing data in an entity shares the data across servers in a ClaimCenter cluster (see “Design Plugin Implementations to Support Server Clusters” on page 179). Additionally, even standard instance variables (not just static variables) can be dangerous because there is only one instance of the plugin.

If you are experienced with multi-threaded programming and you are certain that static variables are necessary, you must ensure that you *synchronize* access to static variables. Synchronization refers to a feature of Java (but not natively in Gosu) that locks access between threads to shared resources such as static variables.

---

**WARNING** Avoid static variables in plugins if at all possible. ClaimCenter may call plugins from multiple process threads and in some cases this could be dangerous and unreliable. Additionally, this type of problem is extremely difficult to diagnose and debug.

---

For important information about concurrency, see “Concurrency” on page 369 in the *Gosu Reference Guide*.

For more information about concurrency and related APIs in Java, see:

<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>

The following sections list some common approaches for thread safety with static variable in Java:

- “Using Java Concurrent Data Types, Even from Gosu” on page 177
- “Using Synchronized Methods (Java Only)” on page 177
- “Using Java Synchronized Blocks of Code (Java only)” on page 178

For thread safety issues in Gosu, see “Gosu Static Variables and Gosu Thread Safety” on page 176.

Additionally, note some similar issues related to multi-server (cluster) plugin design in “Design Plugin Implementations to Support Server Clusters” on page 179.

## Gosu Static Variables and Gosu Thread Safety

The challenges of static variables and thread safety applies to Gosu classes, not just Java. This affects Gosu in plugin code and also for Gosu classes triggered from rules sets. The most important thing to know is that static variables present special challenges to ensure your code is thread safe.

The typical way to create a Gosu class with static variable is with code like:

```
class MyClass {  
    static var _property1 : String;  
}
```

You must be as careful in Gosu with static variables and synchronizing data in them to be thread safe as in Java. Use the Java concurrent data types described in this section to ensure safe access.

---

**WARNING** Thread safety APIs that use blocking can affect performance negatively. For highest performance, use such APIs wisely and test your code under heavy loads that test the concurrency.

---

For important information about concurrency, see “Concurrency” on page 369 in the *Gosu Reference Guide*.

### Using Java Concurrent Data Types, Even from Gosu

The simplest way to synchronizing access to a static variable in Java is to store data as an instance of a Java classes defined in the package `java.util.concurrent`. The objects in that package automatically implement synchronization of their data, and no additional code or syntax is necessary to keep all access to this data thread-safe. For example, to store a mapping between keys and values, instead of using a standard Java `HashMap` object, instead use `java.util.concurrent.ConcurrentHashMap`.

These tools protect the integrity of the keys and values in the map. However, you must ensure that if multiple threads or user sessions use the plugin, the business logic still does something appropriate with shared data. You must test the logic under multi-user and multi-thread situations.

---

**WARNING** All thread safety APIs that use blocking can affect performance negatively. For high performance, use such APIs carefully and test all code under heavy loads that test the concurrency.

---

For important information about concurrency, see “Concurrency” on page 369 in the *Gosu Reference Guide*.

### Using Synchronized Methods (Java Only)

Java provides a feature called synchronization that protects shared access to static variables. It lets you tag some or all methods so that no more than one of these methods can be run at once. Then, you can add code safely to these methods that get or set the object’s static class variables, and such access are thread safe.

If an object is visible to more than one thread, and one thread is running a synchronized method, the object is locked. If an object is locked, other threads cannot run a synchronized method of that object until the lock releases. If a second thread starts a synchronized method before the original thread finishes running a synchronized method on the same object, the second thread waits until the first thread finishes. This is known as *blocking* or *suspending execution* until the original thread is done with the object.

Mark one or more methods with this special status by applying the `synchronized` keyword in the method definition. This example shows a simple class with two synchronized methods that use a static class variable:

```
public class SyncExample {
    private static int contents;

    public int get() {
        return contents;
    }

    // Define a synchronized method. Only one thread can run a syncced method at one time for this object
    public synchronized void put1(int value) {
        contents = value;
        // do some other action here perhaps...
    }

    // Define a synchronized method. Only one thread can run a syncced method at one time for this object
    public synchronized void put2(int value) {

        contents = value;
        // do some other action here perhaps...
    }
}
```

Synchronization protects invocations of all synchronized methods on the object: it is not possible for invocations of two different synchronized methods on the same object to interleave. For the earlier example, the Java virtual machine does all of the following:

- Prevents two threads simultaneously running `put1` at the same time
- Prevents `put1` from running while `put2` is still running
- Prevents `put2` from running while `put1` is still running.

This approach protects integrity of access to the shared data. However, you must still ensure that if multiple threads or user sessions use the plugin, your code does something appropriate with this shared data. Always test your business logic under multi-user and multi-thread situations.

ClaimCenter calls the plugin method initialization method `setParameters` exactly once, hence only by one thread, so that method is automatically safe. The `setParameters` method is a special method that ClaimCenter calls during plugin initialization. This method takes a Map with initialization parameters that you specify in the Plugins registry in Studio. For more information about plugin parameters, see “Special Notes For Java Plugins” on page 170.

On a related note, Java class constructors cannot be synchronized; using the Java keyword `synchronized` with a constructor generates a syntax error. Synchronizing constructors does not make sense because only the thread that creates an object has access to it during the time Java is constructing it.

For important information about concurrency, see “Concurrency” on page 369 in the *Gosu Reference Guide*.

### Using Java Synchronized Blocks of Code (Java only)

Java code can also synchronize access to shared resources by defining a block of statements that can only be run by one thread at a time. If a second thread starts that block of code, it waits until the first thread is done before continuing. Compared to the method locking approach described earlier in this section, synchronizing a block of statements allows much smaller granularity for locking.

To synchronize a block of statements, use the `synchronized` keyword and pass it a Java object or class identifier. In the context of protecting access to static variables, always pass the class identifier `ClassName.class` for the class hosting the static variables.

For example, this demonstrates statement-level or block-level synchronization:

```
class MyPluginClass implements IMyPluginInterface {  
    private static byte[] myLock = new byte[0];  
    public void MyMethod(Address f){  
        // SYNCHRONIZE ACCESS TO SHARED DATA!  
        synchronized(MyPluginClass.class){  
            // Code to lock is here....  
        }  
    }  
}
```

This finer granularity of locking reduces the frequency that one thread is waiting for another to complete some action. Depending on the type of code and real-world use cases, this finer granularity could improve performance greatly over using synchronized methods. This is particularly the case if there are many threads. However, you might be able to refactor your code to convert blocks of synchronized statements into separate synchronized methods. For more information, see “Using Synchronized Methods (Java Only)” on page 177.

Both approaches protect integrity of access to the shared data. However, you must plan to handle multiple threads or user sessions to use your plugin, and do safely access any shared data. Also, test your business logic under realistic heavy loads for multi-user and multi-thread situations.

---

**WARNING** Thread safety APIs that use blocking can affect performance negatively. For highest performance, use such APIs wisely and test your code under heavy loads that test the concurrency.

---

For important information about concurrency, see “Concurrency” on page 369 in the *Gosu Reference Guide*.

## Avoid Singletons Due to Thread-Safety Issues

The thread safety problems discussed in the previous section apply to any Java object that has only a single instance (also referred to as a *singleton*) implemented using static variables. Because static variable access in multi-threaded code is complex, Guidewire strongly discourages using singleton Java classes. You must synchronize access to all data singleton instances just as for other static variables as described earlier in this section. This restriction is important for all Gosu Java that ClaimCenter runs.

This is an example of creating a singleton using a class static variable:

```
public class MySingleton {  
    private static MySingleton _instance =  
        new MySingleton();  
  
    private MySingleton() {  
        // construct object . . .  
    }  
  
    public static MySingleton getInstance() {  
        return _instance;  
    }  
}
```

For more information about singletons in Java, see:

<http://java.sun.com/developer/technicalArticles/Programming/singletons>

If you absolutely must use a singleton, you must synchronize updates to class static variables as discussed at the beginning of “Plugin Thread Safety” on page 175.

**WARNING** Avoid creating singletons, which are classes that enforce only a single instance of the class. If you really must use singletons, you must use the synchronization techniques discussed in “Plugin Thread Safety” on page 175 to be thread safe.

For important other information about concurrency, see “Concurrency” on page 369 in the *Gosu Reference Guide*.

## Design Plugin Implementations to Support Server Clusters

Generally speaking, if your plugin deploys in a ClaimCenter server cluster, there are instances of the plugin deployed on every server in the cluster. Consequently, design your plugin code (and any associated integration code) to support concurrent instances. If the Gosu code calls out to Java for any network connections, that code must support concurrent connections.

**Note:** There is an exception for this cluster rule: messaging plugins exist only for the single server designated the batch server. See “Messaging Flow Details” on page 305.

Because there may be multiple instances of the plugin, you must ensure that you update a database from Java code carefully. Your code must be thread safe, handle errors fully, and operate logically for database transactions in interactions with external systems. For example, if several updates to a database must be treated as one action or several pieces of data must be modified as one atomic action, design your code accordingly.

The thread safety synchronization techniques in “Plugin Thread Safety” on page 175 are insufficient to synchronize data shared across multiple servers in a cluster. Each server has its own Java virtual machine, so it has its own data space. Write your plugins to know about the other server’s plugins but not to rely on anything other than the database to communicate among each other across servers.

You must implement your own approach to ensure access to shared resources safely even if accessed simultaneously by multiple threads and on multiple servers.

For important information about concurrency, see “Concurrency” on page 369 in the *Gosu Reference Guide*.

## Reading System Properties in Plugins

You might want to test plugins in multiple deployment environments without recompiling plugins. For example, perhaps if a plugin runs on a test server, then the plugin queries a test database. If it runs on a production server, then the plugin queries a production database.

Or, you might want a plugin that can be run on multiple machines within a cluster with each machine knowing its identity. You might want to implement unique behavior within the cluster. Alternatively, add this information to log files on the local machine and in the external system also.

For these cases, plugins can use environment (`env`) and server ID (`serverId`) deployment properties to deploy a single plugin with different behaviors in multiple contexts or across clustered servers. Define these system properties in the server configuration file or as command-line parameters for the command that launches your web server container. In general, use the default system property settings. If you want to customize them, use the Plugins registry editor in Studio.

Gosu plugins can query system properties using the methods `getEnv`, `getServerId` and `isBatchServer`, all on the `ServerUtil` class. For example the following Gosu expression evaluate to the current value of the `env` system property, the server ID, and whether this server is the batch server:

```
var envValue = gw.api.system.server.ServerUtil.getEnv("gw.cc.env")
var serverID = gw.api.system.server.ServerUtil.getServerId()
var isBatchServer = gw.api.system.server.ServerUtil.isBatchServer()
```

Java plugins can query system properties using code such as the following example, which gets the `env` property:

```
java.lang.System.getProperty("gw.cc.env");
```

### See also

- “Clustering Application Servers” on page 79 in the *System Administration Guide*.
- For more information about `env` and `serverId` settings, see “Specifying Environment Properties in the <registry> Element” on page 15 in the *System Administration Guide*.
- “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.

## Do Not Call Local Web Services From Plugins

Do not call locally-hosted web service APIs (SOAP APIs) from within a plugin or the rules engine in production systems. If the web service hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data for your plugin implementation. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance.

If you have questions about how to refactor your code to avoid local loopback API calls over web services, contact Guidewire Customer Support.

## Creating Unique Numbers in a Sequence

Typical ClaimCenter implementations need to reliably create unique numbers in a sequence for some types of objects. For example, to enforce a series of unique IDs, such as public ID values, within a sequence. You can generate new numbers in a sequence using sequence generator APIs.

These methods take two parameters:

- An initial value for the sequence, if it does not yet exist.
- A `String` with up to 256 characters that uniquely identifies the sequence. This is the sequence key (`sequenceKey`).

For example, suppose you want to get a new number from a sequence called `WidgetNumber`.

In Gosu, use code like the following:

```
nextNum = gw.api.system.database.SequenceUtil.next(10, "WidgetNumber")
```

If this is the first time any code has requested a number in this sequence, the value is 1. If other code calls this method again, the return value is two, three, or some larger number depending on how many times any code requested numbers for this sequence key.

Similarly, in Java, use the code:

```
nextNum = gw.api.system.database.SequenceUtil.next(10, "WidgetNumber");
```

## Restarting and Testing Tips for Plugin Developers

If you frequently modify your plugin code, you might need to frequently redeploy ClaimCenter to test your plugins. If it is a non-production server, you may not want to shut down the entire web application container and restart it. For development (non-product) use only, reload only the ClaimCenter application rather than the web application container. If your web application container supports this, replace your plugin class files and reload the application.

For example, Apache Tomcat provides a web application called Manager that provides this functionality. For documentation on this Apache Tomcat Manager, refer to:

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/manager-howto.html>

## Summary of All ClaimCenter Plugins

The following table summarizes the plugin interfaces that ClaimCenter defines. For a table that summarizes the plugin interfaces that ContactManager defines, see “ContactManager Plugins” on page 293 in the *Contact Management Guide*.

ClaimCenter Plugin Interface	Description	Maximum enabled implementations
<b>Authentication plugins (see “Authentication Integration” on page 187)</b>		
AuthenticationServicePlugin	The authentication service plugin authorizes a user from a remote authentication source, such as a corporate LDAP or other single-source sign-on system.	1
AuthenticationSource	A marker interface representing an authentication source for user interface login or RPCE web services. The implementation of this interface must provide data to the authentication service plugin that you register in ClaimCenter. All classes that implement this interface must be serializable. Any object contained with those objects must be serializable as well.  For an alternative plugin interface to use with WS-I web services only, see <code>WebservicesAuthenticationPlugin</code> below.	1
AuthenticationSourceCreatorPlugin	Creates an authentication source (an <code>AuthenticationSource</code> ) for user interface login. This takes an HTTP protocol request (from an <code>HTTPRequest</code> object). The authorization source must work with your registered implementation of the <code>AuthenticationServicePlugin</code> plugin interface.	1

ClaimCenter Plugin Interface	Description	Maximum enabled implementations
DBAuthenticationPlugin	Allows you to store the database username and password in a way other than plain text in the config.xml file. For example, retrieve it from an external system, decrypt the password, or read a file from the file system. The resulting username and password substitutes into the database configuration for each instance of that \${username} or \${password} in the database parameters.	1
WebservicesAuthenticationPlugin	For WS-I web services only, configures custom authentication logic. This plugin interface is documented with other WS-I information. For more information, see “WS-I Web Services Authentication Plugin” on page 56.  For an alternative plugin interface to use with RPCE web services, see <a href="#">AuthenticationSource</a> above.	1
ABAAuthenticationPlugin	Advanced authentication between ClaimCenter and ContactManager, such as hiding/encrypting name and password information. For more information, see “ABAAuthenticationPlugin for ContactManager Authentication” on page 197.	1
<b>Document content and metadata plugins (see “Document Management” on page 199)</b>		
IDocumentContentSource	Provides access to a remote repository of documents, for example to retrieve a document, store a document, or remove a document from a remote repository. ClaimCenter implements this with a default version, but for maximum data integrity and feature set, implement this plugin and link it to a commercial document management system.	1
IDocumentDataSource	Stores metadata associated with a document, typically to store it in a remote document management system; see <a href="#">IDocumentContentSource</a> mentioned earlier for a related plugin. By default, ClaimCenter stores the metadata locally in the ClaimCenter database if you do not define it. For maximum data integrity and feature set, implement this plugin and link it to a commercial document management system.	1
<b>Document production plugins (see “Document Production” on page 213)</b>		
IDocumentProduction	Generates documents from a template. For example, from a Gosu template or a Microsoft Word template. This plugin can create documents synchronously and/or asynchronously.  <b>Note:</b> Only register one implementation of this plugin. However, there are multiple other classes that implement this interface and handle one document type. The registered implementation of this plugin interface dispatches requests to the other classes that implement this same interface. See “Document Production” on page 213.	1 (see note)
IDocumentTemplateSource	Provides access to a repository of document templates that can generate forms and letters, or other merged documents. An implementation may simply store templates in a local repository. A more sophisticated implementation might interface with a remote document management system.	1
IEmailTemplateSource	Gets email templates. By default, ClaimCenter stores the email templates on the server but you can customize this behavior by implementing this plugin.	1

ClaimCenter Plugin Interface	Description	Maximum enabled implementations
IDocumentTemplateSerializer	<i>Use the built-in version of this plugin using the “Plugins registry” but generally it is best not implement your own version.</i> This plugin serializes and deserializes document template descriptors. Typically, descriptors persist as XML, as such implementations of this class understand the format of document template descriptors and can read and write them as XML.	1
<b>Messaging plugins (see “Messaging and Events” on page 299)</b>		
MessageTransport	<p>This is the main messaging plugin interface. This plugin sends a message to an external/remote system using any transport protocol. This could involve submitting to a messaging queue, calling a remote API call, saving to special files in the file system, sending e-mails, or anything else you require. Optionally, this plugin can also acknowledge the message if it is capable of sending synchronously (that is, as part of a synchronous send request).</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Studio, Studio prompts you for a name for the plugin. That is called the <i>plugin name</i>. Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.</p>	Multiple
MessageRequest	<p>Optional pre-processing of messages, and optional post-send-processing (separate from post-acknowledgement processing).</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Studio, Studio prompts you for a name for the plugin. That is called the <i>plugin name</i>. Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.</p>	Multiple
MessageReply	<p>Handles asynchronous acknowledgements of a message. After submitting an acknowledgement to optionally handles other post-processing afterward such as property updates. If you can send the message synchronously, do not implement this plugin. Instead, implement only the transport plugin and acknowledge each message immediately after it sends the message.</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Studio, Studio prompts you for a name for the plugin. That is called the <i>plugin name</i>. Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.</p>	Multiple
<b>Financials plugins</b>		
IBulkInvoiceValidationPlugin	Validates a bulk invoice before the bulk invoice submits. This plugin must confirm that a bulk invoice does not violate your business logic. Bulk invoice validation is separate from bulk invoice approval. In general, validation determines whether the data appears to be correct, such as checking whether only certain vendors can submit any bulk invoices. In contrast, submitted invoices go through the approval process with business rules or human approvers to approve or deny an invoice before final processing. For more information, see “Bulk Invoice Integration” on page 391.	1

ClaimCenter Plugin Interface	Description	Maximum enabled implementations
IInitialReserveAdapter	Allows a financial initial reserve to be set automatically as part of the set-up of a new exposure. Called as part of the process of saving and setting up a new exposure (after running Segmentation). By default, uses a rule set for determining the initial reserve. For more information, see “Initial Reserve Initialization for Exposures” on page 409.	1
IExchangeRateSetPlugin	To make financial transactions in multiple currencies, ClaimCenter needs a way of describing current currency exchange rates around the world. Do this using the exchange rate set plugin ( <b>IExchangeRateSetPlugin</b> ) interface, whose main task is to create ExchangeRateSet entities encapsulated in a ExchangeRateSet entity. For more information, see “Exchange Rate Integration” on page 409.	1
IBackupWithholdingPlugin	Handles backup withholding on checks. For more information, see “Deduction Calculations for Checks” on page 407.	1
IDeductionAdapter	Allows financial deductions to be made from the amount otherwise owed on a payment (for example, back-up withholding or a negotiated discount for a preferred vendor). Called as part of the process of creating a new check. By default, calculates back-up withholding deduction, if any, based on the payee’s tax information in his address book record. For more information, see “Handling Other Deductions” on page 408.	1
<b>Contact-related plugins</b>		
IContactSystemPlugin	Search for contacts and retrieve contacts from an external system. For more information. For more information, see “Integrating with a Contact Management System” on page 501.	1
OfficialIdToTaxIdMappingPlugin	Determines whether ClaimCenter treats an official ID type as a tax ID for contacts. For more information, see “Official IDs Mapped to Tax IDs Plugin” on page 267.	1
<b>Other ClaimCenter plugins</b>		
IClaimNumGenAdapter	Generates unique claim numbers for new claims. Generates temporary claim numbers for draft claims. You must implement this plugin. However, ClaimCenter includes a demo version of this plugin. For more information, see “Claim Number Generator Plugin” on page 263.	1
IPolicySearchAdapter	(1) Searches policies to find likely ones to link to a claim and (2) retrieves full policy information for the selected policy. Used during the process of setting up a new claim. Searches can return zero rows if there is no integration to a policy system. You must implement this plugin. However, ClaimCenter includes a demo version of this plugin. For more information, see “Policy Search Plugin” on page 517.	1
IPolicyRefreshPlugin	Supports policy refresh. For more information, see “Policy Refresh Overview” on page 539.	1
IReinsurancePlugin	Retrieves reinsurance information from a policy administration system. For more information, see “Reinsurance Integration” on page 247.	1

ClaimCenter Plugin Interface	Description	Maximum enabled implementations
IApprovalAdapter	(1) Determines whether a user has any authority to submit something, typically a financial transaction. If so, determines whether the user has sufficient authority or requires further approval. If no authority, then rejects the request. (2) Determines who needs to give approval next. In general do not implement this plugin because approval rule sets are sufficient for most cases. By default, ClaimCenter uses internal logic to compare financial totals to authority limits. Also, by default uses the approval rule sets to determine the approving person. For more information, see "Approval Plugin" on page 264.	1
IGeocodePlugin	Geocoding support within ClaimCenter and ContactManager. For more information, see "Geographic Data Integration" on page 235.	1
IArchivingSource	Stores and retrieves archived claims from an external backing source. For more information, see "Archiving Integration" on page 575.	1
PolicyLocationSearchPlugin	Performs a policy location search in a policy system such as PolicyCenter. See "Policy Location Search Plugin" on page 539.	1
IAggregateLimitTransactionPlugin	Provides an optional capability to determine if a Transaction applies to an Aggregate Limit, beyond the usual values on the Aggregate Limit screen. See "Defining Aggregate Limits" on page 463 in the <i>Configuration Guide</i> .	1
<b>Other plugins common to multiple Guidewire applications</b>		
InboundIntegrationStartablePlugin	High performance inbound integrations, with support for multi-threaded processing of work items. See "Multi-threaded Inbound Integration" on page 279	Multiple
ITestingClock	Used for testing complex behavior over a long span of time, such as multiple billing cycles or timeouts that are multiple days or weeks later. This plugin is <i>for development (non-production) use only</i> . It programmatically changes the system time to accelerate the perceived passing of time within ClaimCenter.	1
	<b>WARNING</b> The testing clock plugin is for application development only. You must never use it on a production server. For more information, see "Testing Clock Plugin (Only For Non-Production Servers)" on page 265.	
IAddressAutocompletePlugin	Configures how address automatic completion and fill-in operate. For more information, see "Automatic Address Completion and Fill-in Plugin" on page 265.	1
IPhoneNormalizerPlugin	Normalizes phone numbers that users enter through the application and that enter the database through data import. For more information, see "Phone Number Normalizer Plugin" on page 265.	1
IEncryptionPlugin	Encodes or decodes a String based on an algorithm you provide to hide important data, such as bank account numbers or private personal data. ClaimCenter does not provide any encryption algorithm in the product. ClaimCenter simply calls this plugin implementation, which is responsible for encoding an unencrypted String or reversing that process. The built-in implementation of this plugin does nothing. For more information, see "Encryption Integration" on page 249.	Multiple
	The IEncryption plugin interface supports multiple implementations to support changes to encryption algorithms. See "Changing Your Encryption Algorithm Later" on page 254.	

ClaimCenter Plugin Interface	Description	Maximum enabled implementations
IBaseURLBuilder	Generates a base URL to use for web application pages affiliated with this application, given the HTTP servlet request URI ( <code>HttpServletRequest</code> ). For more information, see “Defining Base URLs for Fully-Qualified Domain Names” on page 269.	1
ManagementPlugin	The external management interface for ClaimCenter, which allows you to implement management systems such as JMX, SNMP, and so on. For more information, see “Management Integration” on page 259.	1
IScriptHost	<i>You can use the built-in version of this plugin using the Plugins registry APIs but do not attempt to implement your own version.</i> Get the current implementation using the Plugins registry and use it to evaluate Gosu code to provide context-sensitive data to other services. For example, property data paths defined with Gosu expressions. For example, this service could evaluate a String that has the Gosu expression <code>"policy.myFieldName"</code> at run time.	1
IProcessesPlugin	Creates new batch processes (new custom batch process sub-types). For more information, see “Custom Batch Processes” on page 587.	1
IStartablePlugin	Creates new plugins that immediately instantiate and run on server startup. For more information, see “What are Startable Plugins?” on page 271.  You can register multiple startable plugin implementations for different tasks.	Multiple
IPreupdateHandler	Implements your preupdate handling in plugin code rather than in the built-in rules engine. For more information, see “Preupdate Handler Plugin” on page 267.	1
IWorkItemPriorityPlugin	Calculates a work item priority. For more information, see “Work Item Priority Plugin” on page 267.	1
IActivityEscalationPlugin	Overrides the behavior of activity escalation instead of simply calling rule sets. For more information, see “Exception and Escalation Plugins” on page 270.	1
IGroupExceptionPlugin	Overrides the behavior of group exceptions instead of simply calling rule sets. For more information, see “Exception and Escalation Plugins” on page 270.	1
IUserExceptionPlugin	Overrides the behavior of user exceptions instead of simply calling rule sets. For more information, see “Exception and Escalation Plugins” on page 270.	1
ClusterBroadcastTransportFactory ClusterFastBroadcastTransportFactory ClusterUnicastTransportFactory	<i>For internal use only.</i> A plugin implementation is registered in the Plugins registry in the default configuration for these plugin interfaces. However, they are unsupported for customer use. Do not change settings in the Plugins registry for these interfaces. Do not use or implement these plugin interfaces.	n/a

# Authentication Integration

To authenticate ClaimCenter users, ClaimCenter by default uses the user names and passwords stored in the ClaimCenter database. Integration developers can optionally authenticate users against a central directory such as a corporate LDAP directory. Alternatively, use single sign-on systems to avoid repeated requests for passwords if ClaimCenter is part of a larger collection of web-based applications. Using an external directory requires a *user authentication plugin*.

To authenticate database connections, you might want ClaimCenter to connect to an enterprise database but need flexible database authentication. Or you might be concerned about sending passwords as plaintext passwords openly across the network. You can solve these problems with a *database authentication plugin*. This plugin abstracts database authentication so you can implement it however necessary for your company.

This topic discusses *plugins*, which are software modules that ClaimCenter calls to perform an action or calculate a result. For information on plugins, see “Plugin Overview” on page 163. For the complete list of all ClaimCenter plugins, see “Summary of All ClaimCenter Plugins” on page 181.

This topic includes:

- “Overview of User Authentication Interfaces” on page 187
- “User Authentication Source Creator Plugin” on page 189
- “User Authentication Service Plugin” on page 191
- “Deploying User Authentication Plugins” on page 194
- “Database Authentication Plugins” on page 195
- “ABAAuthenticationPlugin for ContactManager Authentication” on page 197

## Overview of User Authentication Interfaces

There are several mechanisms to log in to ClaimCenter:

- Log in by using the web application user interface.
- Log in to the server through Guidewire Studio.
- Authenticate RPCE web service calls to the current server.

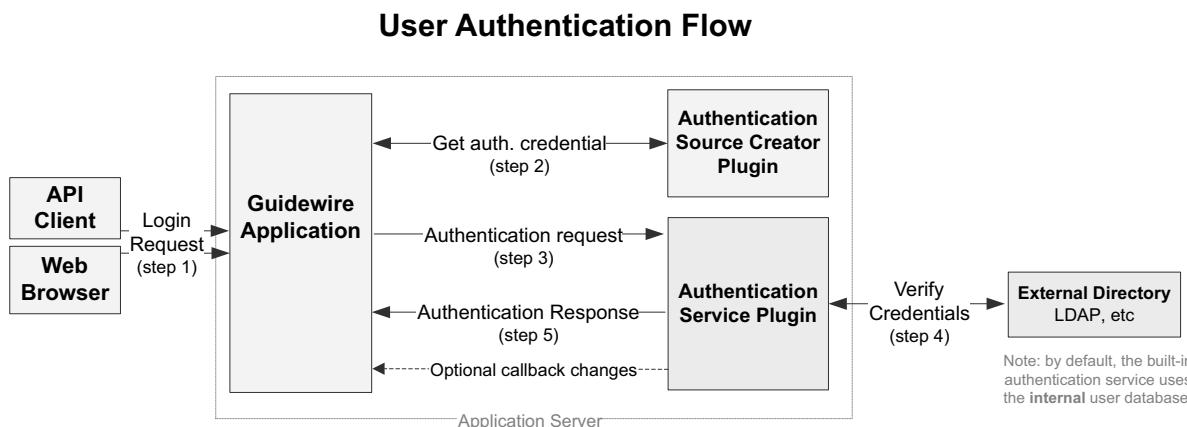
- Authenticate WS-I web service calls to the current server.

Authentication plugins must handle all types of logins other than WS-I web services. For authenticating WS-I web services, see the separate plugin interface in “WS-I Web Services Authentication Plugin” on page 56.

The authentication of ClaimCenter through the user interface is the most complex, and it starts with an initial request from another web application or other HTTP client. To pass authentication parameters to ClaimCenter, the HTTP request must submit the username, the password, and any additional properties as *HTTP parameters*. The parameters are name/value pairs submitted within HTTP GET requests as parameters in the URL, or using HTTP POST requests within the HTTP body (not the URL).

Additionally, authentication-related properties can be passed as *attributes*, which are similar to parameters except that they are passed by the servlet container itself, not the requesting HTTP client. For example, suppose Apache Tomcat is the servlet container for ClaimCenter. Apache Tomcat can pass authentication-related properties to ClaimCenter using attributes that were not on the requesting HTTP client URL from the web browser or other HTTP user agent. Refer to the documentation for your servlet container (such as Apache Tomcat) for details of how to pass attributes to a servlet.

The following diagram gives a conceptual view of the steps that occur during login to ClaimCenter:



The chronological flow of user authentication requests is as follows:

- An initial login request** – User authentication requests can come from web browsers and RPCE web service API calls (including command line tools). If the specified user is not currently logged in, ClaimCenter attempts to log in the user.
- An authentication source creator plugin extracts the request's credentials** – The user authentication information can be initially provided in various ways, such as browser-based form requests or API requests. ClaimCenter externalizes the logic for extracting the login information from the initial request and into a structured credential called an *authentication source*. This plugin creates the *authentication source* from information in *HTTPRequests* from browsers and return it to ClaimCenter. ClaimCenter provides a default implementation that decodes username/password information sent in a web-based form. Exposing this as a plugin allows you to use other forms of authentication credentials such as client certificates or a single sign-on (SSO) credentials. In the reference implementation, the PCF files that handle the login page set the username and password as attributes that the authentication source can extract from the request:
 

```
String userName = (String) request.getAttribute("username");
String password = (String) request.getAttribute("password");
```
- The server requests authentication using an authentication service** – ClaimCenter passes the *authentication source* to the *authentication service*. The authentication service is responsible for determining whether or not to permit the user to log in.
- The authentication service checks the credentials** – The built-in authentication service checks the provided username and password against information stored in the ClaimCenter database. However, a custom imple-

mentation of the plugin can check against external authentication directories such as a corporate LDAP directory or other single sign-on system.

5. **Authentication service responds to the request** – The authentication service responds, indicating whether to permit the login attempt. If allowed, ClaimCenter sets up the user session and give the user access to the system. If rejected, ClaimCenter redirects the user to a login page to try again or return authentication errors to the API client. This response can include connecting to the ClaimCenter *callback handler*, which allows the authentication service to search for and update user information as part of the login process. Using the callback handler allows user profile information and user roles to optionally be stored in an external repository and updated each time a user logs in to ClaimCenter.

## User Authentication Source Creator Plugin

The authentication source creator plugin (`AuthenticationSourceCreatorPlugin`) creates an *authentication source* from an HTTP request. The authentication source is represented by an `AuthenticationSource` object and is typically an encapsulation of username and password. However, it also contains the ability to store a cryptographic hash. The details of how to extract authentication from the request varies based on the web server and your other authentication systems with which ClaimCenter must integrate. This plugin is in the `gw.plugin.security` package namespace.

You must not use ClaimCenter RPCE web service (SOAP) APIs from within authentication plugins. Doing so requires authentication.

### Handling Errors in the Authentication Source Plugin

In typical cases, code in an *authentication source* plugin implementation operates only locally. In contrast, an *authentication service* plugin implementation typically goes across a network and must test authentication credentials with many more possible error conditions.

Try to design your authentication source plugin implementation to not need to throw exceptions.

If you do need to throw exceptions from your authentication source plugin implementation:

- Typically, the login user interface displays a default general message for failure to authenticate and ignores the text in the actual exception.
- The recommended way to display a custom message with an error is to throw the exception class `DisplayableLoginException`:

```
throw new DisplayableLoginException("The application shows this custom message to the user")
```

 Optionally you can subclass `DisplayableLoginException` to track specific different errors for logging or other reasons.
- Only if that approach is insufficient for your login exception handling, you can create an entirely custom exception type as follows.
  - a. Create a subclass of `javax.security.auth.login.LoginException` for your authentication source processing exception.
  - b. Create a subclass of `gw.api.util.LoginForm`.
  - c. In your `LoginForm` subclass, override the `getSpecialLoginExceptionMessage(LoginException)` method. ClaimCenter only calls this method for exception types that are not built-in. Your version of the method must return the `String` to display to the user for that exception type. Note that the only supported method of `LoginForm` for you to override is `getSpecialLoginExceptionMessage`.
  - d. Modify the `Login.pcf` page, which controls the user interface for that form. That PCF page instantiates `gw.api.util.LoginForm`. Change the PCF page to instantiate your `LoginForm` subclass instead of the default `LoginForm` class.

## Authentication Data in HTTP Attributes, Such as the ClaimCenter PCF Login Page

In the default of implementation ClaimCenter, login-related PCF files set the username and password as HTTP request attributes. HTTP request attributes are hidden values in the request. Do not confuse HTTP attributes with URL parameters. To extract data from the URL itself, see “Authentication Data in Parameters in the URL” on page 190.

The authentication source can extract these attributes from the request in the `HttpServletRequest` object:

```
String userName = (String) request.getAttribute("username");
String password = (String) request.getAttribute("password");
```

This plugin interface provides only one method, which is called `createSourceFromHTTPRequest`. The following example of how to implement this method:

```
public class BasicAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {
    public void init(String rootDir, String tempDir) {
    }

    public AuthenticationSource createSourceFromHTTPRequest(HttpServletRequest request)
        throws InvalidAuthenticationSourceData {
        AuthenticationSource source;

        // in real code, check for errors and throw InvalidAuthenticationSourceData if errors...
        String userName = (String) request.getAttribute("username");
        String password = (String) request.getAttribute("password");
        source = new UserNamePasswordAuthenticationSource(userName, password);
        return source;
    }
}
```

## Authentication Data in Parameters in the URL

If you need to extract parameters from the URL itself, use the `getParameter` method rather than the `getAttribute` method on `HttpServletRequest`.

For example from a URL with the syntax:

```
https://myserver:8080/cc/ClaimCenter.do?username=aapplegate&password=sheridan&objectID=12354
```

For example, use the following code:

```
package gw.plugin.security
uses javax.servlet.http.HttpServletRequest
uses com.guidewire.pl.plugin.security.AuthenticationSource
uses com.guidewire.pl.plugin.security.UserNamePasswordAuthenticationSource

class MyAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {

    override function createSourceFromHTTPRequest(request : HttpServletRequest) : AuthenticationSource {
        var source : AuthenticationSource
        var userName = request.getParameter( "username" )
        var password = request.getParameter( "password" )
        source = new UserNamePasswordAuthenticationSource( userName, password )

        return source
    }
}
```

## Authentication Data in HTTP Headers for HTTP Basic Authentication

The `ClaimCenter/java-api/examples` directory also contains a simple example implementation of a plugin that gets authentication information from HTTP basic authentication. Basic authentication encodes the data in HTTP headers for some web servers, such as IBM’s WebSeal.

The example implementation turns this information into an Authentication Source if it is sent encoded in the HTTP request header, for example if using IBM’s WebSeal. The plugin decodes a username and password stored in the header and constructs a `UserNamePasswordAuthenticationSource`.

This plugin interface provides only one method, which is called `createSourceFromHTTPRequest`. The following example shows how to implement this method:

```
public class BasicAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {  
    public void init(String rootDir, String tempDir) {}  
  
    public AuthenticationSource createSourceFromHTTPRequest(HttpServletRequest request)  
        throws InvalidAuthenticationSourceData {  
        AuthenticationSource source;  
        String authString = request.getHeader("Authorization");  
        if (authString != null) {  
            byte[] bytes = authString.substring(6).getBytes();  
            String fullAuth = new String(Base64.decodeBase64(bytes));  
            int colonIndex = fullAuth.indexOf(':');  
            if (colonIndex == -1) {  
                throw new InvalidAuthenticationSourceData("Invalid authorization header format");  
            }  
            String userName = fullAuth.substring(0, colonIndex);  
            String password = fullAuth.substring(colonIndex + 1);  
            if (userName.length() == 0) {  
                throw new InvalidAuthenticationSourceData("Could not find username");  
            }  
            if (password.length() == 0) {  
                throw new InvalidAuthenticationSourceData("Could not find password");  
            }  
            source = new UserNamePasswordAuthenticationSource(userName, password);  
            return source;  
        } else {  
            throw new InvalidAuthenticationSourceData("Could not find authorization header");  
        }  
    }  
}
```

You can implement this authentication source creator interface and store more complex credentials. If you do this, you must also implement an authentication service that knows how to handle these new sources. To do that, implement a user authentication service plugin (`AuthenticationServicePlugin`), described in the next section.

To view the source code to this example, refer to

[ClaimCenter/java-api/examples/plugins/authenticationsourcecreator/](#)

## User Authentication Service Plugin

An authentication service plugin (`AuthenticationServicePlugin`) implementation defines an external service that could authenticate a user. Typically, this would involve sending authentication credentials encapsulated in an *authentication source* and sending them to some separate centralized server on the network such as an LDAP server. This plugin is in the `gw.plugin.security` package namespace.

There are several mechanisms to log in to ClaimCenter:

- Log in by using the web application user interface.
- Log in to the server through Guidewire Studio.
- Authenticate RPCE web service calls to the current server.
- Authenticate WS-I web service calls to the current server.

Any `AuthenticationServicePlugin` implementation must handle all types of logins other than WS-I web services. For authenticating WS-I web services, see the separate plugin interface in “WS-I Web Services Authentication Plugin” on page 56.

For Guidewire Studio users and SOAP API calls, the credentials passed to this plugin are the standard `UserNamePasswordAuthenticationSource`, which contains a basic username and password. In addition, if you design a custom authentication source with data extracted from a web application login request, this plugin must be able to handle those credentials too.

There are three parts of implementing this plugin, each of which is handled by a plugin interface method:

- **Initialization** – All authentication plugins must initialize itself in the plugin’s `init` method.

- **Setting callbacks** – A plugin can look up and modify user information as part of the authentication process using the plugin's `setCallback` method. This method provides the plugin with a call back handler (`CallbackHandler`) in this method. Your plugin must save the callback handler reference in a class variable to use it later to make any changes during authentication.
- **Authentication** – Authentication decisions from a username and password are performed in the `authenticate` method. The logic in this method can be almost anything you want, but typically would consult a central authentication database. The basic example included with the product uses the `CallbackHandler` to check for the user within ClaimCenter. The JAAS example calls a JAAS provider to check credentials. Then, it looks up the user's public ID in ClaimCenter by username using the `CallbackHandler` to determine which user authenticated.

Every `AuthenticationServicePlugin` must support the default `UserNamePasswordAuthenticationSource` because this source is used by Guidewire Studio and RPCE web service API clients if connecting to the ClaimCenter server. A custom implementation must also support any other Authentication Sources that may be created by your custom *authentication source creator plugin*, if any.

Almost every authentication service plugin uses the `CallbackHandler` provided to it, if only to look up the public ID of the user after verifying credentials. Find the Javadoc for this interface in the class `AuthenticationServicePluginCallbackHandler`. This utility class includes four utility methods:

- `findUser` – Lets your code look up a user's public ID based on login user name
- `verifyInternalCredential` – Supports testing a username and password against the values stored in the main user database. This method is used by the default authentication service.
- `modifyUser` – After getting current user data and making changes, perhaps based on contact information stored externally, this method allows the plugin to update the user's information in ClaimCenter.

For more details of the method signatures, refer to the Java API Reference Javadoc for `AuthenticationServicePlugin`.

### Synchronizing User Roles

There is a `config.xml` configuration parameter `ShouldSyncUserRolesInLDAP`. If its value is `true`, the application synchronizes contacts with the roles they belong to after authenticating with the external authentication source.

### Authentication Service Sample Code

The product includes the following example authentication services in the directory `ClaimCenter/java-api/examples/plugins/authenticationservice`.

- **Default ClaimCenter authentication example** – This is the basic default service for ClaimCenter provided as source code. It checks the username and password against information stored in the ClaimCenter database and authenticates the user if a match is found.
- **LDAP authentication example** – `LDAPAuthenticationServicePlugin` is an example of authenticating against an LDAP directory.
- **JAAS authentication example** – `JAASAuthenticationServicePlugin` is an example of how you could write a plugin to authenticate against an external repository using JAAS. JAAS is an API that enables Java code to access authentication services without being tied to those services.

In the JAAS example, the plugin makes a `context.login()` call to the configured JAAS provider. The provider in turn calls back to the `JAASThCallbackHandler` to request credential information needed to make a decision. The plugin provides this callback handler to the JAAS provider. This is a JAAS object, different from the callback handler provided by ClaimCenter to the plugin as a ClaimCenter API. If the user cannot authenticate to JAAS, then the JAAS provider throws an exception. Otherwise, login continues.

## Error Handling in Authentication Service Plugins

Your code must be very careful to catch and categorize different types of errors that can happen during authentication. This is important for preserving this information for logging and diagnostic purposes. Additionally, ClaimCenter provides built-in features to clarify for the user what went wrong.

In general, the login user interface displays a default general message for failure to authenticate and ignores the text in the actual exception. However, `DisplayableLoginException` allows you to throw a more specific message to show to the user from your authentication service plugin code.

The following table lists exception classes that you can throw from your code for various authentication problems. In general, ClaimCenter classes in the `com.guidewire` hierarchy are internal and not for customer use. However, the classes in the `com.guidewire.pl.plugin` hierarchy listed below are supported for customer use.

Exception name	Description
<code>javax.security.auth.login.FailedLoginException</code>	Throw this standard Java exception if the user entered an incorrect password.
<code>com.guidewire.pl.plugin.security.InactiveUserException</code>	This user is inactive.
<code>com.guidewire.pl.plugin.security.LockedCredentialException</code>	Credential information is inaccessible because it is locked for some reason. For example, if a system is configured to permanently lock a user out after too many attempts. Also see related exception <code>MustWaitToRetryException</code> .
<code>com.guidewire.pl.plugin.security.MustWaitToRetryException</code>	The user tried too many times to authenticate and now has to wait for a while before trying again. The user must wait and retry at a later time. This is a temporary condition. Also see related exception <code>LockedCredentialException</code> .
<code>com.guidewire.pl.plugin.security.AuthenticationException</code>	Other authentication issues not otherwise specified by other more specific authentication exceptions.
<code>com.guidewire.pl.plugin.security.DisplayableLoginException</code>	This is the only authentication exception for which the login user interface uses the text of the actual exception to present to the user. For example, throw the exception as follows: <pre>throw new DisplayableLoginException("The application shows this custom message to the user")</pre>

You can make subclasses of the following exception classes:

- If `DisplayableLoginException` does not meet your needs, you can subclass the Java base class `javax.security.auth.login.LoginException`. To ensure the correct text displays for that class:
  - a. Create a subclass of `gw.api.util.LoginForm`.
  - b. In your `LoginForm` subclass, override the `getSpecialLoginExceptionMessage(LoginException)` method. ClaimCenter only calls this method for exception types that are not built-in. Your version of the method must return the `String` to display to the user for that exception type. Note that the only supported method of `LoginForm` for you to override is `getSpecialLoginExceptionMessage`.
  - c. Modify the `Login.pcf` page, which controls the user interface for that form. That PCF page instantiates `gw.api.util.LoginForm`. Change the PCF page to instantiate your `LoginForm` subclass instead of the default `LoginForm` class.
- You can also subclass the exception `guidewire.pl.plugin.security.DisplayableLoginException` if necessary for tracking unique types of authentication errors with custom messages.

## SOAP API Limitations

You must not use ClaimCenter RPCE web service (SOAP) APIs from within authentication plugins. Doing so would require authentication.

## SOAP API User Permissions and Special-Casing Users

Guidewire recommends creating separate a ClaimCenter user (or users) for SOAP API access. This user or set of users must have the minimum permissions allowable to perform SOAP API calls. Guidewire strongly recommends this user have few permissions or no permissions in the web application user interface.

From an authentication service plugin perspective, for those users you could create an exception list in your authentication plugin to implement ClaimCenter internal authentication for only those users. For other users, use LDAP or some other authentication service.

## Example Authentication Service Authentication

The following sample shows how to verify a user name and password against the ClaimCenter database and then modify the user's information as part of the login process.

```
public String authenticate(AuthenticationSource source) throws LoginException {
    if (source instanceof UserNamePasswordAuthenticationSource == false) {
        throw new IllegalArgumentException("Authentication source type " +
            source.getClass().getName() + " is not known to this plugin");
    }

    Assert.checkNotNullParam(_handler, "_handler", "Callback handler not set");

    UserNamePasswordAuthenticationSource uNameSource = (UserNamePasswordAuthenticationSource) source;

    Hashtable env = new Hashtable();

    env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "LDAP://" + _serverName + ":" + _serverPort);
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    String userName = uNameSource.getUsername();
    if (StringUtils.isNotBlank(_domainName)) {
        userName = _domainName + "\\" + userName;
    }

    env.put(Context.SECURITY_PRINCIPAL, userName);
    env.put(Context.SECURITY_CREDENTIALS, uNameSource.getPassword());

    try {
        // Try to login.
        new InitialDirContext(env);
        // Here would could get the result to the earlier and
        // modify the user in some way if you needed to
    } catch (NamingException e) {
        throw new LoginException(e.getMessage());
    }

    String username = uNameSource.getUsername();
    String userPublicId = _handler.findUser(username);
    if (userPublicId == null) {
        throw new FailedLoginException("Bad user name " + username);
    }
}

return userPublicId;
```

## Deploying User Authentication Plugins

Like other plugins, there are two steps to deploying custom authentication plugins:

1. Move your code to the proper directory.
2. Register your plugins in the Studio plugin editor.

First, move your code to the proper directory. Place your custom `AuthenticationSourceCreator` plugin in the appropriate subdirectory of `ClaimCenter/modules/configuration/plugins/authenticationsourcecreator/classes`. For example, if your class is `custom.authsource.myAuthSource`, put it in `../classes/custom/authsource/myAuthSource.class`. If your code depends on any Java libraries other than the ClaimCenter generated libraries, place the libraries in `../plugins/authenticationsourcecreator/lib/`.

Similarly, place your `AuthenticationService` plugin in the appropriate subdirectory of `ClaimCenter/modules/configuration/plugins/authenticationservice/classes`. For example, if your class is `custom.authservice.myAuthService`, put it in `../classes/custom/authservice/myAuthService.class`. Again, if your code depends on any Java libraries other than ClaimCenter generated libraries, place the libraries in `../plugins/authenticationservice/lib/`.

For ClaimCenter to find and use your custom plugins, you must register them in the plugin editor in Studio. See “*Messaging Editor*” on page 137 in the *Configuration Guide*. Remember that in the editor, the plugin name and the plugin interface name must match, even though Studio permits you to enter a different value for the plugin name. Otherwise, ClaimCenter does not recognize the plugin. The value for both must be the name of the plugin interface you are trying to use.

In the main `config.xml` file but not the plugin registry there is a `sessiontimeoutsecs` parameter that configures the duration of inactivity in seconds to allow before requiring reauthentication. This timeout period controls both the user’s web (HTTP) session and the user’s session within the ClaimCenter application. Users who connect to ClaimCenter using the API, for example, do not have an HTTP session, only an application session.

The following is an example of this inactive session timeout parameter:

```
<param name="SessionTimeoutSecs" value="10800"/>
```

## Database Authentication Plugins

You might want the ClaimCenter server to connect to an Enterprise database, but require a flexible database authentication system. Or, you might be concerned about sending passwords as plaintext passwords openly across the network. Solve either of these problems by implementing a *database authentication plugin*.

A custom database authentication plugin can retrieve name and password information from an external system, encrypt passwords, read password files from the local file system, or any other desired action. The resulting user-name and password substitutes into the database configuration file anywhere that  `${username}`  or  `${password}`  are found in the database parameter elements.

It is important to understand that *database authentication plugins* are different from *user authentication plugins*. Whereas user authentication plugins authenticate users into ClaimCenter (from the user interface or using API), database authentication plugins help the ClaimCenter server connect to its database server.

To implement a database authentication plugin, implement a plugin that implements the class `DBAuthenticationPlugin`, which is defined in the Java package `com.guidewire.pl.plugin.dbauth`.

This class has only one method you need to implement: `retrieveUsernameAndPassword`, which must return a username and password. Store the username and password combined together as properties within a single instance of the class `UsernamePasswordPair`.

The one method parameter for `retrieveUsernameAndPassword` is the name of the database (as a `String`) for which the application requests authentication information. This will match the value of the `name` attribute on the `database` or `archive` elements in your `config.xml` file.

If you need to pass additional optional properties such as properties that vary by server ID, pass parameters to the plugin in the Studio configuration of your plugin. Get these parameters in your plugin implementation using the standard `setParameters` method of `InitializablePlugin`. For more information, see “*Example Gosu Plugin*” on page 169.

The username and password that this method returns need not be a plaintext username and password, and it typically would **not** be plaintext. A plugin like this typically encodes, encrypts, hashes, or otherwise converts the data into a secret format. The only requirement is that your database (or an intermediate proxy server that pretends to be your database) knows how to authenticate against this username and password.

The following example demonstrates this method by pulling this information from a file:

```
public class FileDBAuthPlugin implements DBAuthenticationPlugin, InitializablePlugin {
    private static final String PASSWORD_FILE_PROPERTY = "passwordfile";
    private static final String USERNAME_FILE_PROPERTY = "usernamefile";

    private String _passwordfile;
    private String _usernamefile;

    public void setParameters(Map properties) {
        _passwordfile = (String) properties.get(PASSWORD_FILE_PROPERTY);
        _usernamefile = (String) properties.get(USERNAME_FILE_PROPERTY);
    }

    public UsernamePasswordPair retrieveUsernameAndPassword(String dbName) {
        try {
            String password = null;
            if (_passwordfile != null) {
                password = readLine(new File(_passwordfile));
            }
            String username = null;
            if (_usernamefile != null) {
                username = readLine(new File(_usernamefile));
            }
            return new UsernamePasswordPair(username, password);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    private static String readLine(File file) throws IOException {
        BufferedReader reader = new BufferedReader(FileUtil.getFileReader(file));
        String line = reader.readLine();
        reader.close();
        return line;
    }
}
```

Guidewire ClaimCenter includes an example database authentication plugin that simply reads a username and password from files specified in the `usernamefile` or `passwordfile` parameters that you define in Studio.

ClaimCenter replace the  `${username}`  and  `${password}`  values in the `jdbcURL` parameter with values returned by your plugin implementation. For this example, the values to use are the text of the two files (one for username, one for password).

For the source code, refer to the `FileDBAuthPlugin` sample code in the `examples.plugins.dbauthentication` package.

## Configuration for Database Authentication Plugins

For ClaimCenter to find and use your custom plugins, you must register them in the plugin editor in Studio. See “*Messaging Editor*” on page 137 in the *Configuration Guide*. Remember that in the editor, the plugin name and the plugin interface name must match, even though Studio permits you to enter a different value for the plugin name. Otherwise, ClaimCenter does not recognize the plugin. The value for both must be the name of the plugin interface you are trying to use. Add parameters as appropriate to pass information to your plugin.

ClaimCenter also supports looking up database passwords in a password file by setting “`passwordfile`” as a `<database>` attribute in your main `config.xml` file.

At run time, the username and password returned by your database authentication plugin replaces the  `${username}`  and  `${password}`  parts of your database initialization `String` values.

## ABAAuthenticationPlugin for ContactManager Authentication

Typical authentication from ClaimCenter to ContactManager uses the name and password defined in the `IAddressBookAdapter` plugin in Studio for the ClaimCenter application. See “Integrating ContactManager Using QuickStart” on page 46 in the *Contact Management Guide* for details. If you want to use unusual authentication or to hide authentication information in a separate file, you can use an almost identical class to the database authentication plugin interface (`DBAuthenticationPlugin`). The ContactManager version is called `ABAAuthenticationPlugin`.

To use standard authentication to connect ClaimCenter to ContactManager, do **not** implement `ABAAuthenticationPlugin` at all. Instead, define the username and password in the `IAddressBookAdapter` plugin configuration, as described in “Integrating ContactManager Using QuickStart” on page 46 in the *Contact Management Guide*.

### To use custom authentication from ClaimCenter to ContactManager

1. Write a `ABAAuthenticationPlugin` plugin implementation with the same design as described in the previous section about database authentication plugins, “Database Authentication Plugins” on page 195.
2. In Studio, register the `ABAAuthenticationPlugin` plugin implementation as a Gosu plugin. See “Plugin Overview” on page 163.
3. In Studio, in the Plugins editor for the `CCAddressBookPlugin`, add the parameter `authPlugin` with the value `true`.
4. The built-in address book plugin that connects to ContactManager (`IAddressBookAdapter`) detects the `ABAAuthenticationPlugin`. The plugin ignores the RPCE web service default connection authentication information (in the Studio user interface). Instead, ClaimCenter calls your `ABAAuthenticationPlugin` to get the user name and password.

For more information about implementing this plugin, see “Database Authentication Plugins” on page 195. The `ABAAuthenticationPlugin` plugin uses the same approach as the database authentication plugins.



# Document Management

ClaimCenter provides a user interface and integration APIs for creating documents, downloading documents, and producing automated form letters. You can integrate ClaimCenter with a separate external document management system (DMS) that stores the documents. Optionally, the external DMS can store the document metadata, such as the list of documents and their file types.

This topic includes:

- “Document Management Overview” on page 199
- “Choices for Storing Document Content and Metadata” on page 201
- “Document Storage Plugin Architecture” on page 203
- “Implementing a Document Content Source for External DMS” on page 204
- “Storing Document Metadata In an External DMS” on page 206
- “The Built-in Document Storage Plugins” on page 208
- “Asynchronous Document Storage” on page 209
- “APIs to Attach Documents to Business Objects” on page 210
- “Retrieval and Rendering of PDF or Other Input Stream Data” on page 211

**See also**

- For general information on plugins, which are an important part of document management in ClaimCenter, see “Plugin Overview” on page 163.
- “Document Production” on page 213

## Document Management Overview

The ClaimCenter user interface provides a **Documents** section within the claim file.

The **Documents** section lists documents such as letters, faxes, emails, and other attachments stored in a document management system (DMS).

## Document Storage Overview

ClaimCenter can store existing documents in a document management system. For example, you can attach outgoing notification emails, letters, or faxes created by business rules to an insured customer. You can also attach incoming electronic images or scans of paper witness reports, photographs, scans of police reports, or signatures. You can optionally integrate ClaimCenter with an external document management system (DMS).

Within the ClaimCenter user interface, you can find documents attached to a business object, and view the documents. You can also search the set of all stored documents.

Transfer of large documents through the application server to an external document storage system requires significant memory and resources. Even in the best case scenario of memory and CPU resources, the external document storage system (or intermediate network) may be slow. If so, synchronous actions with large documents may appear unresponsive to a ClaimCenter web user. To address these issues, ClaimCenter provides a way to asynchronously send documents to the document management system without bringing documents into application server memory. For maximum user interface responsiveness with an external document storage system, choose asynchronous document storage. In the default configuration, asynchronous document storage is enabled. For more information, refer to “Asynchronous Document Storage” on page 209.

It is important to understand the various types of IDs for a Document entity instance:

- `Document.PublicID` – the unique identifier for a single document in the DMS. If the DMS supports versions, the `PublicID` property does not change for each new version.
- `Document.DocUID` – the unique identifier for a single revision of a single document in the DMS. If the DMS supports versions, the `DocUID` property might change for each new version, depending on how your DMS works. See also the reference to extension properties later in this bullet list.
- **extension properties** – You can extend the base `Document` entity to contain version-related properties or other properties if it makes sense for your DMS.
- `Document.Id` – *This is an internal ClaimCenter property.* Never use the `Id` property to identify a document. Never get or set this property for document management.

## Document Production Overview

ClaimCenter can create new documents from forms, form letters, or other structured data. For example, notification emails, letters, or faxes created by business rules to an insured customer. Typically you would persist the document and attach the resulting new document to a business data object.

There are two ways to create documents:

- ClaimCenter can facilitate creation of documents on the user’s desktop using local application such as Microsoft Word or Excel. This is *client-side document production*.
- ClaimCenter can create some types of new documents from a server-stored template without user intervention. This is *server-side document production*.

For more information, see “Document Production” on page 213.

## Document Retrieval Mode Overview

There are several document retrieval modes, also known as response types:

- `URL` – A URL to a local content store to display the content. The `URL` response type is the recommended response type if your DMS supports it. The `URL` response type permits the highest performance for ClaimCenter.
- `DOCUMENT_CONTENTS` – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 211

For more information, see “Retrieving Documents” on page 204.

## Choices for Storing Document Content and Metadata

There are two separate kinds of data that a document storage system (DMS) processes for ClaimCenter. Before implementing a document management integration, you must decide where to store document content and metadata. The following table compares and contrasts these kinds of data.

Type of data	Description	Plugin interface that handles this data	Default behaviors
Document content	<p>Document content can be anything that represents incoming or outgoing documents.</p> <p>For example:</p> <ul style="list-style-type: none"> <li>• a fax image</li> <li>• a Microsoft Word file</li> <li>• a photograph</li> <li>• a PDF file</li> </ul> <p>A <i>document content source</i> is code that stores and retrieves the document content.</p>	IDocumentContentSource	<p>It is important to note that for the highest performance and data integrity, use an external DMS.</p> <p>In the initial ClaimCenter configuration, built-in classes implement asynchronous storage onto the local file system. For more information, see “The Built-in Document Storage Plugins” on page 208 and “Asynchronous Document Storage” on page 209. If you use an external DMS, you would not use local file system storage.</p>
Document metadata	<p>Document metadata describes the file:</p> <ul style="list-style-type: none"> <li>• name, which typically is the file name</li> <li>• MIME type</li> <li>• description</li> <li>• the full set of metadata on the DMS so users can search for documents.</li> <li>• which business objects are associated with each document</li> <li>• other fields defined by the application or customer extensions, such as recipient, status, or security type</li> </ul> <p>A <i>document metadata source</i> is code that manages and searches document metadata.</p>	IDocumentMetadataSource	<p>In the default ClaimCenter configuration, this plugin is disabled. In other words, in the Studio user interface for this plugin interface, the <b>Enabled</b> checkbox is unchecked.</p> <p>If this plugin is disabled, ClaimCenter stores the document metadata internally in the database, with one Document entity instance for each document. If you store the metadata internally in this way, searching for documents is very fast. Many DMS systems are not designed to withstand high capacity for external searching and metadata lookup. Additionally, if you use internal metadata, the metadata is available even if the DMS is offline.</p> <p>There is a built-in plugin implementation called LocalDocumentMetadataSource. This is for testing only and is unsupported in production. To correctly configure fast internal document metadata, uncheck the <b>Enabled</b> checkbox for this plugin.</p>

### Deciding Where to Store Document Content and Metadata

The best approach for configuring these plugins depends on whether you want to use an external DMS and how your company uses documents. For the highest performance and data integrity, use an external DMS.

#### To configure ClaimCenter for internal document storage (no external DMS):

1. For document content, use the default implementations of the IDocumentContentSource plugin. For more information, see “The Built-in Document Storage Plugins” on page 208 and “Asynchronous Document Storage” on page 209.
2. For document metadata, do not enable the IDocumentMetadataSource plugin. If this plugin is disabled, ClaimCenter stores the document metadata internally in the database, with one Document entity instance for

each document. Internal metadata storage makes document search fast. Additionally, if you use internal metadata, you can search document metadata even when the DMS is offline.

---

**WARNING** Choose your document metadata location carefully. See “Choosing Internal or External Metadata Permanently Affects Some Objects” on page 202.

---

**To configure ClaimCenter for an external DMS:**

1. For the document content, write your own implementation of the `IDocumentContentSource` plugin to store and retrieve content. Your plugin implementation stores and retrieves files using the proprietary API for your external DMS. If you use an external DMS, you can still use the built-in asynchronous document storage system. See “Asynchronous Document Storage” on page 209.
2. For the document metadata, the best solution depends on how you use documents:

---

**WARNING** Choose your document metadata location carefully. See “Choosing Internal or External Metadata Permanently Affects Some Objects” on page 202.

---

- If most documents in the DMS related to ClaimCenter exist because users upload or create documents in ClaimCenter, use internal metadata storage. Do not enable the `IDocumentMetadataSource` plugin in Studio. If this plugin is disabled, ClaimCenter stores the document metadata internally in the database, with one `Document` entity instance for each document. Internal metadata storage using the database makes document search very fast. Additionally, if you use internal metadata, the metadata is available when the DMS is offline.
- If most documents in the DMS related to ClaimCenter are added directly into the DMS without ClaimCenter, write your own implementation of the `IDocumentMetadataSource` plugin. Your plugin implementation manages and searches the metadata. If you want to share document metadata searching across multiple Guidewire applications, it is best to use external metadata.

## Choosing Internal or External Metadata Permanently Affects Some Objects

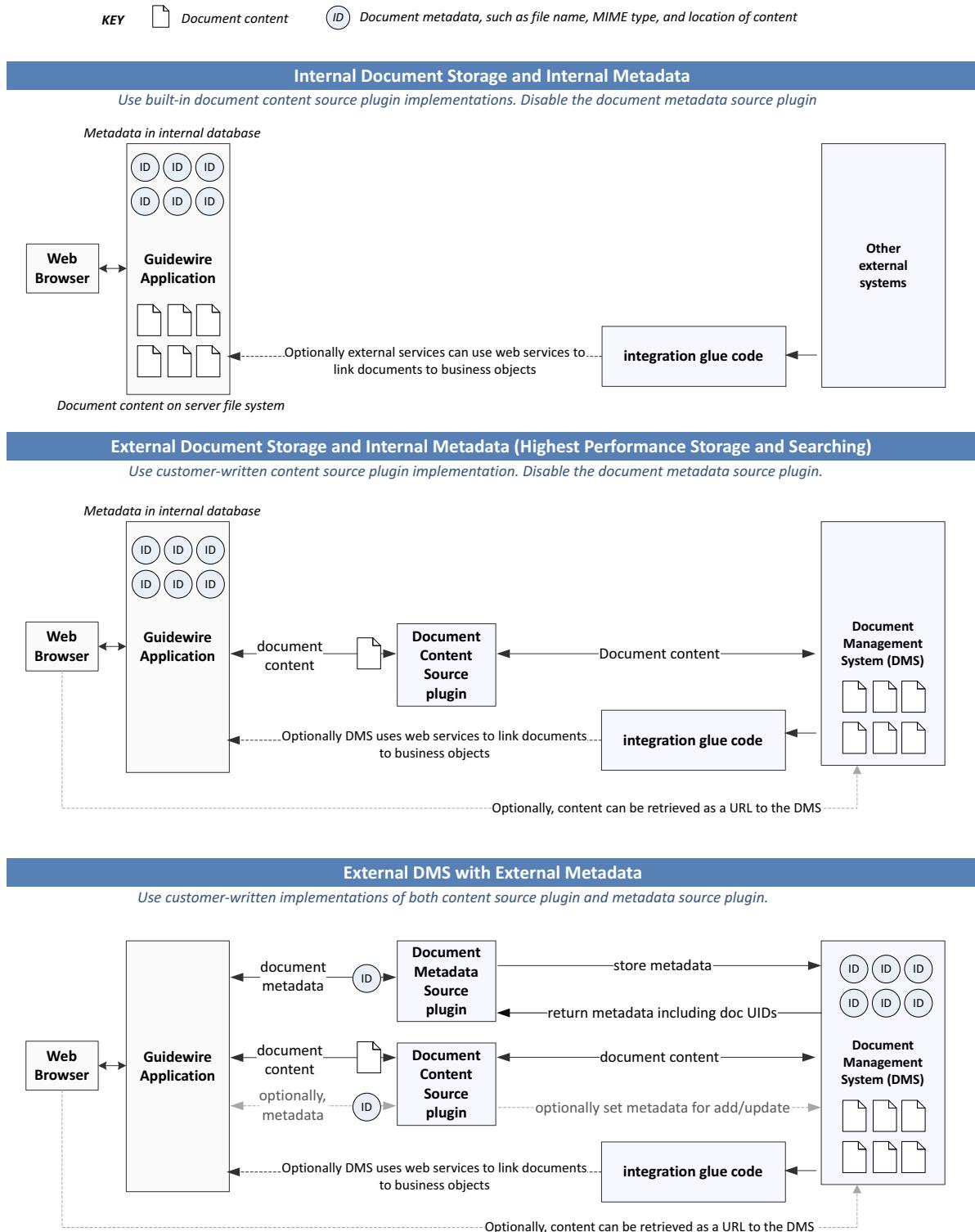
Whichever approach to handling document metadata you choose (internal or external), be aware that you must continue that approach permanently. Note objects and other similar text blocks reference a document within text as a specially-formatted hyperlink within the text itself. The format of this hyperlink within text blocks is slightly different based on whether ClaimCenter or a remote system handles the document metadata. If this quality changes from internal to external, or from external to internal, those document hyperlinks fail.

If you are considering this type of transition, contact Guidewire Customer Support for advice before proceeding.

# Document Storage Plugin Architecture

The following diagram summarizes implementation architecture options for document management.

## Document Storage Architecture Options



# Implementing a Document Content Source for External DMS

Document storage systems vary in how they transfer documents and how they display documents in the user interface. To support this variety, ClaimCenter supports multiple document retrieval modes called *response types*. For the list of options, see “Document Retrieval Mode Overview” on page 200.

To implement a new document content source, you must write a class that implements the `IDocumentContentSource` plugin interface. The following sections describe the methods that your class must implement.

## Adding Documents and Metadata

A new document content source plugin must fulfill a request to add a document by implementing the `addDocument` method. This method adds a new document object to the repository.

The method takes as arguments:

- the document metadata in a `Document` object
- the document contents as an `InputStream` object

The `addDocument` method also may copy some of the metadata contained in the `Document` entity into the external system. Independent of whether the document stores any metadata in the external system, the plugin must update certain metadata properties within the `Document` object:

- the method must set an implementation-specific document universal ID in the `Document.DocUID` property.
- the method must set the modified date in the `Document.DateModified` property.

The method must persist these changes to the ClaimCenter database.

The return value from the `addDocument` method is very important:

- If you do not enable and implement the `IDocumentMetadataSource` plugin, the return value is ignored.
- If you enable and implement the `IDocumentMetadataSource` plugin implementation:
  - If you stored metadata from the `Document` object in addition to document content, return `true`.
  - Otherwise, return `false`. ClaimCenter calls the registered `IDocumentMetadataSource` plugin implementation to store the metadata.

For related discussion, see “Deciding Where to Store Document Content and Metadata” on page 201.

## Error Handling

If your plugin code encounters errors during storage before returning from this method, throw an exception from this method. If document storage errors are detected later, such as for asynchronous document storage, the document content storage plugin must handle errors in an appropriate fashion. For example, send administrative e-mails or create new activities to investigate the problem. You could optionally persist the error information and design a system to track document creation errors and document management problems in a separate user interface for administrators.

## Retrieving Documents

A new document content source plugin must fulfill a request to retrieve a document. To fulfill this kind of request, implement the `getDocumentContentsInfo` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the external repository. Your code could use additional data model extensions on `Document` as needed.

The `getDocumentContentsInfo` method can read the `Document` object but must not modify the `Document` object.

The plugin must return its results within an instance of `gw.document.DocumentContentsInfo`, which is a simple object with the following properties:

- **Response type** – An enumeration in the `ResponseType` property indicates the type of the data:
  - URL – A URL to a local content store to display the content. The URL response type is the recommended response type if your DMS supports it. The URL response type permits the highest performance for ClaimCenter.
  - DOCUMENT\_CONTENTS – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 211
- **Hidden target frame as Boolean** – For response types other than DOCUMENT\_CONTENTS, the Boolean property `TargetHiddenFrame` specifies whether ClaimCenter opens the web page in a hidden frame within the user’s web browser. For example, if the document repository is naturally implemented as a small web page that contains JavaScript, set this property to `true`. JavaScript code from that page might open another web page as a popup window that displays the real document contents.
- **Contents as input stream** – The `InputStream` property contains the document contents in the form of an input stream, which is raw bytes as an `InputStream` object.

#### The `includeContents` Boolean Parameter

The `getDocumentContentsInfo` method has a Boolean parameter `includeContents`.

- If this parameter is `true`, include an input stream with the contents of the document. Set only the properties, `ResultResponseType`, `TargetHiddenFrame`, `InputStream`.
- If this parameter is `false`, only the response type is needed. Set only the property `ResultResponseType`. Do not set any other properties.

ClaimCenter calls some methods in a `IDocumentContentSource` plugin twice for each document: `isDocument` and `getDocumentContentsInfo`. Your plugin implementation must support this design. For the `getDocumentContentsInfo` method, the `includeContents` parameter is set to `false` on the first call and `true` on the second call. Your plugin must return a `DocumentContentsInfo` object with the same response type in response to both calls. Returning any other result is unsupported and results in undefined behavior.

#### MIME Type

The `DocumentContentsInfo.ResponseMimeType` property includes a MIME type. However, your `getDocumentContentsInfo` method must ignore this property. ClaimCenter sets the MIME type based on the `Document` properties after calling the `getDocumentContentsInfo` method. However, it is unsupported to rely on its value at the time that ClaimCenter calls `getDocumentContentsInfo`.

### Checking for Document Existence

A document content source plugin must fulfill requests to check for the exist of a document. To check document existence, ClaimCenter calls the plugin implementation’s `isDocument` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the repository. If the document with that document UID exists in the external DMS, return `true`. Otherwise return `false`.

The `isDocument` method must not modify the `Document` entity instance in any way.

ClaimCenter calls some methods in a `IDocumentContentSource` plugin twice for each document: `isDocument` and `getDocumentContentsInfo`. Your plugin implementation must support this design.

### Removing Documents

A new document content source plugin must fulfill a request to remove a document. To remove a document, ClaimCenter calls the plugin implementation’s `removeDocument` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the repository.

ClaimCenter calls this method to notify the document storage system that the document corresponding to the `Document` object will soon be removed from metadata storage. Your `IDocumentContentSource` plugin implementation can decide how to handle this notification. Choose carefully whether to delete the content, retire it, archive it, or another action that is appropriate for your company. Other `Document` entities may still refer to the same content with the same `Document.DocUID` value, so deletion may be inappropriate.

Be careful writing your document removal code. If the `removeDocument` implementation does not handle metadata storage, then a server problem might cause removal of the metadata to fail even after successful removing document contents.

If the `removeDocument` method removed document metadata from metadata storage and no further action is required by the application, return `true`. Otherwise, return `false`.

## Updating Documents and Metadata

A new document content source plugin must fulfill requests from ClaimCenter to update documents. To update documents, implement the `updateDocument` method. This method takes two arguments:

- a replacement document, as a stream of bytes in an `InputStream` object.
- a `Document` object, which contains document metadata. The `Document.DocUID` property identifies the document in the DMS.

At a minimum, the DMS system must update any core properties in the DMS that represent the change itself, such as the date modified, the update time, and the update user. If the document `UID` property implicitly changed because of the update, set the `DocUID` property on the `Document` argument. ClaimCenter persists changes to the `Document` object to the database.

Optionally, your document content source plugin can update other metadata from `Document` properties. The DMS must update the same set of properties as in your document content source plugin `addDocument` method. If your DMS has a concept of versions, you can extend the base `Document` entity to contain a property for the version. If you extend the `Document` entity with version information, `updateDocument` method sets this information as appropriate.

---

**IMPORTANT** During document update, you can optionally set `Document.DocUID` and any extension properties that represent versions. However, do not change the `Document.PublicID` property. On a related note, never get or set the `Document.ID` property. See “Document Storage Overview” on page 200 for related discussion.

---

The return value from the `updateDocument` method is very important:

- If you do not enable and implement the `IDocumentMetadataSource` plugin, the return value is ignored.
- If you enable and implement the `IDocumentMetadataSource` plugin implementation:
  - If you stored metadata from the `Document` object in addition to document content, return `true`.
  - Otherwise, return `false`. ClaimCenter calls the registered `IDocumentMetadataSource` plugin implementation to store the metadata.

For related discussion, see “Deciding Where to Store Document Content and Metadata” on page 201.

## Storing Document Metadata In an External DMS

In the default ClaimCenter configuration, the `IDocumentMetadataSource` plugin is disabled. In other words, in the Studio user interface for this plugin interface, the `Enabled` checkbox is unchecked. If this plugin is disabled, ClaimCenter stores the document metadata locally in the database, with one `Document` entity instance for each document. If you store the metadata locally in this way, searching for documents is very fast. Many DMS

systems are not designed for high capacity for external searching and metadata lookup. Additionally, if you use local metadata, the metadata is available even if the DMS is offline.

Optionally you can store document metadata in the document source content plugin so the content plugin handles the content and the metadata.

---

**IMPORTANT** It is critical to carefully read “Choices for Storing Document Content and Metadata” on page 201 before beginning implementation.

---

If you are sure you want to store document metadata in the DMS, write a class that implements the document metadata source (`IDocumentMetadataSource`) plugin interface. Typically this plugin sends the metadata to the same external system as the document content source plugin, but it does not have to do so.

It is important to understand the various of IDs for a `Document` entity instance:

- `Document.PublicID` – the unique identifier for a single document in the DMS. If the DMS supports versions, the `PublicID` property does not change for each new version.
- `Document.DocUID` – the unique identifier for a single revision of a single document in the DMS. If the DMS supports versions, the `DocUID` property might change for each new version, depending on how your DMS works. See also the reference to extension properties later in this bullet list.
- **extension properties** – You can extend the base `Document` entity to contain version-related properties or other properties if it makes sense for your DMS.
- `Document.Id` – *This is an internal ClaimCenter property.* Your implementation of the document metadata source plugin must never track documents by the ID (`Document.Id`) property.

---

**IMPORTANT** Always track a document by using the `PublicID` property, not the `Id` property.

---

The required plugin methods include:

- `saveDocument` – Persist document metadata in a `Document` object to the document repository. If document content source plugin methods `addDocument` or `updateDocument` return false to indicate they did not handle metadata, ClaimCenter calls the document metadata source plugin `saveDocument` method.
- `retrieveDocument` – Returns a completely initialized `Document` object with the latest information from the repository.
- `searchDocuments` – Return the set of documents in the repository that match the given set of criteria. This method’s parameters include a `RemotableSearchResultSpec` entity. This entity contains sorting and paging information for the PCF list view infrastructure to specify results.
- `removeDocument` – Remove a document metadata in a `Document` object from the document repository.
- `linkDocumentToEntity` – Associate a document with a ClaimCenter entity. Currently, the only supported entities for linking in this sense are `CheckSet`, `ReserveSet`, and `Activity`. The links between a document and a claim happen automatically if you set the `Document.claim` property.
- `getDocumentsLinkedToEntity` – Return all documents associated with a ClaimCenter entity. Currently, the only supported entities for linking in this sense are `CheckSet` and `ReserveSet`. The links between a document and a claim happen automatically if you set the `Document.claim` property.
- `isDocumentLinkedToEntity` – Check if a document is associated with a ClaimCenter entity. Currently, the only supported entities for linking in this sense are `CheckSet` and `ReserveSet`. The links between a document and a claim happen automatically if you set the `Document.claim` property.
- `unlinkDocumentFromEntity` – Remove the association between a document and a ClaimCenter entity. Currently, the only supported entities for linking in this sense are `CheckSet` and `ReserveSet`. The links between a document and a claim happen automatically if you set the `Document.claim` property.

#### See also

- For more details, refer to the API Reference Javadoc documentation for `IDocumentMetadataSource`.

- For more information about the document source content plugin, see “[Implementing a Document Content Source for External DMS](#)” on page 204.
- For more information about the included reference implementation, see “[The Built-in Document Storage Plugins](#)” on page 208.

## The Built-in Document Storage Plugins

ClaimCenter includes a reference implementation for the `IDocumentContentSource` plugin. This plugin implementation simply stores documents on the ClaimCenter server’s file system. By default, ClaimCenter stores the document metadata such as document names in the database. Document names are what people typically think of as document file names. You can override this behavior by implementing the `IDocumentMetadataSource` plugin.

To decide whether to use the built-in document storage plugin for your project, carefully read these topics:

- “[Choices for Storing Document Content and Metadata](#)” on page 201
- “[Document Storage Plugin Architecture](#)” on page 203

### Built-in Document Storage Directory and File Name Patterns

The document content and storage plugins use the `documents.path` parameter in their XML definition files to determine the storage location. If the value of that parameter is an absolute path, then the specific location is used. If the value is a relative path, then the location is determined by using the value of the temporary directory parameter (`javax.servlet.context.tempdir`) is used instead.

The temporary directory property is the root directory of the servlet container’s temp directory structure. In other words, this is a directory that servlets can write to as scratch space but with no guarantee that files persist from one session to another. The temporary directory of the built-in plugins in general are for testing only. Do not rely on temporary directory data in a production (final) ClaimCenter system, and even the built-in plugins are for demonstration only, as indicated earlier.

Documents are partitioned into subdirectories by claim and by relationships within the claim. The following table explains this directory structure:

Entity	Directory naming	Example
Claim	<code>Claim + claimNumber</code>	<code>/documents/Claim02-02154/</code>
Exposure	<code>claimDirectory/Exposure + exposurePublicID</code>	<code>/documents/Claim02-02154/ExposureABC:01/</code>
Claimant	<code>claimDirectory/Claimant + claimantPublicID</code>	<code>/documents/Claim02-02154/ClaimantABC:99/</code>

Notice that for exposure and claimant, the public ID is used, whereas for a claim the claim number is used.

ClaimCenter stores documents by the name you choose in the user interface as you save and add the file. If you add a file from the user interface and it would result in a duplicate file name, ClaimCenter warns you. The new file does not quietly overwrite the original file. If you create a document using business rules, the system forces uniqueness of the document name. The server appends an incrementing number in parentheses. For example, if the file name is `myfilename`, the duplicates are called `myfilename(2)`, `myfilename(3)`, and so on.

The built-in document metadata source plugin provides a unique document ID back to ClaimCenter. That document ID identifies the file name and relative path within the repository to the document.

For example, an exposure document’s repository-relative path might look something like this:

`/documents/claim02-02154/exposureABC:01/myFile.doc`

## Remember to Store Public IDs in the External System

In addition to the unique document IDs, remember to store the `PublicID` property for Guidewire entities such as `Document` in external document management systems.

ClaimCenter uses public IDs to refer to objects if you later search for the entities or re-populate entities during search or retrieval. If the public ID properties are missing on document entities during search or retrieval, the ClaimCenter user interface may have undefined behavior.

## Asynchronous Document Storage

Some document production systems generate documents slowly. When many users try to generate documents at the same time, multiple CPU threads compete and that makes the process even slower. One alternative is to create documents asynchronously so that user interaction with the application does not block waiting for document production.

Similarly, transfer of large documents through the application server to an external document storage system requires significant memory and resources. Even in the best case scenario of memory and CPU resources, the external document storage system (or intermediate network) may be slow. If so, synchronous actions with large documents may appear unresponsive to a ClaimCenter web user. To address these issues, ClaimCenter provides a system to asynchronously send documents to the document management system without bringing documents into application server memory. A separate thread on the batch server sends documents to your real document management system using the messaging system.

To implement this, ClaimCenter includes two software components:

- **Asynchronous document content source** – ClaimCenter includes an document content source plugin implementation `gw.plugin.document.impl.AsyncDocumentContentSource`. When it gets a request to send the document to the document management system, it immediately saves the files to a temporary directory on the local disk. In a clustered environment, map the local path to this temporary directory so that it references a server directory shared by all elements of the cluster. For example, map this to an SMB server.
- **Document storage messaging transport** – A separate task uses the ClaimCenter messaging system architecture to send documents as a separate task on the server. In a clustered environment, any server can create (submit) a new message, however only the batch server runs messaging plugins to actually send them. In the built-in configuration, the transport uses with the document storage plugin that you write. In a clustered environment, this means that your actual document content source implementation only runs on the batch server.

By default, the asynchronous document storage plugin is enabled. You specify to that plugin a class that implements `IDocumentContentSource` that you write to send the documents to the external system. By default, it simply uses the built-in document content storage implementation.

For maximum document data integrity and document management features, use a complete commercial document management system (DMS) and implement new `IDocumentMetadataSource` and `IDocumentContentSource` plugins to communicate with the DMS. Where at all possible, store the metadata in the DMS if it can natively store this information. In all cases, store the metadata in only one location: either in the DMS or ClaimCenter built-in metadata storage, but not both. Avoid duplicating metadata in the ClaimCenter database itself for production systems.

## Configuring Asynchronous Document Storage

### To configure asynchronous sending

1. Write a content source plugin as described earlier in this topic. Your plugin implementation must implement the interface `IDocumentContentSource`. However, it is important that you do not register it itself as the plugin implementation for the `IDocumentContentSource` interface in the Plugins Editor in Studio.

2. Instead, in Studio, click on Resources → Plugin → gw → plugin → IDocumentContentSource. This opens the editor for the current implementation (the default implementation) for the built-in version of this plugin.
3. In the Parameters tab, set the documents.path parameter to the desired local file path for your stored files. In a clustered environment, map the local path to this temporary directory so that it references a server directory shared by all elements of the cluster. For example, map this to an SMB server.
4. In the Parameters tab, set the SyncedContentSource parameter to class name of the class that you wrote that implements IDocumentContentSource.

#### To temporarily disable asynchronous sending

1. In Studio, click on Resources → Plugin → gw → plugin → IDocumentContentSource.
2. In the Parameters tab, find the TrySyncedAddFirst parameter.
3. Set this parameter to false.

#### To permanently disable asynchronous sending

1. In Studio, click on Resources → Plugin → gw → plugin → IDocumentContentSource.
2. Set the class name to your fully-qualified class name rather than the built-in AsyncDocumentContentSource implementation class.

## APIs to Attach Documents to Business Objects

### Gosu APIs to Attach Documents to Business Objects

ClaimCenter provides document-related Gosu APIs. Your business rules can add new documents to certain types of objects (including a claim) using document-related Gosu APIs.

To create a new object in Java, use the EntityFactory class discussed in “Accessing Entity and Typecode Data in Java” on page 631.

You can create a new document with Java code such as:

```
Document doc = (Document) EntityFactory.getInstance().newEntity(Document.class);
```

To implement document search, the searchDocument method might look similar to the following example. This example creates a new DocumentSearchResult for the result set and a new Document for a specific document summary:

```
DocumentSearchResult result =
(DocumentSearchResult)EntityFactory.getInstance().newEntity(DocumentSearchResult.class);

Document document = (Document)EntityFactory.getInstance().newEntity(Document.class);
document.setName("Test Document");
document.setType(DocumentType.DIAGRAM);

...
result.addToSummaries(document);
```

### Web Service APIs to Attach Documents to Business Objects

ClaimCenter provides a web services API for adding linking business objects (such as a claim) to a document. This allows external process and document management systems to work together to inform ClaimCenter after creation of new documents related to a business object. For example, in paperless operations, new postal mail might come into a scanning center to be scanned. Some system scans the paper, identifies with a business object, and then loads it into an electronic repository.

The repository can notify ClaimCenter of the new document and create a new ClaimCenter activity to review the new document. Similarly, after sending outbound correspondence such as an e-mail or fax, ClaimCenter can add a reference to the new document. After the external document repository saves the document and assigns an identifier to retrieve it if needed, ClaimCenter stores that document ID.

The ClaimAPI interface includes methods to add documents to ClaimCenter objects:

- `addDocument` – Add a new document to a claim, exposure, or other supported object that can have a document. See “Adding a Document from External Systems” on page 159.

## Retrieval and Rendering of PDF or Other Input Stream Data

You can display arbitrary `InputStream` content in a window. For example, you can display a PDF returned from code that returns PDF data as a byte stream (`byte[]`) from a plugin, encapsulated in a `DocumentContentsInfo` object.

Use the following utility method from Gosu including PCF files:

```
gw.api.document.DocumentsUtil.renderDocumentContentsDirectly(fileName, docInfo)
```

There are other utility methods on the `DocumentsUtil` object that may be useful to you. Refer to the API reference documentation in the Studio Help menu for more details.

For more information about PDF creation, see “Server-side PDF Licensing” on page 220.



# Document Production

ClaimCenter provides a user interface and integration APIs for creating documents, downloading documents, and producing automated form letters. The APIs include the ability to integrate the ClaimCenter document user interface with a separate corporate document management system that can store the documents and optionally the document metadata.

This topic includes:

- “Document Production Overview” on page 213
- “Document Template Descriptors” on page 221
- “Generating Documents from Gosu” on page 230
- “Template Web Service APIs” on page 233

**See also**

- For general information on plugins, which are an important part of document management in ClaimCenter, see “Plugin Overview” on page 163.
- For an overview of document management and storage, see “Document Management” on page 199

## Document Production Overview

ClaimCenter can create new documents from forms, form letters, or other structured data. For example, notification emails, letters, or faxes created by business rules to an insured customer. The resulting new document optionally can be attached to business data objects. ClaimCenter can create some types of new documents from a server-stored template without user intervention.

Users can create new documents locally from a template and then attach them to a claim or other ClaimCenter objects. This is useful for generating forms or form letters or any type of structured data in any file format. Or, in some cases ClaimCenter creates documents from a server-stored template without user intervention. For example, ClaimCenter creates a PDF document of an outgoing notification email and attaches it to the claim.

The `IDocumentProduction` interface is the plugin interface used to create new document contents from within the Guidewire application. It interfaces with a document production system in a couple of different ways,

depending on whether the new content can be returned immediately or requires delayed processing. These two modes are:

- **Synchronous production** – The document contents are returned immediately from the creation methods.
- **Asynchronous production** – The creation method returns immediately, but the actual creation of the document is performed elsewhere and the document may not exist for some time.

There are different integration requirements for these two types of document production.

There are several document response types:

- **URL** – A URL to a local content store to display the content. The URL response type is the recommended response type if your DMS supports it. The URL response type permits the highest performance for ClaimCenter.
- **DOCUMENT\_CONTENTS** – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 211

These settings are defined as the `ResponseType` property within `DocumentContentsInfo`, which is the return result from synchronous production methods.

The details of these plugins are discussed in “User Interface Flow for Document Production” on page 214. For locations of templates and descriptors, see “Template Source Reference Implementation” on page 229.

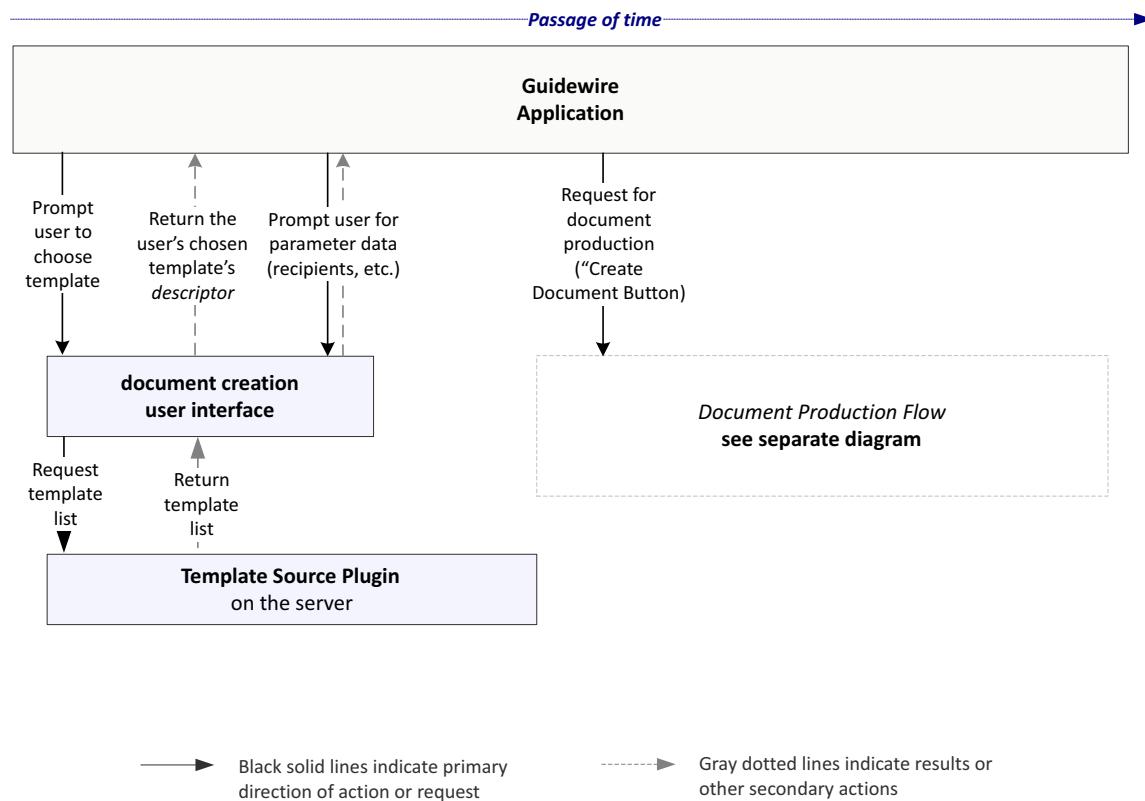
After document production, storage plugins store the document in the document management system.

## User Interface Flow for Document Production

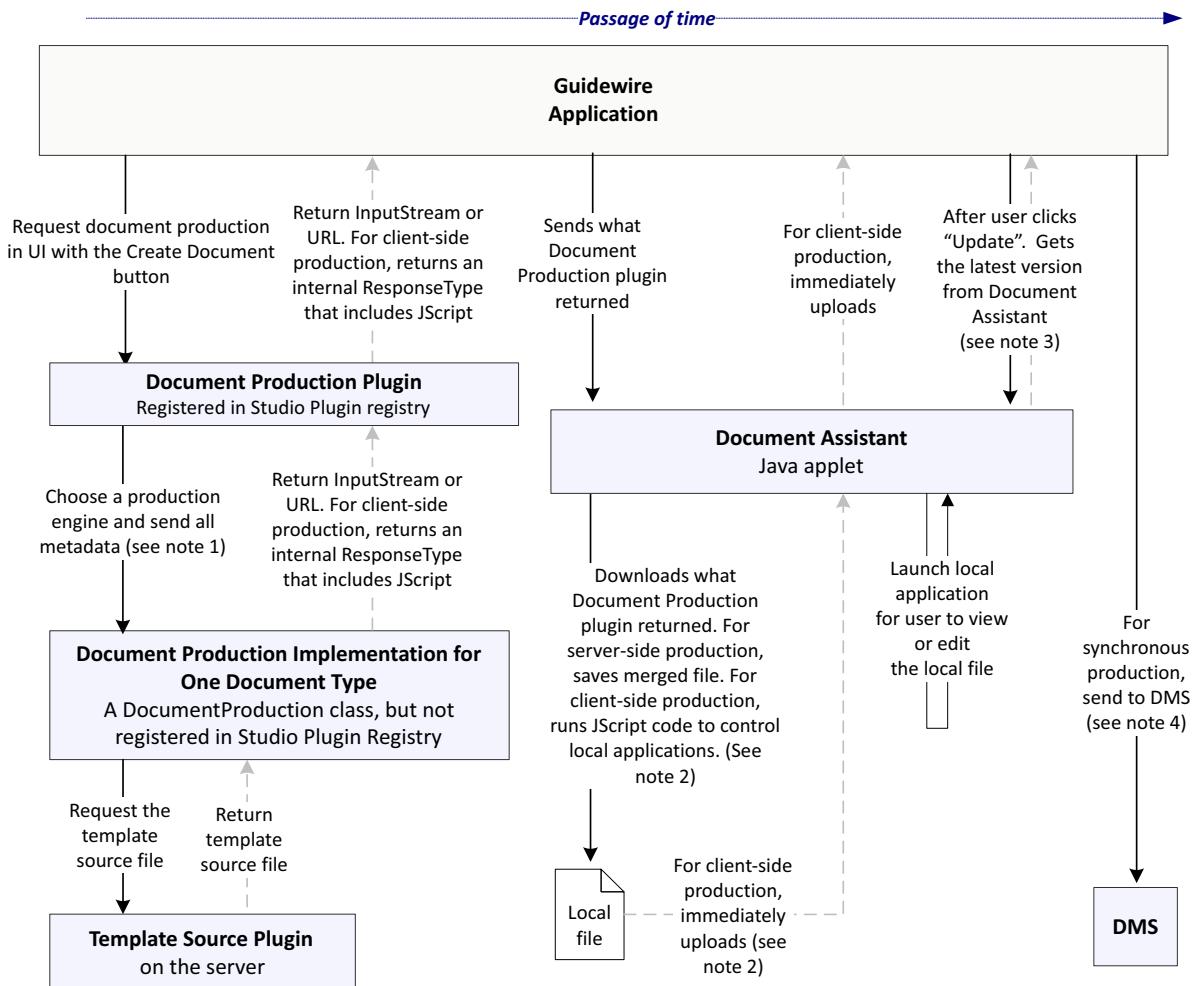
From the ClaimCenter user interface, create forms and letters and choose the desired template. You can select other parameters (some optional) to pass to the document production engine. ClaimCenter supports Gosu-initiated automatic document creation from business rules. The automatic and manual processes are similar, but the automatic document creation skips the first few steps of the process. The automated creation process skips some steps relating to parameters that the user chooses in the user interface. Instead, the Gosu code that requests the new document sets these values.

The following diagrams illustrates the interactions between ClaimCenter, the various plugin implementations, the Document Assistant, and an external document management system to manage the generation of a form or letter.

#### Interactive Document Creation UI Flow



## Document Production Flow



### Notes

1. The template descriptor for this request determines the document production implementation. It specifies a template handler String or a MIME type, both of which map to a document production implementation.
2. For client-side production, if Document Assistant is not present, the Guidewire application sends the web browser a JScript file. The JScript file does the equivalent of Document Assistant if you properly configure the PC and approve the request. To upload, the user interface prompts you to locate the merged file on the local file system.
3. If the Document Assistant is not present, you must manually locate the merged file on the local file system to upload.
4. For asynchronous production, after completion, the document production plugin sends the document to the DMS.



Primary direction of action or request



Results or other secondary actions

The chronological steps are as follows:

1. ClaimCenter invokes the document creation user interface to let you select a form or letter template.
2. The ClaimCenter document creation user interface requests a list of templates from the *template source*, which is a plugin that acts like a repository of templates.
3. The template source returns a list of templates.
4. The document creation user interface displays a list of templates and lets you choose one, perhaps after searching or filtering among the list of templates. Searching and filtering uses the template metadata, which is information about the templates themselves.
5. After you select a template, the document creation user interface returns the *template descriptor* information for the chosen template to ClaimCenter.
6. ClaimCenter prompts you through the document creation user interface for other document production parameters for the document merge. For example, choose the recipient of a letter or other required or optional properties for the template. After you enter this data, the user interface returns the parameter data to ClaimCenter.
7. ClaimCenter chooses the appropriate document production plugin from values in the template descriptor. The descriptor file can indicate a specific document production plugin or let one be chosen based on the MIME type of the file. The document production plugin gets a template descriptor and the parameter data values.
8. The document production plugin obtains the actual template file from the template source.
9. The document production plugin takes whatever steps are necessary to launch an editor application or other production engine for merging the template file with the parameter data. ClaimCenter includes document production plugins that process Microsoft Word files, Adobe Acrobat (PDF) files, Microsoft Excel files, Gosu templates, and plain text files. For Gosu templates, the Gosu document production plugin executes the Gosu template using the parameter data.

The result of this step is a *merged document*. It is possible to produce documents using applications on the user's computer, a process known as *client-side document production*. For example, the built-in production plugins use client-side production for Microsoft Word and Microsoft Excel documents.
10. If the document was created from the user interface, ClaimCenter automatically downloads the file to your local machine if the configuration parameter `AllowDocumentAssistant` is enabled. The file opens in the appropriate application on the desktop. You can print the file, and if the editor application supports editing, you can change the file and save it in the editor application. This describes the reference implementation, but the user interface flow can work differently if you customize the PCF files for a different workflow.
11. If the file is on your PC, you upload the completed, locally merged document, or you can choose an alternative document to upload. You can then specify some additional parameters about the document. If the configuration parameter `AllowDocumentAssistant` is enabled, the upload step by the Guidewire Document Assistant supplies the location of the locally merged document. If `AllowDocumentAssistant` is disabled or the Guidewire Document Assistant control is not installed, then you must browse manually to the local location of the merged document to upload.
12. Additionally, depending on the type of document production, the production engine could add the document to the document storage system. The production system or the storage system could notify ClaimCenter using plugin code or from an external system using the web services API. For more information about document storage, see "Document Management Overview" on page 199.

The preceding steps describe the reference implementation, but you can configure many parts of the flow to suite your needs. You can configure the user interface flow to work differently by modifying the user interface PCF configuration files. In some cases, you can configure the flow by modifying the document-related plugins.

## Document Production Plugins

As mentioned in “User Interface Flow for Document Production” on page 214, ClaimCenter supports two types of document production:

- *synchronous production* – the document contents return immediately
- *asynchronous production* – the creation method returns immediately but creation happens later

These two types of document production have different integration requirements.

For synchronous production, ClaimCenter and its installed *document storage* plugins are responsible for persisting the resulting document, including both the document metadata and document contents. In contrast, for asynchronous document creation, the *document production* (`IDocumentProduction`) plugins are responsible for persisting the data.

The `IDocumentProduction` plugin is an interface to a document creation system for a certain type of document. The document creation process may involve extended workflow and/or asynchronous processes, and it may depend on `Document` entity or set properties in the `Document`.

There are some additional related interfaces that assist the `IDocumentProduction` plugin:

- `IDocumentTemplateSource` and `IDocumentTemplateDescriptor` – Encapsulate the basic interface for searching and retrieving templates that describe the document to be created.  
The descriptive information includes the basic metadata (name, MIME type, and so on) and a pointer to the template content. Specifically, `IDocumentTemplateDescriptor` describes the templates used to create documents and `IDocumentTemplateSource` plugin actually lists and retrieves the document templates
- `IPDFMergeHandler` – Used in creation of PDF documents. View the built-in implementation to set parameters in the default implementation.

The `IDocumentProduction` plugin has two main methods: `createDocumentSynchronously` and `createDocumentAsynchronously`.

- The `createDocumentSynchronously` method returns document contents as:
  - `URL` – A URL to a local content store to display the content. The `URL` response type is the recommended response type if your DMS supports it. The `URL` response type permits the highest performance for ClaimCenter.
  - `DOCUMENT_CONTENTS` – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 211

These settings are defined as the `ResponseType` property within `DocumentContentsInfo`, which is the return result from synchronous production methods.

For documents created synchronously, the caller of the `createDocumentSynchronously` method must pass the contents to the document content plugin for persistence. The `Document` parameter can be modified if the `DocumentProduction` plugin wants to set any properties for persistence in the database.

- The `createDocumentAsynchronously` method returns the document status, such as a status URL that could display the status of the document creation. In the reference implementation, none of the built-in document production plugins support asynchronous document creation. The default user interface does not actually use the results of the `createDocumentAsynchronously` method. To support this type of status update, customize the user interface PCF files to generate a new user interface. Also, customize the included `DocumentProduction` Gosu class that generates documents from Gosu. By default, `DocumentProduction` does not use the result of the `createDocumentAsynchronously` method.
- For documents created asynchronously, your `createDocumentAsynchronously` method must put the newly created contents into the DMS. Next, your external system can use web service APIs to add the document to notify ClaimCenter that the document now exists. See “Web Service APIs to Attach Documents to Business Objects” on page 210.

If your code to add the document is running within the server in local Gosu or Java code, do not call the SOAP APIs to call back the same server. Calling back to the same server using SOAP is not generally safe. Instead, use domain methods on the entity to add the entity. For example:

```
newDocumentEntity = Claim.addDocument()
```

In either case, immediately throw an exception if any part of your creation process fails.

It is possible to produce documents using applications on the user's computer, a process known as *client-side document production*. For example, the built-in production plugins use client-side production for Microsoft Word and Microsoft Excel documents. The Document Assistant can call the local application to open the file and merge in necessary properties. On Windows, the Document Assistant sends JScript that can control local applications.

For more information about the built-in production plugins, see "Built-in Document Production Plugins" on page 230.

For more details on specific methods of these plugins, refer to the Java API Reference Javadoc. In the Javadoc, there are two versions of the `IDocumentProduction` interface. There is a base class Guidewire platform version with ".pl." in the fully-qualified package name. To implement the plugin, you must implement the other version, the ClaimCenter-specific version, which has ".cc." in its fully-qualified package name.

## Configuring Document Production and MIME Types

Document production configuration is defined by the registry for the `IDocumentProduction` plugin in the Plugins editor in Studio. The registry includes a list of parameters that define *template types* and their corresponding `IDocumentProduction` plugin implementations. In the registry, a template type is either a MIME type that ClaimCenter recognizes or a text value that you provide to define a template type.

### Defining a Template Type for Document Production

You define template types by adding parameters to the list in the registry for the `IDocumentProduction` plugin. Each parameter in the list specifies the name of a template type and a class that implements the `IDocumentProduction` plugin interface to handle production for that template type. Separately, you must deploy the implementation class to the `ClaimCenter/modules/configuration/plugins/document/classes` directory and necessary libraries to the `ClaimCenter/modules/configuration/plugins/document/lib` directory.

### Built-in Implementations of the Document Production Plugin

A typical document production configuration includes parameters for template types that reference the built-in `IDocumentProduction` implementations for common file types. For example, the template type named `application/msword` specifies

`com.guidewire.cc.plugin.document.internal.WordDocumentProductionImpl` as the implementation class.

Refer to the plugin registry for the `IDocumentProduction` plugin to see additional built-in plugin implementations.

### How ClaimCenter Determines which Plugin Implementation to Use for Document Production

ClaimCenter determines which `IDocumentProduction` implementation to use to produce a document from a specific template by following this procedure:

1. ClaimCenter searches for a non-MIME-type entry in the plugin parameter list that matches the `documentProductionType` property of the template. If a match is found, ClaimCenter proceeds with document production.
2. If no match is found, ClaimCenter searches for a MIME-type entry in the plugin parameter list that matches the  `mimeType` property of the template. If a match is found, ClaimCenter proceeds with document production.
3. If no match is found, document production fails.

You can replace the default dispatching implementation with your own `IDocumentProduction` implementation.

## Server-Side PDF Document Production

ClaimCenter supports server-side PDF production but not client-side PDF production.

### Adding a custom MIME type for Document Production

Adding a custom MIME type that ClaimCenter recognizes requires several steps.

1. In `config.xml`, add the new MIME type to the `<mimetypemapping>` section. The information for each `<mimetype>` element contains these attributes:
  - `name` – The name of the MIME type. Use the same name as in the plugin registry, such as `text/plain`.
  - `extension` – The file extensions to use for the MIME type.  
If more than one extension applies, use a pipe symbol (" | ") to separate them. ClaimCenter uses this information to map between MIME types and file extensions. To map from a MIME type to a file extension, ClaimCenter uses the first extension in the list. To map from file extension to MIME type, ClaimCenter uses the first `<mimetype>` entry that contains the extension.
  - `icon` – The image file for documents of this MIME type. At runtime, the image file must be in the `SERVER/webapps/cc/resources/images` directory.

2. Add the MIME type to the configuration of the application server, if required.

How you add a MIME type depends on the brand of application server. For Tomcat, you configure MIME types in the `web.xml` configuration file by using `<mime-mapping>` elements. Assure the MIME type that you need is not in the list already you try to add it.

#### See also

- For basic instructions, see “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.
- For more about information built-in `IDocumentProduction` implementations classes, see “Built-in Document Production Plugins” on page 230.

## Server-side PDF Licensing

In the ClaimCenter reference implementation, the server-side PDF production software is made by a company called Big Faceless Org (BFO). Without a license, the generated documents contain a large watermark that reads “DEMO” on the face of each generated page, rather than preventing document creation entirely. To remove the watermark, you must obtain a license key through Guidewire Customer Support.

All server-side PDF production licenses are licensed per server *CPU*, not per server.

With your copy of ClaimCenter, you are entitled to four CPU licenses for PDF production. However, you must still go through a process to obtain the keys associated with your licenses. In addition, you may purchase additional licenses through the same process.

To obtain a license key:

1. Contact your Customer Support Partner or email [support@guidewire.com](mailto:support@guidewire.com) with your request for a PDF production license for ClaimCenter.
2. A support engineer sends you an Authorization Form.
3. Fill out the Authorization Form, including the number of CPUs to use. For multi-CPU servers, include the total number of CPUs.
4. Fax the filled-out form to Guidewire prior to issuing the license.
5. The Guidewire support engineer requests license keys from the appropriate departments.
6. Once the license keys are obtained, Guidewire emails the designated customer contact (per the information on the form) with the license information.

If you have additional questions about this process, email [support@guidewire.com](mailto:support@guidewire.com).

## Configuration

The default configuration appears in the Plugins registry in Studio. Navigate to **configuration** → **config** → **Plugins** → **registry** and click on **IPDFMergeHandler**. For information about using the plugins editor, see “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.

Ensure the implementation class is set to `gw.plugin.document.impl.BFOMergeHandler`, which is the default.

In the list of plugin parameters, set the following two parameter values:

- Set parameter `BatchServerOnly` value to `true` for typical usage. The `BatchServerOnly` parameter determines whether PDF production happens on each server or solely on the batch server.
- Set parameter `LicenseKey` value to your server key. The `LicenseKey` value is empty in the default configuration. You must acquire your own BFO licenses from customer support.

ClaimCenter uses the `IPDFMergeHandler` plugin interface only for server-side PDF production in the default implementation of the `IDocumentProduction` interface.

## Document Template Descriptors

A *document template descriptor* describes an actual document template, such as a Microsoft Word mail merge template, an Adobe PDF form, or a Gosu template. An interface called `IDocumentTemplateDescriptor` defines the API interface for an object that represents a document template descriptor. In most cases, it is best for you to use the built-in implementation of this interface. The built-in implementation reads template information from a standard XML file. You can modify the XML file if desired. However, you probably do not need to modify the code that reads or writes this XML file.

The template descriptor contains four different kinds of information:

- **Template metadata** – Template metadata is metadata about the template itself, not the file. Template metadata includes the template ID, the template name, and the calendar dates that limit the availability of the template.
- **Document metadata defaults** – Document metadata defaults are attributes that are applied to documents after their creation from the template, or as part of their creation, for example the default document status.
- **Context objects** – Context objects are objects that can be inserted into the document template, or have properties extracted from them before inserting them into the document template. For example, an email document template might include To and CC recipients as context objects, each of which is of the type `Contact`. The context objects include default values to be inserted, as well as a set of legal alternative values for use in the ClaimCenter document creation user interface. A context object is an object attached to the merge request that can either be inserted into the document or certain properties within the object extracted from it.
- **Property names and values to merge** – Each descriptor defines a set of template field names and values to insert into the document template, including optional formatting information. Effectively, this describes which ClaimCenter data values to merge into which fields within the document template. For example, an email document template might have To and CC recipients as context objects called `To` and `CC` of type `Contact`. The template might have a context object called `InsuredName` that extract the value `To.DisplayName`.

The `IDocumentTemplateDescriptor` interface is closely tied to the XML file format which corresponds to the default implementation of the `IDocumentTemplateSerializer` interface. The `IDocumentTemplateDescriptor` API consists entirely of getters, plus one setter (for `DateModified`).

For locations of templates and descriptors, see “Template Source Reference Implementation” on page 229.

### Template Descriptor Fields for Metadata About Each Template

The following are property names and type information for `IDocumentTemplateDescriptor`, broken down by the four categories of information. For each property, the property is optional within the XML files, unless stated as required.

- `templateId` – Required. The unique ID of the template.
- `name` – Required. A human-readable name for the template.
- `identifier` – An additional human-readable identifier for the template. This often corresponds to a well-known domain-specific document code, to indicate (for example) which state-mandated form this template corresponds to.
- `scope` – Required. The contexts in which this template may be used. Possible values are
  - `ui` – the document template must only be used from the document creation user interface
  - `rules` – the document template must only be used from rules or other Gosu and must not appear in a list the user interface template.
  - `all` – the document template may be used from any context
- `description` – Required. A human-readable description of the template and/or the document it creates.
- `password` – If present, holds the password which is required for the user to create a document from the template. May not be supported by all document formats (for example, not supported for Gosu templates). In Microsoft Word, use the option to **Protect Document...** and select the **Forms** option. Then use the same password in the descriptor file as used to protect the document. To merge claim data into the form, ClaimCenter needs to unlock it, merge the data, and relock the form. ClaimCenter checks whether a form is locked and attempts to unlock it (and later relock it) using the password provided.
- `mimeType` – Required. The type of document to create from this document template. In the built-in implementation of document source, this determines which `IDocumentProduction` implementation to use to create documents from this template.
- `documentProductionType` – If present, a specified document production type can be used to control which implementation of `IDocumentProduction` must be used to create a new document from the template. This is not the only way to select a document production plugin implementation. You can also use a MIME type.
- `dateModified` – The date the template was last modified. In the default implementation, this is set from the information on the XML file itself. Both getter and setter methods exist for this property so that the date can be set by the `IDocumentTemplateSource` implementation. This property is not present in the XML file. However, the built-in implementation of the `IDocumentTemplateDescriptor` interface generates this property.
- `dateEffective, dateExpiration` – Required. The effective and expiration dates for the template. If you search for a template, ClaimCenter displays only those for which the specified date falls between the effective and expiration dates. However, this does not support different versions of templates with the same ID. The ID for each template must be unique. You can still create documents from templates from Gosu (from PCF files or rules) independent of these date values. Gosu-based document creation uses template IDs but ignores effective dates and expiration dates.
- `keywords` – A set of keywords search for within the template. Delimit keywords with the pipe (|) symbol in the XML file.
- `requiredPermission` – The code of the `SystemPermissionType` value required for the user to see and use this template. Templates for which the user does not have the appropriate permission do not appear in the user interface. This setting does not prevent creation of a document by Gosu (PCF files or rules).
- `getMetadataPropertyNames` – This method returns the set of extra metadata properties which exist in the document template definition. This is used in conjunction with `getMetadataPropertyValue()` as a flexible extension mechanism. You can add arbitrary new fields to document template descriptors. ClaimCenter passes new properties to the internal entities that display document templates in the user interface. Also, if the extra property names correspond to properties on the `Document` entity, the ClaimCenter passes values to documents created from the template.
- `getMetadataPropertyValue(String propName)` – See `getMetadataPropertyNames()`.
- `mailmergetype` – Optional configuration of pagination of client-side Microsoft Word production. By default, ClaimCenter uses Microsoft Word *catalog pagination*, which correctly trims the extra blank page at the end. However, catalog pagination forbids template substitution in headers and footers. In contrast, *standard pagination* allows template substitution in headers and footers.

*nation* adds a blank page to the end of the file but enables template substitution in headers and footers. Set this attribute to the value `catalog` to use catalog pagination. To use standard pagination, do not set this attribute.

### Template Descriptor Fields and Defaults for Document Metadata

The following template descriptor fields define the defaults for document type and security types:

- `templateType` – Corresponds to the `DocumentType` typelist. Documents created from this template have their `type` fields set to this value. You use a different name in the XML file for this property compared to how it appears in the descriptor. In the XML this property is `type` instead of `templateType`. This is the only property in the XML with a name difference like this.
- `defaultSecurityType` – Security type in the `DocumentSecurityType` typelist. This is the security type that becomes the default value for the corresponding document metadata fields for documents created using this template

### Template Descriptor Fields and Context Objects

As you write templates that include Gosu expressions, you need to reference business data such as the current `Claim` entity. Reference entities within templates as objects called *context objects*. Context objects create variables that form field expressions can refer to by name.

In addition to context objects that you define, form fields can always reference the claim using the `claim` symbol, for example, `claim.property` or `claim.methodName()`.

Also, if a related exposure or claimant is selected in the document creation screen, that object also is available through the symbol `RelatedTo`. For example, `RelatedTo.DisplayName`.

Only having access to one or two high-level root objects would get challenging. For example, suppose you want to address a claim acknowledgement letter to the main contact on the claim. Without context objects, each form field would have to repeat the same prefix (`Claim.MainContact.`) many times:

```
<FormField name="ToName">Claim.MainContact.DisplayName</FormField>
<FormField name="ToCity">Claim.MainContact.PrimaryAddress.City</FormField>
<FormField name="ToState">Claim.MainContact.PrimaryAddress.State</FormField>
...

```

This could become very tedious and hard to read, especially with complex lengthy prefixes. Fortunately, you can simplify template code by creating a context object that refers to the intended recipient. Each context object must have two attributes: a unique name (such as "To") and a type (as a string, such as "Contact"; see the following discussion for the legal values). In addition, each context object must contain a `DefaultObjectValue` tag. This tag can contain any valid Gosu expression that identifies the default value to use for this `ContextObject`. You can construct a context object for your recipient as follows:

```
<ContextObject name="To" type="Contact">
<DefaultObjectValue>Claim.MainContact</DefaultObjectValue>
</ContextObject>
```

You can simplify the form fields to the following:

```
<FormField name="ToName">To.DisplayName</FormField>
<FormField name="ToInsuredCity">To.PrimaryAddress.City</FormField>
<FormField name="ToState">To.PrimaryAddress.State</FormField>
<FormField name="ToZip">To.PrimaryAddress.PostalCode</FormField>
...

```

Context objects serve a second purpose by allowing you to manually specify the final value of each context object within the user interface. If you select a document template from the chooser, ClaimCenter displays a series of choices, one for each context object. The name of the context object appears as a label on the left, and the default value appears on the right. The document wizard can optionally allow users to pick from a list of possible values using the `PossibleObjectValues` tag as follows:

```
<ContextObject name="To" type="Contact">
<DefaultObjectValue>Claim.MainContact</DefaultObjectValue>
<PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
</ContextObject>
```

The `PossibleObjectValues` value must be a Gosu expression that evaluates to an array, typically an array of Guidewire entities although that is not required. ClaimCenter does not enforce that the `DefaultObjectValue` is of the same type as `PossibleObjectValues`. While this makes it possible to have two different types for one context object, generally Guidewire recommends against this approach. If you were to do so, you must write form field expressions that work with two different types for that context object.

Context objects must be of one of the following types:

Context object type	Meaning
<code>EntityName</code>	The name of any ClaimCenter entity such as <code>Claim</code> , <code>Exposure</code> , or <code>Activity</code> . The possible object values appear as a drop-down list of options. If that type of entity has a special type of picker, it also displays. For example, if you set the context object type to "Contact", users can use the Contact picker to search for a different value. Similarly, if you set the context object type to "User", ClaimCenter displays a user picker.
<code>Entity</code>	This is a sort of "wildcard" type that indicates support for any keyable entity. Unlike the entry listed earlier with a specific entity name, this type is actually the literal string " <code>Entity</code> ". This is useful for heterogeneous lists of objects.
<code>TypekeyName</code>	The name of any ClaimCenter typekey such as " <code>YesNo</code> ".
<code>string</code>	A case-sensitive all-lower-case string to appear in the user interface as a single line of text. The <code>&lt;DefaultObjectValue&gt;</code> tag must be present, and its contents indicates the default text for this context object. Ignores the <code>&lt;PossibleObjectValues&gt;</code> tag.
<code>text</code>	Case sensitive, must be all lower case. Appears in the user interface as several lines of text. The <code>&lt;DefaultObjectValue&gt;</code> tag must be present, and its contents indicates the default text for this context object. If you use this, ClaimCenter ignores the <code>&lt;PossibleObjectValues&gt;</code> tag.

By default, the `name` attribute of the context object becomes the user-visible name. If you want to use a different user-visible name, set the context object's `display-name` attribute to the text that you want to be visible to the user.

The `type` attribute on the `ContextObject` is used to indicate how the user interface presents the object in the document creation user interface. Valid options include: `String`, `text`, `Contact`, `User`, `Entity`, `Claim`, or any other ClaimCenter entity name or typekey type.

If the context object is of type `String`, then the user would typically be given a single-line text entry box.

If the context object has type `Text`, the user sees a larger text area. However, if the `ContextObject` definition includes a `PossibleObjectValues` tag containing Gosu that returns a `Collection` or array of `String` objects, the user interface displays a selection picker. For example, use this approach to offer a list of postal codes from which to choose. If the object is of type `Contact` or `User`, in addition to the drop-down box, you see a picker button to search for a particular contact or user. All other types (`Entity` is the default if none is specified) are presented as a drop-down list of options. If the `ContextObject` is a typekey type, then the default value and possible values fields must generate Gosu objects that resolve to `TypeKey` objects, not text versions of typecodes values.

There are a few instances in ClaimCenter system in which entity types and typekey types have the same name, such as `Contact`. In this case, ClaimCenter assumes you mean the entity type. If you want the typelist type, or generally want to be more specific, use the fully qualified name of the form `entity.EntityName` or `typekey.TypeKeyName`.

- `String[] getContextObjectNames()` – Returns the set of context object names defined in the document template. See the XML document format for more information on what the underlying configuration looks like.
- `String getContextObjectType(String objName)` – Returns the type of the specified context object. Possible values include "string", "text", "Entity", or the name of any system entity such as "Claim".
- `boolean getContextObjectAllowsNull(String objName)` – Returns `true` if `null` is a legal value, `false` otherwise.

- `String getContextObjectDisplayName(String objName)` – Returns a human-readable name for the given context object, to display in the document creation user interface.
- `String getContextObjectDefaultValueExpression(String objName)` – Returns a Gosu expression which evaluates to the desired default value for the context object. Use this to set the default for the document creation user interface, or as the value if a document is created automatically.
- `String getContextObjectPossibleValuesExpression(String objName)` – Returns a Gosu expression which evaluates to the desired set of legal values for the given context object. Used to display a list of options for the user in the document creation user interface.

### Template Descriptor Fields Related to Form Fields and Values to Merge

Form fields dictate a mapping between ClaimCenter data and the merge fields in the document template.

For example, you might want to merge the claim number into a document field using the simple Gosu expression "`Claim.ClaimNumber`".

The full set of template descriptor fields relating to form fields are as follows:

- `String[] getFormFieldNames()` – Returns the set of form fields defined in the document template. Refer to the following XML document format for more information on what the underlying configuration looks like.
- `String getFormFieldValueExpression(String fieldName)` – Returns a Gosu expression that evaluates to the desired value for the form field. The Gosu expression is usually written in terms of one or more Context Objects, but any legal Gosu expression is allowed.
- `String getFormFieldDisplayValue(String fieldName, Object value)` – Returns the string to insert into the completed document given the field name and the value. The value is typically the result of evaluating the expression returned from the method `getFormFieldValueExpression`. Use this method to rewrite values if necessary, such as substituting text. For example, display text that means “not applicable” (“`<n/a>`”) instead of `null`, or format date fields in a specific way.

## XML Format of Built-in IDocumentTemplateSerializer

The default implementation of `IDocumentTemplateSerializer` uses an XML format that closely matches the fields in the `DocumentTemplateDescriptor` interface. This is intentional. The purpose of `IDocumentTemplateSerializer` is to serialize template descriptors and let you define the templates within simple XML files. The XML format is suitable in typical implementations. ClaimCenter optionally supports different potentially elaborate implementations that might directly interact with a document management system storing the template configuration information.

**IMPORTANT** Many of the fields in this section are defined in more detail in the previous section, “Document Template Descriptors” on page 221. The XML format described in this section is basically a serialization of the fields in the `IDocumentTemplateDescriptor` interface. In the reference implementation, the built-in `IDocumentTemplateDescriptor` and `IDocumentTemplateSerializer` classes implement the serialization.

The default implementation `IDocumentTemplateSerializer` is configured by a file named `document-template.xsd`. The default implementation uses XML that looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<DocumentTemplateDescriptor
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="document-template.xsd"
    id="ReservationRights.doc"
    name="Reservation Rights"
    description="The initial contact letter/template."
    type="letter_sent"
    lob="GL"
    state="CA"
    mime-type="application/msword"
    date-effective="Apr 3, 2003"
    date-expires="Apr 3, 2004"
```

```

    keywords="CA, reservation">
  <ContextObject name="To" type="Contact">
    <DefaultObjectValue>Claim.MainContact</DefaultObjectValue>
    <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
  </ContextObject>
  <ContextObject name="From" type="Contact">
    <DefaultObjectValue>Claim.AssignedUser.Contact</DefaultObjectValue>
    <PossibleObjectValues>Claim.getRelatedUserContacts()</PossibleObjectValues>
  </ContextObject>
  <ContextObject name="CC" type="Contact">
    <DefaultObjectValue>Claim.Driver</DefaultObjectValue>
    <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
  </ContextObject>
<FormFieldGroup name="main">
  <DisplayValues>
    <NullDisplayValue>No Contact Found</NullDisplayValue>
    <TrueDisplayValue>Yes</TrueDisplayValue>
    <FalseDisplayValue>No</FalseDisplayValue>
  </DisplayValues>
  <FormField name="ClaimNumber">Claim.ClaimNumber</FormField>
  <FormField name="InsuredName">To.DisplayName</FormField>
  <FormField name="InsuredAddress1">To.PrimaryAddress.AddressLine1</FormField>
  <FormField name="InsuredCity">To.PrimaryAddress.City</FormField>
  <FormField name="InsuredState">To.PrimaryAddress.State</FormField>
  <FormField name="InsuredZip">To.PrimaryAddress.PostalCode</FormField>
  <FormField name="CurrentDate">Libraries.Date.currentDate()</FormField>
  <FormField name="ClaimNoticeDate">Claim.LossDate</FormField>
  <FormField name="AdjusterName">From.DisplayName</FormField>
  <FormField name="AdjusterPhoneNumber">From.WorkPhone</FormField>
  <FormField name="InsuranceCompanyName">Claim.Policy.UnderwritingCo</FormField>
  <FormField name="InsuranceCompanyAddress">From.PrimaryAddress.AddressLine1</FormField>
  <FormField name="InsuranceCompanyCity">From.PrimaryAddress.City</FormField>
  <FormField name="InsuranceCompanyState">From.PrimaryAddress.State</FormField>
  <FormField name="InsuranceCompanyZip">From.PrimaryAddress.PostalCode</FormField>
</FormFieldGroup>
</DocumentTemplateDescriptor>

```

At run time, this XSD is referenced from a path relative to the module config\resources\doctemplates directory. If you want to change this value, in the plugin registry for this plugin interface, in Studio in the Plugins editor, set the DocumentTemplateDescriptorXSDLocation parameter. To use the default XSD in the default location, set that parameter to the value "document-template.xsd" assuming you keep it in its original directory.

The attributes on the DocumentTemplateDescriptor element correspond to the properties on the IDocumentTemplateDescriptor API.

### Date Formats in the Document Template XML File

Date values may be specified in the XML file in any of the following formats for systems in the English locale:

Date format	Example
MMM d, yyyy	Jun 3, 2005
MMMM d, yyyy Note: Four M characters mean the entire month name)	June 3, 2005
MM/dd/yy	10/30/06
MM/dd/yyyy	10/30/2006
MM/dd/yy hh:mm a	10/30/06 10:20 pm
yyyy-MM-dd HH:mm:ss.SSS	2005-06-09 15:25:56.845
yyyy-MM-dd HH:mm:ss	2005-06-09 15:25:56
yyyy-MM-dd'T'HH:mm:ss zzz	2005-06-09T15:25:56 -0700
EEE MMM dd HH:mm:ss zzz yyyy	Thu Jun 09 15:24:40 -0700 2005

Refer to the following codes for the date formats listed in the earlier table:

- a = AM or PM
- d = day
- E = Day in week (abbrev.)

- `h` = hour (24 hour clock)
- `m` = minute
- `M` = month (MMMM is entire month name)
- `s` = second
- `S` = fraction of a second
- `T` = parse as time (ISO8601)
- `y` = year
- `z` = Time Zone offset.

The first three formats typically are the most useful because templates typically expire at the end of a particular day rather than at a particular time. If you use any of the `template_tools` command line tools commands, you cannot rely on the date format in input files remaining. Although ClaimCenter preserves the values, the date format may change.

For text elements, such as month names, ClaimCenter requires the text representations of the values to match the current international locale settings of the server. For example, if the server is in the French locale, you must provide the month April as "Avr", which is short for Avril, the French word for April.

### Context Objects in the Document Template Descriptor XML File

A context object is an object attached to the merge request. You can insert a context object into the document or insert only certain fields within the object to the document.

For each `ContextObject` tag, the `DefaultObjectValues` expression determines the object (a contact in this case) to initially select. The `PossibleObjectValues` expression determines the set of objects to display in the select control.

Notice that all Gosu expression are in the contents of the tag, rather than attributes on the tag, which makes formatting issues somewhat easier. Also, there is still always a `Claim` entity called `claim` in context as the template runs. If you have time-intensive lookup operations, perform lookups once for the entire document, rather than once for each field that uses the results. To do this, write Gosu classes that implement the lookup logic and cache the lookup results as needed.

### Form Fields in the Document Template Descriptor XML File

*Form fields* dictate a mapping between ClaimCenter data and merge fields in the document template. For example, to merge the claim number into the field `ClaimInfo`, use the following expression:

```
<FormField name="ClaimInfo">Claim.ClaimNumber</FormField>
```

`FormField` tags can contain any valid Gosu expression. This capability is most useful to combine with Studio-based utility libraries or class extensions. For example, to render the claim number and also other information about the claim, encapsulate that logic into an entity enhancement method called `Claim.getClaimInformation()`. Reference that function in the form field as follows:

```
<FormField name="ClaimInfo">Claim.getClaimInformation()</FormField>
```

Form fields can have two additional attributes: `prefix` and `suffix`. Both attributes are simple text values. Use these in cases in which you want to always display text such as "The claim information is \_\_\_\_.", so you can rewrite the form field to look like:

```
<FormField name="MainContactName"
  prefix="The claim information is "
  suffix=".">
  Claim.getClaimInformation()
</FormField>
```

### Form Fields Groups in the Document Template Descriptor XML File

You can optionally define form field groups that logically group a set of form fields. Groups are most useful for defining common attributes across the set of form fields, such as a common date string format. You can also use groups to define a `String` to display for the values `null`, `true`, or `false`. For example, you could check a

Boolean value and display the text "Police report was filed." and "Police report was NOT filed." based on the results. Or, display a special message if a property is null. For example, for a doctor name field, display "No Doctor" if there is no doctor.

Form field groups are implemented as a `FormFieldGroup` element composed of one or more `FormField` elements. A descriptor file can have any number of `FormFieldGroup` elements. Typically, the Gosu in the `FormField` tags refer to a context object defined earlier in the file (see "Template Descriptor Fields and Context Objects" on page 223).

You can group display values for multiple fields by defining a `DisplayValues` tag within the `FormFieldGroup`. You can specify only one value (for example, `NullDisplayValue`) or some other smaller subset of values; you do not need to specify all possible display values. The possible choices are `NullDisplayValue` (if `null`), `TrueDisplayValue` (if `true`), and `FalseDisplay` (if `false`).

For Gosu expressions with subobjects that are pure entity path expressions and any object in the path evaluates to `null`, the expression silently evaluates to `null`. For example, if `Obj` has the value `null`, then `Obj.SubObject.Subsubobject` always evaluates to `null`. Consequently, the `NullDisplayValue` is a simple way of displaying something better if any part of the a multi-step field path expression is `null`.

ClaimCenter also uses the `NullDisplayValue` if an invalid array index is encountered. For example, the expression "`Claim.doctor[0].DisplayName`" results in displaying the `NullDisplayValue` if there are no doctors on the claim.

However, this approach does not work with method invocations on a `null` expression. For example, the expression `Obj.SubObject.Field1()` throws an exception if `Obj` is `null`.

In addition to checking for specific values, the `DisplayValues` tag can contain a `NumberFormat` tag, and either a `DateFormat` tag or a `TimeFormat` tag. These tags allow the user to specify a number format (such as "`####,###.###"`) or Date format (such as "`MM/dd/yyyy`"). The number formats modify the display of all form field values that are numbers or dates. This is useful in cases in which every number value in a form must be formatted in a particular way. Specify the format just once rather than repeatedly. Number format codes must contain the # symbol for each possible digit. Use any other character to separate the digits.

### Customization of the XML Descriptor Mechanism

If the default XML-based template descriptor mechanism is used, the set of attributes can still be modified to suit your needs. To extend the set of attributes on document templates, a few steps are required.

First, modify the `document-template.xsd` file, or create a new one. The location of the `.xsd` file used to validate the document is specified by the `DocumentTemplateDescriptorXSDLocation` parameter within the application `config.xml` file. This location is specified relative to the `WEB_APPLICATION/WEB-INF/platform` directory in the deployed web application directory.

Any number of attributes can be added to the definition of the `DocumentTemplateDescriptor` element. This is the only element which can be modified in this file, and the only legal way in which it can be modified. For example, you could add an attribute named `myattribute` as shown in bold in the following example:

```
<xsd:element name="DocumentTemplateDescriptor">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ContextObject" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="HtmlTable" minOccurs="0" maxOccurs="unbounded" />
      <xsd:element ref="FormFieldGroup" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="identifier" type="xsd:string" use="optional"/>
    <xsd:attribute name="scope" use="optional">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="all"/>
          <xsd:enumeration value="gosu"/>
          <xsd:enumeration value="ui"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

```

    </xsd:attribute>
    <xsd:attribute name="password" type="xsd:string" use="optional"/>
    <xsd:attribute name="description" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="lob" type="xsd:string" use="required"/>
    <xsd:attribute name="myattribute" type="xsd:string" use="optional"/>
    <xsd:attribute name="section" type="xsd:string" use="optional"/>
    <xsd:attribute name="state" type="xsd:string" use="required"/>
    <xsd:attribute name="mime-type" type="xsd:string" use="required"/>
    <xsd:attribute name="date-modified" type="xsd:string" use="optional"/>
    <xsd:attribute name="date-effective" type="xsd:string" use="required"/>
    <xsd:attribute name="date-expires" type="xsd:string" use="required"/>
    <xsd:attribute name="keywords" type="xsd:string" use="required"/>
    <xsd:attribute name="required-permission" type="xsd:string" use="optional"/>
    <xsd:attribute name="default-security-type" type="xsd:string" use="optional"/>
</xsd:complexType>
</xsd:element>

```

Next, enable the user to search on the `myattribute` attribute and see the results. Add an item with the same name (`myattribute`) to the PCF files for the document template search criteria and results. See the “Using the PCF Editor” on page 295 in the *Configuration Guide* for more information about PCF configuration.

Now the new `myattribute` property shows up in the template search dialog. The search criteria processing happens in the `IDocumentTemplateSource` implementation. The default implementation, `LocalDocumentTemplateSource`, automatically handles new attributes by attempting an exact match of the attribute value from the search criteria. If the specified value in the descriptor XML file contains commas, it splits the value on the commas and tries to match any of the resulting values.

For example, if the value in the XML is `test`, then only a search for "test" or a search that not specifying a value for that attribute finds the template. If the value in the XML file is "test,hello,purple", then a search for any of "test", "hello", or "purple" finds that template.

Finally, once ClaimCenter creates the merged document, it tries to match attributes on the document template with properties on the document entity. For each match found, ClaimCenter copies the value of the attribute in the template descriptor to the newly created Document entity. The user can either accept the default or change it to review the newly-created document.

## Template Source Reference Implementation

The ClaimCenter reference implementation keeps a set of document templates on the ClaimCenter application server. The document templates reside initially in the directory:

ClaimCenter/modules/configuration/config/resources/doctemplates

At run time, the document templates are typically at the path:

SERVER/webapps/cc/modules/configuration/config/resources/doctemplates

For each template, there are two files:

File	Naming	Example	Purpose
The template	Normal file name	"Test.doc"	Contains the text of the letter, plus named fields that fill in with data from ClaimCenter.
The template descriptor	Template name + ".descriptor"	"Test.doc.descriptor"	This XML-formatted file provides various information about the template: <ul style="list-style-type: none"> <li>• How to search and find this template</li> <li>• Form fields that define what information populates each merge field</li> <li>• Form fields that define context objects</li> <li>• Form fields that define root objects to merge</li> </ul>

To set up a new form or letter, you must create a template file and a template descriptor file. Then, deploy them to the `templates` directory on the web server.

**See also**

- For details on the XML formatting of template descriptor files, see:
  - “Document Template Descriptors” on page 221
  - “XML Format of Built-in IDocumentTemplateSerializer” on page 225.

## Document Template Descriptor Optional Cache

By default, ClaimCenter calculates the list of document templates from files locally on disk each time the application needs them. If you have only a small list of document templates, this is a quick process. However, if you have a large number of document templates, you can tell ClaimCenter to cache the list for better performance.

You might prefer to use the default behavior (no caching) during development, particularly if you are frequently changing templates while the application is running. However, for production, set the optional parameter in the document template source plugin to cache the list of templates.

### To enable document template descriptor caching

1. In Guidewire Studio, under Resources, click Plugins → IDocumentTemplateSource.
2. Under the parameters editor in the right pane, add the `cacheDescriptors` parameter with the value `true`.

## Built-in Document Production Plugins

Each application uses a different mechanism for defining fields to be filled in by the document production plugins. The following table describes the built-in document production plugins and their associated formats:

Application or format	Description of built-in template
Microsoft Word	The document production plugin takes advantage of Microsoft Word's built-in mail merge functionality to generate a new document from a template and a descriptor file. ClaimCenter includes with a sample Reservation of Rights Word document and an associated CSV file. That CSV file is present so that you can manually open the Word template. ClaimCenter does not require or use this document.
Adobe Acrobat (PDF)	PDF file production happens on the server and requires a license to server-based software. See “Server-side PDF Licensing” on page 220.
Microsoft Excel	The document production plugin takes advantage of Microsoft Excel's built-in named fields functionality. The “names” in the Excel spreadsheet must match the <code>FormField</code> names in the template descriptor. After the merge, the values of the named cells become the values extracted from the descriptor file.
Gosu template	Gosu provides you with a sophisticated way to generate any kind of text file. This includes plain text, RTF, CSV, HTML, or any other text-based format.  The document production plugin retrieves the template and uses the built-in Gosu language to populate the document from Gosu code and values defined in the template descriptor file. Based on the file's MIME type and the local computer's settings, the system opens the resultant document in the appropriate application. See “Data Extraction Integration” on page 607.

## Generating Documents from Gosu

ClaimCenter can generate PDF documents and Gosu-based forms without user intervention from Gosu business rules or from any other place Gosu runs.

ClaimCenter automatic form generation does not support any client-side creation such as Microsoft Word and Microsoft Excel templates. In client-side creation, the Document Assistant generates the new document locally. On Windows, the Document Assistant uses JScript to control local applications on the client PC. At the time business rules run, there may or may not be a user manipulating the application user interface. Because there may

be no client to run the client-side code, client-side document production is impossible for automatic document creation.

However, server-side creation is possible using text formats:

- You can convert Word documents convert to the text-based RTF format
- You can convert many Excel templates to CSV files.

Therefore, you can convert most Microsoft application documents into equivalent Gosu templates that can generate text in RTF or CSV format. You can also use Gosu to generate any other text-based format, most notably plain text and HTML.

The automatic form generation process is very similar to the manual document generation process, but effectively skips some steps previously described for manual form generation.

The following list describes differences in automatic form generation compared to steps in “User Interface Flow for Document Production” on page 214. See that section for the step numbers:

- **Automatic template choosing** – Because template choosing is automatic, this effectively skips step 1 through step 6. Instead, you specify the appropriate template and parameter information within Gosu code.
- **No user editing** – Since there is no user intervention, there is no optional step for user intervention within the middle of this flow of step 9
- **No user uploading** – Because the Gosu code is executed on the ClaimCenter server, there is no user uploading step, which is step 11 in that section.

To create a document, first create a map of values that specifies the value for each context object. Using `java.util.HashMap` is recommended, but any `Map` type is legal. This value map must be non-`null`. The values in this map are unconstrained by either the default object value or the possible object values. Be careful to pass valid objects of the correct type.

Within business rules, the Gosu class that handles document production is called `DocumentProduction`. It has methods for synchronous or asynchronous creation, which call the respective synchronous or asynchronous methods of the appropriate document production plugin, `createDocumentSynchronously` or `createDocumentAsynchronously`. Additionally there is a method to store the document: `createAndStoreDocumentSynchronously`. You can modify this Gosu class as needed.

If synchronous document creation fails, the `DocumentProduction` class throws an exception, which can be caught and handled appropriately by the caller. If document storage errors happen later, such as for asynchronous document storage, the document content storage plugin must handle errors appropriately. For example, the plugin could send administrative e-mails or create new activities using SOAP APIs to investigate the issues. Or you could design a system to track document creation errors and document management problems in a separate user interface for administrators. In the latter case, the plugin could register any document creation errors with that system.

If the synchronous document creation succeeds, next your code must attach the document to the claim by setting `Document.Claim`.

---

**IMPORTANT** New documents always use the latest in-memory versions of entity instances at the time the rules run, not the versions as persisted in the database.

---

Be careful creating documents within Pre-Update business rules or in other cases where changes can be rolled back due to errors (Gosu exceptions) or validation problems. If errors occur that roll back the database transaction even though rules added a document an external document management system, the externally-stored document is *orphaned*. The document exists in the external system but no ClaimCenter persisted data links to it.

#### See also

- “Important Notes About Cached Document UIDs” on page 232.

### Example Document Creation After Sending an Email

You can use code like the following to send a standard email and then create a corresponding document:

```
// First, construct the email
var toContact : Contact = myClaim.Insured
var fromContact : Contact = myClaim.AssignedUser.Contact.Person
var subject : String = "Email Subject"
var body : String = "Email Body"

// Next, actually *send* the email
gw.api.email.EmailUtil.sendEmailWithBody(myClaim, toContact, fromContact, subject, body)

// Next, create the document that records the email
var document : Document = new Document(myClaim)
document.Claim = myClaim
document.Name = "Create by a Rule"
document.Type = "letter_sent"
document.Status = "draft"
// ...perhaps add more property settings here

// Create some "context objects"
var parameters = new java.util.HashMap()
parameters.put("To", toContact)
parameters.put("From", fromContact)
parameters.put("Subject", subject)
parameters.put("Body", body)
parameters.put("RelatedTo", myClaim)
parameters.put("Claim", myClaim)

// Create and store the document using the context objects
DocumentProduction.createAndStoreDocumentSynchronously("EmailSent.gosu.htm", parameters, document)
```

This particular example assumes that the document production plugin for that template supports synchronous production. This is true for all built-in document production plugins but not necessarily for all document production plugins. The calling code must either know in advance whether the document production plugin for that template type supports synchronous and/or asynchronous creation, or checks beforehand. If necessary, you can check which types of production are supported with code such as:

```
var plugin : IDocumentProduction;
plugin = PluginRegistry.getPluginRegistry().getPlugin(IDocumentProduction) as IDocumentProduction;
if (plugin.synchronousCreationSupported(templateDescriptor)) {
    ...
}
```

For more examples of creating documents from Gosu, see “Document Creation” on page 145 in the *Rules Guide*.

### Important Notes About Cached Document UIDs

If a new document is created and an error occurs within the same database transaction, the error typically causes the database transaction to roll back. This means that no database data was changed. However, if the local ClaimCenter transaction rolls back, there is no stored reference in the ClaimCenter database to the document unique ID (UID). The UID describes the location of the document in the external system. This information is stored in the Document entity in ClaimCenter in the same transaction, so the Document entity was not committed to the database. The new document in the external system is *orphaned*, and additional attempts to change ClaimCenter data regenerates a new, duplicate version of the document.

For the common case of validation errors, ClaimCenter avoids this problem. If a validatable entity fails validation, ClaimCenter saves the document UID in local memory. If the user fixes the validation error in that user session, ClaimCenter adds the document information as expected so no externally-stored documents are orphaned.

However, if other errors occur that cause the transaction to roll back (such as uncaught Gosu exceptions), externally-stored documents associated with the current transaction could be orphaned. The document is stored externally but the ClaimCenter database contains no Document entity that references the document UID for it. Avoiding orphaned documents is a good reason to ensure your Gosu rules properly catches exceptions and

handles errors elegantly and thoroughly. Write good error-handling code and logging code always, but particularly in document production code.

## Template Web Service APIs

To manipulate Gosu document templates from external system, you can use the ClaimCenter `TemplateToolsAPI` web service. This web service is WS-I compliant.

### Validating Templates

To validate templates, the `TemplateToolsAPI` web service provides several methods:

`validateTemplate` – Validate that the given template descriptor is in a valid format, and that all template descriptor context objects and form fields are valid given the current datamodel:

- Check that the Gosu expressions in the descriptor are valid. Context object default and possible value expressions must use the available objects. Form field expressions must use available objects or context objects.
- Check that the `permissionRequired` attribute, if specified, is a valid system permission code.
- Check that the `default-security-type` attribute, if specified, is a valid document security type code.
- Check that the `type` attribute, if specified, is a valid document type code.
- Check that the `section` attribute, if specified, is a valid section type code.

The method takes a template ID as a String value such as "ReservationRights.doc". The method returns human-readable string detailing the operations performed and any errors.

- `validateTemplateInLocale` – Same as `validateTemplate`, but specifies a specific locale code as an extra method argument.
- `validateAllTemplates` – validates all templates. The method returns human-readable string detailing the operations performed and any errors.
- `validateAllTemplatesInLocale` – Same as `validateAllTemplates`, but specifies a specific locale code as an extra method argument.
- `listTemplates` – List the templates which the server currently knows about. Useful for sanity-checking arguments to the validation commands. The method takes no arguments. It returns a single String value that lists all template names.
- `importFormFields` – Imports context objects, field groups, and fields from a comma-separated values (CSV) file into the corresponding template descriptor file. Arguments include:
  - `contextObjectFileContents` – The contents of a file containing the context objects to be imported, in CSV format.
  - `fieldGroupFileContents` – The contents of a file containing the field groups to be imported, in CSV format.
  - `fieldFileContents` – The contents of a file containing the fields to be imported, in CSV format.
  - `descriptorFileContents` – The contents of the descriptor file.

The method returns a results object that lists the fields for the new contents of the descriptor file, and a human-readable string detailing operations performed and any errors encountered.



# Geographic Data Integration

ClaimCenter and Guidewire ContactManager provide a user interface and an integration API for assigning a latitude and a longitude to an address. These two decimal numbers identify a specific location in degrees north or south of the equator and east or west of the prime meridian. The process of assigning these geographic coordinates to an address is called *geocoding*. Additionally, ClaimCenter and ContactManager support routing services, such as getting a map of an address and getting driving directions between two addresses, provided the addresses are geocoded already.

This topic includes:

- “Geocoding Plugin Integration” on page 235
- “Steps to Deploy a Geocoding Plugin” on page 237
- “Writing a Geocoding Plugin” on page 238
- “Geocoding Status Codes” on page 244

**See also**

- For general information about implementing plugins, see “Plugin Overview” on page 163.

## Geocoding Plugin Integration

Guidewire ClaimCenter and Guidewire ContactManager use the Guidewire geocoding plugin to provide geocoding services in a uniform way, regardless of the external geocoding service that you use. The application requests geocoding services from the registered `GeocodePlugin` implementation. `GeocodePlugin` implementations typically do not apply geocode coordinates directly to addresses in the application database. Instead, the application requests geocode coordinates from the plugin, and the application decides how to apply them.

In addition, the `GeocodePlugin` interface supports routing services, such as retrieving driving directions and maps from external geocoding services that support these features. The interface also defines a method for *reverse geocoding*, which gets an address from geocode coordinates. The plugin interface defines methods that let callers of the plugin determine whether the registered implementation supports routing services and reverse geocoding.

## How ClaimCenter Uses Geocode Data

ClaimCenter uses geocode data for actions within rules and other geographic searches. The base configuration of ClaimCenter supports these high-level geocoding features:

- **ClaimCenter user assignment and searching** – ClaimCenter can assign claims to users based on proximity between two addresses, such as an insured's address and user addresses. You can also search for users by proximity on the administration user search page. This feature requires ClaimCenter to have the geocoding plugin enabled and its database contain geocoded addresses
- **ClaimCenter catastrophe search and heat maps** – ClaimCenter lets you search for claims associated with a catastrophe display them in a heat map the [Catastrophe Search](#) page.
- **ContactManager address book searches (proximity search of vendors)** – ClaimCenter and ContactManager let you search for nearby service providers in the address book based, on geographic proximity. This feature requires that:
  - ClaimCenter and ContactManager to be installed.
  - ClaimCenter and ContactManager have the `GeocodingPlugin` enabled.
  - ContactManager contains geocoded addresses in its database.

To support user assignment and searching using geocoding, you must install and register a `GeocodePlugin` implementation for an external geocoding service in ClaimCenter. To support address book searches that use geocoding, must install and register a `GeocodePlugin` implementation in both ClaimCenter and ContactManager.

---

**IMPORTANT** To support address book searches with geocoding, configure and install both ClaimCenter and ContactManager, linking the two applications. For details, see “[Integrating ContactManager with Guidewire Core Applications](#)” on page 45 in the *Contact Management Guide*.

---

## What the Geocoding Plugin Does

A `GeocodePlugin` implementation generally performs the following tasks:

- Assign latitude and longitude coordinates (required)
- List possible address matches (optional)
- Return driving directions (optional)
- Find an address from coordinates (optional)
- Find maps for arbitrary addresses (optional)

## Synchronous and Asynchronous Calls to the Geocoding Plugin

From the perspective of ClaimCenter calling a `GeocodePlugin` method, the call is always synchronous. The caller waits until the method completes. For user-initiated requests, such as proximity searches, the user interface blocks until the plugin responds.

ClaimCenter and ContactManager also support a distributed system that allows all servers in the cluster to perform geocoding asynchronously from the application. This *distributed work queue system* is especially useful after an upgrade with new addresses to geocode. You can use the geocoding work queues to geocode addresses in the background. You can use the work queues even if your application instance comprises a single server instead of a cluster of servers.

The default `config.xml` file and scheduling files are pre-configured with default settings for the work queue system. Change those settings only if the defaults are inappropriate. Register and enable the plugin correctly, and the distributed work queue system handles background geocoding automatically.

**Note:** The `Geocode` and `ABGeocode` work queues use only the `geocodeAddressBestMatch` method of the `GeocodePlugin` interface.

**See also**

- “Scheduling and Configuring the Geocode Distributed Work Queue” on page 22 in the *System Administration Guide*.

## Using a Proxy Server with the Geocoding Plugin

If you want to prevent ClaimCenter and the geocoding plugin from accessing external Internet services directly, you must use a proxy server for outgoing requests. If you use a proxy server for the geocoding plugin, you must configure the built-in Bing Maps implementation to connect with the proxy server, not the Bing Maps geocoding service.

The geocoding plugin only initiates communications with geocoding services. It never responds to communications initiated external from the Internet. Therefore, you do not need a *reverse proxy server* to insulate the geocoding plugin from incoming Internet requests.

**See also**

- “Proxy Servers” on page 619.
- To learn how to configure the Bing Maps geocoding plugin to use a proxy server, see “Bing Maps Geocoding Service Communication” on page 622.

## The Built-in Bing Maps Geocoding Plugin

ClaimCenter includes a fully functional and supported implementation of the `GeocodePlugin` to connect to the Microsoft Bing Maps Geocode Service. For details about configuring Bing Maps support, see “Configuring Geocoding” on page 20 in the *System Administration Guide*.

## Batch Geocoding Only Some Addresses

The `Address` entity has a property called `BatchGeocode`. The `Geocode` writer in ClaimCenter and the `ABGeocode` writer in `ContactManager` use `BatchGeocode` to filter which addresses to pass to the plugin for geocoding. If the property is `true`, they pass the address to the plugin for geocoding.

Implementations of the `GeocodePlugin` ignore the `BatchGeocode` property. Callers of the plugin are responsible for determining which addresses to geocode. The `GeocodePlugin` geocodes any address it receives.

For more information, see “Geocoding and Proximity Searches for Vendor Contacts” on page 99 in the *Contact Management Guide*.

## Steps to Deploy a Geocoding Plugin

Follow these high-level steps to deploy a `GeocodePlugin` implementation:

### Step 1: Implement the Plugin Interface in Gosu

If you want to use a geocoding service other than the one supported by the built-in `GeocodePlugin` implementation, write your own implementation and register it in ClaimCenter Studio. See “Writing a Geocoding Plugin” on page 238.

If you want to support proximity searches of vendors within the `ContactManager` application, you must repeat this step in `ContactManager` Studio.

### Step 2: Register the Plugin Implementation in Studio

- In Studio, navigate to `Resources` → `Plugins` → `gw` → `plugin` → `geocode` → `GeocodePlugin`.

2. In the pane on the right, select the **Enabled** checkbox.

A dialog box informs you that editing the plugin creates a copy in the current module.

3. Click **Yes**.

4. In the **Class** text box, enter the name of the plugin implementation class that you want to use.

5. Edit the **Parameters** table to specify parameters and values that your plugin implementation requires.

Typically, a `GeocodePlugin` implementation has security parameters for connecting to the external geocoding service, such as a username and password.

If you want support for proximity searches of vendors, which uses the `ContactManager` application, you must repeat this step in `ContactManager Studio`.

## Step 3: Enable the User Interface for Desired Geocoding Features

You must enable the user interface for the desired geocoding features within ClaimCenter by adjusting parameters in the `config.xml` file.

For ClaimCenter, modify the `UseGeocodingInPrimaryApp` parameter. This specifies whether ClaimCenter displays the user interface for proximity searches local to ClaimCenter in assignment and user search pages and pickers. For example:

```
<param name="UseGeocodingInPrimaryApp" value="true"/>
```

If you want support for proximity searches of vendors, which uses the `ContactManager` application, and want to support the user interface for geocoding, set the `UseGeocodingInAddressBook` parameter. Set this parameter in each application that requires support. For example, to support the user interface in both ClaimCenter and `ContactManager`, set this in both ClaimCenter and `ContactManager`.

```
<param name="UseGeocodingInAddressBook" value="true"/>
```

## Writing a Geocoding Plugin

To use a geocoding service other than Microsoft Bing Maps Geocode Service, write your own `GeocodePlugin` implementation in Gosu and register your implementation class in ClaimCenter Studio.

The high level features and related plugin methods of the `GeocodePlugin` interface are:

- **Geocode an address** – `geocodeAddressBestMatch`
- **List possible matches for an address** – `geocodeAddressWithCorrections`, `pluginSupportsCorrections`
- **Retrieve driving directions between two addresses** – `getDrivingDirections`,  
`pluginSupportsDrivingDirections`, `pluginReturnsOverviewMapWithDrivingDirections`,  
`pluginReturnsStepByStepMapsWithDrivingDirections`
- **Retrieve a map for an address** – `getMapForAddress`, `pluginSupportsMappingByAddress`
- **Retrieve an address from a pair of geocode coordinates** – `getAddressByGeocodeBestMatch`,  
`pluginSupportsFindByGeocode`
- **List possible addresses from a pair of geocode coordinates** – `getAddressByGeocode`,  
`pluginSupportsFindByGeocodeMultiple`

The `geocodeAddressBestMatch` method is the only method required of a `GeocodePlugin` implementation to be considered functional. The other methods are for optional features of the `GeocodePlugin`.

## Using the Abstract Geocode Java Class

Although you can write your own implementation of the `GeocodePlugin`, Guidewire provides a built-in implementation of the plugin interface, called `AbstractGeocodePlugin`. It is an abstract Java class that your Gosu implementation can extend. The default behaviors of `AbstractGeocodePlugin` may save you work, particularly

if you do not support all the optional features of the plugin. This built-in, abstract implementation is defined in the package `gw.api.geocode`.

If you use `AbstractGeocodePlugin` as the base class of your implementation, your Gosu class must provide implementations of these methods:

- `geocodeAddressBestMatch`
- `getDrivingDirections`
- `pluginSupportsDrivingDirections`

You can add other interface methods to your Gosu class to support other optional features of the `GeocodePlugin`.

## High-Level Steps to Writing a Geocoding Plugin Implementation

1. Write a new class in Studio that extends `AbstractGeocodePlugin`:

```
class MyGeocodePlugin extends AbstractGeocodePlugin {
```

Omit “`implements GeocodePlugin`” because `AbstractGeocodePlugin` already implements the interface.

2. Implement the required method `geocodeAddressBestMatch`.

The method accepts an address and returns a different address with latitude and longitude coordinates assigned.

See “Geocoding an Address” on page 239.

3. To support driving directions, implement these methods:

- `pluginSupportsDrivingDirections` – Return `true` from this method to indicate that your implementation supports driving directions.
- `getDrivingDirections` – If your implementation supports driving directions, return driving directions based on a start address and a destination address that have latitude and longitude coordinates. Otherwise, return `null`.

See “Getting Driving Directions” on page 241.

4. If you want to support other optional features, such as getting a map for an address or getting an address from geocode coordinates, override additional methods. Be sure to let ClaimCenter know your plugin supports these features by implementing the methods that identify feature support.

For more information, see:

- “Supporting Multiple Address Corrections with a List of Possible Matches” on page 241
- “Retrieving Overview Maps” on page 242
- “Adding Segments of the Journey with Optional Maps” on page 242
- “Getting a Map for an Address” on page 243
- “Getting an Address from Coordinates (Reverse Geocoding)” on page 243

## Geocoding an Address

The `GeocodePlugin` interface has one required method, `geocodeAddressBestMatch`. The method takes an address (`Address`) instance and returns a different address instance. The address returned from the plugin has a `GeocodeStatus` value that indicates whether the geocoding request succeeded and how precisely the geocode coordinates match the incoming address. Valid values for `GeocodeStatus` include `exact`, `failure`, `street`, `postalcode`, or `city`. If the status is anything other than `failure`, the `Latitude` and `Longitude` properties in the returned address are correct for the returned address.

Your implementation of the `geocodeAddressBestMatch` method must not modify the incoming address instance with data returned from the geocoding service, or for any reason. Instead, make a copy of the incoming address by using the `clone()` method on it. Or, create a new address instance by using the Gosu expression `new Address()`. Set the properties on the cloned or new address from the values returned by the geocoding service and use that address as the return value of your `geocodeAddressBestMatch` method.

The following example creates a new address and sets the geocoding status and the geocode coordinates.

```
a = new Address()  
a.GeocodeStatus = GeocodeStatus.TC_EXACT  
a.Latitude = 42.452389  
a.Longitude = -71.375942
```

In a real implementation, your code assigns coordinate values obtained from the external geocoding service, not from numeric literals as the example shows.

#### See also

- “Geocoding Status Codes” on page 244.

### [Geocoding an Address from the User Interface](#)

If ClaimCenter wants to geocode an address immediately, ClaimCenter calls one of the geocoding plugin methods. In situations in which ClaimCenter wants only the best match for a geocoding request, it calls the plugin method `geocodeAddressBestMatch`.

If you trigger geocoding from the user interface, geocoding is synchronous. In other words, the user interface blocks until the plugin returns the geocoding result. There is no built-in timeout between the application and the geocoding plugin. Your own geocoding plugin must encode a timeout so it can give up on the external service, throw a `RemoteException`, and let the user interface continue.

#### See also

- “Handling Address Clarifications for a Geocoded Address” on page 240

### [Geocoding an Address from a Batch Process](#)

ClaimCenter geocodes addresses in the background with batch processes that call the geocoding plugin as necessary to geocode an address. For more information about how to configure the batch processes, See the “Working with the Geocode Distributed Work Queue” on page 20 in the *System Administration Guide*.

### [Handling Address Clarifications for a Geocoded Address](#)

If `geocodeAddressBestMatch` is the only method from `AbstractGeocodePlugin` that you override, your plugin does really handle address correction. To support address correction, override the `geocodeAddressWithCorrections` and `pluginSupportsCorrections` methods. Also, you must implement a PCF to display multiple address and let the user select the correct one.

However, the `geocodeAddressBestMatch` method can provide address clarifications or leave some properties blank if the service did not use them to generate the coordinates. If the geocoding service modified properties on the submitted address, set those properties on the address that the your plugin returns to match.

Callers of the plugin must assume that blank properties in a returned address are blank purposely. For example, certain address properties in return data might be unknown or inappropriate if the geocoding status is other than `exact`. If the geocode status represents the weighted center of a city, the street address might be blank because the returned geocode coordinates do not represent a specific street address. ClaimCenter treats the set of properties returned by the geocoding plugin to be the full set of properties to show to the user or log to the geocoding corrections table.

Sometimes a geocoding service returns variations of an address. For example, the street address “123 Main Street” might “123 North Main Street” and “123 South Main Street”, each with different geocode coordinates. The geocoding service might return both results so a user can select the appropriate one. Some variations might be due to differences in abbreviations, such `Street` versus `St`. Some services provide variants with and without suite, apartment, and floor numbers from addresses, or provide variants that contain other kinds of adjustments. For the `geocodeAddressBestMatch` method, return only the best match.

**See also**

- “Supporting Multiple Address Corrections with a List of Possible Matches” on page 241.

### Supporting Multiple Address Corrections with a List of Possible Matches

If your geocoding service can provide a list of potential addresses for address correction, implement the `geocodeAddressWithCorrections` method. Additionally, implement the `pluginSupportsCorrections` method and return `true` to tell ClaimCenter that your implementation supports multiple address corrections.

In contrast to the `geocodeAddressBestMatch` method, the `geocodeAddressWithCorrections` method returns a list of addresses rather than a single address. Both methods can return address corrections or clarifications, or leave some properties blank if they were not used to generate the coordinates. However, the system calls the `geocodeAddressWithCorrections` method if the user interface context can handle a list of corrections. For example, your user interface might support letting a user choose the intended address from a list of near matches.

The result list that your `geocodeAddressWithCorrections` method returns must be a standard `List` (`java.util.List`) that contains only `Address` entities. You declare this type of object in Gosu by using the generic syntax `List<Address>`.

If the geocoding service does not support multiple corrections, this method must return a one-item list that contains the results of a call to `geocodeAddressBestMatch`. If you base your implementation on the built-in `AbstractGeocodePlugin` class, the abstract class implements this behavior for you.

**See also**

- “Handling Address Clarifications for a Geocoded Address” on page 240
- “Gosu Generics” on page 239 in the *Gosu Reference Guide*

### Geocoding Error Handling

If your plugin implementation fails to connect to the external geocoding service, this method throws the exception `java.rmi.RemoteException`. Your implementation must never set the geocode status of an address to `none`. Instead, throw an exception if the error is retryable. For more information, see “Geocoding Status Codes” on page 244.

## Getting Driving Directions

ClaimCenter and ContactManager optionally can display driving directions and other travel information. You can support this by implementing these methods on the `GeocodePlugin` interface:

- `getDrivingDirections` – Get driving directions based on a start address and a destination address, as well as a switch that specifies miles or kilometers.
- `pluginSupportsDrivingDirections` – Return `true` from this simple method.

Driving directions are enabled by default if you have geocoding enabled in the user interface. To disable driving directions even in cases in which geocoding is enabled, you must edit the relevant PCF pages.

It is important to understand that ClaimCenter does not require that the driving directions request be handled by the same external service as geocoding requests. If desired, the plugin could contact different services for these types of requests.

The interface for driving directions is a single method called `getDrivingDirections`. If the user requests driving directions, ClaimCenter calls the geocoding plugin method `getDrivingDirections` once for each request. From a programming perspective, the method and the request are synchronous in the sense that the method must not return until the request is complete. However, from a user perspective it may seem asynchronous because ClaimCenter may request multiple driving directions requests without showing them to the user immediately. For example, ClaimCenter may request directions from one insured’s home address to nine different auto repair shops in preparation for users to request a map for one of them.

This method takes two `Address` entities. The method must send these addresses to a remote driving directions service and return the results. If driving time and a map illustrating the route between the two addresses are available, the method returns those also.

Address properties already include values for latitude and longitude before calling the plugin. Because some services use only the latitude and longitude, driving directions can be to or from an inexact address such as a postal code rather than exact addresses.

The plugin must return a `DrivingDirections` object that encapsulates the results. This class is in the `gw.api.contact` package namespace. To create one, you have two options.

You can directly create a new driving directions object:

```
var dd = new DrivingDirections()
```

Alternatively, you can use a helper method to initialize the object:

```
uses gw.api.geocode.GeocodeUtils  
var dd = GeocodeUtils.createPreparedDrivingDirections(startAddress, endAddress, unitOfDist)
```

**Note:** The driving directions object is implemented as a Guidewire internal Java class, not a Guidewire entity. This distinction is important, because you cannot extend its data model.

## Retrieving Overview Maps

If your geocoding or routing service supports overview maps, first implement the method `pluginReturnsOverviewMapWithDrivingDirections` to return `true`.

Next, set the following properties on the driving directions object.

- `MapOverviewUrl` – a URL of the overview map shows the entire journey, as a `MapImageUrl` object, which is a simple object containing two properties:
  - `MapImageUrl` – A fully-formed and valid URL string
  - `MapImageTag` – The text of the best HTML image element (in other words, an HTML `<IMG>` tag) that properly displays this map. This text may include, for example, the `height` and `width` attributes of the map, if they are known.
- `hasMapOverviewUrl` – Set to `true` if there is a URL of the overview map shows the entire journey

For example:

```
dd.MapOverviewUrl = new MapImageUrl()  
dd.MapOverviewUrl.MapImageUrl = "http://myserver/mapengine?lat=3.9&long=5.5"
```

If you want to know how map data is used in the ClaimCenter user interface, see the PCF file `AddressBookDirectionsDV.pcf`.

## Adding Segments of the Journey with Optional Maps

If you want to provide actual driving directions, you must also add driving directions elements that represent the segments of the journey.

Each `DrivingDirections` object contains various properties that relate to the entire journey, and it contains a list of journey segments in the form of an array of `DrivingDirectionsElem` objects. Each object in the array represents one segment, such as “Turn right on Main Street and drive 40 miles”. Many properties are set automatically if you use `createPreparedDrivingDirections` as described earlier.

For each new segment, you do not need to create `DrivingDirectionsElem` directly, instead call the `addNewElement` method on the `DrivingDirections` object:

```
drivingdirections.addNewElement(String formattedDirections, Double distance, Integer duration)
```

The `formattedDirections` object may represent either a discrete stage in the directions, such as “Turn left onto I-80 for 10 miles”. It may represent a note or milestone not corresponding to a stage, such as “Start

"trip". The exact format and content of the textual description depends greatly on your geocoding service. It may or may not include HTML formatting.

If your service supports multiple individual maps other than the overview, repeatedly call the method `addNewMapURL(urlString)` on the driving directions object to add URLs for maps. There is no requirement for the number of maps to match the number of segments of the journey. The position in the map URL list does not have a fixed correspondence to a segment number. However, always add map URLs in chronological order for the journey. In other words, generate the segments in the expected order from the start address to the end address.

If you add individual maps like this, also implement the method `pluginReturnsStepByStepMapsWithDrivingDirections` to return `true`.

### Extracting Data from Driving Directions in PCF Files

If you want to extract information from `DrivingDirections` in PCF code or other Gosu code, be aware there are properties you can extract, such as `TotalDistance`, `TotalTimeInMinutes`, `GCDistance`, and `GCDistanceString`. Methods with "GC" in the name refer to the great circle and great circle distance, which is the distance between two points on the surface of a sphere. It is measured along a path on the surface of the Earth's curved 3D surface, in contrast to point-to-point through the Earth's interior.

**Note:** In Gosu, the address entity contains utility methods for calculating great circle distances. For example, `address.getDistanceFrom(latitudeValue, longitudeValue)`

The `description` properties from the start and end addresses are also copied into the `Start` and `Finish` properties on `DrivingDirections`.

Refer to the Gosu API Reference in Studio for the full set of properties.

### Error Handling

If the plugin fails to connect to the external geocoding service, throw the exception `java.rmi.RemoteException`.

## Getting a Map for an Address

Some geocoding services support getting a map from an address. If your service supports it, first implement the `pluginSupportsMappingByAddress` method and return `true` to tell ClaimCenter that your implementation supports this feature. Next, implement the method `getMapForAddress`, which takes an `Address` and a unit of distance (miles or kilometers).

Your `getMapForAddress` method must return a map image URL in a `MapImageUrl` object. This is a simple wrapper object containing a `String` for a map URL.

The following demonstrates a simple (fake) implementation:

```
override function getMapForAddress(address: Address, unit: UnitOfDistance) : MapImageUrl {  
    var i = new MapImageUrl()  
    i.MapImageUrl = "http://myserver/mapengine?lat=3.9&long=5.5"  
    return i;  
}
```

## Getting an Address from Coordinates (Reverse Geocoding)

Some geocoding services support getting an address from latitude and longitude coordinates. This is sometimes called *reverse geocoding*. If your service supports it, you can override the following methods in `AbstractGeocodePlugin` to implement reverse geocoding.

ClaimCenter supports two types of reverse geocoding: return a single address, and return multiple addresses. You can support one or both. Implementing this method is required if you want to support all mapping features in ClaimCenter.

To implement single-result reverse geocoding, first implement the method `pluginSupportsFindByGeocode` and return `true`. Next, implement the `getAddressByGeocodeBestMatch` method, which takes a latitude coordinate and a longitude coordinate. Return an address with as many properties set as your geocoding service provides.

To implement multiple-result reverse geocoding, first implement the method `pluginSupportsFindByGeocodeMultiple` and return `true`. Next, implement the `getAddressByGeocode` method, which takes a latitude coordinate, a longitude coordinate, and a maximum number of results. If the maximum number of results parameter is zero or negative, this method must return all results. As with the geocoding multiple-result methods, this returns a list of `Address` entities, specified with the generics syntax `List<Address>`. For more information, see “Gosu Generics” on page 239 in the *Gosu Reference Guide*.

## Geocoding Status Codes

Geocoding services typically provide a set of status codes to indicate what happened during the geocoding attempt. For example, even if the external geocoding service returns latitude and longitude coordinates successfully, it is useful to know how precisely those coordinates represent the location of an address. Do they represent an exact address match? If the service could not find the address or the address was incomplete, do the coordinates identify the weighted center of the postal code or city? The status codes `exact`, `street`, `postalcode`, and `city` indicate the precision with which the `Latitude` and `Longitude` properties identify the global location of an address.

Additionally, the status code `failure` indicates that geocoding failed. That means that any values in the `Latitude` and `Longitude` properties of the address are unreliable. The status code `none` indicates that an address has not been geocoded since it was created or last modified, which also means that `Latitude` and `Longitude` are unreliable.

The status values must be values from the `GeocodeStatus` type list, described in the following table:

Geocode status code	Description
<code>none</code>	No attempt at geocoding this address occurred. This is the default geocoding status for an address. Do not set an address to this geocoding status. If you experience an error that must retrigger geocoding later, throw an exception instead. When an address is modified, ClaimCenter sets the address to this status, too, which indicates that the address has not been geocoded since it was last modified.
<code>failure</code>	An attempt at geocoding this address was made but failed completely. If an address could not be geocoded, use this code. The Geocode distributed work queue that runs in the background retries the failed address. Do not use this code for an error that is retryable, such as a network failure. If you experience an error that must retrigger geocoding later, throw an exception instead.
<code>exact</code>	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match exactly the complete address.
<code>street</code>	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the street but not the complete address. This can be more or less precise than <code>postalcode</code> , depending on the complete address and the length of the street.
<code>postalcode</code>	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the postal code but not the complete address. The meaning of a postal code match depends on the geocoding service. A geocoding service may use the geographically weighted center of the area, or it may use a designated address within the area, such as a postal office.
<code>city</code>	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the city but not the complete address. The meaning of a city match depends on the geocoding service. A geocoding service may use the geographically weighted center of the city, or it may use a designated address, such as the city hall.

**IMPORTANT** The `GeocodeStatus` type list is final. You cannot extend it with type codes of your own.





# Reinsurance Integration

This topic explains how to integrate external systems with ClaimCenter to retrieve reinsurance information for policies on claims.

This topic includes:

- “Reinsurance Overview” on page 247
- “Reinsurance Plugin” on page 247

## Reinsurance Overview

Reinsurance is insurance risk transferred to another insurance company for all or part of an assumed liability. Reinsurance can be thought of as insurance for insurance companies. When a company reinsures its liability with another company, it cedes business to that company. The amount an insurer keeps for its own account is its retention. When an insurance company or a reinsurance company accepts part of another company’s business, it assumes risk. It thus becomes a reinsurer.

**Note:** The insurance company directly selling the policy is also known in the industry as the carrier, the reinsured, or the ceding company. This topic uses the term *carrier* to refer to this company. An insurance company accepting ceded risks is known as the *reinsurer*.

## Reinsurance Plugin

ClaimCenter provides the `IReinsurancePlugin` to synchronize reinsurance information for policies on claims in ClaimCenter with current reinsurance information from an external system. To edit the registry for the `IReinsurancePlugin`, in the **Resources** pane in ClaimCenter Studio, navigate to **resources** → **gw** → **plugin** → **policy** → **reinsurance** → `IReinsurancePlugin`.

## Implementations of the Reinsurance Plugin

The default configuration of ClaimCenter provides these built-in implementations:

- `ReinsuranceDemoPlugin.gs` – Simulates integration with an external system by returning sample data to ClaimCenter
- `PCReinsurancePlugin.gs` – Integrates PolicyCenter and Reinsurance Management with ClaimCenter through the `RICoverageAPI` web service that PolicyCenter publishes.

## Enabling the Reinsurance Plugin

In the default configuration of ClaimCenter, the `IReinsurancePlugin` is enabled and uses the `ReinsuranceDemoPlugin` implementation class. This can be useful when you first begin working with reinsurance in ClaimCenter and to demonstrate the built-in reinsurance features of ClaimCenter. You must not use this implementation in a production setting.

### To enable the `PCReinsurancePlugin` implementation

1. Start ClaimCenter Studio.

At a command prompt, navigate to `ClaimCenter\bin` and enter the following command:

```
gwcc studio
```

2. In the Resources pane, navigate to `configuration → Plugins → gw → plugin → policy → reinsurance → IReinsurancePlugin`.
3. Click the `Enabled` check box to enable the plugin. If you see a message asking if you want to create a copy in the current module, click `Yes`.
4. In the `Class` field, specify:  
`gw.plugin.policy.reinsurance.pc700.PCReinsurancePlugin`
5. Save your changes.

# Encryption Integration

You can store certain data model properties encrypted in the database. For example, you can hide important data, such as bank account numbers or private personal data, by storing the data in the database in a non-plaintext format. This topic is about how to integrate ClaimCenter with your own encryption code.

This topic includes:

- “Encryption Integration Overview” on page 249
- “Changing Your Encryption Algorithm Later” on page 254
- “Encryption Changes with Snapshots” on page 256
- “Encryption Features for Staging Tables” on page 257

## Encryption Integration Overview

You can store certain data model properties encrypted in the database. For example, hide important data, such as bank account numbers or private personal data, in the database in a non-plaintext format. The actual encryption and decryption is implemented through the `IEncryption` plugin. You can define your own algorithm.

ClaimCenter does not provide an encryption algorithm in the product. ClaimCenter simply calls this plugin implementation, which is responsible for encoding an unencrypted `String` or reversing that process.

---

**IMPORTANT** ClaimCenter provides a sample encryption algorithm that simply reverses the string value. You must disable this plugin if you do not wish to use encryption. You must implement the plugin to properly encrypt your data with your own algorithm.

All encryption and decryption is done within ClaimCenter automatically as the application accesses and loads entities from the database. All Gosu rules and web service interfaces automatically operate on plain text (unencrypted text) without special coding to support encryption within rules, web services, and messaging plugins.

From Gosu, any encrypted fields appear unencrypted. If you must avoid sending this information as plain text to external systems, you must design your own integrations to use a secure protocol or encrypt that data.

For example, use the plugin registry and use your own encryption plugin to encrypt and decrypt data across the wire in your integration code. Create classes to contain your integration data for each integration point. You can make some properties contain encrypted versions of data that require extra security between systems. Such properties do not need to be database-backed. You can implement enhancement properties on entities that dynamically return the encrypted version. If your messaging layer uses encryption (for example, SSL/HTTPS) or is on a secure network, additional encryption might not be necessary. It depends on the details of your security requirements.

---

**WARNING** From Gosu, any encrypted fields appear unencrypted. Carefully consider security implications of any integrations with external systems for properties that normally are encrypted.

---

For communication between Guidewire applications, the built-in integrations do not encrypt any encrypted properties during the web services interaction. This affects the following integrations:

- ClaimCenter and ContactManager
- ClaimCenter and PolicyCenter

---

**IMPORTANT** The built-in integrations between Guidewire applications do not encrypt properties in the web service connection between the applications. To add additional security, you must customize the integration to use HTTPS or add additional encryption of properties sent across the network.

---

## Setting Encrypted Properties

You can change the encryption settings for a column in data model files by overriding the column information (adding a *column override*). You can only set encryption for character-based column types such as `varchar`. Encryption is unsupported on other types, including binary types (`varbinary`) and date types.

There are two ways to mark the column as encrypted:

- Make a column encrypted by specifying its type as `encrypted`:

```
<column name="encrypted_column" type="encrypted" size="10"/>
```

This makes the column `encrypted_column` have the type `EncryptedString`. ClaimCenter generates visible masks for all input widgets for this value.

- Alternatively, mark a column as encrypted through the `encryption` attribute, which only applies to types based on `varchar`. This masks values that are accessed directly as bean properties. For example:

```
<column name="encrypted_column" type="varchar" size="10" encryption="true"/>
```

This example makes the property `encrypted_column` to be the type `String`. In this case, input widgets for this value are masked only if the value is a bean property access. For example, suppose the earlier column is within an entity called `TestBean`. ClaimCenter generates an input mask if the value expression is the form `bean.encrypted_column`. However, there is no input mask if the expression is a method invocation like `pageHelper.getMyEncryptedColumnValue()`.

ClaimCenter prevents you from encrypting properties with a denormalized column. In other words, no encryption on a property that creates a secondary column to support case-insensitive search. For example, the contact property `LastNamesDenorm` is a denormalized column added to mirror the `LastNames` property, and thus the `LastNames` property cannot be encrypted.

## Querying Encrypted Properties

Gosu database query builder queries (and older-style `find` expressions) work as expected if you compare the property with a literal value. For example, looking up a social security number or other unique ID is comparing with a literal value. However, the encrypted data for every record is not decrypted from the database to test the results of the query. It actually works in the opposite way. If you compare a constant value against an encrypted

property, ClaimCenter encrypts the constant in the SQL query. The query embeds the equality comparison logic directly into the SQL.

For comparison logic, the SQL that Gosu generates always compares either:

- Two properties from the database, either both encrypted or both unencrypted
- A static literal, encrypted if necessary, and a property from the database.

One side effect of this is that equality comparison is the only supported comparison for encrypted properties. For example you cannot use “greater than” comparisons or “less than” comparisons.

This means that your `find` queries can compare encrypted properties against other values in only two ways:

- Direct equality comparison (`==`) or not equals (`!=`) a Gosu expression of type `String`
- Direct equality comparison (`==`) or not equals (`!=`) with a field path to another encrypted property

For example, suppose `Claim.SecretVal` and `Claim.SecretVal2` are encrypted properties, suppose `Claim.OtherVal` and `Claim.OtherVal2` are unencrypted properties, and suppose `tempValue` is a local variable containing a `String`.

The following queries work:

```
q.compare("SecretVal", Equals, "123")
q.compare("SecretVal", NotEquals, "123")
q.compare("SecretVal", Equals, tempValue)
q.compare("SecretVal", NotEquals, tempValue)
q.compare("SecretVal", Equals, q.getColumnRef("SecretVal2")) // both encrypted
q.compare("SecretVal", NotEquals, q.getColumnRef("SecretVal2")) // both encrypted
q.compare("OtherVal", Equals, q.getColumnRef("OtherVal2")) // both unencrypted
q.compare("OtherVal", NotEquals, q.getColumnRef("OtherVal2")) // both unencrypted
```

The following queries do not work:

```
q.compare("SecretVal", Equals, q.getColumnRef("OtherVal")) // bad mix of encrypted and unencrypted
q.compare("SecretVal", NotEquals, q.getColumnRef("OtherVal")) // bad mix of encrypted and unencrypted
q.compare("SecretVal", GreaterThan, "123") // no greater than or less than
```

**IMPORTANT** If a Gosu `find` query involves encrypted properties, you can use equality comparison with other encrypted properties or against `String` values constants in the query. However, there are limitations with what you can do in the query. Refer to the instructions in this section for details.

## Duplicate Check Search Template and Encryption

In the base configuration, the ClaimCenter duplicate check search template supports encryption.

## Writing Your Encryption Plugin

Write and implement a class that implements the `IEncryption` plugin interface. Its responsibility is to encrypt and decrypt data with one encryption algorithm.

To encrypt, implement an `encrypt` method that takes an unencrypted `String` and returns an encrypted `String`, which may or may not be a different length than the original text. If you want to use strong encryption and are permitted to do so legally, you are free to do so. ClaimCenter does not include any actual encryption algorithm.

To decrypt, implement a `decrypt` method that takes an encrypted `String` and returns the original unencrypted `String`.

You must also specify the maximum length of the encrypted string by implementing the `getEncryptedLength` method. Its argument is the length of decrypted data. It must return the maximum length of the encrypted data. ClaimCenter primarily uses the encryption length at application startup time during upgrades. During the upgrade process, the application must determine the required length of encrypted columns. If the length of the column must increase to accommodate inflation of encrypted data, then this method helps ClaimCenter know how far to increase space for the database column.

To uniquely identify your encryption algorithm, your plugin must return an encryption ID. Implement a `getEncryptionIdentifier` method to return a unique identifier for your encryption algorithm. This identifier tracks encryption-related change control. This is exposed to Gosu as the property `EncryptionIdentifier`:

```
override property get EncryptionIdentifier() : String {
    return "ABC:DES3"
}
```

The encryption ID is very important and must be unique among all encryption plugins in your implementation. The application decides whether to upgrade the encryption data with a new algorithm by comparing:

- The encryption ID of the current encryption plugin
- The encryption ID associated with the database last time the server ran.

For important details, see “[Changing Your Encryption Algorithm Later](#)” on page 254.

---

**IMPORTANT** Be careful that your encryption plugin returns an appropriate encryption ID and that it is unique among all your encryption plugins.

---

The following example is a simple demonstration encryption plugin. It simple fake encryption algorithm is to append a reversed version of the unencrypted text to the end of the text. For example, encrypting the text “hello” becomes “helloolleh”.

To decrypt the text, it merely removes the second half of the string. The second half of the string is the reversed part of the text that it appended earlier when encrypting the string. It is important to note that this fake algorithm doubles the size of the encrypted data, hence the `getEncryptedLength` doubles and returns the input size argument.

The following Gosu code implements this plugin

```
package Plugins
uses gw.plugin.util.IEncryption
uses java.lang.StringBuilder

class EncryptionByReversePlugin implements IEncryption {

    override function encrypt(value:String) : String {
        return reverse(value)
    }

    override function decrypt(value:String) : String {
        return value.subString(value.length() / 2, value.length());
    }

    override function getEncryptedLength(size:int) : int {
        return size * 2 // encrypting doubles the size
    }

    override property get EncryptionIdentifier() : String {
        return "mycompany:reverse"
    }

    private function reverse(value:String) : String {
        var buffer = new StringBuilder(value)
        return buffer.reverse().append(value).toString()
    }
}
```

## Detecting Accidental Duplication of Encryption or Decryption

You can mitigate the risk of accidentally encrypting already-encrypted data if you design your encryption algorithm to authoritatively detect whether the property data is already encrypted.

For example, suppose you put a known series of special unusual characters that could not appear in the data both before and after your encrypted data. You can now detect whether data is encrypted or unencrypted:

- During an encryption request, if your encryption plugin detects that the data is already encrypted, throw an exception.
- During a decryption request, if your encryption plugin detects that the data is already decrypted, throw an exception.

### If You Import Data from Staging Tables

If you import data from staging tables and use encrypted columns, there are special tools that you need to know about. See “Encryption Features for Staging Tables” on page 257.

## Installing Your Encryption Plugin

After you write your implementation of the `IEncryption` plugin, you must register your plugin implementation. You can register multiple `IEncryption` plugin implementations if you need to support changing the encryption algorithm.

### To enable your encryption plugin implementation

1. Create a new class that implements the `IEncryption` plugin interface. Be certain that your `EncryptionIdentifier` method returns a unique encryption identifier. If you change your encryption algorithm, it is critical that you change the encryption identifier for your new implementation.

**IMPORTANT** Guidewire strongly recommends you set the encryption ID for your current encryption plugin to a name that describes or names the algorithm itself. For example, "encryptDES3".

2. In Studio, navigate to **Resources** → **Configuration** → **config** → **Plugins** → **registry**
3. Right-click on **registry**, and choose **New** → **Plugin**.
4. Studio prompts you to name your new plugin. Remember that you can have than one registered implementation of an `IEncryption` plugin interface. The name field must be unique among all your plugins. This is called the *plugin name* and is particularly important for encryption plugins.

**IMPORTANT** Guidewire strongly recommends you set the plugin name for your current encryption plugin to a name that describes the algorithm itself. For example, `encryptDES3`.

5. In the **interface** field, type `IEncryption`.
6. Edit standard plugin fields in the Plugins editor in Studio. See “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*. Also see “Plugin Overview” on page 163.
7. In `config.xml`, set the `CurrentEncryptionPlugin` parameter to the plugin name.

The `CurrentEncryptionPlugin` parameter specifies which encryption plugin is the current encryption algorithm for the main database. Specify the plugin name, not the class name nor the encryption ID.

**WARNING** If the `CurrentEncryptionPlugin` parameter is missing or specifies an implementation that does not exist, the server does not start.

8. Start the server.

If the upgrader detects data model fields marked as encrypted but the database contains unencrypted versions, the upgrader encrypts the field in the main database using the current encryption plugin.

**See also**

- “Changing Your Encryption Algorithm Later” on page 254
- “Encryption Changes with Snapshots” on page 256

## Adding or Removing Encrypted Properties

If you later add or remove encrypted properties in the data model, upgrader automatically runs on server startup to update the main database to the new data model.

**IMPORTANT** The upgrader does not upgrade claim snapshots or archive databases during the normal upgrade process. For important information about upgrading claim snapshots and archive databases, see “Encryption Changes with Snapshots” on page 256.

## Set Encryption Algorithm For Archived Objects with Encrypted Properties

The `config.xml` configuration parameter `DefaultXmlExportIEncryptionId` specifies the unique encryption ID of an encryption plugin.

If archiving is enabled, the application uses that encryption plugin to encrypt any encrypted fields during XML serialization. For more about encryption unique IDs, see “Writing Your Encryption Plugin” on page 251.

**WARNING** To avoid accidentally archiving encrypted properties as unencrypted data, be sure to set the parameter `DefaultXmlExportIEncryptionId`.

## Changing Your Encryption Algorithm Later

You can register any number of `IEncryption` plugins. For an original upgrade of your database to a new encryption algorithm, you register two implementations at the same time. You might register more than two implementations if you have encrypted claim snapshots in archive databases.

However, only one encryption plugin is the *current encryption plugin*. The `config.xml` configuration parameter `CurrentEncryptionPlugin` controls this setting. It specifies which encryption plugin, among potentially multiple implementations, is the current encryption algorithm for the main database. Set the parameter to the plugin name, not the class name nor the encryption ID, for the current encryption plugin.

**Note:** When you use the Plugins editor in Studio to add an implementation of `IEncryption`, Studio prompts you for a text value to use as the plugin name for this implementation. Guidewire strongly recommends you set the plugin name for encryption plugins to names that describe the algorithm. For example, “`encryptDES3`” or “`encryptRSA128`”. Any legacy encryption plugins (if you did not originally enter a name) have the name “`IEncryption`”.

During server startup, the upgrader checks the encryption ID of data in the main database. The server compares this encryption ID with the encryption ID associated with the current encryption plugin. If the encryption IDs are different, the upgrader decrypts encrypted fields with the old encryption plugin, found by its encryption ID. Next, the server encrypts the fields to be encrypted with the new encryption plugin, found by its plugin name as specified by the parameter `CurrentEncryptionPlugin`.

The most important things to remember whenever you change encryption algorithms are:

- All encryption plugins must return their appropriate encryption IDs correctly.
- All encryption plugins must implement `getEncryptedLength` correctly.
- You must set `CurrentEncryptionPlugin` to the correct plugin name.

The server uses an internal lookup table to map all previously used encryption IDs to an incrementing integer value. This value is stored with database data. Internally, the upgrader manages this lookup table to determine whether data needs to be upgraded to the latest encryption algorithm. Do not attempt to manage this table directly. Instead, assure every encryption plugin returns its appropriate encryption ID, and assure `CurrentEncryptionPlugin` specifies the correct plugin name.

Be careful not to confuse the encryption ID of a plugin implementation with its plugin name or class name. The server relies on the encryption ID saved with the database and the encryption ID of the current encryption plugin to identify whether the encryption algorithm changed.

For claim snapshots, the encryption ID used to encrypt the snapshot is saved in the claim snapshot with each encrypted field. The upgrader does not upgrade claim snapshots or archive databases during the normal upgrade process.

#### See also

[“Encryption Changes with Snapshots” on page 256](#)

## Changing Your Encryption Algorithm

The following procedure describes how to change your encryption algorithm. It is extremely important to follow it exactly and very carefully. If you have questions about this before doing it, contact Guidewire Professional Services before proceeding.

---

**WARNING** Do not follow this procedure until you are sure you understand it and test your encryption algorithm code. Before proceeding, be confident of your encryption code, particularly your implementation of the plugin method `getEncryptedLength`. Failure to perform this procedure correctly risks data corruption.

---

1. Shut down your server.
2. Register a new plugin implementation of the `IEncryption` plugin for your new algorithm. When you add an implementation of the plugin in Studio, it prompts you for a plugin name for your new implementation. Name it appropriately to match the algorithm. For example, "encryptDES3".
3. Be sure your plugin returns an appropriate and unique encryption ID. Name it appropriately to match the algorithm. For example, "encryptDES3".
4. Set the `config.xml` configuration parameter `CurrentEncryptionPlugin` to the plugin name of your new encryption plugin.
5. Continue to register your older encryption plugins so that ClaimCenter can decrypt any encrypted data in claim snapshots in archived claims.
6. Start the server. The upgrader uses the old encryption plugin to decrypt your data and then encrypts it with the new algorithm.

#### See also

- [“Writing Your Encryption Plugin” on page 251](#)
- [“Encryption Changes with Snapshots” on page 256](#)

## Critical Warning: If You Change Encryption Algorithms, You May Still Need the Old Algorithm

If you change your encryption algorithm, it is important to realize that you may need to continue to register your encryption plugin implementations for your older algorithm.

For claim snapshots, changes to encryption do not occur immediately in the main database as part of the normal upgrader. There is a work queue that converts snapshots over time. See [“Encryption Changes with Snapshots” on page 256](#)

page 256. Even after running the upgrader, you must continue to register your encryption plugins for your old encryption algorithms. The snapshot includes the encryption ID for the plugin that encrypted the snapshot. To restore the claim, the work queue needs that encryption plugin to decrypt the snapshot. Next, ClaimCenter uses the current encryption algorithm to encrypt again.

For archived claims, changes to encryption do not occur immediately in the main database as part of the normal upgrader. Even after running the upgrader, you must continue to register encryption plugin implementations for your old encryption algorithms. The archive data includes the encryption ID for the plugin that originally encrypted the data. When ClaimCenter restores that claim, the application needs the encryption plugin for older algorithms to decrypt encrypted properties. Next, ClaimCenter uses the current encryption algorithm to encrypt any encrypted properties.

Archived claim snapshots in archive databases with outdated encryption remain in their non-upgrade state until the claim is restored.

---

**WARNING** Register one encryption plugin implementation for every encryption algorithm that might be referenced by encryption ID in any claim snapshots or archived data. Ensure that all encryption plugin implementations return the correct encryption ID.

---

## Encryption Changes with Snapshots

On server startup, upgrader updates update encryption settings in the database if you do any of the following:

- Add encrypted properties
- Remove encrypted properties
- Add an encryption plugin for the first time.
- Change the encryption algorithm. The server looks for a changed value for the encryption ID of the current encryption plugin. For related information, see “Changing Your Encryption Algorithm Later” on page 254

However, the upgrader does not upgrade claim snapshots during the normal upgrade process.

After startup, a work queue runs. Eventually that work queue converts all claim snapshots to use the current encryption settings. First the work queue decrypts if necessary then encrypts with the current encryption algorithm.

Additionally, claim snapshots can upgrade on demand for the user interface. Suppose the work queue is running but is not complete and did not upgrade some claim snapshots. If a user views one of those claim snapshots, ClaimCenter immediately upgrades the encryption in the snapshot as part of accessing the snapshot.

To control the number of claim snapshots the work queue upgrades at one time, set the `SnapshotEncryptionUpgradeChunkSize` configuration parameter. Set that parameter to the number of claims to upgrade each time the worker runs. Typically the worker runs once a day, but it is configurable.

To force the work queue to upgrade all snapshots all at once, set the `SnapshotEncryptionUpgradeChunkSize` parameter to 0. Next, after the usual database upgrade completes, manually run the Encryption Upgrade work queue writer from the tools page. However, before setting `SnapshotEncryptionUpgradeChunkSize` to 0 to process all snapshots, consider carefully the implications. Depending on the workload, this work queue can use server resources for a long time. For many companies, it is preferable to convert one chunk of snapshots every night at 1:00 a.m. when server usage is low.

For details of the Encryption Upgrade work queue, see “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide*.

### Special Issues For Changing Your Encryption Algorithm

If you change encryption algorithms, you must continue to register one encryption plugin implementation for every encryption algorithm that might be referenced by encryption ID in any claim snapshots. Ensure that all encryption plugin implementations return the correct encryption ID.

**IMPORTANT** Be sure that you understand how encryption upgrades work with claim snapshots. See “Changing Your Encryption Algorithm Later” on page 254.

## Encryption Features for Staging Tables

If you need to import records using staging tables and any data has encrypted properties, you must encrypt those properties before importing them into operational tables. To support this feature, there is a special tool that encrypts any encrypted fields in staging tables before import. This tool encrypts only the columns in the data model marked as encrypted.

During upgrades, be careful about the order that you make data model modifications. Before beginning staging table import work, do any data model changes including changing encryption settings. Next, let the server upgrade the database. After you modify the data model and increment the data model version number, ClaimCenter automatically encrypts data in the operational tables during server launch as part of upgrade routines. During upgrade, ClaimCenter drops (deletes) all staging tables.

Guidewire strongly recommends you encrypt your staging table data after you complete all work on your data conversion tools and your data passes all integrity checks. Note that staging table integrity checks never depend on the encryption status of encrypted properties. You can run integrity checks independent of whether the encrypted properties are currently encrypted.

There are several ways to encrypt your data:

- Synchronously with web services using `TableImportAPI.encryptDataOnStagingTables()`. This method takes no arguments and returns nothing.
- Synchronously with the command line tool `table_import` with the option `-encryptstagingtbls`. For more information, see “Table Import Command” on page 191 in the *System Administration Guide*.
- Asynchronously with web services using the `TableImportAPI` web service method `encryptDataOnStagingTablesAsBatchProcess`. This method takes no arguments and returns a process ID (`ProcessID`). Use the returned process ID to check the status of the encryption batch process or to terminate the encryption batch process. Use this process ID with methods on the web service `MaintenanceToolsAPI`. See “Maintenance Web Services” on page 146 for details.

#### To encrypt any encrypted properties in staging tables

1. Write and register an encryption plugin as discussed in “Encryption Integration Overview” on page 249. Carefully upgrade your existing production data after your data model change.
2. Perform integrity checks until they all pass. During development of conversion programs that populate the staging tables, you may need to repeatedly run integrity checks and modify your code.
3. Use one of the web service APIs or command line tools that encrypt columns in staging tables, as discussed earlier in this topic.

**WARNING** If the encryption process succeeds, be careful not to run the process a second time because you would corrupt your data by encrypting already-encrypted data. You can mitigate this problem by careful design of your encryption code. See “Detecting Accidental Duplication of Encryption or Decryption” on page 252

If the encryption process fails in any way, no changes are committed to the database. All changes happen in one database transaction. If an exception or other error occurs, partial work is undone. You can safely repeat the process after you fix any errors. For example, if the encryption plugin changes the length of the data field, this process could fail at the database level and throw an exception, which rolls back all changes. Be careful when fixing problems. See the earlier discussion about performing data model changes and upgrades before beginning encryption work.

4. After the encryption process succeeds, use regular staging table loading web service APIs or command line tools. For example, the web service method `integrityCheckStagingTableContentsAndLoadSourceTables` or the `table_import` command line tool. These methods run integrity checks again. Integrity checks do not behave differently with encrypted properties.
5. If you need to change your encryption algorithm later, see “Changing Your Encryption Algorithm Later” on page 254.

**For related information, see these sections:**

- For general information about staging table import, see “Importing from Database Staging Tables” on page 415.
- If you need to change your encryption algorithm after data is encrypted, see “Changing Your Encryption Algorithm Later” on page 254.
- “Table Import Command” on page 191 in the *System Administration Guide*

# Management Integration

This topic discusses *management*, which is a special type of API that allows external code to control ClaimCenter in certain ways through the JMX protocol or some similar protocol. Types of control include viewing or changing cache sizes and modifying configuration parameters dynamically. This topic focuses on writing a new management plugin.

This topic includes:

- “Management Integration Overview” on page 259
- “The Abstract Management Plugin Interface” on page 260
- “Integrating With the Included JMX Management Plugin” on page 261

**See also**

- For more information about plugins, see “Plugin Overview” on page 163.
- For the complete list of all ClaimCenter plugins, see “Summary of All ClaimCenter Plugins” on page 181.

## Management Integration Overview

ClaimCenter provides APIs to view and change some ClaimCenter settings from remote management consoles or APIs from remote integration code:

- **Configuration parameters** – View the values of all configuration parameters, and change the value of certain parameters. These changes take effect in the server immediately, without requiring the server to restart. After the server shuts down, ClaimCenter discards dynamic changes like this. After the server starts next, the server rereads parameters in the `config.xml` configuration file.
- **Batch processes** – View batch processes currently running (if any).
- **Users** – View the number of current user sessions and their user names
- **Database connections** – View the number of active and idle database connections
- **Notifications** – View notifications about locking out users due to excessive login failures.

ClaimCenter exposes this data through three mechanisms:

- **User interface in Server Tools tab, on the Management Beans page** – All of these items are exposed in the user interface on the Management Beans page of the ClaimCenter Server Tools tab. This tab is accessible only to users with the soapadmin permission. For more information, see “Management Beans” on page 163 in the *System Administration Guide*.
- **Abstract management plugin interface** – ClaimCenter also provides an abstract interface called the *management plugin*. This plugin exposes *management beans* to the external world. Management beans are interfaces to read or control system settings. For example, the management plugin could expose the management beans using the Java Management Extension (JMX) protocol, using the Simple Network Management Protocol (SNMP), or something else entirely.
- **A supported JMX management plugin for Apache Tomcat** – ClaimCenter includes an example plugin that implements JMX using the management plugin. This plugin is compiled and automatically installed in `ClaimCenter/modules/configuration/plugins` so that you need only to enable it in Studio. The included JMX management plugin works only with the Apache Tomcat web server. The source code for this example is included in at this path:

`ClaimCenter\java-api\examples\src\examples\plugins\management`

## The Abstract Management Plugin Interface

The following table lists the important interfaces and classes used by the abstract management plugin interface.

Interface or class	Description
<code>ManagementPlugin</code>	The management interface for Guidewire systems. This is the interface that the included JMX plugin implements.
<code>GWMBean</code>	The interface for managed beans exposed by the system. The management plugin must expose these internal management beans to the outside world using JMX, SNMP, or some other management interface.
<code>GWMBeanInfo</code>	Meta-information about a Guidewire managed bean ( <code>GWMBean</code> ), such as its name and description.
<code>ManagementAuthorizationCallbackHandler</code>	A callback handler interface that ClaimCenter invokes if a user attempts management operations.
<code>NotificationSenderMarker</code>	A marker interface that indicates that a <code>GWMBean</code> can send notifications.
<code>Attribute</code>	The <code>Attribute</code> class encapsulates two strings that represent a system attribute name and its value.
<code>AttributeInfo</code>	Metadata about an <code>Attribute</code> , such as name, description, type, and whether the attribute is readable and/or writable.
<code>Notification</code>	A ClaimCenter notification, such as those sent if ClaimCenter locks out a user due to too many failed login attempts.
<code>NotificationInfo</code>	Metadata about a notification, such as a message string, a sequence number, and a notification type.

The main task of the management plugin is to register `GWMBean` objects. Registering the object means that the plugin provides that service to the outside world in whatever way makes sense for that service. For example, the management plugin might create a wrapper for the `GWMBean` and expose it using JMX or SNMP using its own published service.

For details, refer to these Java source files in the directory `ClaimCenter/java-api/examples`:

- `JMXManagementPlugin.java`
- `GWMBeanWrapper.java`
- `JMXAuthenticatorImpl.java`
- `JSR160Connector.java`

**See also**

- To write code for an external system to communicate with the built-in JMX management plugin, see “Integrating With the Included JMX Management Plugin” on page 261.

## Integrating With the Included JMX Management Plugin

ClaimCenter includes an example plugin that implements JMX using the management plugin interface. This plugin is compiled and automatically installed in `ClaimCenter/modules/configuration/plugins` so that you need only to change one setting in the plugin registry in Studio to enable it.

In the Plugins editor, select **Resources** → **Plugins** → `gw` → **plugin** → **management** → **ManagementPlugin**, then check the box for **Enabled**. This supported JMX management plugin works only with the Apache Tomcat web server. The source code for this example is included as part of the ClaimCenter examples directory, at the following path:

`ClaimCenter\java-api\examples\src\examples\plugins\management`

**Note:** JMX support differs between web servers, but the JMX management plugin is designed for and supported in Guidewire production environments that use Apache Tomcat only. The JMX management plugin is supported for jetty in development environments. Modifications to the source code and use of different JMX libraries could enable the management plugin for other web servers. Guidewire provides the source code in case such changes are desired.

After you enable the JMX plugin, it publishes a JMX service called a *JMX JSR160 connector* under Apache Tomcat. A remote management console or your integration code can call the JMX JSR160 connector by creating *JMX connector client* code that connects to the published service.

The following example demonstrates a simple Java application that can retrieve a system attribute programmatically from a remote system. This example relies on the following conditions:

- The ClaimCenter server must be running under the Apache Tomcat web server.
- The JMX plugin must be installed (which it is by default).
- The JMX plugin must be enabled in the Plugins editor

This example retrieves the `HolidayList` system attribute programmatically from a remote system:

```
package com.mycompany.cc.integration.jmx;

import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;

// Create a "JMX JSR160 connector" to connect to ClaimCenter
//
public class TestJMXClientConnector {

    public static void main(String[] args) {

        try {
            // The address of the connector server
            JMXServiceURL address = new JMXServiceURL(
                "service:jmx:rmi:///jndi/rmi://akitio:1099/jrmp");

            // The creation environment map, null in this case
            Map creationEnvironment = null;

            // Create the JMXConnectorServer
            JMXConnector ctor = JMXConnectorFactory.newJMXConnector(address,
                creationEnvironment);

            // May contain - for example - user's credentials
            Map environment = new HashMap();
        }
    }
}
```

```
environment.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.rmi.registry.RegistryContextFactory");
environment.put(Context.PROVIDER_URL, "rmi://localhost:1099");
String[] credentials = new String[]{"su", "cc"};
environment.put(JMXConnector.CREDENTIALS, credentials);

// Connect
cntor.connect(environment);

// Obtain a "stub" for the remote MBeanServer
MBeanServerConnection mbsc = cntor.getMBeanServerConnection();

// Call the remote MBeanServer
String domain = mbsc.getDefaultDomain();
ObjectName delegate =
    ObjectName.getInstance("com.guidewire.pl.system.configuration:type=configuration");
String holidayList = (String)mbsc.getAttribute(delegate, "HolidayList");
System.out.println(holidayList);
}
catch (Exception e) {
    e.printStackTrace();
}
}
}
```

# Other Plugin Interfaces

*Plugins* are software modules that ClaimCenter calls to perform an action or calculate a result. This topic describes plugin interfaces that are not discussed in detail elsewhere in this documentation.

This topic includes:

- “Claim Number Generator Plugin” on page 263
- “Approval Plugin” on page 264
- “Automatic Address Completion and Fill-in Plugin” on page 265
- “Phone Number Normalizer Plugin” on page 265
- “Testing Clock Plugin (Only For Non-Production Servers)” on page 265
- “Work Item Priority Plugin” on page 267
- “Official IDs Mapped to Tax IDs Plugin” on page 267
- “Preupdate Handler Plugin” on page 267
- “Defining Base URLs for Fully-Qualified Domain Names” on page 269
- “Exception and Escalation Plugins” on page 270

**See also**

- For general information about plugins, see “Plugin Overview” on page 163.
- For messaging plugins, see “Messaging and Events” on page 299.
- For authentication plugins, see “Authentication Integration” on page 187.
- For document and form plugins, see “Document Management” on page 199.
- For geocoding plugins, see “Geographic Data Integration” on page 235.

## Claim Number Generator Plugin

The claim number generator plugin (`IClaimNumGenAdapter`) is responsible for generating a claim number. There are two methods of this plugin that you must define, `generateNewClaimNumber` and `generateTempClaimNumber`, each of which must generate a unique claim number.

The `generateTempClaimNumber` can generate a different format of claim number appropriate for temporary use. For example, if you create the claim but it is not yet complete, the claim number is temporary. If the claim data is complete and soon opens, ClaimCenter calls the plugin method `generateNewClaimNumber` for its permanent claim number.

Both methods take a template data string generated by the `IClaimNumGenAdapter_Claim` plugin template. It is a Gosu template file that extracts data from an entity and sends specific properties to the plugin. For more information about plugin templates, see “Writing Plugin Templates For Plugins That Take Template Data” on page 172.

For an example implementation of this plugin:

- ...as a Gosu plugin, see “Example Gosu Plugin” on page 169.
- ...as a Java plugin, see “Special Notes For Java Plugins” on page 170.

ClaimCenter includes a feature called *field validators* that help you ensure accurate input within the web application user interface. In the case of claim numbers, field validators ensure that user input of a claim number exactly matches the correct format. For example, the following example shows a claim number field validator.

```
<ValidatorDef name="ClaimNumber" value="[0-9]{3}-[0-9]{5}"  
    description="Validator.ClaimNumber"  
    input-mask="###-#####"/>
```

If you implement claim number generator code, remember to update the claim number field validator so that it matches the actual format of your claim numbers. For details about field validator configuration, see “Field Validation” on page 251 in the *Configuration Guide*.

---

**WARNING** If you first implement a claim number generator or later change the claim number format, always update the field validators for claim numbers or data entry of claim numbers fails.

---

Typically, ClaimCenter calls this plugin only once for each claim. However, there are rare cases in which the new claim number generator may get called twice for the same claim. For example, if the claim gets a validation error, then the user logs out of ClaimCenter, finds the claim as a draft, and then saves it again.

There is also a method called `cancelNewClaimNumber`, which you must implement, but it is acceptable to have the method do nothing. This method gives plugin implementors a chance to cancel a previously allocated claim number. It is called in the rare case that the `generateNewClaimNumber` method was called to allocate a claim number but the claim fails validation and is subsequently canceled or otherwise unfinished. In this case, ClaimCenter calls the `cancelNewClaimNumber` method to give the claim number generator chance to reuse the previously-allocated number. It takes a claim number and also a template data string that you provide.

---

**WARNING** Even if you implement `cancelNewClaimNumber` and perform some intelligent reacquiring algorithm for the number, a small chance remains for losing a claim number. For example, a power failure after allocating the claim number but before the change commits to the database.

---

## Approval Plugin

The approval plugin implementation answers two questions. First, does the set of transactions need approval? Second, if so, who must give that approval? If no external plugin is implemented, ClaimCenter answers the first question by comparing the transaction set to the user’s authority limits as described in the ClaimCenter System Administration Guide. If approval is required, then ClaimCenter consults the Approval rule set to answer the second question.

Your approval plugin must have two methods that correspond to these two questions:

- `requiresApproval` – Determines whether the user has any authority and, if so, whether the user has sufficient authority. The `ApprovalResult` object returns these two answers along with a list of messages indicating why approval is required.

- `getApprovingUser` – Determines the user (and group) to assign the approval activity

Both of these methods are passed some information about the transactions in the `templateData` parameter. ClaimCenter looks for a template named `Approval_TransactionSet.gs` with root object `TransactionSet`.

## Automatic Address Completion and Fill-in Plugin

To support automatic address completion, implement the `IAddressAutocompletePlugin` plugin interface.

There is a default implementation called `DefaultAddressAutocompletePlugin`, which handles the default behavior.

The address automatic completion plugin interface defines the following functionality.

- The plugin provides a `createHandler` factory method for creating `AutocompleteHandler` objects. The default implementation creates an `AddressAutocompleteHandler`.
- You can add fields for automatic completion by extending `AddressAutofillable.eti`, `AddressAutofillDelegate.gs` and `AddressFillableExtension.gs`. Guidewire recommends that these three files always support exactly the same set of fields.

Contact Guidewire Customer Support for more information.

### See also

- “Automatic Address Completion and Fill-in” on page 112 in the *Globalization Guide*

## Phone Number Normalizer Plugin

To support automatic address completion, implement the `IPhoneNormalizerPlugin` plugin interface.

There is a default implementation called `DefaultPhoneNormalizerPlugin`, which handles the default behavior.

Contact Guidewire Customer Support for more information.

### See also

- “DefaultNANPACountryCode” on page 65 in the *Configuration Guide*

## Testing Clock Plugin (Only For Non-Production Servers)

---

**WARNING** The `ITestingClock` plugin is supported only for testing on non-production development servers. Do not register this plugin on production servers.

---

Testing ClaimCenter behavior over a long span of time during multiple billing cycles can be challenging. For testing on development servers only, you can implement the `ITestingClock` plugin and programmatically change the system time to simulate the passing of time. For example, you can define a plugin that returns the real time except in special cases in which you artificially increase the time to represent a time delay. The delay could be one week, one month, or one year.

This plugin interface has only two methods, `getCurrentTime` and `setCurrentTime`, which get and set the current time using the standard ClaimCenter format of milliseconds stored in a `long` integer.

If you cannot set the time in the `setCurrentTime` function, for example if you are using an external “time server” and it temporarily cannot be reached, throw the exception `java.lang.IllegalArgumentException`.

Time must always increase, not go back in time. Going back in time is likely to cause unpredictable behavior in ClaimCenter.

**Notes:**

- The plugin method `setCurrentTime` advances the time only for `gw.api.util.DateUtil.currentDate`. The class `java.util.Date` always states the server time. Therefore, you must use `gw.api.util.DateUtil.currentDate` in your implementation code.
- The advanced date does not change in all parts of the product. There are certain areas that intentionally do not use the rolled-over date. For example, the time shown in the next scheduled run for batch processes is not affected.
- Today's date on the **Calendar** does not change.

## Using the Testing Clock Plugin

The base application has an implementation of the `ITestingClock` plugin interface. This topic shows you how to use it.

1. Start ClaimCenter Studio.
2. Navigate in the Project pane to **configuration** → **config** → **Plugins** → **registry**.
3. Right-click **registry** and click **New** → **New Plugin**.
4. In the name field, type `ITestingClock`.
5. In the interface field, type `ITestingClock`.
6. Click **OK**.
7. In the Plugin editor, click the green plus (+) sign and choose **Java Plugin**
8. In the **Java class** field, type the following:  
`com.guidewire.pl.plugin.system.internal.OffsetTestingClock`
9. In the Project window, navigate to **configuration** → **config** → **config.xml**.
10. In `config.xml` under Environment Parameters, set `EnableInternalDebugTools` to true. If you see a message asking if you want to edit the file, click Yes.  
`<!-- Enable internal debug tools page http://localhost:8080/app/InternalTools.do -->`  
`<param name="EnableInternalDebugTools" value="true"/>`
11. Save your changes.
12. Start ClaimCenter.
13. Log in as user `su` with password `gw`.
14. Enter the following text in the Quick Jump field on the upper right, and then click **Go**:  
`Run Clock addDays 10`
15. In the yellow message area near the top of the screen, a message shows the current date to be ten days from today.

## Testing Clock Plugin in ClaimCenter Clusters

If you are operating a cluster of ClaimCenter servers, you must use the following procedure to change the time.

**To change the testing clock time for ClaimCenter clusters**

1. Stop all servers with the exception of the *batch server*.
2. Advance the testing clock.

3. Restart all the cluster nodes.

## Work Item Priority Plugin

Customize how ClaimCenter calculates work item priority by implementing the work item priority plugin (`IWorkItemPriorityPlugin`). It is a simple interface with one method, `getWorkItemPriority`, which takes a `WorkItem` parameter.

Your method must return a priority number, which must be a non-negative integer. A higher integer indicates to handle this work item before lower priority work items. The default work item priority if there is no plugin implementation is zero. You can raise the priority of certain work items in your implementation by returning positive priority numbers.

If more than one work item exists with the same priority number, ClaimCenter chooses the one with the earliest creation time.

## Official IDs Mapped to Tax IDs Plugin

Contacts in the real world have many official IDs. However, only one of a contact's official IDs is used as a tax ID. In the default ClaimCenter data model, each `Contact` instance has a single `TaxID` field. Also, each `Contact` instance has an `OfficialIDs` array field. The array holds all sorts of official IDs for a contact, including the contact's official tax ID.

You configure ClaimCenter with official ID types in the `OfficialID` typelist. For example, the base configuration includes type keys for the U.S. Social Security Number (SSN) and the U.S. Federal Employer Identification Number (FEIN). The `OfficialIDs` array on a `Contact` instance contains instances of the `OfficialID` entity type, which has a field for the `OfficialIDType` typekey of the instance.

Create a Gosu implementation of the `OfficialIdToTaxIdMappingPlugin` interface to configure which typekeys from the `OfficialID` typelist you want to treat as official tax IDs. The plugin interface has one method, `isTaxId`, which takes a typekey from the `OfficialIDType` typelist (`oIdType`). The method returns `true` if you want to treat that official ID type as a tax ID. The default Java implementation that ClaimCenter provides always returns `false`.

The following sample Gosu implementation of the `OfficialIdToTaxIdMappingPlugin` interface returns `true` for official ID typekeys that have SSN or FEIN as codes.

```
uses gw.plugin.contact.OfficialIdToTaxIdMappingPlugin

class MyOfficialIdToTaxIdMappingPlugin implements OfficialIdToTaxIdMappingPlugin {

    /**
     * Return true if the official ID type is either an SSN or a FEIN.
     */
    override function isTaxId(oIdType : OfficialIDType) : boolean {
        return OfficialIDType.TC_SSN == oIdType or OfficialIDType.TC_FEIN == oIdType
    }
}
```

## Preupdate Handler Plugin

In some cases it makes sense to implement preupdate handling in plugin code rather than in a rule set. To implement preupdate handling in plugin code, register an implementation of the `IPreUpdateHandler` plugin interface.

It is best to use `IPreUpdateHandler` plugin instead of `Preupdate Rules` in the following cases:

- **If you need to create entity instances** – If you have tasks that must create new entity instances, the plugin approach is best. The application calls the `IPreUpdateHandler` plugin before calling the rule set. The Preupdate rules run on all new entities created by the `IPreUpdateHandler` plugin implementation. You can separate your code into tasks that create objects in the plugin, and other tasks in the rule set. Without using the plugin, a Preupdate rule that creates a new entity must incorporate the logic in Preupdate rules that explicitly check the new entity instances also.
- **If you need to manipulate removed / deleted entity instances** – Similarly, this plugin is good for tasks that must deal with removed objects, entity instances that on commit will delete or retire in the database. A Preupdate rule set for a particular entity type does not run simply for removal of the deleted object. Only add or change of an entity instance triggers Preupdate and Validation rule sets for that entity type. However, the `IPreUpdateHandler` has direct access to removed entity instances in the bundle. Therefore, it is best that work that involves removed objects, and optionally new and changed objects, happen in the `IPreUpdateHandler` plugin implementation.

For example in ClaimCenter, deleting a check also deletes its payments and other subobjects. The action only retires `Transaction` entities, and no `Transaction` entities are added or changed. Therefore, the `TransactionSet` Preupdate rules do not run. This behavior might interact with other customizations you might add. For example, suppose you maintain a property on another entity type to track payment totals of a certain type. You must track add, change, and delete for `Payment` objects to update the total. However, a `TransactionSet` Preupdate rule to do this fails to catch removal of payments when the entire `Check` deletes in the user interface. Relying on a `TransactionSet` Preupdate rule thus requires additional code to handle this edge case. In contrast, you can do all this work in one place by using the `IPreUpdateHandler` plugin.

Registering and enabling this plugin is sufficient for ClaimCenter to call the plugin for preupdate, independent of the server `config.xml` parameter `UseOldStylePreUpdate`.

Next, if the server `config.xml` parameter `UseOldStylePreUpdate` is set to `true`, ClaimCenter additionally calls the validation-graph based Preupdate rule set after calling the plugin.

Your plugin implementation must implement one method, `executePreUpdate`. This method takes a single preupdate context object, `PreUpdateContext`, and returns nothing.

The `PreUpdateContext` object has several properties you can get, which return a list (`java.util.List`) of entities in the current database transaction.

From Gosu they look like the following properties:

- `InsertedBeans` – An unordered list of entities added in this transaction.
- `UpdatedBeans` – An unordered list of entities changed in this transaction.
- `RemovedBeans` – An unordered list of entities added in this transaction.

From Java, these properties appear as three methods: `getInsertedBeans`, `getUpdatedBeans`, and `getRemovedBeans`.

For an overview of preupdate rules, see “Preupdate Rule Set Category” on page 63 in the *Rules Guide*

### Default Plugin Implementation

In the base configuration, ClaimCenter provides the plugin implementation `gw.plugin.preupdate.impl.CCPreUpdateHandlerImpl`.

By default, this plugin implementation class collects all inserted and updated beans.

Any exception cause the bounding database transaction to roll back, effectively undoing the update.

The ClaimCenter default implementation creates `Exposure` for Workers Compensation claims and calls Reinsurance plugins as appropriate.

## Defining Base URLs for Fully-Qualified Domain Names

If ClaimCenter generates HTML pages, it typically generates a *base URL* for the HTML page using a tag such as `<base href="...">` at the top of the page. In almost all cases, ClaimCenter generates the most appropriate base URL, based on settings in `config.xml`.

In some cases, this behavior is inappropriate. For example, suppose you hide ClaimCenter behind a load balancing router that handles Secure Socket Layer (SSL) communication. In such a case, the external URL would include the prefix `https://`. The load balancer handles security and forwards a non-secure HTTP request to ClaimCenter with a URL prefix `http://`. The default implementation of the base URL includes the URL prefix `http://`.

The load balancer would not typically parse the HTML enough to know about this problem, so the base URL at the user starts with `http` instead of `https`. This breaks image loading and display because the browser tries to load the images relative to the `http` URL. The load balancer rejects the requests because they are insecure because they do not use HTTPS/SSL.

Avoid this problem by writing a custom base URL builder plugin (`IBaseURLBuilder`) plugin and registering it with the system.

You can base your implementation on the built-in example implementation found at the path:

```
ClaimCenter/java-api/examples/src/examples/plugins/baseurlbuilder
```

To handle the load balancer case mentioned earlier, the base URL builder plugin can look at the HTTP request's header. If a property that you designate exists to indicate that the request came from the load balancer, return a base URL with the prefix `https` instead of `http`.

The built-in plugin implementation provides a parameter `FqdnForUrlRewrite` which is not set by default. If you enable browser-side integration features, you must specify this parameter to rewrite the URL for the external fully-qualified domain name (FQDN). The JavaScript security model prevents access across different domains. Therefore, if ClaimCenter and other third-party applications are installed on different hosts, the URLs must contain fully-qualified domain names. The fully-qualified domain name must be in the same domain. If the `FqdnForUrlRewrite` parameter is not set, the end user is responsible for entering a URL with a fully-qualified domain name.

There is another parameter called `auto` which tries to auto-configure the domain name. This setting is not recommended for clustering environments. For example, do not use this if the web server and application server are not on the same machine, or if multiple virtual hosts live in the same machine. In these cases, it is unlikely for the plugin to figure out the fully-qualified domain name automatically.

In Studio, under **Resources**, click **Plugins** → `IBaseURLBuilder`, then add the parameter `FqdnForUrlRewrite` with the value of your domain name, such as `"mycompany.com"`. The domain name must specify the Fully Qualified Domain Name to be enforced in the URL. If the value is set to `"auto"`, the default plugin implementation makes the best effort to calculate the server FQDN from the underlying configuration.

### Implement `IBaseURLBuilder` and `InitializablePlugin`

Your `IBaseURLBuilder` plugin must explicitly implement `InitializablePlugin` in the class definition. Otherwise, ClaimCenter does not initialize your plugin.

For example, suppose your plugin implementation's first line looks like this:

```
class MyURLBuilder implements IBaseURLBuilder {
```

Change it to this:

```
class MyURLBuilder implements IBaseURLBuilder, InitializablePlugin {
```

To conform to the new interface, you must also implement a `setParameters` method even if you do not need parameters from the plugin registry:

```
function setParameters(map: java.util.Map) { // this is part of InitializablePlugin
```

```
// access values in the MAP to get parameters defined in plugin registry in Studio  
}
```

## Exception and Escalation Plugins

There are several optional exception and escalation plugins. By default, they just call the associated rule sets and perform no other function. Implement your own version of the plugin and register it in Studio if you want something other than the default behavior.

Use these plugins for the following tasks:

- Add additional logic before or after calling the rule set definitions in Studio
- Completely replace the logic of the rule set definitions in Studio.

One reason you might want to completely replace the logic of the rule set definitions in Studio is to make your code more easily tested using unit tests.

The following table lists the exception and escalation plugins:

Name	Plugin interface	Default action
Activity escalation	IActivityEscalationPlugin	Calls the activity escalation rule set
Group exceptions	IGroupExceptionPlugin	Calls the group exception rule set
User exceptions	IUserExceptionPlugin	Calls the user exception rule set

# Startable Plugins

## What are Startable Plugins?

You can register custom code that runs at server startup in the form of a *startable plugin* implementation. You can use this type of plugin as a listener, such as listening to a JMS queue. You can selectively start or stop each startable plugin in the Internal Tools interface, unlike other types of plugins.

For a cluster, by default startable plugins only operate on the batch server. However, you can configure it to run on all servers on the cluster. For more information, see “Configuring Startable Plugins to Run on All Servers” on page 274.

Startable plugins have similarities with the messaging reply plugin. They typically implement some sort of listener code that initially runs at start up. A messaging transport’s main task is to send a message to an external system and listen for the reply acknowledgment. However, this metaphor might not apply to some integrations if you do not need send outgoing messages about data changes using the event and messaging system. However, if you need listener code and it must initialize with server startup, use a startable plugin.

**Note:** Some integrations require external code to trigger an action within ClaimCenter but do not need custom listener code. Depending on your needs, consider publishing a ClaimCenter web service instead of a startable plugin. For more information, see “Web Services Introduction” on page 31.

For ClaimCenter users with administration privileges, you can view the status of each startable plugin and also start or stop each one. To view the status, go to **Server Tools** → **Startable Plugin**. You can change the PCF files for this user interface. For example, you could customize this view to provide detailed information on your startable plugin, on its underlying transport mechanism, or any other information.

Register your startable plugin in Studio just like any other plugin. Find this plugin interface in the Studio Plugins Registry editor in the following location **Configuration** → **Plugins** → **gw** → **api** → **startable** → **IStartablePlugin**. Right-click on that interface name and choose **Add implementation...** This location is slightly different than most plugins, which are in **Configuration** → **Plugins** → **gw** → **plugin**.

---

**IMPORTANT** For another type of customer-defined background process, see “Custom Batch Processes” on page 587.

---

## Writing a Startable Plugin

The basics for a startable plugin are relatively straightforward. Most of your coding of a startable plugin is implementing your custom listener code or other special service. Write a new class that implements the `IStartablePlugin` interface and implement the following methods:

- A `start` method to start your service (at startup, or from the [Server Tools → Startable Plugin](#) page)
- A `stop` method to stop your service (from the [Server Tools → Startable Plugin](#) page)
- A property accessor function to get the `State` property from your startable plugin. This method returns a typecode from the typelist `StartablePluginState`: the value `Stopped` or `Started`. The administration user interface uses this property accessor to show the state to the user. Define a private variable to hold the current state and your property accessor (`get`) function can look simple:

```
// private variables...
var _state = StartablePluginState.Stopped;

...
// property accessor (get) function...
override property get State() : StartablePluginState {
    return _state // return our private variable
}
```

Alternatively, combine the variable definition with the shortcut keyword as to simplify your code. You can combine the variable definition with the property definition can be combined with the single variable definition:

```
var _state : StartablePluginState as State
```

The plugin does include a constructor called on system startup. However, start your service code in the `start` method, not the constructor. Your start method must appropriately set the state (started or stopped) using an internal variable that you define.

At minimum, your start method must set your internal variable that tracks your started or stopped state, with code such as:

```
_state = Started
```

Additionally, in your `start` method, start any listener code or threads such as a JMS queue listener.

Your `start` method has a method parameter that is a callback handler of type `StartablePluginCallbackHandler`. This callback is important if your startable plugin modifies any entity data, which is likely the case for most real-world startable plugins.

Your plugin must run any code that affects entity data within a code block that you pass to the callback handler method called `execute`. The `execute` method takes as its argument a *Gosu block*, which is a special type of in-line function. The Gosu block you pass to the `execute` method takes no arguments. For more information, see “[Gosu Blocks](#)” on page 231 in the *Gosu Reference Guide*.

**Note:** If you do not need a user context, use the simplest version of the callback handler method `execute`, whose one argument is the Gosu block. To run your code as a specific user, see “[User Contexts for Startable Plugins](#)” on page 273.

You do not need to create a separate bundle. If you create new entities to the current bundle, they are in the correct (default) database transaction the application sets up for you. Use the code `Transaction.getCurrent()` to get the current bundle if you need a reference to the current (writable) bundle. For more information, see “[Bundles and Database Transactions](#)” on page 331 in the *Gosu Reference Guide*.

---

**IMPORTANT** The Java language does not directly support blocks. If you implement your plugin in Java, you cannot use a Gosu block but you can use an anonymous class to do the same thing. For more information, see “[If You Define Your Startable Plugin In Java](#)” on page 277.

The plugin’s `start` method also includes a `boolean` variable that indicates whether the server is starting. If `true`, the server is starting up. If `false`, the start request comes from the [Server Tools](#) user interface.

The following shows a simple Gosu block that changes entity data. This example assumes your listener defined a variable `messageBody` with information from your remote system. If you get entities from a database query, remember to add them to the current bundle. Refer to “Updating Entity Instances in Query Results” on page 170 in the *Gosu Reference Guide* for more information about bundles.

This simple example queries all User entities and sets a property on results of the query:

```
//variable definition earlier in your class...
var _callback : StartablePluginCallbackHandler;
...

override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) : void
{
    _callback = cbh
    _callback.execute( \ -> {

        var q = gw.api.database.Query.make(User) // run a query
        var b = gw.Transaction.Transaction.Current // get the current bundle

        for (e in q.select()) {

            // add entity instance to writable bundle, then save and only modify that result
            var writable_object = bundle.add(e)

            // modify properties as desired on the result of bundle.add(e)
            writable_object.Department = "Example of setting a property on a writable entity instance."
        }
    })
//Note: You do not need to commit the bundle here. The execute method commits after your block runs.
}
```

Note that to make an entity instance writable, save and use the return result of the bundle add method. For more information, see “Adding Entity Instances to Bundles” on page 334 in the *Gosu Reference Guide*.

Just like your `start` method must set your internal variable that sets its state to started, your `stop` method must set your internal state variable to `StartablePluginState.Stopped`. Additionally, stop whatever background processes or listeners you started in your `start` method. For example, if your `start` method creates a new JMS queue listener, your `stop` method destroys the listener. Similar to the `start` method’s `isStartingUp` parameter, the `stop` method includes a `boolean` variable that indicates whether the server is shutting down now. If `true`, the server is shutting down. If `false`, the stop request comes from the `Server Tools` user interface.

## User Contexts for Startable Plugins

If you use the simplest method signature for the `execute` method on `StartablePluginCallbackHandler`, your code does not run with a current ClaimCenter user. Any code that directly or indirectly runs due to this plugin (including preupdate rules or any other code) must be prepared for the current user to be `null`. You must not rely on non-null values for current user if you use this version.

However, there are alternate method signatures for the `execute` method. Use these to perform your startable plugin tasks as a specific `User`. Depending on the method variant, pass either a user name or the actual `User` entity.

On a related note, the `gw.transaction.Transaction` class has an alternate version of the `runWithNewBundle` method to create a bundle with a specific user associated with it. You can use this in contexts in which there is no built-in user context or you need to use different users for different parts of your tasks. The method signature is:

```
gw.transaction.Transaction.runWithNewBundle(\ bundle -> YOUR_BLOCK_BODY, user)
```

For the second method argument to `runWithNewBundle`, pass either a `User` entity or a `String` that is the user name.

## Simple Startable Plugin Example

The following is a complete simple startable plugin. This example does not do anything useful in its start method, but demonstrates the basic structure of creating a block that you pass to the callback handler's execute method:

```
package gw.api.startableplugin
uses gw.api.startable.IStartablePlugin
uses gw.api.startable.StartablePluginCallbackHandler
uses gw.api.startable.StartablePluginState

class HelloWorldStartablePlugin implements IStartablePlugin
{
    var _state = StartablePluginState.Stopped;
    var _callback : StartablePluginCallbackHandler;

    construct()
    {

    }

    override property get State() : StartablePluginState
    {
        return _state
    }

    override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) : void
    {
        _callback = cbh
        _callback.execute( \ -> {
            // Do some work:
            // [...]
        } )
        _state = Started
        _callback.log( "*** From HelloWorldStartablePlugin: Hello world." )
    }

    override function stop( isShuttingDown: boolean ) : void
    {
        _callback.log( "*** From HelloWorldStartablePlugin: Goodbye." )
        _callback = null
        _state = Stopped
    }
}
```

## Configuring Startable Plugins to Run on All Servers

For a cluster, by default startable plugins only operate on the batch server. However, you can configure it to run on all servers on the cluster by declaring the startable plugin is *distributed*. To declare the plugin is distributed, add the @Distributed annotation to the startable plugin implementation class.

Guidewire strongly recommends that startable plugins very clearly and consistently save the state (started or stopped) across all servers in a cluster. This approach handles edge cases like a server joining the cluster late after other servers have started. To keep the state consistent across all servers, the state must persist in the database. Even though the state information in the database is not automatically persisted, the broadcast to other servers happens automatically.

The series of steps is as follows:

1. On any server, the server starts. The plugin method reads the second argument to the `start` method. Because the server is starting up, this value is `true`. The plugin implementation detects this condition and reads the database status of the started or stopped state and sets its own state appropriately.
2. Later, the state may changes on a server to start or stop from the user interface or through APIs. The application propagates this information to all servers in the cluster. Each server in the cluster decides whether it is appropriate to respect this request:
  - If the startable plugin is distributed, the server respects the request and calls the plugin `start` or `stop` method as appropriate.
  - If the startable plugin is not distributed, only the batch server handles the request.

However, if the startable plugin is not distributed and the original changed server is the batch server, the behavior is different. The batch server does not bother with the distributed broadcast system because it is the only running version of the startable plugin. Instead, that server immediately calls the plugin `start` or `stop` method as appropriate.

3. On any server, the server shuts down. The plugin method reads the second argument to the `stop` method. Because the server is shutting down, this value is `true`. The plugin implementation detects this condition and simply stops its local state but does not set the database status in the database.

Implement getting and setting the state information in the database using the plugin callback handler object, which is an instance of `StartablePluginCallbackHandler`. This object is a parameter to the `start` method of the plugin. Save a copy of this object as a private variable and then call the following methods on the object as needed:

- `getState` – Gets the state information in the database: `true` for started, `false` for stopped. If this is the first time in history that this startable plugin has ever run on this server-database cluster combination, its state information is not in the database. This method takes one arguments that represents the default assumption to use the very first time this query is checked. For example, if you expect this startable plugin to always run, pass the value `Started`. If this startable plugin runs only in rare situations and you expect only an administrator or API to trigger it to start, pass the value `Stopped`. If the server state has ever been set in the database, `getState` ignores this argument.
- `setState` – Sets the state information in the database for this startable plugin. This method takes one argument, which is true if and only if the startable plugin is running. This request attempts to set this information in the database if it is not yet set to the expected value. If the startable plugin state is already matching the database value for the state according to the `getState` method, do not call `setState`. Be careful to catch any exceptions. You must create your own bundle to make this change, see the example for how to use the `runWithNewBundle` API.
- `logStart` – Writes a line in the log about the plugin starting. This method takes one argument, which is line to log, as a `String`. Include the plugin name and the state, such as `"HelloWorldDistributedStartablePlugin:Started"`.
- `logStop` – Writes a line in the log about the plugin stopping. This method takes one argument, which is line to log, as a `String`. Include the plugin name and the state, such as `"HelloWorldDistributedStartablePlugin:Stopped"`.

The following is an example implementation of the `start` and `stop` methods that implement a trivial thread and the proper setting of startable plugin state.

```
package gw.api.startableplugin
uses gw.api.startable.IStartablePlugin
uses gw.api.startable.StartablePluginCallbackHandler
uses gw.api.startable.StartablePluginState
uses gw.transaction.Transaction
uses java.lang.Thread
uses gw.api.util.Logger

@Distributed
class HelloWorldDistributedStartablePlugin implements IStartablePlugin {
    var _state : StartablePluginState;
    var _startedHowManyTimes = 0
    var _callback : StartablePluginCallbackHandler;
    var _thread : Thread

    override property get State() : StartablePluginState {
        return _state
    }

    override function start( handler : StartablePluginCallbackHandler, serverStartup: boolean ) : void {
        _callback = handler
        if (serverStartup) {
            // if the server is starting up, read the value from the database, and default to stopped
            // if this plugin NEVER saved its state before. You might want to change this to Started instead.
            _state = _callback.getState(Stopped)

            if (_state == Started) {
                _callback.logStart("HelloWorldDistributedStartablePlugin:Started")
            }
        }
    }
}
```

```

        else {
            _callback.logStart("HelloWorldDistributedStartablePlugin:Stopped")
        }
    } else {
        _state = Started
        if (_callback.State != Started) {
            changeState(Started) // call our internal function that sets the database state if necessary
        }
        _callback.logStart("HelloWorldDistributedStartablePlugin")
    }

    // if the thread already existed for some reason, briefly stop it before starting it again
    if (_state == Started) {
        if (_thread != null) {
            _thread.stop()
        }
        _startedHowManyTimes++
    }

    // create your thread and start it
    var t = new Thread() {
        function run() {
            print("hello!")
        }
    }
    t.Daemon=true
}

override function stop( serverStopping : boolean ) : void {
    if (_thread != null) {
        _thread.stop()
        _thread = null
    }
    if (_callback != null) {
        if (serverStopping) {
            if (_state == Started) {
                _callback.logStop("HelloWorldDistributedStartablePlugin:Started")
            }
            else {
                _callback.logStop("HelloWorldDistributedStartablePlugin:Stopped")
            }
            _callback = null
        }
        else {
            if (_callback.State != Stopped) {
                changeState(Stopped) // call our internal function that sets the database state if necessary
            }
            _callback.logStop("HelloWorldDistributedStartablePlugin")
        }
    }
    _state = Stopped
}

// our internal function that sets the database state if necessary. if there are exceptions,
// this implementation tries 5 times. You might want a different behavior.
private function changeState(newState : StartablePluginState) {
    var tryCount = 0
    while (_callback.State != newState && tryCount < 5) {
        try {
            Transaction.runWithNewBundle(\ bundle -> { _callback.setState(bundle, newState)},
                User.util.UnrestrictedUser)
        }
        catch (e : java.lang.Exception) {
            tryCount++
            _callback.log(this.IntrinsicType.Name + " on attempt " + tryCount +
                " caught " + (typeof e).Name + ":" + e.Message)
        }
    }
}
}

```

## Important Notes for Java and Startable Plugins

### If You Define Your Startable Plugin In Java

In Gosu, your startable plugin must call the `execute` method on the callback handler object, as discussed in previous topics:

```
override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) : void {  
    _callback = cbh  
    _callback.execute( \ -> {  
        //...  
    }  
}
```

However, the Java language does not directly support blocks. If you implement your plugin in Java, you cannot use a Gosu block. However, instead you can use an anonymous class.

From Java, the method signatures for the `execute` methods (there are multiple variants) take a `GWRunnable` for the block argument. `GWRunnable` is a simple interface that contains a single method, called `run`. Instead of using a block, you can define an in-line anonymous Java class that implements the `run` method. This is analogous to the standard Java design pattern for creating an anonymous class to use the standard class `java.lang.Runnable`.

For example:

```
GWRunnable myBlock=new GWRunnable() {  
    public void run() {  
        System.out.println("I am startable plugin code running in an anonymous inner class");  
        // add more code here...  
    }  
};  
  
_callbackHandler.execute(myBlock);
```

For information about Gosu blocks and inner classes, see “Gosu Block Shortcut for Anonymous Classes Implementing an Interface” on page 207 in the *Gosu Reference Guide*.

### Where to Put Java Files for Your Startable Plugin

If you have Java files for your startable plugin, place your Java class and libraries files in the same places as with other plugin types.

The instructions are slightly different depending on whether you define the plugin interface implementation itself in Java or in Gosu:

- If your main startable plugin class is a Java class, see “Special Notes For Java Plugins” on page 170.
- If your main startable plugin class is a Gosu class, see “If Your Gosu Plugin Needs Java Classes and Library Files” on page 170.

## Persistence and Startable Plugins

Your startable plugin can manipulate Guidewire entity data. If your startable plugin needs to maintain state for itself, do one of the following:

- Create your own custom persistent entity types that track the internal state information of the startable plugin.
- Use the system parameter table for persistence.

## Starting and Stopping Startable Plugins

You can start and stop startable plugins from SOAP APIs. For more information, see “Starting or Stopping Startable Plugins” on page 146.



# Multi-threaded Inbound Integration

This topic describes a plugin interface that supports high performance multi-threaded processing of inbound requests. ClaimCenter includes default implementations for the most common usages: reading text file data and receiving JMS messages.

This topic includes:

- “Multi-threaded Inbound Integration Overview” on page 279
- “Inbound Integration Configuration XML File” on page 281
- “Inbound File Integration” on page 283
- “Inbound JMS Integration” on page 287
- “Custom Inbound Integrations” on page 289
- “Understanding the Polling Interval and Throttle Interval” on page 295

## Multi-threaded Inbound Integration Overview

There are sometimes situations that require high-performance data throughput for inbound integrations that require special threading or transaction features from the hosting J2EE/JEE application environment. It is difficult to interact with the application server’s transactional facilities and write correct, thread-safe, high-performing code. ClaimCenter includes tools that help you write such inbound integrations. You can focus on your own business logic rather than how to write thread-safe code that works safely in each application server.

### Inbound Integration Configuration XML File

There is a configuration file for inbound integrations called `inbound-integration-config.xml`. Edit this file in Studio to define configuration settings for every inbound integration that you want to use. Each inbound integration requires configuration parameters such as references to the registered plugin implementations. See “Inbound Integration Configuration XML File” on page 281.

## Inbound Integration Core Plugin Interfaces

ClaimCenter provides the multi-threaded inbound integration system in two different plugin interfaces for different use cases. Each defines a contract between ClaimCenter and inbound integration high-performance multi-threaded processing of input data:

- `InboundIntegrationMessageReply` - Inbound high-performance multi-threaded processing of replies to messages sent by a `MessageTransport` implementation. This is a subinterface of `MessageReply`.
- `InboundIntegrationStartablePlugin` - Inbound high-performance multi-threaded processing of input data as a startable plugin. A startable plugin is for all contexts other than handling replies to messages sent by a message transport (`MessageTransport`) implementation. This is a subinterface of `IStartablePlugin`.

You might not need to write your own implementation of these main plugin interfaces. To support common use cases, ClaimCenter includes plugin implementations that are supported for production servers. Both are provided in variants for startable plugin and message reply use.

Type of input data	Description	Related topic
File inbound integrations	Use this integration to read text data from local files. Poll a directory in the local file system for new files at a specified interval. Send new files to integration code and process incoming files line by line, or file by file. See “Inbound File Integration” on page 283. You provide your own code that processes one chunk of work, either one line, or one file, depending on how you configure it.	“Inbound File Integration” on page 283
JMS inbound integration	Use this integration to get objects from a JMS message queue. See “Inbound JMS Integration” on page 287. You provide your own code that processes the next message on the JMS message queue.	“Inbound JMS Integration” on page 287
Custom integration	If you process incoming data other than files or JMS messages, write your own version of the <code>InboundIntegrationMessageReply</code> or <code>InboundIntegrationStartablePlugin</code> plugin interface. In both cases, for custom integrations you must write multiple classes that implement helper interfaces such as <code>WorkAgent</code> .	“Custom Inbound Integrations” on page 289

In all cases, you must register and configure plugin implementations in the Studio plugin registry. See each topic for more information about which implementation classes to register. Additionally, for file and JMS integrations, you write handler classes. See “Inbound Integration Handlers for File and JMS Integrations” on page 280.

When registering a plugin implementation in the Plugins Registry, you must add a plugin parameter called `integrationservice`. That `integrationservice` parameter defines how ClaimCenter finds configuration information within the inbound integration configuration XML file.

To configure the inbound integration configuration XML file, see “Inbound Integration Configuration XML File” on page 281.

For general information about the Plugins Registry, see “Registering a Plugin Implementation Class” on page 166.

## Inbound Integration Handlers for File and JMS Integrations

Whether you use the built-in integrations or write your own, you must write code that handles one chunk of data.

To write a custom integration that supports data other than files or JMS messages, your code primarily implements the interface `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin`. For custom integrations, “Custom Inbound Integrations” on page 289 and skip the rest of this topic.

In contrast, if you use the built-in file or JMS inbound integrations, you register a built-in plugin implementation of `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin`, depending on whether you are using messaging. ClaimCenter includes plugin implementations of those interfaces that know how to process files or JMS data, depending on which one you choose.

For file or JMS inbound integrations, you must write a *handler class* that processes one chunk of data.

ClaimCenter defines a handler plugin interface called `InboundIntegrationHandlerPlugin` that contains one method called `process`. That method handles one chunk of data that ClaimCenter passes as a method of type `java.lang.Object`. Write a handler class that implements the `process` method. Downcast the `Object` to the necessary type:

- for file handling, downcast to `java.nio.file.Path` or a `String`, depending how you configure the integration to process by files or by line
- for JMS handling, downcast to is a JMS message of type `javax.jms.Message`

Next, register the plugin in the Studio plugin registry. See “Registering a Plugin Implementation Class” on page 166.

---

**IMPORTANT** When registering your plugin implementation, you must also add one plugin parameter called `integrationService`. That plugin parameter links your plugin implementation to one XML element within the `inbound-integration-config.xml` file. See “Inbound Integration Configuration XML File” on page 281.

---

If you are using either the file or JMS integrations as the startable plugin variant, your class must implement the `InboundIntegrationHandlerPlugin` interface.

If you are using either the file or JMS integrations as the message reply variant, your class must implement the `InboundIntegrationMessageReplyHandler` interface. This is a subinterface of `InboundIntegrationHandlerPlugin`. Implement the basic `process` method as well as all methods of the `MessageReply` plugin. For example, implement `MessageReply` methods `initTools`, `suspend`, `shutdown`, `resume`, and `setDestinationID`. Save the parameters to your `initTools` method into private variables. Use those private variables during your `process` method find and acknowledge the original `Message` object. For related information, see “Implementing a Message Reply Plugin” on page 346.

## Inbound Integration Configuration XML File

In Studio, there is a configuration file for inbound integrations. In the Project window, navigate to `configuration` → `config` → `integration` and click `inbound-integration-config.xml`.

The file contains the following types of data:

- “Thread Pool Configuration” on page 281
- “Configuring a List of Inbound Integrations” on page 282

### Thread Pool Configuration

The first section of the `inbound-integration-config.xml` file configures thread pools. Within the `<threadpools>` element, there is a list of `<threadpool>` elements, each of which look like the following

```
<threadpool name="gw_default" disabled="false">
  <gwthreadpooltype>FORKJOIN</gwthreadpooltype>
</threadpool>
```

The `name` attribute is a symbolic name that is used later in the XML file to refer uniquely to this thread pool.

The `disabled` attribute is a Boolean value that defines whether to disable that thread pool. If set to `true`, the thread pool is disabled.

Within the `threadpool` element, there are two supported element types for setting thread pool parameters:

- `<gwthreadpooltype>` – The thread pool type with the following supported values, which are case-sensitive:
  - Set to `FORKJOIN` for a self-managing default thread pool.

- Set to COMMONJ for running WebSphere or WebLogic and defining the JNDI name of the thread pool. If you set this value, you must also set the <workmanagerjndi> parameter.
- <workmanagerjndi> – JNDI name of the thread pool. This parameter is required if the thread pool type is set to COMMONJ.

**IMPORTANT** In the default configuration, there are thread pools predefined for default WebSphere or WebLogic thread pools. These are for development but not production use. For best performance, create your own custom thread pool with a unique name and tune that thread pool for the specific work you need, such as your JMS work. Then, update the <threadpool> settings to include the JNDI name for your new thread pool.

## Configuring a List of Inbound Integrations

The second section of the `inbound-integration-config.xml` file contains a list of inbound integrations that you want to use. For example, if you want 5 different JMS inbound integrations, each listening to its own JMS queue, add 5 elements to this section of the file.

Within the <integrations> element, there is a list of subelements of several pre-defined element names. For each inbound integration, define configuration parameters as subelements. For example, you must declare the name of the registered plugin implementations that corresponds to your handler code.

Some of the parameters are required, and some are optional. If you use either the built-in file processing or JMS integration, there are special parameters just for those integrations.

The following table lists the supported integration element names and where to find the complete reference for parameter names.

Type of integration	Configuration element	For more information
File inbound integration	<file-integration>	"Inbound File Integration" on page 283
JMS inbound integration	<jms-integration>	"Inbound JMS Integration" on page 287
Custom inbound integration. Directly implement the <code>InboundIntegrationStartablePlugin</code> or <code>InboundIntegrationMessageReply</code> interface.	<custom-integration>	"Custom Inbound Integrations" on page 289

Your configuration XML element must have the following two attributes:

- `name` – A unique identifying name for this inbound integration in the `inbound-integration-config.xml` file.

**IMPORTANT** The `name` attribute must match the value of the `integrationservice` plugin parameter in the Plugins registry for all registered plugin implementations of any inbound integration interfaces. In the Plugins registry, add the plugin parameter `integrationservice` and set to the value of this unique identifying name. See “Registering a Plugin Implementation Class” on page 166 and “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.

- `disabled` – determines whether to disable this inbound integration. Set to `true` to disable the inbound integration. Otherwise, set to `false`.

Refer to the file in Studio to see examples of these settings.

## Inbound File Integration

ClaimCenter includes built-in code that supports file-based input with high-performance multi-threaded processing. ClaimCenter provides this code in two variants, one for processing message replies, and one as a startable plugin.

You cannot modify the plugin implementation code in Studio, but you can use one or more instances of these integrations to work with your own file data.

The processing flow for file integration is as follows:

1. In response to some inbound event, your own integration code creates a new file in a specified incoming directory on the local file system.
2. The inbound file integration code polls the *incoming directory* at a specified interval and detects any new files since the last time checked.

---

**IMPORTANT** Any files that are in the incoming directory before the plugin initializes are never processed. For example, if you restart the server, files created after the server shuts down but before initialization are never processed. To process files that already existed, wait until the plugin is initialized, then move files outside the *incoming directory*, and then back to the incoming directory again.

The order of the processing of individual files in the directory is undefined. Never rely on the order being any particular order, such as alphabetic or creation date.

3. The inbound file integration code moves all found files to the *processing directory*, which stores inbound files in progress.
4. The inbound file integration code opens each new file using a specified character set. The default is UTF-8, but it is configurable.
5. The inbound file integration code reads one unit of work (one chunk of data) and dispatches it to your handler code. The `processingmode` parameter in the `inbound-integration-config.xml` file defines the type of data processed in one unit of work. If that parameter has the value `line`, ClaimCenter sends one line at a time to the handler as a `String` object. If that parameter has the value `file`, ClaimCenter sends the entire file to the handler as a `java.nio.file.Path` object.

If exceptions occur during processing, the plugin code moves the file to the *error directory*. The file name is changed to add a prefix that includes the time of the error, as expressed in milliseconds as returned from the Java time utilities. For example, if the file name `ABC.txt` has an error, it is renamed in the error directory with a name similar to `1864733246512.error.ABC.txt`.

6. After successfully reading and processing the complete file, the inbound file integration code moves the file to the *done directory*.
7. If there were any other files detected in this polling interval in step 2, the inbound file integration code repeats the process at step 4. Optionally, you can set the integration to operate on the most recent batch of files in parallel. For related information, see the `ordered` parameter, mentioned later in this section.
8. The inbound file integration waits until the next polling interval, and repeats this process at step 3.

### To create an inbound file integration

1. In the Project window, navigate to `configuration` → `config` → `integration` and click `inbound-integration-config.xml`.
2. Configure the thread pools. See “Thread Pool Configuration” on page 281.
3. In the list of integrations, create one `<integration>` element of type `<file-integration>`. Follow the pattern in the file to correctly set the XML/XSD namespace for the element. Set the `name` and `disabled` attributes as described in “Configuring a List of Inbound Integrations” on page 282.

**4. Set configuration parameter subelements as follows:**

File integration configuration parameters	Required	Description	Example value
pluginhandler	Required	The name in the Plugins registry for an implementation of the InboundIntegrationHandlerPlugin plugin interface. Note that this is the .gwp file name, not the implementation class name.	InboundFileIntegrationExample
processingmode	Required	To process one line at a time, set to <code>line</code> . In your handler class in the <code>process</code> method, you must downcast to <code>String</code> .  To process one file at a time, set to <code>file</code> . In your handler class in the <code>process</code> method, you must downcast to <code>java.nio.file.Path</code> .	line
threadpool	Required	The unique name of a thread pool as configured earlier in the file. See “Thread Pool Configuration” on page 281.	gw_default
osgiservice	Required	Always set to <code>true</code> . This is for internal use. This value is independent of whether you choose to register your handler class as an OSGi plugin.	true
transactional	Required	You must always set this to <code>false</code> .	false
stoponerror	Required	If true, ClaimCenter stops the integration if an error occurs. Otherwise, ClaimCenter just skips that item. Be sure to log any errors or notify an administrator.	true
incoming	Required	The full path of the configured incoming events directory	/tmp/inbound/incoming
processing	Required	The full path of the configured processing events directory	/tmp/inbound/processing
done	Required	The full path of the configured done events directory	/tmp/inbound/done
error	Required	The full path of the configured error events directory	/tmp/inbound/error
pollinginterval	Optional	The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the <code>ordered</code> parameter. See “Understanding the Polling Interval and Throttle Interval” on page 295. The default is 60 seconds.	15

File integration configuration parameters	Required	Description	Example value
throttleinterval	Optional	The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 295. The default is 60 seconds.	15
ordered	Optional	<p>By default, the inbound file integration handles multiple files at a time in parallel in multiple server threads. If you want files handled in a single thread sequentially, set this value to true. The default is false.</p> <p><b>WARNING:</b> The order of the processing of individual files in the directory is undefined. Never rely on the order being any particular order, such as alphabetic or creation date.</p> <p>Also see “Understanding the Polling Interval and Throttle Interval” on page 295.</p>	false

5. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 166 and “Using the Plugins Registry Editor” on page 113 in the Configuration Guide.
6. Studio prompts for a plugin name and plugin interface. For the plugin name, use a name that represents the purpose of this specific inbound integration. For the **Interface** field:
  - For a message reply plugin, type `InboundIntegrationMessageReply`.
  - For a startable plugin for non-messaging use, type `InboundIntegrationStartablePlugin`.
7. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
8. In the **Java class** field, type:
  - For a message reply plugin, type  
`com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationMessageReply`.
  - For a startable plugin for non-messaging use, type  
`com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationPlugin`.
9. Add a plugin parameter with the key **integrationservice**. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
10. Write your own *inbound integration handler plugin* implementation.
  - For a message reply plugin, your handler code must implement the plugin interface `gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin`.
  - For a startable plugin for non-messaging use, your handler code must implement the plugin interface `gw.plugin.integration.inbound.InboundIntegrationMessageReply`.

This interface has one method called `process`, which has a single argument of type `Object`. The method returns no value. The file integration calls that method to process one chunk of data. The data type depends on the value you set for the `processingmode` parameter:

  - If you set the `processingmode` parameter to `Line`, downcast the `Object` to `String` before using it.

- If you set the `processingmode` parameter to `file`, downcast the `Object` to `java.nio.file.Path` before using it.

If your code throws an exception, ClaimCenter moves the file to the error directory. Note the `stoponerror` parameter in the configuration file. If that value is `true`, ClaimCenter stops the integration if an error occurs. Otherwise, skips that item. Be sure to log any errors or notify an administrator.

In Studio, within the Plugins registry, register your handler plugin implementation class. Add a row in the Plugins registry editor for your plugin implementation class. See “Registering a Plugin Implementation Class” on page 166 and “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*. Set the interface name to the handler interface that you implemented (see step 10).

---

**IMPORTANT** Within the Plugins registry for your handler plugin implementation, the Name field must match the `pluginhandler` parameter you use in the `inbound-integration-config.xml` file for this integration.

---

11. Add a plugin parameter with the key `integrationservice`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
12. Start the server and test your new inbound integration.

## Example File Integration

The following is an example file integration configuration in the `inbound-integration-config.xml` file:

```
<ci:file-integration name="exampleFileIntegration" disabled="true">
    <pluginhandler>InboundFileIntegrationHandler</pluginhandler>
    <pollinginterval>1</pollinginterval>
    <throttleinterval>5</throttleinterval>
    <threadpool>gw_default</threadpool>
    <ordered>true</ordered>
    <stoponerror>false</stoponerror>
    <transactional>false</transactional>
    <osgiservice>true</osgiservice>
    <processingmode>line</processingmode>
    <incoming>/tmp/incoming</incoming>
    <processing>/tmp/processing</processing>
    <error>/tmp/error</error>
    <done>/tmp/done</done>
    <charset>UTF-8</charset>
    <createdirectories>true</createdirectories>
```

The following is a simple handler class called `mycompany.integration.SimpleFileIntegration`, which prints the lines in the file:

```
package mycompany.integration

uses gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin

class SimpleFileIntegration implements InboundIntegrationHandlerPlugin
{
    // this example assumes the inbound-integration-config.xml file
    // sets this to use "line" not "entire file" processing
    // See the <processingmode> element
    construct()
        print("***** SimpleFileIntegration startup ");
    }

    override function process(obj: Object) {
        // downcast as needed (to String or java.nio.file.Path, depending on value of <processingmode>
        var line = obj as String
        print("***** SimpleFileIntegration processing one line of file: ${line} (!)");
    }
}
```

For this example, in the plugin registry there are two plugin implementations in the Plugins registry:

- `InboundFileIntegration.gwp` – Registers an implementation of `InboundFileIntegrationPlugin` with the required Java class `com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationPlugin`. The `integrationservice` plugin parameter is set to `exampleFileIntegration`.
- `InboundFileIntegrationHandler.gwp` – Registers an implementation of `InboundIntegrationHandlerPlugin` with the Gosu class `mycompany.integration.SimpleFileIntegration`. The `integrationservice` plugin parameter is set to `exampleFileIntegration`.

When you start up the server, you will see the log line in the console for startup:

```
***** SimpleFileIntegration startup
```

After the server starts, add files to the `/tmp/incoming` directory. You will see additional lines for each processed line. As mentioned earlier, do not add files to the directory until after the plugin is initialized. Files that are in the incoming directory on startup are never processed.

## Inbound JMS Integration

ClaimCenter includes a built-in high-performance multi-threaded integration with inbound queues of Java Message Service (JMS) messages. The inbound JMS integration supports application servers that implement the `commonj.work.WorkManager` interface specification. These currently include the IBM WebSphere and Oracle Weblogic application servers. Define your own code that processes an individual message, and the inbound JMS framework handles message dispatch and thread management.

ClaimCenter can use JMS implementations on the application server but ClaimCenter does not include its own JMS implementation. For additional advice on setting up or configuring an inbound JMS integration with ClaimCenter, contact Guidewire Customer Support.

### To create an inbound JMS integration

1. In the Project window, navigate to `configuration` → `config` → `integration` and click `inbound-integration-config.xml`
2. Configure the thread pools. See “Thread Pool Configuration” on page 281.
3. In the list of integrations, create one `<integration>` element of type `<jms-integration>`. Follow the pattern in the file to correctly set the XML/XSD namespace for the element. Set the `name` and `disabled` attributes as described in “Configuring a List of Inbound Integrations” on page 282.
4. Set configuration parameter subelements as follows:

Plugin parameters in Plugins editor, description	Required	Description	Example value
<code>pluginhandler</code>	Required	The name in the Plugins registry for an implementation of the <code>InboundIntegrationHandlerPlugin</code> plugin interface. Note that this is the <code>.gwp</code> file name, not the implementation class name.	<code>InboundJMSIntegrationExample</code>
<code>transactional</code>	Required	You must always set this to <code>true</code> .	<code>true</code>
<code>threadpool</code>	Required	The unique name of a thread pool as configured earlier in the file. See “Thread Pool Configuration” on page 281.	<code>gw_default</code>

Plugin parameters in Plugins editor, description	Required	Description	Example value
ordered	Optional	For typical use, set to true to maintain the processing of inbound JMS messages in order.  The default value is false, which means the JMS integration dispatches the inbound messages in parallel with no guarantees of strict ordering.  The behavior of the ordered flag with the polling and throttle interval works the same in the JMS integration as in the file integration. See “Understanding the Polling Interval and Throttle Interval” on page 295.	true
batchlimit	Required	The maximum number of messages to receive in a poll interval	2
connectionfactoryjndi	Required	The application server configured JNDI connection factory	jms/gw/queueCF
destinationjndi	Required	The application server configured JNDI destination	jms/gw/queue
user	Optional	User name for authenticating on the JMS queue.	jsmith
password	Optional	Password for authenticating on the JMS queue.	pw123
osgiservice	Required	Always set to true. This is for internal use. This value is independent of whether you choose to register your handler class as an OSGi plugin.	true
stoponerror	Required	If true, ClaimCenter stops the integration if an error occurs. Otherwise, ClaimCenter just skips that message. Be sure to log any errors or notify an administrator.	true
messagereceivetimeout	Optional	The maximum time in seconds to wait for an individual JMS message. The default is 15.	15
pollinginterval	Optional	The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 295. The default is 60 seconds.	60
throttleinterval	Optional	The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 295. The default is 60 seconds.	60

If you throw an exception in your code, the transaction of the message processing is rolled back. The original message is back in the queue.

5. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 166 and “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.
6. Studio prompts for a plugin name and plugin interface. For the name, use a name that represents the purpose of this specific inbound integration. For the interface field, type `InboundIntegrationStartablePlugin`.

7. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
8. In the **Java class** field, type:
  - For a message reply plugin, type  
`com.guidewire.pl.integration.inbound.jms.DefaultJMSInboundIntegrationMessageReply`.
  - For a startable plugin for non-messaging use, type  
`com.guidewire.pl.integration.inbound.jms.DefaultJMSInboundIntegrationPlugin`.
9. Add a plugin parameter with the key **integrationService**. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
10. Write your own *inbound integration handler plugin* implementation. Your handler code must implement a plugin interface
  - For a message reply plugin, type  
`gw.plugin.integration.inbound.InboundIntegrationMessageReplyHandler`.
  - For a startable plugin for non-messaging use, implement the interface  
`gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin`.This interface has one method called `process`, which has a single argument of type `Object`. The method returns no value. The JMS integration calls that method to process one message. Downcast this `Object` to `javax.jms.Message` before using it.  
If you throw an exception in your code, the behavior depends on the configuration parameter `stoponerror`. If that parameter has the value `true`, processing on that queue stops. If it has the value `false`, that message is skipped. Be sure to catch any errors in your processing code and log any issues so that an administrator can follow up later.
11. In Studio, within the Plugins registry, register your handler plugin implementation class. Add a row in the Plugins registry editor for your Gosu, Java, or OSGi plugin implementation class. See “[Registering a Plugin Implementation Class](#)” on page 166 and “[Using the Plugins Registry Editor](#)” on page 113 in the *Configuration Guide*. Set the interface name to the handler interface that you implemented (see step 10).

**IMPORTANT** Within the Plugins registry for your handler plugin implementation, the Name field must match the `pluginhandler` parameter you use in the `inbound-integration-config.xml` file for this integration.

12. Add a plugin parameter with the key **integrationService**. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
13. Start the server and test your new inbound integration.

## Custom Inbound Integrations

ClaimCenter includes built-in inbound integrations of file-based input and JMS messages. If these built-in integrations do not serve your needs, write your own integration based on the plugin interfaces in the `gw.plugin.integration.inbound` package:

- `InboundIntegrationMessageReply` – message reply plugin
- `InboundIntegrationStartablePlugin` – startable plugin for other non-messaging contexts

**IMPORTANT** You must write your plugin implementation for this plugin interface in Java, not Gosu.

The `InboundIntegrationStartablePlugin` plugin interface extends several other interfaces:

- `InitializablePlugin` – This interface requires on method that passes plugin parameters. The plugin parameters are passed to the `setup` method in the `WorkAgent` interface, which your plugin must implement.

- **WorkAgent** – This *work agent* interface defines the core behavior of the inbound integration framework. This is the most complex part of writing your own custom inbound integration. See “Writing a Work Agent Implementation” on page 290.
- **IStartablePlugin** – The startable plugin interface defines methods such as `start`, `stop`, and `getState`. See “What are Startable Plugins?” on page 271. Be aware that the main `WorkAgent` interface defines `start` and `stop` methods with no arguments. `IStartablePlugin` adds additional method signatures of those methods that take arguments.

The `InboundIntegrationMessageReply` plugin interface extends several other interfaces:

- **MessageReply** – The interface for classes that handle replies from messages sent by a `MessageTransport` implementation. See “Implementing Messaging Plugins” on page 343 for the relationship between the interfaces `MessageTransport`, `MessageRequest`, and `MessageReply`.
- **WorkAgent** – This *work agent* interface defines the core behavior of the inbound integration framework. This is the most complex part of writing your own custom inbound integration. See “Writing a Work Agent Implementation” on page 290.

There are no other methods on `InboundIntegrationStartablePlugin` not defined in one of those other interfaces.

After you write your own custom implementation of `InboundIntegrationStartablePlugin`, use the ClaimCenter Studio Plugins Registry to register your plugin implementation with ClaimCenter. See “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.

## Writing a Work Agent Implementation

The `gw.api.integration.inbound.WorkAgent` interface defines methods that coordinate and process work. You must write your own class that implements this interface. In addition to the primary functions of each method, you may also want to perform some logging to help debug your code.

To write a complete work agent implementation, you must write multiple related classes that work together. Refer to the following table for a summary of each class

Class that you write	Implements this interface	Description
A work agent	<code>WorkAgent</code>	Your plugin is the top level class that coordinates work for this service. Instantiates your class that implements the interface <code>Factory</code> .
<b>Finding and preparing work during each polling interval</b>		
A factory	<code>Factory</code>	A factory is a class that for each polling interval. Instantiates your class that implements the interface <code>WorkSetProcessor</code> .
A work set processor	<code>WorkSetProcessor</code> If your plugin supports the optional feature of being <i>transactional</i> , implement the subinterface <code>TransactionalWorkSetProcessor</code>	An object that knows how to acquire and divide resources. Instantiates your class that implements the interface <code>Inbound</code> .  The main method for processing one unit of work within a <code>WorkData</code> object is the <code>process</code> method within this object.
Inbound	<code>Inbound</code>	A class that knows how to find work in its <code>findWork</code> method and return new work data sets. Instantiates your class that implements the interface <code>WorkDataSet</code> .

Class that you write	Implements this interface	Description
<b>Representing the work itself</b>		
A work data set	WorkDataSet	Represents the set of all data found in this polling interval. Encapsulates a set of work data (WorkData) objects and any necessary context information to operate on the data. Instantiates your class that implements the interface WorkData.
Work data	WorkData	Represents one unit of work

## Setup and Teardown the Work Agent

In your `WorkAgent` implementation class, write a `setup` method that initializes your resources. ClaimCenter calls the `setup` method before the `start` method. The method signature is:

```
public void setup(Map<String, Object> properties);
```

The `java.util.Map` object that is the method argument is the set of plugin parameters from the Studio Plugins Editor for your plugin implementation.

Implement the `teardown` method to release resources acquired in the `start` method. ClaimCenter calls the `teardown` method before the `stop` method.

---

**IMPORTANT** If you use the startable plugin variant of in the inbound integration plugin (`InboundIntegrationStartablePlugin`), be aware that the main `WorkAgent` interface defines `start` and `stop` methods with no arguments. The startable plugin variant also implements `IStartablePlugin`, which means you must add additional method signatures of those methods that take arguments. See related topic “What are Startable Plugins?” on page 271.

---

## Start and Stop the Plugin

In your `WorkAgent` implementation class, implement the plugin method `start` to start the work listener and perform any necessary initialization that must happen each time you start the work agent.

Implement the plugin method `stop` and perform any necessary logic to stop your work agent.

Compare and contrast the `start` and `stop` methods with the different methods `setup` and `teardown`. See “Setup and Teardown the Work Agent” on page 291.

## Declare Whether Your Work Agent is Transactional

ClaimCenter calls the plugin method `transactional` to determine whether your agent is *transactional*. A transactional work agent creates work items with a slightly different interface. There are additional method that you must implement to begin work, to commit work, and to roll back transactional changes to partially finished work. To specify your work agent is transactional, return `true` from the `transactional` method. Otherwise, return `false`.

If your work agent is transactional, your implementation of `Factory.createWorkUnit()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of (extends from) `WorkSetProcessor`. See “Get a Factory for the Work Agent” on page 291.

## Get a Factory for the Work Agent

In your `WorkAgent` implementation class, implement the `factory` method. This method must return a `Factory` object, which represents an object that creates work data sets.

The Factory interface has only one method, which is called `createWorkProcessor`. It takes no arguments and returns an instance of your own custom class that implements the `WorkSetProcessor` interface.

If your agent is transactional, your implementation of `Factory.createWorkProcessor()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of `WorkSetProcessor`. See “Declare Whether Your Work Agent is Transactional” on page 291. Both interfaces are in the `gw.api.integration.inbound` package.

## Writing a Work Set Processor

Most of your actual work happens in code called a work set processor, which is a class that you create that implements the `WorkSetProcessor` interface or its subinterface `TransactionalWorkSetProcessor`. Both interfaces are in the `gw.api.integration.inbound` package. See “Declare Whether Your Work Agent is Transactional” on page 291

The basic `WorkSetProcessor` interface defines two methods

- `getInbound` – Gets an object that knows how to acquire and divide resources to create work items. This method returns an object of type `Inbound`, which is an interface with only one method, called `findWork`. Define your own class that implements the `Inbound` interface. The `findWork` method must get the *work data set*, which represent multiple work items. If your plugin supports unordered multi-threaded work, each work item represents work that can be done by its own thread. For example, for the inbound file integration, a work data set is a list of newly-added files. Each file is a separate work item. The `findWork` method returns the data set encapsulated in a `WorkDataSet` object. The polling process of the inbound integration framework calls the `findWork` method to do the main work of getting new data to process. See “Error Handling” on page 292. From Gosu, this method appears as a getter for the `Inbound` property rather than as a method.
- `process` – Processes one work data item within a work data set. The method takes two arguments of type `WorkContext` and `WorkData`. The `WorkData` is one work item in the work data set. The `WorkContext` that you can optionally use to declare a resource or other context necessary to process the data item. Your own implementation of the work data set (`WorkDataSet`) is responsible for populating this context information if you need it. For example, if your inbound integration is listening to a message queue, you might store the connection or queue information in the `WorkContext` object. Your `WorkSetProcessor` can then access this connection information in the `process` method when processing one message on the queue. See “Error Handling” on page 292.

Only if your agent is transactional, your implementation of `Factory.createWorkUnit()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of `WorkSetProcessor`. See “Declare Whether Your Work Agent is Transactional” on page 291.

The `TransactionalWorkSetProcessor` interface defines several additional methods:

- `begin` – Begin any necessary transactional context. You are responsible for management of any transactions.
- `commit` – Commit any changes. You are responsible for management of any transactions.
- `rollback` – Rollback any changes. You are responsible for management of any transactions.

All three methods take a single argument of type `TxContext`. The `TxContext` interface is a subinterface of `WorkContext`. Use it to represent customer-specific work context information that also contains transaction-specific information. Create your own implementation of this class in your `getContext` method of your `WorkDataSet`. See “Creating a Work Data Set” on page 293.

## Error Handling

Any exceptions in `WorkDataSet.findWork()` causes ClaimCenter to immediately stop processing until the plugin is restarted or the server is restarted.

Any exceptions in the `WorkSetProcessor` in the `process` method are logged and the item is skipped.

**WARNING** It is important to catch any exceptions in the `process` method to ensure that you correctly handle error conditions. For example, you may need to notify administrators or place the work item in a special location for special handling.

## Creating a Work Data Set

You must implement your own class that encapsulates knowledge about a work data set, which represents the set of all data found in this polling interval. The work data set is created by your own implementation of the `Inbound.findWork()` method. For example, an inbound file integration creates a work data set representing a list of all new files in an incoming directory. See “Get a Factory for the Work Agent” on page 291.

Create a class that implements the `gw.api.integration.inbound.WorkDataSet` interface. Your class must implement the following methods:

- `getData` – Get the next work item and move any iterator that you maintain forward one item so that the next call returns the next item after this one. Return a `WorkData` object if there are more items to process. Return `null` to indicate no more items. For example, an inbound file integration might return the next item in a list of files. The `WorkData` interface is a marker interface, so it has no methods. Write your own implementation of a class that implements this interface. Add any object variables necessary to store information to represent one work item. It is the `WorkDataSet.getData()` method that is responsible for instantiating the appropriate class that you write and populating any appropriate data fields. For example, for an inbound file integration, one `WorkData` item might represent one new file to process.

**Note:** From Gosu, the `getData` method appears as a getter for the `Data` property, not a method.

- `hasNext` – Return `true` if there are any unprocessed items, otherwise return `false`. In other words, if this same object’s `getData` method would return a non-null value if called immediately, return `true`.
- `getContext` – Your implementation of a work data set can optionally declare a resource or other context necessary to process the data item. You are responsible for populating this context information if you need it. For example, if your inbound integration listens to a message queue, store the connection or queue information in an instance of a class that you write that implements `gw.api.integration.inbound.WorkContext`. In your code that processes each item (your `WorkSetProcessor` implementation), you can access this connection information from the `WorkSetProcessor` object’s `process` method when processing each new message. If your plugin supports transactional work items (the `transactional` plugin parameter), your class must implement a subinterface called `TxContext`, which requires two additional methods:
  - `isRollback` – Returns a boolean value that indicates the transaction will be rolled back.
  - `setRollbackOnly` – Set your own boolean value to indicate that a rollback will occur.
- `close` – Close and release any resources acquired by your work data set.

## Getting Parameters

Like all other plugin types, your plugin implementation can get parameter values. See “Plugin Parameters” on page 167. The `Map` argument to the `setup` method includes all parameters that you set in the `inbound-integration-config.xml` file for that integration. Save the map or the values of important parameters in private variables in your plugin implementation.

The map argument to the `setup` method also includes any arbitrary parameters that you set in the `<parameters>` configuration element. See the parameter called `parameters` in “Installing a New Custom Inbound Integration” on page 294.

## Installing a New Custom Inbound Integration

### To create and register a new custom inbound integration

1. Design and write all required implementation Java classes as described in “Installing a New Custom Inbound Integration” on page 294.

**IMPORTANT** You must write your plugin implementation for this plugin interface in Java, not Gosu.

2. In the Project window, navigate to configuration → config → integration and click inbound-integration-config.xml.
3. Configure the thread pools. See “Thread Pool Configuration” on page 281.
4. In the list of integrations, create one <integration> element of type <custom-integration>. Follow the pattern in the file to correctly set the XML/XSD namespace for the element. Set the name and disabled attributes as described in “Configuring a List of Inbound Integrations” on page 282.
5. Set configuration parameter subelements as follows:

Plugin parameters in Plugins editor, description	Required	Description	Example value
workagentimpl	Required	Set this parameter to the fully-qualified name of your InboundIntegrationStartablePlugin or InboundIntegrationMessageReply implementation.  <b>IMPORTANT:</b> You must write your plugin implementation for this plugin interface in Java, not Gosu.	mycompany.integ.MyInboundPlugin
pluginhandler	n/a	<i>This parameter is unused in custom inbound integrations. This parameter is used only for file and JMS integrations.</i>	n/a
transactional	Optional	Always set this boolean value to the same value returned by your plugin implementation's transactional method. See “Declare Whether Your Work Agent is Transactional” on page 291.  If not set, defaults to false.	true
threadpool	Optional	The unique name of a thread pool as configured earlier in the file. See “Thread Pool Configuration” on page 281.	gw_default
stoponerror	Required	If true, ClaimCenter stops the integration if an error occurs. Otherwise, ClaimCenter just skips that item. Be sure to log any errors or notify an administrator.	true
ordered	Optional	Set to true to process in a single thread.  Set to false to process items in multiple threads, as managed by the thread pool.  The behavior of the ordered flag interacts with the behavior of the polling interval and throttle interval. See “Understanding the Polling Interval and Throttle Interval” on page 295.	true

Plugin parameters in Plugins editor, description	Required	Description	Example value
pollinginterval	Optional	The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 295. The default is 60 seconds.	60
throttleinterval	Optional	The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 295. The default is 60 seconds.	60
parameters	Optional	Arbitrary parameters that you can define, for example to store server names and port numbers. Define subelements that alternate between key and value elements, contain key/value pairs. For example:  The plugin interface has a <code>setup</code> method. These parameters are in the <code>java.util.Map</code> that ClaimCenter passes to that initialization method.	<parameters> <key>key1</key> <value>myvalue</value> </parameters>

6. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 166 and “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.
7. Studio prompts for a plugin name and plugin interface. For the name, use a name that represents the purpose of this specific inbound integration.
8. For the **Interface** field, type the plugin interface you implemented, either `InboundIntegrationStartablePlugin` or `InboundIntegrationMessageReply`.
9. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
10. In the **Java class** field, type the fully-qualified name of your plugin implementation.
11. Add a plugin parameter with the key `integrationservice`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
12. Start the server and test your new inbound integration. Add logging code as appropriate to confirm the integration.

## Understanding the Polling Interval and Throttle Interval

There are two different configuration parameters for coordinating when the work begins: `pollinginterval` and `throttleinterval`.

The primary mechanism is the *polling interval* (`pollinginterval`). Additionally, there is a throttle interval (`throttleinterval`) which you can use to reduce the impact on the server load or external resources.

1. At the beginning of the polling interval, the integration begins polling for new work and creating new work items (but not yet beginning the items).
2. ClaimCenter begins working on the work items.
  - If the work is ordered (the `ordered` parameter is `true`), the work happens in the same thread.

- If the work is unordered (the `ordered` parameter is `false`), the work happens in separate additional threads with no guarantee of strict ordering.
3. After the current work is complete, the system determines how much time has transpired and compares with the two interval parameters. The behavior is slightly different for ordered and unordered work.
- If the work is ordered (the `ordered` parameter is `true`), the following table describes the behavior.

If the total time since last polling is...	Behavior
Less than the <code>pollinginterval</code>	The server waits until the remaining part of the polling interval and then waits the complete <code>throttleinterval</code> time
Greater than the <code>pollinginterval</code> but less than the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code>	The server waits until the end of the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code> times
Greater than the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code>	The server immediately polls for new work

If the work is unordered, the same time check occurs with some differences:

- the work proceeds in parallel
- the time check happens immediately after new work is created but does not wait for the work to be done.

---

part IV

# Messaging



# Messaging and Events

You can send *messages* to external systems after something changes in ClaimCenter, such as a changed exposure or an added check. The changes trigger *events*, which trigger your code that sends messages to external systems. For example, if you create a new check in ClaimCenter, your messaging code notifies a corporate financials system or notifies a check printing service.

ClaimCenter defines a large number of events of potential interest to external systems. Write rules to generate messages in response to events of interest. ClaimCenter queues these messages and then dispatches them to the receiving systems.

This topic explains how ClaimCenter generates messages in response to events and how to connect external systems to receive those messages.

This topic discusses *plugins*, which are software modules that ClaimCenter calls to perform an action or calculate a result. For information on plugins, see “Plugin Overview” on page 163. For the complete list of all ClaimCenter plugins, see “Summary of All ClaimCenter Plugins” on page 181.

For financials-specific events and messaging, read this topic for messaging concepts and implementation details. Next, refer separately to “Financials Integration” on page 363 for custom financials events and APIs.

Register new messaging plugins in Studio. As you register plugins in Studio, Studio prompts you for a plugin interface name (in a picker) and, in some cases, for a plugin name. Use that plugin *name* as you configure the messaging destination in the Messaging editor in Studio.

This topic includes:

- “Messaging Overview” on page 300
- “Message Destination Overview” on page 311
- “Filtering Events” on page 319
- “List of Messaging Events in ClaimCenter” on page 320
- “Generating New Messages in Event Fired Rules” on page 329
- “Message Ordering and Multi-Threaded Sending” on page 334
- “Late Binding Data in Your Payload” on page 339
- “Reporting Acknowledgements and Errors” on page 340

- “Tracking a Specific Entity With a Message” on page 343
- “Implementing Messaging Plugins” on page 343
- “Resynchronizing Messages for a Primary Object” on page 350
- “Message Payload Mapping Utility for Java Plugins” on page 353
- “Monitoring Messages and Handling Errors” on page 354
- “Messaging Tools Web Service” on page 356
- “Batch Mode Integration” on page 358
- “Included Messaging Transports” on page 358

## Messaging Overview

To understand this topic, it might help to get a high-level overview of terminology and an overview of events in ClaimCenter. The following subtopics summarizes important messaging concepts.

### Event

An event is an abstract notification of a change in ClaimCenter that might be interesting to an external system. An event most often represents a user action to application data, or an API that changed application data. For some (but not all) entities in the data model configuration files, if a user action or API adds, changes, or removes an entity instance, the system triggers an event. Each event has a name, which is a `String` value.

For example, in ClaimCenter, adding a new check on a claim triggers an event. The event name is "`ClaimAdded`".

One user action or API call might trigger multiple events for different objects in one database transaction. In some cases, the object might have multiple events occur in one database transaction.

Triggering an event triggers one call to the Event Fired rule set for each external system that is interested in that event.

Event Fired rules for an event name run only if you tell ClaimCenter that at least one external system is interested in that event. To request that ClaimCenter run Event Fired rules for that event and that destination ID, see “Message Destination Overview” on page 311.

Additionally, only entity types defined with the `<events>` tag by default generate added, changed, or deleted events. For more information, see “`<events>`” on page 196 in the *Configuration Guide*.

### Message

A message is information to send to an external system in response to an event. ClaimCenter can ignore the event or send one or more messages to each external system that cares about that event. In addition to an ID and some status information, each message has a *message payload*. The payload is the main data content of the message. The payload is `String` in the `Message.Payload` property.

Your code creates messages in the Event Fired rule set. For more information, see “Generating New Messages in Event Fired Rules” on page 329. Because message creation impacts user response times, avoid unnecessarily large or complex messages if possible.

### Message History

After a message is sent, the application converts a message (`Message`) object to a message history (`MessageHistory`) object. The message history object has the same properties as a message object.

You can use the message history database table to understand the messaging history to your external systems. This can be useful to help understand problems.

Additionally, message history objects are important when detecting duplicate messages. To report a duplicate message, you must find the message history object. Call the `reportDuplicate` method on the message history object, which represents the original message. See “[Reporting Acknowledgements and Errors](#)” on page 340.

## Messaging Destinations

A messaging destination is an external system to which to send messages. Generally speaking, a messaging destination represents one external system. Register one destination for each external system, even if that external system is used for multiple types of data.

Register your messaging destinations in Studio. In the **Messaging** editor, specify which classes implement the messaging plugin interfaces for that messaging destination. The most important message plugin interface is `MessageTransport`. The `MessageTransport` plugin interface is responsible for actually dispatching the message (most importantly, its payload) to your external system.

Each destination registers a *list of event names* for which it wants notifications. Each destination may listen for different events compared to other destinations. The Event Fired rule set runs once for every combination of an event name and a destination interested in that event. To request that ClaimCenter run Event Fired rules for that event and that destination ID, see “[Message Destination Overview](#)” on page 311.

If more than one messaging destination requests an event name, ClaimCenter runs the Event Fired rule set once for every destination that requested it. To determine which destination this event trigger is for, your Event Fired rules must check the `DestinationID` property of the message context object at run time.

In the **Messaging** editor there are other important settings for each destination, including:

- **Alternative Primary Entity** – See “[Primary Entity and Primary Object](#)” on page 301
- **Message Without Primary** – See “[How Destination Settings Affects Ordering](#)” on page 338

## Root Object

A root object for an event is the entity instance most associated with the event. This might be a small subobject or a more prominent high-level object.

A root object for a message is the entity instance most associated with the message. This might be a small subobject or a more prominent high-level object.

By default, the message’s root object is the same as the root object for the event that created the message in your Event Fired rules. This default makes sense in most cases. You can override this default for a message if desired. See “[Setting a Message Root Object or Primary Object](#)” on page 332.

## Primary Entity and Primary Object

A *primary entity* represents a type of high-level object that a Guidewire application uses to group and sort related messages. A *primary object* is a specific instance of a primary entity. Each Guidewire application specifies a default *primary entity* type for the application, or no default primary entity type.

Additionally, messaging destinations can override the primary entity type, and that setting applies just to that messaging destination. Only specific entity types are supported for each Guidewire application.

**IMPORTANT** Determining which primary object, if any, applies for a messaging destination is critical to understanding how ClaimCenter orders messages. See “[Safe Ordering](#)” on page 302 and “[Message Ordering and Multi-Threaded Sending](#)” on page 334.

In ClaimCenter:

- The default primary entity is `Claim`. Most objects are subobjects of a claim. An `Exposure` object is part of a claim. A `ClaimContact` object is part of a claim.

- A messaging destination can specify the `Contact` entity as an alternative primary object, in which case that setting applies just to that messaging destination.
- No other entity types can be alternative primary entities for a messaging destination.

Some objects are not associated with any primary object. For example, `Catastrophe` and `User` objects are not associated with a single claim. Messages associated with such objects are called *non-safe-ordered messages*. See “Safe Ordering” on page 302 and “Message Ordering and Multi-Threaded Sending” on page 334.

Do not confuse the *root object* for a message with the *primary object* associated with a message. The root object is the generally the object that triggered the event. The primary object is the highest-level object related to the root object.

To configure how a message is associated with a primary entity, there are some automatic behaviors when you set the message root object. You can manually set the message root object and the primary entity properties for a message. See “Setting a Message Root Object or Primary Object” on page 332.

## Acknowledgement (ACK and NAK)

An acknowledgement is a formal response from an external system back to a ClaimCenter that declares how successfully the system processed the message:

- A *positive acknowledgement* (ACK) means that the external system processed the message successfully.
- A *negative acknowledgement* (NAK) means the external system failed to handle the message due to errors.

It is very important for integration programmers to understand that ClaimCenter distinguishes between the following errors:

- An error that throws a Gosu or Java exception during initial sending in the `MessageTransport` method called `send`. Errors during this phase typically indicate network errors or other automatically retryable errors. If the `send` method throws an exception, ClaimCenter automatically retries sending the message multiple times.
- An error reported from the external system, which are also called a NAK.

For more information about error handling, see “Error Handling in Messaging Plugins” on page 348.

## Safe Ordering

Safe ordering is a messaging feature that causes related messages to be received in the same order they were sent. This is called *safe order*. Messages are grouped by their related *primary object* for each messaging destination. For important definitions relevant to this topic, see “Primary Entity and Primary Object” on page 301.

Messages send in creation order with other messages associated with that same primary object for that destination. Any messages associated with a primary object for that destination are called *safe-ordered messages*. The application waits for an acknowledgement before processing the next safe-ordered message for that same primary object for that destination. In other words, delays or errors for that destination block further sending of messages for that same destination for that same primary object.

For example, by default ClaimCenter sends any messages associated with a claim to each destination grouped by claim and sent in creation order. These are *safe-ordered messages*. In Guidewire terminology, the *primary object* for ClaimCenter is a claim (a `Claim` object). If a messaging destination sets `Contact` as its alternative *primary entity*, messages associated with a contact send grouped by contact for each messaging destination and sent in creation order.

There are ClaimCenter objects that are not associated with a primary entity. For example, `Catastrophe`. All messages that are not associated with a claim (the primary object) are called *non-safe-ordered messages*.

Not all messages are associated with a primary object. The logic for how and when to send these *non-safe-ordered messages* is different from the logic for safe-ordered messages. Additionally, the behavior varies based on the messaging destination configuration setting called **Strict Mode**. For details, see “Message Ordering and Multi-Threaded Sending” on page 334.

## Transport Neutrality

ClaimCenter does not assume any specific type of transport or message formatting.

Destinations deliver the message any way they want, including but not limited to the following:

- **Submit the message by using remote API calls** – Use a web service (SOAP) interface or a Java-specific interface to send a message to an external system.
- **Submit the message to a guaranteed-delivery messaging system** – For instance, a message queue.
- **Save to special files in the file system** – For large-scale batch handling, you could send a message by implementing writing data to local text files that are read by nightly batch processes. If you do this, remember to make your plugins thread-safe when writing to the files.
- **Send e-mails** – The destination might send e-mails, which might not guarantee delivery or order, depending on the type of mail system. This approach is acceptable for simple administrative notifications but is inappropriate for systems that rely on guaranteed delivery and message order, which includes most real-world installations.

## Messaging Flow Overview

The high-level steps of event and message generation and processing are as follows. As an example for this list, let us suppose you want to detect new checks so you can send the information to a check printing system.

1. **Application startup** – At application startup, ClaimCenter checks its configuration information and constructs messaging destinations. Each destination registers for specific events for which it wants notifications.

In the earlier example, the destination registers for the `CheckAdded` and `CheckChanged` events.

2. **Users and APIs trigger messaging events** – Events trigger after data changes. For example, a change to data in the user interface, or an outside system calls a web service that changes data. The messaging event represents the changes to application data as the entity commits to the database.

3. **Event Fired rules create messages** – ClaimCenter runs the Event Fired rule set for each event name that triggers. ClaimCenter runs the rule set multiple times for an event name if multiple destinations listen for it. Your rules can choose to generate new messages. Messages have a text-based *message payload*.

In ClaimCenter, for example you might write rules that check if the event name is `CheckAdded` or `CheckChanged`, and if certain other conditions occur. If so, the rules might generate a new message with an XML payload that describes the added check. The rules might also link entities with the message for later use.

4. **ClaimCenter sends message to a destination** – Messages are put in a queue and handed one-by-one to the *messaging destination*.

In ClaimCenter, a check printing destination might take an XML payload and submit the message to an external message queue to notify the external system of the check.

5. **ClaimCenter waits for an acknowledgement** – The external system replies with an acknowledgement to the destination after it processes the message, and the destination's messaging plugins process this information. If the message successfully sent, the messaging plugins submit an ACK and ClaimCenter sends the next message. For details of the messaging ordering system, see “Message Ordering and Multi-Threaded Sending” on page 334. The code that submits the acknowledgement might also make changes to application data, such as updating a messaging-specific property on a persisted object or advancing a workflow. If you change data, see the warnings in the section “Messaging Flow Details” on page 305. Remember that you can also submit an error (a negative acknowledgement, a NAK) instead of a positive acknowledgement.

Most messaging integration code that you write for a typical deployment is writing your Event Fired rules that create message payloads and writing `MessageTransport` plugin implementations to send messages. There are two other optional types of messaging plugins and discussed later.

Once you write messaging plugin implementation classes, you register your new messaging plugin classes in Studio. Register your messaging plugin implementations first in the `Plugins` editor and then in the `Messaging` editor.

For plugin interfaces that can have multiple implementations, which includes all messaging plugin interfaces, Studio asks you to *name* your plugin implementation when registering your plugin class. Use that *plugin implementation class name* when you configure the messaging destination in the **Messaging** editor.

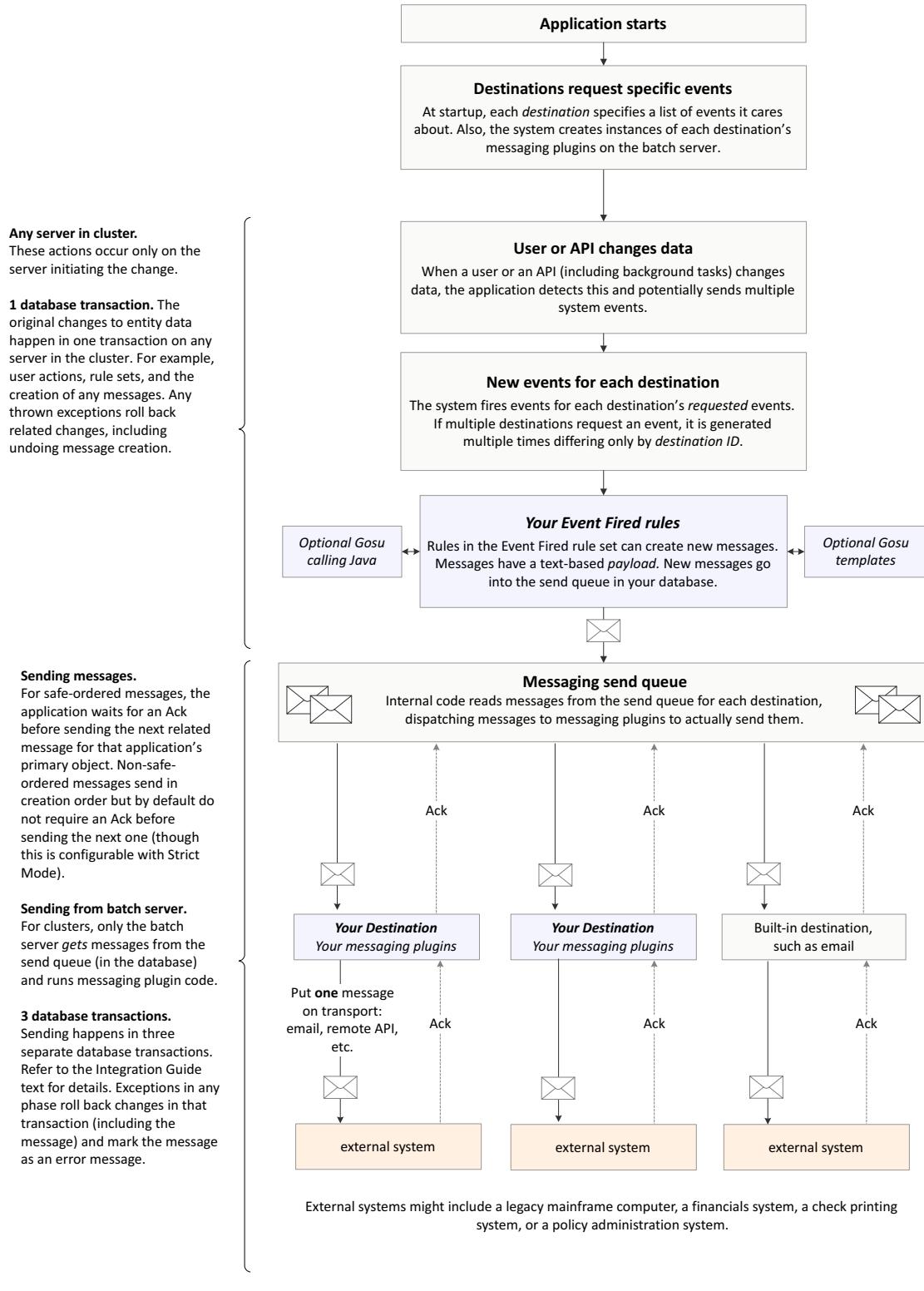
**See Also**

- “Using the Messaging Editor” on page 137 in the *Configuration Guide*.
- “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*.
- “Plugin Overview” on page 163.
- “Message Destination Overview” on page 311.

## Messaging Flow Details

The following diagram illustrates the chronological flow of events and messaging, starting from the top of the diagram. Refer to the section following the diagram for a detailed explanation of each step.

## Messaging Overview



The following list describes a detailed chronological flow of event-related actions:

- 1. Destination initialization at system startup** – After the ClaimCenter application server starts, the application initializes all destinations. ClaimCenter saves a list of events for which each destination requested notifications. Because this happens at system startup, if you change the list of events or destination, you must restart ClaimCenter. Each destination encapsulates all the necessary behavior for that external system, but uses three different plugin interfaces to implement the destination. Each plugin handles different parts of what a destination does. The message request plugin handles message pre-processing. The message transport plugin handles message transport, and the message reply plugin handles message replies.

Register new messaging plugins in Studio first in the Plugins editor. When you create a new implementation, Studio prompts you for a plugin interface name (in a picker) and, in some cases, for a plugin name. Use that plugin name in the Messaging editor in Studio to register each destination. Remember that you need to register your plugin in two different editors in Studio.

- 2. A user or an API changes something** – A user action, API, or a batch process changed data in ClaimCenter.

For example, ClaimCenter triggers an event if you add an exposure to a claim, add a note to anything, or issue a payment.

It is critical to understand that the change does not fully commit to the database until step 4. Any exceptions that occur before step 4 undo the change that triggered the event. In other words, unless all follow-up actions to the change succeed, the database transaction rolls back. For related information, see “Messaging Database Transactions During Sending” on page 318.

**ClaimCenter generates messaging events** – The same action, such as a single change to the ClaimCenter database, might trigger *more* than one event. ClaimCenter checks whether each destination has listed each relevant event in its messaging configuration. For each messaging destination that listens for that event, ClaimCenter calls your Event Fired business rules. If multiple destinations want notifications for a specific event, ClaimCenter duplicates the event for each destination that wants that event. To the business rules, these duplicates look the same except with different *destination ID* properties. It is critical to understand that a change to ClaimCenter data might generate events for one destination but not for another destination. To change this list for destination, review the **Messaging** editor in Studio and select the row for your destination. Additionally, only entity types defined with the `<events>` tag by default generate added, changed, or deleted events. For more information, see “`<events>`” on page 196 in the *Configuration Guide*.

- 3. ClaimCenter invokes Event Fired rules (and they generate messages)** – ClaimCenter calls the Event Fired rule set for each destination-event pair. Event Fired rules must rules check the event name and the messaging destination ID to determine whether to send a message for that event. Your Event Fired rules generate messages using the Gosu method `messageContext.createMessage(payload)`. The rule actions choose whether to generate a message, the order of multiple messages, and the text-based message *payload*. For more information, see “Generating New Messages in Event Fired Rules” on page 329. Your rules can use the following techniques:

- Optionally export an entity to XML using generated XSDs** – Studio includes a tool that helps you export business data entities (and other types like Gosu classes) to XML. You can select which properties are required or optional for each integration point. You can export an XSD to describe the data interchange format you selected. Then, you can edit your Event Fired rules to generate a payload for the entity that conforms to your custom XSD. For more details, see “Creating XML Payloads Using Guidewire XML (GX) Models” on page 333.

The Guidewire XML (GX) modeler is a powerful tool for integrations. Create custom XML models that contain only the subset of entity object data that is appropriate for each integration point. It can output a custom XSD that your external system can use. You can also use GX models to parse (import) XML data into in-memory objects that describe the XML structure. See “Creating XML Payloads Using Guidewire XML (GX) Models” on page 333.

- Optionally use Gosu templates to generate the payload** – Rules can optionally use templates with embedded Gosu to generate message content.

- **Optionally use Java classes (called from Gosu) to generate the payload** – Rules can optionally use Java classes to generate the message content from Gosu business rules. See “Calling Java from Gosu” on page 119 in the *Gosu Reference Guide*.
- **Optional late binding** – You can use a technique called *late binding* to include parameters in a message payload at message creation time but evaluate them immediately before sending. See “Late Binding Data in Your Payload” on page 339 for details.

**WARNING** Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309.

**4. New messages are added to the send queue** – After all rules run, ClaimCenter adds any new messages to the send queue in the database. The submission of messages to the send queue is part of the same database transaction that triggered the event to preserve atomicity. If the transaction succeeds, all related messages successfully enter the send queue. If the transaction *fails*, the change rolls back including all messages added during that transaction. Messages might wait in this queue for a while, depending on the state of acknowledgements and the status of safe ordering of other messages (see step 5).

**5. On the batch server only, ClaimCenter dispatches messages to messaging destinations** – ClaimCenter retrieves messages from the send queue and dispatches messages to destinations. The messaging plugins for each destination sends each message. For details of how ClaimCenter retrieves messages and orders them for sending, see “Safe Ordering” on page 302 and “Message Ordering and Multi-Threaded Sending” on page 334.

To send a message, ClaimCenter finds the messaging destination’s transport plugin and calls its `send` method. The message transport plugin sends the message in whatever native transport layer is appropriate. See “Transport Neutrality” on page 303. If the `send` method throws an exception, ClaimCenter automatically retries the message.

**6. Acknowledging messages** – Some destination implementations know success or failure immediately during sending. For example, a messaging transport plugin might call a synchronous remote procedure call on a destination system before returning from its `send` method.

In contrast, a messaging destination might need to wait for an *asynchronous*, time-delayed reply from the destination to confirm that it processed the message successfully. For example, it might need to wait for an incoming message on an external messaging queue that confirms the system processed that message.

In either case, the messaging destination code or the external system must confirm that the message arrived safely by submitting an *acknowledgement* (an ACK) to ClaimCenter for that specific message. Alternatively, it can submit an error, also called a negative acknowledgement or NAK. You can submit an ACK or NAK in several places. For synchronous sending, submit it in your `MessageTransport` plugin during the `send` method. For asynchronous sending, submit it in your `MessageReply` plugin. For asynchronous sending, an external system could optionally use a SOAP API to submit an acknowledgement or error.

If using messaging plugins to submit the ACK, you can also make changes to data during the ACK, such as updating properties on entities. For financials objects, acknowledgements of messages have special behavioral side effects, such as changing the status of a check. For more information, see “Financials Integration” on page 363.

Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309.

**7. After an ACK or NAK for a safe-ordered message, ClaimCenter dispatches the next related message** – An ACK for a safe-ordered message affects what messages are now sendable. If there are other messages for that destination in the send queue for the same primary object, ClaimCenter soon sends the next message for that primary object. For details of how messages are retrieved and ordered, see “Safe Ordering” on page 302 and “Message Ordering and Multi-Threaded Sending” on page 334. Read that section for details of the setting

called Strict Mode. Strict Mode affects whether ClaimCenter waits for acknowledgements for non-safe-ordered messages.

## Important Entity Data Restrictions in Messaging Rules and Messaging Plugins

Event Fired rules and messaging plugin implementations have limitations about changing entity instance data. Messaging code in these locations must perform only the minimal data changes necessary for integration on the message entity.

**WARNING** For data integrity and server reliability, you must carefully follow restrictions regarding what data you can change in Event Fired rules and messaging plugin implementations.

### Event Fired Rule Set Restrictions for Entity Data Changes

**WARNING** Entity changes in Event Fired must be very limited. Changes do not trigger validation rules, or pre-update rules. Read this topic carefully for additional restrictions.

The following important restrictions apply to code within the Event Fired rule set:

- In general, perform any data updates in your preupdate rules, not your Event Fired rules.
- A property is safe to change in Event Fired rules only if it is messaging-specific and users can never modify it from the user interface, even indirectly. Carefully consider the role of every property you might modify.
- The only object safe to add is a new message using the `createMessage` method on the message context object. Never create new objects of any other types, even indirectly through other APIs.
- Never delete objects.
- Never call business logic APIs that might change entity data, even in edge cases.
- Never rely on any entity data changes triggering these common rule sets:
  - Pre-update rule set
  - Validation rule set
  - Event Fired rule set
- These restrictions apply to all entity types, including custom entity types.

Design your messaging rules carefully around these restrictions to avoid data corruption and logical errors that are difficult to diagnose.

### Messaging Plugin Restrictions for Entity Data Changes

**WARNING** Entity changes in messaging plugin code must be very limited. Changes do not trigger validation rules, pre-update rules, or concurrent data change exceptions. Read this topic carefully for additional restrictions.

The following important restrictions apply to code triggered by a `MessageTransport` or `MessageReply` plugin implementation:

- A property is safe to change only if it is messaging-specific and users can never modify it from the user interface, even indirectly. Carefully consider the role of every property you might modify.
- You cannot create any new objects in general. In the default configuration, only the following objects are safe to create within a messaging plugin:
  - activities
  - notes
  - workflows and workflow items

- work queues

You cannot rely on preupdate rules or validation rules running for those objects.

All other object types are dangerous and unsupported to add from within messaging plugins.

If you modify the data model such that there are additional foreign keys on these objects, even these objects explicitly listed may be unsafe to add. For advice on specific changes, contact Guidewire Customer Support.

- Never delete objects.

- You must not rely on any entity data changes eventually triggering these common rule sets:

- Pre-update rule set
- Validation rule set
- Event Fired rule set for standard events *ENTITYAdded*, *ENTITYChanged*, and *ENTITYRemoved*.

However, Event Fired rule set still triggers for events that you explicitly add. You can use the API `entity.addEvent("messageName")` to add events. ClaimCenter calls the Event Fired rule set once for each custom event for each messaging destination that listens for it. Your Event Fired code must conform to all restrictions in “Event Fired Rule Set Restrictions for Entity Data Changes” on page 309.

- Never call business logic APIs that might change entity data, even in edge cases.
- From messaging plugin code, entity instance changes do not trigger concurrent data exceptions except in special rare cases. To avoid data integrity issues with concurrent changes, avoid changing data in these code locations. See later in this topic for additional guidance.
- In some cases, consider adding or advancing a workflow as an alternative to direct data modifications from messaging code. The workflow can asynchronously perform code changes in a separate bundle outside your messaging-specific code.
- If you must update messaging-specific data, consider how absence of detecting concurrent data changes from messaging plugins might affect your extensions to the data model. For example, suppose you intend to modify an entity type to add a property with simple data. Instead, you could add a property with a foreign key to an instance of a custom entity type. First, create the instance of your custom entity type at an early part of the lifecycle of your main objects long before messaging code runs. As mentioned earlier, it is unsupported to create a entity instance in Event Fired rules or in messaging plugins. This restriction applies to all entity types, including custom entity types. In Event Fired rules or in messaging plugins, modify the messaging-specific entity instance. With this design, there is less chance of concurrent data change conflicts from a simple change on the main business entity instance from within the user interface.

In a `MessageRequest` plugin implementation, do no data changes at all.

Separate from the concurrent data exception differences mentioned earlier, there is an optional feature to *lock* related objects during messaging actions. If you use optional locking and code tries to access locked data (the primary entity instance or the message), the calling code waits for it to be unlocked.

---

**IMPORTANT** For maximum data integrity, enable optional entity locking during messaging. See “Messaging Database Transactions During Sending” on page 318. For maximum performance, disable optional entity locking during messaging.

---

Design your messaging code carefully around these restrictions to avoid data corruption and logical errors that are difficult to diagnose.

## Database Transactions When Creating Messages

All steps up to and including adding messages to the send queue occur in one database transaction. In the list earlier in “Messaging Flow Details” on page 305, this is step 2 through step 4. All changes commit in the same transaction. This is always the same database transaction that triggers the initial event.

The database transaction rolls back if any of the following occur:

- Exceptions in rule sets that run before message creation
- Exceptions in Event Fired rules (where you create your messages)
- Exceptions in rule sets that run after Event Fired rules but before committing the bundle to the database
- Errors committing the bundle to the database, and remember that this bundle includes new Message objects

If any of these errors occur, ClaimCenter rolls back all messages added to the send queue in that transaction.

There are special rules about database transactions during message sending at the destination level. See “Messaging Database Transactions During Sending” on page 318.

## Messaging in ClaimCenter Clusters

If you run ClaimCenter in a cluster, be aware that step 1 through step 4 can occur on any ClaimCenter server within the cluster. ClaimCenter processes events on the same server as the user action or API call that triggered the event.

Once a message is in the send queue (step 5 through step 7), any further action with the message occurs *only* on the batch server.

Consequently, the batch server is the only server on which messaging plugins run. Configure your batch server (and any backup batch servers) to communicate with your destination external systems. For example, remember to configure your firewalls accordingly including your backup batch servers.

For ClaimCenter clusters, only the batch server actually uses your messaging plugins.

## Messaging Plugins Must Not Call SOAP APIs on the Same Server

In general, avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data.

Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance. If you have questions about how to convert some particular use, locally-hosted SOAP APIs from plugins or rules, contact Customer Support.

This is true for all types of local loopback SOAP calls to the same server. This includes the `soap.local.*` Gosu APIs, the SOAP API libraries, and any Studio-registered web services that call the same server as the client.

Those limitations are true for all plugin code. In addition, there are messaging-specific limitations with this approach. Specifically, ClaimCenter locks the root entity for the message in the database. Any attempts to modify this entity from outside your messaging plugin (and SOAP APIs are included) result in concurrent data exceptions.

---

**WARNING** Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine. There are various problems if you call SOAP APIs that modify entities that are currently in use, including but not limited to APIs that might change the message root entity.

---

## Message Destination Overview

To represent each external system that receives a message, you must define a *messaging destination*. Typically, a destination represents a distinct remote system. However, you could use destinations to represent different remote APIs or different message types to send from ClaimCenter business rules.

Carefully choose how to structure your messaging code. For example, if there are several related remote systems or APIs, you must choose whether they are logically one messaging destination or multiple destinations. Your choice affects messaging ordering. The ClaimCenter messaging system ensures there is no more than one in-flight message per primary object per destination. This means that the definition of a destination is important for message ordering and multithreading. See “Message Ordering and Multi-Threaded Sending” on page 334.

Each destination specifies a list of events for which it wants notifications and various other configuration information. Additionally, a destination encapsulates a list of your plugins that perform its main destination functions:

- **Message preparation** – (Optional) If you need message preparation before sending, write a plugin that implements the message request (`MessageRequest`) plugin interface. For example, this plugin might take simple name/value pairs stored in the message payload and construct an XML message in the format required by a transport or final message recipient. If the destination requires no special message preparation, omit the request plugin entirely for the destination. For implementation details and optional post-send processing, see “Implementing a Message Request Plugin” on page 344.
- **Message transport** – (Required) The main task of a destination is to send messages to an external system. Its underlying protocol might be an external message queue, web service request to external systems, FTP, or a proprietary legacy protocol. You actually send your messages in your own implementation of the `MessageTransport` plugin interface. Every destination must provide a message transport plugin implementation. Multiple messaging destinations might use the same messaging transport plugin implementation if they really do use the underlying transport code. For implementation details, see “Implementing a Message Transport Plugin” on page 345.
- **Message reply handling** – (Required only if asynchronous) If a destination requires an asynchronous callback for acknowledgements, implement the `MessageReply` plugin. If the destination requires no asynchronous acknowledgement, omit the reply plugin. For implementation details, see “Implementing a Message Reply Plugin” on page 346.

**Note:** ClaimCenter includes multi-threaded inbound integration APIs that you can optionally use in conjunction with `MessageReply` plugins. For example, listen for JMS messages or process text files that represent message replies. If you want to use input data other than JMS messages or text files for message replies, write a custom integration by implementing the `InboundIntegrationMessageReply` plugin. `InboundIntegrationMessageReply` is a subinterface of `MessageReply`. See “Multi-threaded Inbound Integration Overview” on page 279 and “Custom Inbound Integrations” on page 289.

After you write code that implements your messaging plugins, register them in Studio. When registering an implementation of new messaging plugin, Studio prompts you for a plugin name. The plugin name is different from the implementation class name. The plugin name is a short arbitrary name that identifies a plugin implementation. Studio only prompts you for a plugin name for plugin interfaces that support more than one implementation.

Use that plugin name to configure the messaging destination in the Messaging editor in Studio to register each destination. For details and examples of this editor, see “Using the Messaging Editor” on page 137 in the *Configuration Guide*.

For more information about plugins, see “Plugin Overview” on page 163.

To create new destinations, configure the messaging registry to specify the list of destinations, each of which includes the following information:

- The *plugin name* class for a `MessageTransport` plugin in Java or Gosu.
- Optionally, the *plugin name* for the implementation of a `MessageRequest` plugin.
- Optionally, the *plugin name* for the implementation of a `MessageReply` plugin.
- **Retry Interval** – The amount of time in milliseconds (`initialretryinterval`) after a retryable error to retry a sending a message.
- **Max Retries** – The number of automatic retries (`maxretries`) to attempt before suspending the messaging destination.

- **Backoff Multiplier** – The amount to increase the time between retries, specified as a multiplier of the time previously attempted. For example, if the last retry time attempted was 5 minutes, and the multiplier (`retrybackoffmultiplier`) is set to 2, ClaimCenter attempts the next retry in 10 minutes.
- **Event Names** – A list of events to listen for, by name. Each event triggers the Event Fired rule set for that destination. To specify that the destination wants to listen for all events, use the special event name string "`(\w)*`".
- **Destination ID**, which typically you use in your Event Fired rules to check the intended messaging destination for the event notification. If five different destinations request an event that fires, the Event Fired rule set triggers five times for that event. They differ only in the *destination ID* property (`destID`) within each message context object. Each messaging plugin implementation must have a `setDestinationID` method. This method allows your destination to get its own destination ID to store it in a private variable. Your code can use the stored value for logging messages or send it to integration systems so that they can programmatically suspend/resume the destination if necessary.

The valid range for your destination IDs is 0 through 63, inclusive. Guidewire reserves all other destination IDs for built-in destinations such as the email transport destination.

- **Alternative Primary Entity** – See “Primary Entity and Primary Object” on page 301
- **Strict Mode** – See “How Destination Settings Affects Ordering” on page 338
- **Poll interval** – Each messaging destination pulls messages from the database (from the send queue) in batches of messages on the batch server. The application does not query again until this amount of time passes. Use this field to set the value of the polling interval to wait. After the current round of sending, the messaging destination sleeps for the remainder of the poll interval. If the current round of sending takes longer than the poll interval, then the thread does not sleep at all and continues to the next round of querying and sending. See “Message Ordering and Multi-Threaded Sending” on page 334 for details on how the polling interval works. If your performance issues primarily relate to many messages per primary object per destination, then the polling interval is the most important messaging performance setting.
- **Sender Threads** – To send messages associated with a primary object, ClaimCenter can create multiple sender threads for each messaging destination to distribute the workload. These are threads that actually call the messaging plugins to send the messages. Use this field to configure the number of sender threads for safe-ordered messages. This setting is ignored for non-safe-ordered messages, since those are always handled by one thread for each destination. If your performance issues primarily relate to many messages but few messages per claim for each destination, then this is the most important messaging performance setting. For more information, see “Message Ordering and Multi-Threaded Sending” on page 334.
- **Shutdown timeout** – Messaging plugins have methods to handle the administrative commands suspend, resume, and preparing for the messaging system to shutdown. The shutdown timeout value is the length of time to wait before attempting to shutdown the messaging subsystem. For more information about suspend and shutdown actions, see “Message Destination Overview” on page 311.

Manage these settings in Guidewire Studio in the Messaging editor. See “Using the Messaging Editor” on page 137 in the *Configuration Guide*.

**IMPORTANT** If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor; see “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*. Remember the plugin implementation name that you use. You need it to configure the messaging destination in the Messaging editor in Studio to register each destination. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 137 in the *Configuration Guide*.

To write your messaging plugins, see “Implementing Messaging Plugins” on page 343.

**WARNING** Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309.

### Sharing Plugin Classes Across Multiple Destinations

It is common to implement messaging plugin classes that multiple destinations share. For example, a transport plugin might manage the transport layer for multiple destinations that use that physical protocol. In such cases, be aware of the following things:

- The class instantiates once for each destination. The instance is not shared across destinations. However, you still must write your plugin code as threadsafe, since you might have multiple sender threads. The number of sender threads only affects safe-ordered messaging, and is a field in the Messaging editor in Studio for each destination.
- Each messaging plugin instance distinguishes itself from other instances by implementing the `setDestinationID` method and saving the destination ID in a private class variable. Use this later for logging, exception handling, or notification e-mails. For more information, see “Saving the Destination ID for Logging or Errors” on page 349.

## Handling Acknowledgements

Due to differences in external systems and transports, there are two basic approaches for handling replies. ClaimCenter supports synchronous and asynchronous acknowledgements, although in different ways:

1. **Synchronous acknowledgement at the transport layer** – For some transports and message types, acknowledging that a message was successfully sent can happen synchronously. For example, some systems can accept messages through an HTTP request or web service API call. For such a situation, use the synchronous acknowledgement approach. The synchronous approach requires that your transport plugin `send` method actually send the message and immediately submit the acknowledgement with the message method `reportAck`. For error and duplicate handling:
  - In most cases, including most network errors, throw an exception in the `send` method. This triggers automatic retries of the message sending using the default schedule for that messaging destination.
  - For other errors or flagging duplicate messages, call the `reportError` or `reportDuplicate` methods. See “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 341.
2. **Asynchronous acknowledgement** – Some transports might finish an initial process such as submitting a message on an external message queue. However in some cases, the transport must wait for a delayed reply before it can determine if the external system successfully processed the message. The transport can wait using polling or through some other type of callback. Finally, submit the acknowledgement as successful or an error. External systems that send status messages back through a message reply queue fit this category. There are several ways to handle asynchronous acknowledgements, as described later.

For asynchronous acknowledgement, the messaging system and code path is much more complex. In this case, the message transport plugin does not acknowledge the message during its main `send` method.

The typical way to handle asynchronous replies is through a separate plugin called the message reply plugin. The message reply plugin uses a callback function that acknowledges the message at a later time. For example, suppose the destination needed to wait for a message on a special incoming messaging queue to confirm receipt of the message. The destination’s message reply plugin registers with the queue. After it receives the remote acknowledgement, the destination reports to ClaimCenter that the message successfully sent.

One special step in asynchronous acknowledgement with a message reply plugin is setting up the callback routine’s database *transaction* information appropriately. Your code must retrieve message objects safely and commit any updated objects (the ACK itself and or additional property updates) to the ClaimCenter database.

To set up the callbacks properly, Guidewire provides several interfaces, including:

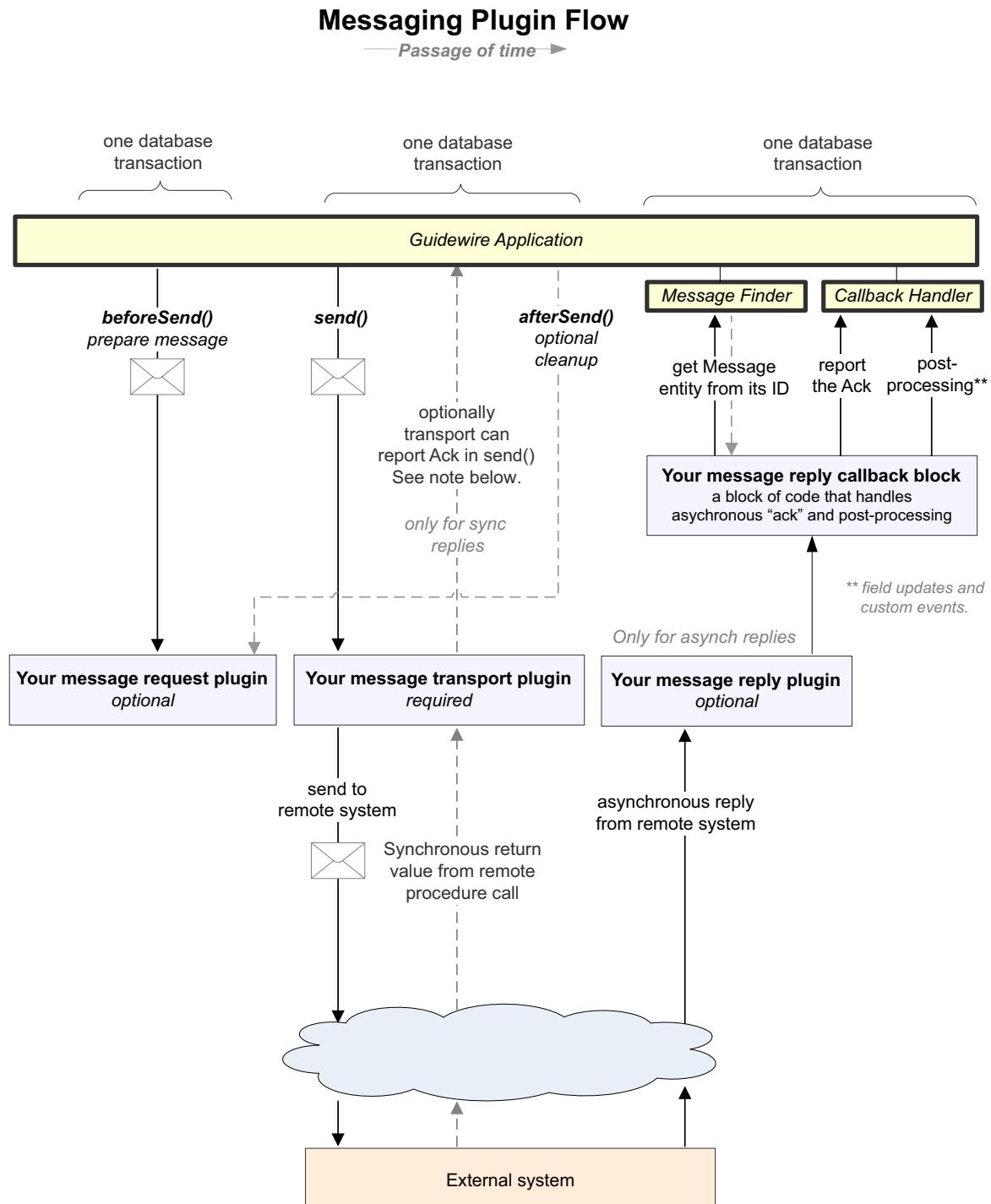
- **A message finder** – A class that returns a `Message` object from its message ID (the `MessageID` property) or from its sender reference ID and destination ID (`SenderRefID` and `DestinationID`). ClaimCenter provides an instance of this class to a message reply plugin upon initialization in its `initTools` method.

- **A plugin callback handler** – A class that can execute the message reply callback block in a way that ensures that any changes commit. ClaimCenter provides an instance of this class to a message reply plugin upon initialization in its `initTools` method.
- **Message reply callback block interface** – The actual code that the callback handler executes is a block of code that you provide called a *message reply callback block*. This code block is written to a very simple interface with a single `run` method. This code can acknowledge a message and perform post-processing such as property updates or triggering custom events.

For more information about these objects and how to implement them, see “[Implementing a Message Reply Plugin](#)” on page 346.

An alternative to this approach is for the external system to call a ClaimCenter web service (SOAP) API to acknowledge the message. ClaimCenter publishes a web service called `MessagingToolsAPI` that has an `ackMessage` method. Set up an `Acknowledgement` object with the `MessageID` set to the message ID as a `String`. If it was an error, set the `Error` property to `true`. Pass the `Acknowledgement` to the `ackMessage` method.

The following diagram illustrates the destination-related plugins, associated components, along with the chronological flow of actions between elements in the system.



*Note:* A message transport can acknowledge the message from within the transport's `send()` method if possible. In that case, post-processing such as field updates would be done during the `send()` method. For destinations where the reply must be asynchronous (delayed), that destination must define a *message reply plugin* and its *callback block* to listen for and report the acknowledgement.

## Rule Sets Must Never Call Message Methods for ACK, Error, or Skip

From within rule set code, you must never call any message acknowledgment or skipping methods such as the `Message` methods `reportAck`, `reportError`, or `skip`. Use those `Message` methods only within messaging plugins.

This prohibition also applies to Event Fired rules.

## Messaging Database Transactions During Sending

As mentioned in “Database Transactions When Creating Messages” on page 310, all steps up to and including adding the message to the send queue occurs in one database transaction. This is the same database transaction that triggered the event.

Additionally, there are special rules about database transactions during message sending at the destination level.

1. ClaimCenter calls `MessageRequest.beforeSend(...)` in one database transaction and commits changes assuming no exceptions occurred.
2. ClaimCenter calls `MessageTransport.send(...)` and then `MessageRequest.afterSend(...)` in one database transaction and commits changes assuming no exceptions occurred.
3. The `MessageReply` plugin, which optionally handles asynchronous acknowledgements to messages, does its work in a separate database transaction and commits changes assuming no exceptions occurred.

## Message and Entity Locking

At the start of each transaction, ClaimCenter locks the message object itself at a database level. This ensures that the application does not try to acknowledge a skipped message or similar conditions. If some code tries to access the message while it is locked, the calling code waits for the current lock holder to unlock the message. See the diagram earlier in this section for the timing of messaging database transactions.

ClaimCenter can optionally lock the primary entity instance for a message if there is an associated safe ordered object for the message. The entity instance locking only affects the one entity instance, not any subobjects.

Similar to message locking, if some code tries to access the message while it is locked, the calling code waits for the current lock holder to unlock the message. ClaimCenter checks the `config.xml` parameter

`LockPrimaryEntityDuringMessageHandling`. If it is set to `true`, ClaimCenter locks the primary entity instance for that destination during the following operations:

- during send
- during message reply handling
- while marking a message as skipped

For example, if the message is associated with a claim, during all of these operations you can request that ClaimCenter optionally lock the associated `Claim` object at the database level. This can reduce problems in edge cases in which other threads try to modify objects associated with this claim.

**WARNING** This entity instance (and message) lock handling is separate from the concurrent data exception system. Be aware that concurrent data exception checking is disabled during messaging access. For important related information, see “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309.

## Built-In Destinations

Your rule sets can send standard e-mails and optionally attach the email to the claim as a document. The built-in email APIs use the built-in email destination. For more information about email configuration and API to send emails, see “Sending Emails” on page 135 in the *Rules Guide*. ClaimCenter always creates the built-in email destination, independent of your configuration settings.

Additionally, ClaimCenter includes a built-in destination to communicate with the Insurance Services Office (ISO), which is an optional type of ClaimCenter integration. Unlike the built-in email destination, ClaimCenter creates the ISO destination only if a special line in the server `config.xml` file is present. For more information about ISO, see “Insurance Services Office (ISO) Integration” on page 447.

Similarly, ClaimCenter includes a built-in destination to communicate with the Metropolitan Reporting Bureau, which is a police accident and police report inquiry service. ClaimCenter creates the Metropolitan destination only if a special line in the server `config.xml` file is present. For more information about Metropolitan, see “Metropolitan Reporting Bureau Integration” on page 487.

## Filtering Events

This section focuses on how ClaimCenter recognizes events and how to decide whether to notify external systems of these events.

### Validity and Rule-Based Event Filtering

ClaimCenter allows entry and creation of claims with fewer restrictions than might be true for an external system, such as a mainframe. ClaimCenter does this so that a new claim can be committed to the database with minimal information and the claim can be improved later as you gather more information.

However, in some cases, an external system might not want to know about the claim until a much more complete (or perhaps more correct) claim commits to the database. You might choose to not send a *claim added* message to the mainframe if there is not enough information to create the record on the mainframe. Express this level of correctness or completeness by setting a *validation level* for the claim. Each external system’s *destination* defined in ClaimCenter may have completely different validation requirements.

For example, suppose an external system wants to be notified of every claim, but its standards were high about what kinds of claims it wants to know about. Additionally, you might want to omit notifying an external system about new notes on a claim that it does not know about yet. In other words, you did not send a message notifying the external system about the claim, so you probably do not want to send updates about its subobjects.

To implement this, there are two parts of the integration implementation:

- 1. Your validation rules set the validation level** – There are two rule sets that govern validation checking, one for claims (Claim Validation Rules) and one for exposures (Exposure Validation Rules). These rules can set the validation level. Set the property `claim.ValidationLevel`.
- 2. Your event rules check validation level (and other settings)** – Your Event Fired rules define whether to send a message to an external system. For example, the rule might listen for the events `ExposureAdded`, `ExposureChanged`, `ClaimAdded`, `ClaimChanged`, and perhaps other events. If the event name and destination ID matches what it listens for and the validation level was greater than a certain level, the rule creates a new message. Such a rule in Gosu might look like:

```
if (claim.ValidationLevel > externalSystemABCLevel) {  
    // create message...  
}
```

Similarly, if you did not want to send events for a note before its associated claim is valid, you could write a rule condition such as the following:

```
note.claim.ValidationLevel > externalSystemABCLevel
```

It is important to understand that you define your own standards for event filtering and processing. ClaimCenter does not enforce any requirements about validation levels. The `EntitlenameAdded` events and `EntitlenameChanged` events trigger independent of the object’s validation level.

Generally speaking, ClaimCenter does not use the validation levels. You can define rules that set or get the validation level for some purpose. Your messaging rules might send a message to an external system because of an event, but ignore the event in some cases due to the validation rules.

There is one rule set that governs claim checking, called Claim Validation Rules. You can set validation levels in these rules that are checked later within the Event Fired rule set.

ClaimCenter itself does not actually use the validation level for any purpose. However, if you use the ISO integration, the validation level defines whether the claim or exposure is ready for ISO.

## List of Messaging Events in ClaimCenter

ClaimCenter generates events associated with a specific entity instance as the root object for the event. For more information about root objects, see “Root Object” on page 301. Also, contrast this with the concept of a primary object, see “Primary Entity and Primary Object” on page 301.

Sometimes an event root object is a higher-level object such as claim. In other cases, the event is on a subobject, and your Event Fired rules must do some work to determine what high-level object it is about. For example, the Address subobject is common and changing it is common. However, what larger object contains this address? You might need this additional context to do something useful with the event.

For example, was the address a claim’s loss location? Was it the claimant’s temporary address?

In ClaimCenter, all the child objects of a claim have a `Claim` property that references the claim.

ClaimCenter triggers events for many objects if an entity is added, removed (or retired), or if a property changes on the object. For example, selecting a different claim type on any claim generates a *changed* event on the claim.

The changes are about the change to that database row itself, not on subobjects. For example, ClaimCenter reports a change to an object if a foreign key reference to a subobject changes but not if properties on the subobject changes.

There are exceptions to this rule.

For example, selecting a different claimant on a claim is a change to the claim. A change to an exposure is an exposure change, but not a claim change.

The following table describes the events that ClaimCenter raises. In this table, *standard* events refer to the *added*, *changed*, and *removed* events for entities that generate events. For example, `Claim` entity would generate events whenever code adds, changes, or removes entities of that type in the database. In those cases, the Event Fired business rules would see `ClaimAdded`, `ClaimChanged`, and `ClaimRemoved` events if one or more destinations registered for those event names.

Entity	Events	Description
Activity	<code>ActivityAdded</code> <code>ActivityChanged</code> <code>ActivityRemoved</code>	Standard events for the Activity entity. <code>ActivityChanged</code> indicates that an activity changed, including marking the activity completed or skipped.
Assignable Entities: • Claim • Exposure • Activity • Matter	<code>AssignmentAdded</code> <code>AssignmentChanged</code> <code>AssignmentRemoved</code>	Assignment events for assignable entities. The assignable entity is the root entity for the event. If an assignment added to this entity, <code>AssignmentAdded</code> triggers. If the entity previously had an assignment and now has no assignment, <code>AssignmentRemoved</code> triggers. If assignment data such as assigned user, group, date changed, <code>AssignmentChanged</code> triggers.

Entity	Events	Description
Claim	ClaimAdded ClaimChanged ClaimRemoved	Standard events for claims. It is important to understand that the ClaimAdded and ClaimChanged events are sent for claims and exposures independent of their validity level. If you want to send claims and exposures to external systems only after reaching a specific <i>validation level</i> , your Event Fired rules can decide to ignore that event. To prevent duplicate messages of certain types (such as initial message to external systems), create data model extensions on claim or exposure and set them within Event Fired rules. The rules mark a claim or exposure as already sent to that external system. The ClaimChanged event triggers if a claim closes or reopens. ClaimCenter does not trigger these events for a claim resync (see ClaimResync in this table).
	ClaimResync	Resync a claim. This is an administrator request to drop all pending and <i>in error</i> messages for a claim. Then, your Event Fired rules tries to resync the claim with the external system to recover from integration problems. Administrators can request a claim resync from the application user interface or through the web services API. For more information, see "Resynchronizing Messages for a Primary Object" on page 350.
ClaimInfo	ClaimInfoAdded ClaimInfoChanged ClaimInfoRemoved	Standard events for ClaimInfo objects.
Coverage	CoverageAdded CoverageChanged CoverageRemoved	Coverage entities are abstract entities that exist in the form of coverage subtypes, such as PolicyCoverage. CoverageAdded, CoverageChanged, and CoverageRemoved events are sent for coverage subtypes. Most customers do not let users remove existing coverages. Thus, in practice, ClaimCenter may never trigger the CoverageRemoved event. Changing a property on a Coverage entity triggers the Policy_Changed event in addition to the Coverage_Changed event.
Document	DocumentAdded DocumentChanged DocumentRemoved	Standard events for documents. The DocumentAdded events trigger if a document links to the claim file. Most implementations do not permit users to remove documents once added, so this event may never trigger.
Exposure	ExposureAdded ExposureChanged ExposureRemoved	Standard events for exposures. Note: <ul style="list-style-type: none"> <li>If it is important to send claims and exposures to external systems only after reaching a specific <i>validation level</i>, Event Fired rules decide whether to ignore or log that event.</li> <li>To prevent duplicate messages of certain types such as initial message to external systems, create data model extension messaging-specific properties on claim or exposure. Set your properties in Event Fired rules. Your rules indicate that a claim or exposure was already sent to that external system.</li> <li>The ExposureChanged event triggers if a claim is closes or reopens.</li> <li>If your implementation does not allow users to remove exposures once added, the ExposureRemoved event may never trigger.</li> </ul>
Matter	MatterAdded MatterChanged MatterRemoved	Standard events for the Matter entity. The MatterAdded event triggers after a legal matter attaches to a claim. If your implementation does not allow users to remove legal matters from a claim once added, the MatterRemoved event may never trigger.

<b>Entity</b>	<b>Events</b>	<b>Description</b>
MetroReport	MetroReportAdded MetroReportChanged MetroReportRemoved	Standard events for MetroReport entities. For more information about the optional Metropolitan service, see “Metropolitan Reporting Bureau Integration” on page 487.
Negotiation	NegotiationAdded NegotiationChanged NegotiationRemoved	Standard events for Negotiation entities.
Note	NoteAdded NoteChanged NoteRemoved	Standard events for Note entities.
Policy	PolicyAdded PolicyChanged PolicyRemoved	Standard events for policies. Notes: <ul style="list-style-type: none"><li>• The PolicyAdded event triggers if something adds a policy snapshot to ClaimCenter. This happens after entering a new claim or after changing the policy on an existing claim.</li><li>• The PolicyChanged event triggers after a policy or any subobject updates. This includes property, vehicle, stat code, but not coverages.</li><li>• ClaimCenter sends these events independent of the claim validation level of the associated claim. Filter (ignore) events as needed</li><li>• Changes to a property on a Coverage entity triggers the PolicyChanged event in addition to the CoverageChanged event.</li></ul>
PolicyCoverage	CoverageAdded CoverageChanged CoverageRemoved	Standard events for Coverage entities.
PropertyCoverage	CoverageAdded CoverageChanged CoverageRemoved	Standard events for Coverage entities.
VehicleCoverage	CoverageAdded CoverageChanged CoverageRemoved	Standard events for Coverage entities.
<b>General-purpose events</b>		
Workflow	WorkflowAdded WorkflowChanged WorkflowRemoved	Standard events for Workflow entities.
ProcessHistory	ProcessHistoryAdded ProcessHistoryChanged ProcessHistoryRemoved	Some integrations can be done as a batch process. For example, the financials escalation process starts manually or on a timer. You could use the event/messaging system to write records to a batch file (or rows to a database table) as each transaction processes. Perhaps during the night, you can submit the batch data to some downstream system. To coordinate this activity, you can use the ProcessHistory entity. As a batch process starts, a ProcessHistoryAdded event triggers. Listen for the ProcessHistoryChanged event in your Event Fired rules, and check the <code>processHistory.CompletionTime</code> property for the date-stamp in a datetime object. See “Batch Processes and Work Queues” on page 123 in the <i>System Administration Guide</i> .
SOAPCallHistory	SOAPCallHistoryAdded SOAPCallHistoryChanged SOAPCallHistoryRemoved	Standard events for SOAPCallHistory entities. The application creates one for each incoming web service call.
StartablePluginHistory	StartablePluginHistoryAdded StartablePluginHistoryChanged StartablePluginHistoryRemoved	Standard events for StartablePluginHistory entities. The application creates these to track when a startable plugin runs.
InboundHistory	InboundHistoryAdded InboundHistoryChanged InboundHistoryRemoved	Standard events for root entity Invoice.
<b>Administration events</b>		

Entity	Events	Description
Catastrophe	CatastropheAdded CatastropheChanged CatastropheRemoved	Standard events for catastrophes. The CatastropheChanged event triggers if the catastrophe retires or gets a new catastrophe code.
Group	GroupAdded GroupChanged GroupRemoved	Standard events for groups. The GroupChanged event triggers after additions or removals of users from a group.
Role	RoleAdded RoleChanged RoleRemoved	Standard events for Role entities.
User	UserAdded UserChanged UserRemoved	Standard events for users. ClaimCenter triggers the UserChanged event only for changes made directly to the user record, not for changes to roles or group memberships. Be aware that changes to the user's contact record such as a phone number cause a ContactChanged event, not a UserChanged event.
UserSettings	UserSettingsAdded UserSettingsChanged UserSettingsRemoved	Standard events for user settings.
GroupUser	GroupUserAdded GroupUserChanged GroupUserRemoved	Standard events for root entity GroupUser.
UserContact	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
<b>Financial transaction events</b>		
BulkInvoice	BulkInvoiceAdded BulkInvoiceChanged BulkInvoiceRemoved	Standard events for the BulkInvoice entity.
	BulkInvoiceStatusChanged	The bulkinvoice.status property changed. See "Financials Integration" on page 363 and "Claim Financials Web Services (ClaimFinancialsAPI)" on page 369 for details.
Check	CheckAdded CheckChanged CheckRemoved	Standard events for the Check entity.
	CheckStatusChanged	The check.status property changed. See "Financials Integration" on page 363 and "Check Integration" on page 372 for details.
Reserve	ReserveAdded ReserveChanged ReserveRemoved	Standard events for reserves. Check for changes to the status property within rules that catch the ReserveChanged event.
	ReserveStatusChanged	The reserve.status property changed. See "Financials Integration" on page 363 and "Reserve Transaction Integration" on page 390 for details.
AuthorityLimit-Profile	AuthorityLimitProfileAdded AuthorityLimitProfileChanged AuthorityLimitProfileRemoved	Standard events for the AuthorityLimitProfile entity.
Payment	PaymentAdded PaymentChanged PaymentRemoved	Standard events for Payment entities. Check for changes to the status property within rules that catch the PaymentsChanged event.
	PaymentStatusChanged	The payment.status property changed, possibly but not necessarily because of a change to the associated check. See "Financials Integration" on page 363 and "Reserve Transaction Integration" on page 390 for details.
RecoveryReserve	RecoveryReserveAdded RecoveryReserveChanged RecoveryReserveRemoved	Standard events for recovery reserves. The RecoveryAdded event triggers after some code adds a recovery reserve change. Typically the recovery is initially in the submitting status.

<b>Entity</b>	<b>Events</b>	<b>Description</b>
	RecoveryReserveStatusChanged	The recoveryreserve.status property changed. See “Financials Integration” on page 363 and “Recovery Reserve Transaction Integration” on page 387 for details.
Recovery	RecoveryAdded RecoveryChanged RecoveryRemoved	Standard events for recovery transactions.
	RecoveryStatusChanged	The check.status property changed. See “Financials Integration” on page 363 and “Recovery Transaction Integration” on page 388 for details.
Transaction	TransactionAdded TransactionChanged TransactionRemoved	Standard events for Transaction, which is the supertype of other transactions.
RITransaction	RITransactionAdded RITransactionChanged RITransactionRemoved	Standard events for RITransaction
<b>Contacts and address book events</b>		
ABContact	ABContactAdded ABContactChanged ABContactRemoved	Standard events for ABContact entities. (ContactManager only)
Adjudicator	ContactAdded ContactChanged ContactRemoved	Standard events for the Contact entity.
Attorney	ContactAdded ContactChanged ContactRemoved	Standard events for the Contact entity.
AutoTowingAgcy	ContactAdded ContactChanged ContactRemoved	Standard events for the Contact entity.
AutoRepairShop	ContactAdded ContactChanged ContactRemoved	Standard events for the Contact entity.
ClaimAssociation	ClaimAssociationAdded ClaimAssociationChanged ClaimAssociationRemoved	Standard events for the ClaimAssociation entity.
ClaimContact	ClaimContactAdded ClaimContactChanged ClaimContactRemoved	Standard events for ClaimContact entities. The ClaimContactChanged event is particularly important if you track changes to claim contacts within ClaimCenter. For example, if the Tax ID or phone number of a claimant changed, you might want to update the associated claimant or exposure records in an external system. However, most implementations do not listen for the ClaimContactRemoved event. Instead, handle removals within Event Fired rules for the event ClaimContactRoleRemoved. (Contrast this entity with Contact and ClaimContactRole.) Also refer to the event for ClaimContactContactChanged later in this table.
ClaimContact	ClaimContactContactChanged	ClaimCenter triggers this if either the contactID changes on the claimContact or if the referenced contact changes.
ClaimContactRole	ClaimContactRoleAdded ClaimContactRoleChanged ClaimContactRoleRemoved	Standard events for ClaimContactRole entities. The ClaimContactRoleAdded event triggers after a contact associates with a new role for a claim, policy, exposure, matter, negotiation, or evaluation. For example, this event triggers after some code adds a new exposure and the insured is now the claimant role for that new exposure. This is an important event if you track contact changes. For many implementations, roles are more important to track than contacts. Similarly, the ClaimContactRoleRemoved event triggers if a contact no longer has a certain role for the claim, policy, exposure, matter, negotiation, or evaluation. For example, if the contact is the main contact for the claim, but the main contact changes to a different person.

Entity	Events	Description
CompanyVendor	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
Company	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
Contact	ContactAdded ContactChanged ContactRemoved	Contact entities are abstract entities that exist in the form of contact subtypes, such as Person and Doctor. The ContactAdded, ContactChanged, and ContactRemoved events are sent for all contact subtypes. (Also, contrast Contact with ClaimContact and ClaimContactRole.)
Doctor	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
LawFirm	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
LegalVenue	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
MedicalCareOrg	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
Organization	OrganizationAdded OrganizationChanged OrganizationRemoved	Standard events for Organization entities.
Person	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
VendorVendor	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
Place	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.

#### At What Time Does a Remove-related Event Trigger?

The `EntityNameRemoved` events trigger after either of the following occurs:

- some code deletes an entity from ClaimCenter
- some code marks the entity as *retired*. Retiring means a *logical delete* but leaves the entity record in the database. In other words, the row remains in the database and the `Retired` property changes to indicate that the entity data in that row is inactive. Only some entities in the data model are retrievable. Refer to the *Data Dictionary* for details.

## Triggering Custom Event Names

Business rules can trigger custom events using the `addEvent` method of claim entities and most other entities. This method can also be called from Java code that uses the entity libraries, specifically from Java plugins or from Java classes called from Gosu.

You might choose to create custom events in response to actions within the ClaimCenter application user interface or using data changes originally initiated from the web service APIs. Triggering custom events is useful also if you want to use event-based rules to encapsulate code that logically represents one important action that could generate events. You could handle the event in your Event Fired rules but trigger it from another rule set such as validation, or from PCF pages.

To raise custom events within Gosu, use the `addEvent` method of the relevant object. In other words, call `myEntity.addEvent(eventname)`. In this case, the event name is whatever `String` you pass as the parameter. The entity whose `addEvent` method you call is the root object for the event.

You might also want to trigger custom events during message acknowledgement. If acknowledging the message from a messaging plugin, use the entity `addEvent` method described earlier.

### Custom Events From SOAP Acknowledgements

Integrations that use SOAP API to acknowledge a message can use a separate mechanism for triggering custom events as part of the message acknowledgement. This is a common approach for some financials events.

First, your web service API client code in Java creates a new SOAP entity `Acknowledgement`. Next, call its `setCustomEvents` method to store a list of custom events to trigger as part of the acknowledgement:

```
String[] myCustomEvents = {"ExternalABC", "ExternalDEF", "ExternalGHI"};
myAcknowledgement.setCustomEvents(myCustomEvents);
```

Then, submit the acknowledgement using the SOAP APIs:

```
messagingToolsAPI.acknowledgeMessage(myAcknowledgement);
```

### How Custom Events Affect Pre-Update and Validation

Be aware that pre-update and validation rules do not run solely because of a triggered event. An entity's pre-update and validation rules run only if actual entity data changes. In cases where triggered events do not correspond to entities with modified properties, the event firing alone does not trigger pre-update and validation rules. This does not affect most events, since almost all events correspond to entity data changes.

However, for the `ClaimResync` event (triggered from a claim resync from the user interface), no entity data inherently changes due to this event, so this difference affects resync handling. This also affects any other custom event firing through the `addEvent` entity method. If you also use `ContactManager`, this also affects `ContactManager`'s entity validation for events such as the `ResyncABCContact` event. If you require the pre-update and/or validation rules to run as part of custom events, you must modify some property on the entity or those rule sets do not run.

## Events for Contact Changes

The `Contact` object gets special handling because it is so widely used and in very different ways by different destinations. Some external systems have a relational model for contacts – a separate contact table or database, to which other entities reference. Other systems use a hierarchical model where the contact information is simply part of the claim record. Typically in these systems, contact information appears throughout the claim data model and linked information.

Accordingly, ClaimCenter reports `Contact` changes in two ways. A system that uses a relational model for contacts only needs to know if a contact itself changes. It can then update its contact record, and all references to the contact point to the new information. In ClaimCenter, there are actually two levels at which contact information exists.

First, a single copy of a contact is on each claim. If the contact has multiple roles on the claim, the contact has a single record. For example, one contact might have more than one of the following roles: insured, witness, claimant, alternative contact for another claimant. This single `Contact` entity links to the claim by the claim contact (`ClaimContact`) entity. The claim contact entity contains multiple roles describing the different roles, which are `ClaimContactRole` entities.

Second, a contact may have a master record in an external address book (typically Guidewire `ContactManager`) shared across multiple claims (and even multiple systems).

An external system might want to know:

- if the master record changed
- if the contact snapshot associated with a single claim changed

- if the roles played by the contact within the claim changed.

Each of these situations has a corresponding event in ClaimCenter.

A non-relational system needs additional context. It need to know not only what information on the contact changed, but what objects see the contact so that it can make updates to those objects internally. For example, suppose the same contact is a claimant on multiple exposures and the claimant address exists on each exposure record in the external system. In this case, a single change to a **Contact** entity in ClaimCenter requires updates to multiple exposure records in the external system.

Register only for the events that are appropriate for how each destination system works. You must understand how ClaimCenter and your external systems handle contact information.

The following examples show how ClaimCenter treats some contact changes:

- An adjuster adds a new witness to an existing exposure. She makes a new contact record for the witness. This triggers a **ContactAdded** event for the new person and a **ClaimContactAdded** event for the association of that person with the claim in the role of witness.
- The adjuster later realizes the witness was also a passenger, a custom role you might define. This triggers a **ClaimContactRoleAdded** event.
- The adjuster then updates the contact record for this person. This triggers a **ContactChanged** event for the shared contact record. This triggers a **ClaimContactContactChanged** event to alert you that the contact record changed for a person associated with a particular claim.
- The adjuster decides to promote this person to the master address book in case the person relates to future claims. This triggers an **ABContactAdded** event.
- If it turns out the passenger is not a witness and you remove this role, ClaimCenter triggers the **ClaimContactRoleRemoved** event.
- If a contact is no longer the active person playing that role and the role is inactive, this raises a **ClaimContactRoleChanged** event. For example, this happens if the person was the primary doctor but the claimant switches doctors.
- If you change a person's information in the user interface and indicate changes in the central address book, the system sends two events. ClaimCenter triggers both **ContactChanged** and **ClaimContactContactChanged** events.

These examples lead to some conclusions:

- Contact events are usually not useful. Multiple claims never share a contact. If you want to know that a contact on a claim changed, listen for the more specific **ClaimContactContactChanged** event. This event tells you the claim that it occurs on and which contact changed.
- There are only a few cases where an object directly links directly to a contact rather than through the **ClaimContact** mechanism. For example, lienholders on a vehicle or property link directly link to the contact, so these are the only places where a contact change would not trigger a **ClaimContactContactChanged** event. Otherwise, do not bother listening for **ContactChanged** events.
- For the most part, concentrate on checking **ClaimContact** and **ClaimContactRole** events for whoever is the claimant, insured, or lawyer. Check for **ClaimContactChanged** events for changes to the person's information or company's information.

Further notes:

- If a claim adds a **ClaimContact**, ClaimCenter only triggers the **ClaimContactAdded** event. There are no separate events triggered for each role added as a consequence of adding the **ClaimContact**. This means that the messages that respond to the **ClaimContactAdded** event must look at all roles for the new **ClaimContact**.
- If some code removes a claim contact from a claim, ClaimCenter triggers the **ClaimContactRoleRemoved** event and then the **ClaimContactRemoved** event. The separate events triggered for each role help determine what roles the contact previously played so that these can be cleared in an external system. If the **ClaimContactRemoved** event triggers, the contact's roles are already gone. The **ClaimContactRoleRemoved**

events provide an opportunity to determine what roles the contact used to have so that you can synchronize them with an external system.

- Policies and related objects (vehicles, properties) link to contacts. ClaimCenter reports `ContactChanged` events if the person's address book information changes. However, it does not report anything else for these references because it does not report changes to these objects anyway.
- The *claimant* information links to a contact in notes, documents, activities, evaluations, negotiations, and checks. ClaimCenter does not report changes to top-level objects such as these if the contact record changes. However, it does report a change if these objects relate to a *different* claimant, as it would for any other property that changes on the top-level object.
- Checks reference contacts using the `cc_checkPayee` join table. ClaimCenter does not report changes to a contact's information as a change to the check. This is because the Pay To line on the check derives from the contact but ClaimCenter saves this separately at the time it creates the check.

## Events for Special Subobjects

For changes to some subobjects that are really just extensions of a parent object, ClaimCenter triggers a “changed” event on the parent object. These are:

Subobject type	Can be part of this object
Addresses	claims (for example, loss location), contacts
Vehicles	claims, vehicle damage exposures, and/or a policy
Property	property claims, some exposure types, and/or a policy
Employment Data	workers' compensation claims
Lost Wages Benefits	lost wages exposures
Endorsements	a policy
Stat codes	a policy

## Ordering Events

ClaimCenter generates the events in the order registered by the destination. For each event name in the list, if one or more corresponding object changes occurred in the transaction, ClaimCenter generates an event for each distinct object change. For example, let's assume the registered event list looked like this:

- `ClaimAdded`
- `ExposureAdded`
- `ClaimChanged`
- `ExposureChanged`

If you create an exposure and update two of its exposures in one operation, ClaimCenter generates three events:

- `ExposureAdded`
- `ExposureChanged`
- `ExposureChanged`

## No Events from Import Tool

The web services interface `IImportToolsAPI` and the corresponding `import_tools` command line tool are a generic mechanism for loading system data or sample data into the system. Events do not trigger in response to data added or updated using this interface. Be very careful about using this interface for loading important business data where events might be expected for integration purposes. You must use some other system to ensure your external systems are up to date with this newly-loaded data.

## Generating New Messages in Event Fired Rules

Each time a system event triggers a messaging event, ClaimCenter calls the Event Fired rule set. The application calls this rule set once for each event/destination pair for destinations that are interested in this event. Remember that destinations signal which events they care about in the Messaging editor in Studio, which specifies your messaging plugins by name. The *plugin name* is the name for which Studio prompts you when you register a plugin in the Plugins Editor in Studio. Your Event Fired rules must decide what to do in response to the event. Most importantly, decide whether you want to create a message in response to the event. Because message creation impacts user response times, avoid unnecessarily large or complex messages if possible.

The most important object your Event Fired rules use is a *message context object*, which you can access using the `messageContext` variable. This object contains information such as the event name and destination ID. Typically your rule set generates one or more messages, although the logic can omit creating messages as appropriate. The following sections explain how to use business rules to analyze the event and generate messages.

---

**WARNING** Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309.

---

Studio includes a tool that helps you export business data entities (and other types like Gosu classes) to XML. You can select which properties are required or optional for each integration point. You can export an XSD to describe the data interchange format you selected. Then, you can edit your Event Fired rules to generate a payload for the entity that conforms to your custom XSD. For more details, see “Creating XML Payloads Using Guidewire XML (GX) Models” on page 333.

### Rule Set Structure

If you look at the sample Event Fired rule set, you can see a suggested hierarchy for your rules. The top level creates a different branch of the tree for each destination. You can determine which destination this event applies to by using the `messageContext` variable accessible from Event Fired rule sets. For example, to check the destination ID number, use code like the following:

```
// If this is for destination #1  
messageContext.DestID == 1
```

At the next level in the rules hierarchy, it determines for what root object an event triggered:

```
messageContext.Root typeis Claim // If the root object is a Claim...
```

Finally, at the third level there is a rule for handling each event of interest:

```
messageContext.EventName == "ClaimChanged"
```

In this way, it is easy to organize the rules and keep the logic for handling any single event separate. Of course, if you have shared logic that would be useful to processing multiple events, create a Gosu class that encapsulates that logic. Your messaging code can call the shared logic from each rule that needs it.

### Simple Message Payload

There are multiple steps in creating a message. First, you must convert (*cast*) the root object of the event to a variable of known type:

```
var claim = messageContext.Root as Claim
```

Once the Rule Engine recognizes the root object as a `Claim`, it allows you to access properties and methods on the claim to parameterize the payload of your message.

Next, create a message with a `String` payload:

```
var msg = messageContext.createMessage("The claim number is " + claim.ClaimNumber +  
" and event name is " + messageContext.EventName)
```

## Message Payloads and Setting Message Root or Primary Entities

If you want to use the safe ordering feature of ClaimCenter, you may need additional lines of code, depending on what the root object of the event is.

For more information about these concepts, see:

- “Root Object” on page 301
- “Primary Entity and Primary Object” on page 301
- “Setting a Message Root Object or Primary Object” on page 332

## Multiple Messages for One Event

The Event Fired rule set runs once for each event/destination pair. Therefore, if you need to send multiple messages, create multiple messages in the desired order in your Event Fired rules. For example:

```
var msg1 = messageContext.createMessage("Message 1 for claim " + claim.PublicID +
    " and event name " + messageContext.EventName)
var msg2 = messageContext.createMessage("Message 2 for claim " + claim.PublicID +
    " and event name " + messageContext.EventName)
```

You can also use loops or queries as needed. For example, suppose that if a claim-related event occurs, you want to send a message for the claim and then a message for each note on the claim. The rule might look like the following:

```
var claim = messageContext.Root as Claim
var msg = messageContext.createMessage("message for claim with public ID " + claim.PublicID)

for (note in claim.Notes) {
    msg = messageContext.createMessage(note.Body)
}
```

This creates one message for the claim and also one message for *each* note on the claim.

If you create multiple messages for one event like this, you can share information easily across all of the messages. For example, you could determine the username of the person who made the change, store that in a variable, and then include it in the message payload for all messages.

Remember that if multiple destinations requested notification for a specific event name, your Event Fired rule set runs once for each destination, varying only in the `messageContext.DestID`.

You might need to share information across multiple runs of the Event Fired rule set for the same event or different events. If so, see “Saving Intermediate Values Across Rule Set Executions” on page 331.

## Determining What Changed

In addition to normal access to the `messageContext`'s root object, there is a way to find out what has changed. Your Gosu business rule logic can determine which user made the change, the timestamp, and the original value of changed properties. This information is available only at the time you originally generate the message (*early binding*).

**Note:** You cannot use the `messageContext` object during processing of late bound properties, which are properties that employ *late binding* immediately before sending. For more information about late binding, see “Late Binding Data in Your Payload” on page 339.

At the beginning of your code, use `isFieldChanged` method test whether the property changed. If the field changed (and only if the property changed), call the `getOriginalValue` function to get the original value of that property. To get the new (changed) value, simply access the property directly on an entity. The new value has not yet been committed to the database. There are additional functions similar to `isFieldChanged` and `getOriginalValue` that are useful for array properties and other situations. Refer to “Determining What Data Changed in a Bundle” on page 338 in the *Gosu Reference Guide* for the complete list.

For example, the following Event Fired rule code checks if a desired property changed, and checks its original value also:

```
Var usr = User.util.getCurrentUser() as User  
Var msg = "Current user is " + usr.Credential.UserName + ".  
msg = msg + " current loss type value is " + claim.LossType  
if (claim.isFieldChanged("LossType")) {  
    msg = msg + " old value is " + (claim.getOriginalValue("LossType") as LossType).Code  
}
```

For a complete list of Gosu methods related to finding what data changed, see “Determining What Data Changed in a Bundle” on page 338 in the *Gosu Reference Guide*.

### Rule Sets Must Never Call Message Methods for ACK, Error, or Skip

From within rule sets, you must never call any message acknowledgment or skipping methods such as the Message methods `reportAck`, `reportError`, or `skip`. Use those Message methods only within messaging plugins. This prohibition also applies to Event Fired rules.

### Saving Intermediate Values Across Rule Set Executions

A single action in the user interface can generate multiple events that share some of the same information. Imagine that you do some calculation to determine the user’s ID in the destination system and want to send this in all messages. You cannot save that in a variable in a rule and use it in another rule. The built-in scope of variables within the rule engine is a single rule. You cannot use the information later if the rule set runs again for another event caused by the same user interface action.

ClaimCenter provides a solution to this problem by providing a `HashMap` you can use across all rule set executions for the same action that triggered the system event. In other words, the `HashMap` is available for all Event Fired rules executing in a single database transaction triggered by the same system event.

Let us consider an example. Suppose that in a single action, an activity completes and it creates a note. This causes two different events and hence two separate executions of the `EventFired` rule set. As ClaimCenter executes rules for completing the activity, your rule logic could save the subject of the activity by adding it to the *temporary map* using the `SessionMarker.addToTempMap` method. Later, if the rule set executes for the new note, your code checks if the subject is in the `HashMap`. If it is in the map, your code adds the subject of the activity to the message for the note.

Code to save the activity’s information would look like the following:

```
var session = messageContext.SessionMarker // get the sessionmarker  
var act = messageContext.Root as Activity // get the activity  
  
// Store the subject in the "temporary map" for later retrieval!  
session.addToTempMap("related_activity_subject", act.Subject)
```

Later, to retrieve stored information from the `HashMap`, your code would look like the following:

```
var session = messageContext.SessionMarker // get the sessionmarker  
  
// Get the subject line from the "temporary map" stored earlier!  
var subject = session.getFromTempMap("related_activity_subject") as String
```

### Creating a Payload Using Gosu Templates

You can use Gosu code in business rules to generate Gosu strings using concatenation to design *message payloads*, which are the text body of a message. Generating your message payloads directly in Gosu offers more control over the logic flow for what messages you need and for using shared logic in Gosu classes.

However, sometimes it is simpler to use a text-based template to generate the message payload text. This is particularly true if the template contains far more static content than code to generate content. Also, templates are easier to write than constructing a long string using concatenation with linefeed characters. Particularly for long

templates, templates expose static message content in simple text files. People who might not be trained in Guidewire Studio or Gosu coding can easily edit these files.

You can use Gosu templates from within business rules to create some or all of your message payload.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. Your fully-qualified name of the template is `mycompany.templates.NotifyAdminTemplate`.

Use the following code to generate (run) a template and pass a parameter:

```
var myClaim = messageContext.Root as Claim;
// generate the template and pass a parameter to the template
var x = mycompany.templates.NotifyAdminTemplate.renderToString(myClaim)
// create the message
var msg = messageContext.createMessage("Test my template content: " + x)
```

This code assumes the template supports parameter passing. For example, something like this:

```
<%@ params(myClaimParameter : Claim) %>
The Claim Number is <%= myClaimParameter.ClaimNumber %>
```

There are a couple of steps. First, select the template. Then, you let templates use objects from the template's Gosu context using the `template.addSymbol` method. Finally, you execute the template and get a `String` result you could use as the message payload, or as part of the message payload.

The `addSymbol` method takes the *symbol name* that is available from within the template's Gosu code, an object type, and the actual object to pass to the template. The object type could be any intrinsic type, including ClaimCenter entities such as `Claim` or even a Java class.

For more information about using templates, see “Gosu Templates” on page 347 in the *Gosu Reference Guide*.

## Setting a Message Root Object or Primary Object

From the Gosu environment, the `messageContext.Root` property specifies the *root object* for an event. Typically this same object also the root object for the message generated for that event. Because that is the most common case, by default any new message gets the same root object as the event object root. The message root indicates which object this message is about. For further definition of a root object, see “Root Object” on page 301. Contrast this with the definition of primary object on “Primary Entity and Primary Object” on page 301.

You can override the message root object to be a different object. For example, suppose you added a subobject and caught the related event in your Event Fired rules and added a message. You might want the message root to be the subobject's parent object instead of the default behavior. To override the message root, set the message's `MessageRoot` (not `Root`) property in your Event Fired rules. For example:

```
message.MessageRoot = myObject
```

It is important to note that the *primary object* of a message is different from the *message root*, however. It is actually the primary object of a message that defines the message ordering algorithm. See “Safe Ordering” on page 302 and “Message Ordering and Performance Tuning Details” on page 336.

Unlike the message root, there is not a single property that implements the primary object data for a message. Instead, there are multiple properties on a `Message` object that correspond to each type of data that could be a primary object for that application. As mentioned in “Primary Entity and Primary Object” on page 301, each application can define a default primary entity. Each messaging destination can choose from a small set of primary entities. The properties on the `Message` object for primary entity are strongly typed to the type of the primary entity. In other words, there is a different property on `Message` for each primary entity type.

In ClaimCenter, there are multiple properties on `Message` for the primary entity:

- `message.Claim` – The claim, if any, associated with this `Message` object.
- `message.Contact` – The contact, if any, associated with this `Message` object.

In ClaimCenter, if you set the `message.MessageRoot` property, the following behavior occurs automatically as a side-effect of setting this property from Gosu or Java:

- If the entity is a claim or has a `Claim` property, ClaimCenter automatically sets the `message.Claim` property to set the primary entity.
- If the entity type is `Contact` or has a `Contact` property, ClaimCenter automatically sets the `message.Contact` property to set the primary entity.

You only need to set the primary entity properties manually if the automatic behaviors described in this topic did not set them already. Note that you do not need to set unused primary entity properties to `null`. The only primary entity property used in the `Message` object is the one that matches the primary entity for the destination.

To configure custom behavior of the message ordering system (safe ordering) by manually overriding properties that reference a primary object:

- Set the alternative primary entity in the messaging destination if you are not using an application default primary entity.
- In your Event Fired rules, manually set the primary entity property on the `Message` object that matches the primary entity for that destination. The primary entity for the destination is either the application default primary entity or overridden in the messaging destination configuration. You can set a primary entity property to `null` instead of an object. If the property for the primary entity for that destination is `null`, then ClaimCenter treats the message as a *non-safe-ordered* message. ClaimCenter orders safe-ordered and non-safe-ordered messages differently. For details, see “Message Ordering and Multi-Threaded Sending” on page 334.

---

**WARNING** Be careful with setting message properties that store a reference to a primary object. ClaimCenter uses that information to implement safe ordering of messages by primary object.

---

## Creating XML Payloads Using Guidewire XML (GX) Models

Studio includes a tool that helps you export business data entities (and other types like Gosu classes) to XML. You can select which properties are required or optional for each integration point. You can export an XSD to describe the data interchange format you selected. Then, you can use this model to export XML or import XML in your integrations. For example, your messaging plugins or your Event Fired rules could send XML to external systems. You could also write web services that take XML data its payload from an external system or return XML as its result.

The output XML only includes the properties specified in your custom XSD. It is best to create a custom XSD for each integration. Part of this is to ensure you send only the data you need for each integration point. For example, a check printing system probably needs a smaller subset of object properties than a external legacy financials system might need.

The first step is to create a new XML model in Studio. In Studio, navigate in the resource tree to the package hierarchy in which you want to store your XML model. Next, right-click on the package and from the contextual menu choose **New → Guidewire XML Model**.

For instructions on using the GX modeler, see “The Guidewire XML (GX) Modeler” on page 302 in the *Gosu Reference Guide*.

## Using Java Code to Generate Messages

Business rules, including message-generation rules, can optionally call out to Java modules to generate the message payload string. See “Calling Java from Gosu” on page 119 in the *Gosu Reference Guide*.

## Saving Attributes of the Message

As part of creating a message, you can save a *message code* property within the message to help categorize the types of messages that you send. Optionally, you can use this information to help your messaging plugins know how to handle the message. Alternatively, your destination could report on how many messages of each type were processed by ClaimCenter (for example, for reconciliation).

If you need additional properties on the `Message` entity for messaging-specific data, extend the data model with new properties. Only do this for messaging-specific data.

During the `send` method of your message transport plugin, you could test any of these properties to determine how to handle the message. As you acknowledge the message, you could compare values on these properties to values returned from the remote system to detect possible mismatches.

ClaimCenter also lets you save *entities by name* (saving references to objects) with the message to update ClaimCenter entities as you process acknowledgements. For example, to save a `Note` entity by the name `note1` to update a property on it later, use code similar to the following:

```
msg.putEntityByName("note1", note)
```

For more information about using `putEntityByName`, see “[Reporting Acknowledgements and Errors](#)” on page 340.

These methods are especially helpful to handle special actions in acknowledgements. For example, to update properties on an entity, use these methods to authoritatively find the original entity. These methods work even if public IDs or other properties on the entity change. This approach is particularly useful if public ID values could change between the time Event Fired rules create the message and the time you messaging plugins acknowledge the message. The `getEntityByName` method always returns the correct reference to the original entity.

## Handling Policy Changes on a Claim

ClaimCenter stores a snapshot of policy information for a claim. However, you can enter or edit policy information in ClaimCenter if no policy system integrates with ClaimCenter. In this case, you may need to send changes made to the policy with claim information to an external system. ClaimCenter generates policy and coverage-related events to help you generate messages if the policy information changes.

Multiple claims never share a policy snapshot so the `Policy.claims` array only ever contains one claim.

## Maximum Message Size

In this version of ClaimCenter, messages can contain up to one billion characters.

## Message Ordering and Multi-Threaded Sending

This section explains in detail how ClaimCenter orders messages and in some cases uses multiple threads.

For an overview of safe ordering, see the following topics:

- “Primary Entity and Primary Object” on page 301
- “Safe Ordering” on page 302
- “Setting a Message Root Object or Primary Object” on page 332

The application waits for an acknowledgement before processing the next safe-ordered message for that primary object for each destination. This allows ClaimCenter to send messages as soon as possible and yet prevent prevents errors that might occur if related messages send out of order.

For example, suppose some external system must process an initial new message for a parent object before receiving any messages for subobjects or notes that relate to the parent object. If the external system rejects the

new message for the parent object, it is not safe to send further messages to that system. However, it is likely safe to send messages about unrelated objects, or messages about the same object but to a different destination.

Even if the transport layer guarantees delivery order, it is unsafe for ClaimCenter to send the second message before confirming that the first safe-ordered message succeeds. Doing otherwise could cause difficult error recovery problems.

Safe ordering has large implications for messaging performance. If the send queue contains 10 messages associated with different claims, ClaimCenter can send all these safe-ordered messages immediately. However, if the send queue contains 10 safe-ordered messages for the same claim, ClaimCenter can only send one. ClaimCenter must wait for the message acknowledgement, and then at that point can send the next (only one) message for that claim. Another way of thinking about is that only one message can be in flight for each claim/destination pair.

If you license the ContactManager application for use with ClaimCenter, note that ContactManager also supports safe ordering of messages associated with each ABContact entity. This means that ContactManager only allows one message for each combination of ABContact and destination pair at any given time. For more ContactManager integration information, see “ContactManager Integration Reference” on page 269 in the *Contact Management Guide*.

## Ordering Non-safe-ordered Messages

Some messages are not associated with a primary object such as `Claim` (or `Contact`, if a destination specifies `Contact` as the alternative primary object). These cross-claim messages are *non-safe-ordered messages*.

These messages are sometimes also called *Messages Without Primary*.

See the beginning of this topic for links to topics that define primary entities and safe ordering.

By default, non-safe-ordered messages send as soon as possible, before any queued safe-ordered messages, and send in the order that Event Fired rules generated them. By default ClaimCenter does not wait for acknowledgements for non-safe-ordered messages before sending the next -safe-ordered message.

For example, ClaimCenter sends a new catastrophe code message or a message about a new `User` immediately. ClaimCenter never waits for acknowledgements for other messages before sending this message.

However, if you enable the Strict Mode feature for a destination in the Messaging editor in Studio, ClaimCenter waits for the acknowledgement for every non-safe-ordered message. There are also destination options for multi-threaded sending of non-safe-ordered messages. For details, see “How Destination Settings Affects Ordering” on page 338.

For ClaimCenter, messages for the following events are by default non-safe-ordered messages:

- `Catastrophe` events
- `Group` events
- `User` events

## If Multiple Events Fire, Which Message Sends First?

For each destination, the Event Fired rules run in the order that each destination’s configuration specifies in the **Messaging** editor in Studio. In general, messages send in the order of message creation in Event Fired rules. ClaimCenter runs the Event Fired for one event name before the rules run again for the next listed event name. For messages with both the same event name and same destination, the message order is the order that your Event Fired rules create the messages. For more information about destination setup, see “Message Destination Overview” on page 311.

If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor. See “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 137 in the *Configuration Guide*.

In typical deployments, this means that the event name order in the destination setup is very important. Carefully choose the order of the event names in the destination setup. It is important to remember that changes to the ordering of the event names change the order of the events in Event Fired rules. Such changes can produce radical effects in the behavior of Event Fired rules if they assume a certain event order. For example, typical downstream systems want information about a parent object before information about the child objects.

The event name order in the destination setup is critical. Carefully choose the order of the event names in the destination setup. Be extremely careful about any changes to ordering event names in the destination setup. Changes in the event name order could change message order, and that can force major changes in your Event Fired rules logic.

Because ClaimCenter supports *safe-ordering* of messages related to a primary object, the actual ordering algorithm is more complex. Refer to “Message Ordering and Multi-Threaded Sending” on page 334 for details.

## Message Ordering and Performance Tuning Details

ClaimCenter pulls messages from the database (the send queue) in batches of messages on the batch server only, and then waits for a polling interval before querying again.

You can configure the number of messages that the messaging subsystem retrieves in each round of sending. This is called the chunk size. Configure the chunk size in the messaging destination configuration editor in the **Chunk Size** field. By default, this value is set to 100,000, which typically includes all sendable messages currently in the send queue. You can also change the polling interval in messaging destination configuration editor in the **Polling Interval** field.

You must understand the difference between message readers and message sending threads:

- *Message readers* are threads that query the database for messages. Message readers use the *message send order* (typically this is equivalent to creation order). The message reader never loads more than the maximum number of messages in the chunk size setting at one time.
- *Message sender threads* are threads that actually call the messaging plugins to send the messages. A new feature in this release is support for multiple sender threads per messaging destination for safe-ordered messages. You can configure the number of sender threads for safe-ordered messages in the messaging destination configuration editor in the **Number Sender Threads** field.

The messaging ordering and sending architecture works as follows by default (see “How Destination Settings Affects Ordering” on page 338):

1. Each messaging destination has a worker thread that queries the database for messages for that destination only. In other words, *each destination has its own message reader*. Each message reader thread (each worker thread) acts independently.
2. The destination’s message reader queries the database for one batch of messages, where the maximum size is defined by the chunk size. ClaimCenter does not use the chunk size value in the query itself. Instead, the message reader orders the results by send order and stops iterating across the query results after it retrieves that many messages from the database.

ClaimCenter performs two separate queries:

- a. First, ClaimCenter queries for messages associated with a primary object for that destination, also known as *safe-ordered messages*. The maximum number of messages returned for this query is also the chunk size. The database query itself ensures ClaimCenter that no more than one message per primary object exists in this list. If another message for a primary object is sent but unacknowledged, no messages for that object appears in this list. This enforces the rule that no more than one message can be in-flight for each primary object per destination.

The query ensures that no two messages are in flight (sent but not yet acknowledged) for the same destination for the same primary object. So, if there are 100 messages for one primary object, the query only reads and dispatches one of those messages (out of a possible 100) to the destination subthreads.

- b. Next, ClaimCenter queries for all messages not associated with the primary object for that destination, also known as *non-safe-ordered messages*. If the chunk size is not set high enough, the returned set is not the full set of non-claim-specific messages. Be aware the chunk size affects each query. It is not cumulative for safe-ordered and non-safe-ordered messages.

By default, the chunk size is set to 100,000, which is usually sufficient for customers. Be sure not to lower the chunk size too much. Typically there are dependencies between safe-ordered and non-safe-ordered messages on that destination. If the chunk size is too low, the non-safe-ordered message query might not retrieve all the non-safe-ordered messages that your safe-ordered messages rely upon. For example, a claim message might reference a catastrophe. The downstream system might need to receive the message that applies to many primary objects before any other messages reference that information.

3. For each destination, the worker thread iterates through all non-safe-ordered messages for that destination, sending to the messaging plugins. Settings in the Messaging editor in Studio for each destination affect how non-safe-ordered messages are sent. The choices are single thread, multi-thread, or strict mode. Those settings affect how many threads send messages, and how the application handles errors. See “How Destination Settings Affects Ordering” on page 338.

After each worker thread finishes sending non-safe-ordered messages, it creates subthreads to send safe-ordered messages for that destination. Configure the number of threads in the messaging destination configuration editor in the **Number Sender Threads** field. Each worker thread distributes the list of safe-ordered messages to send to the subthreads.

Assigning the number of sender subthreads for a destination by default affects only the safe-ordered messages for that destination. For non-safe-ordered messages (also called Messages Without Primary), the rules depend on destination settings. See “How Destination Settings Affects Ordering” on page 338.

If the message is associated with a primary entity, during messaging operations you can optionally lock the primary entity at the database level. This can reduce some problems in edge cases in which other threads (including worker threads) try to modify objects associated with this same object.

For example, if using the default `Claim` primary entity, the system locks the associated `Claim` during messaging.

ClaimCenter checks the `config.xml` parameter `LockPrimaryEntityDuringMessageHandling`. If it is set to `true`, ClaimCenter locks the primary entity during message send, during all parts of message reply handling, and while marking a message as skipped.

4. The message reader thread waits until all destination threads send all messages in the queues for each subthread.
  5. ClaimCenter checks how much time passed since the beginning of this round of sending (since the beginning of step 2) and sleeps the remainder of the polling interval. Configure the polling interval in the messaging destination configuration editor in the **Polling Interval** field. If the amount of time since the last beginning of the polling interval is `TIME_PASSED` milliseconds, and the polling interval is `POLLING_INTERVAL` milliseconds. If the polling interval since the last query has not elapsed, the reader sleeps for `(POLLING_INTERVAL - TIME_PASSED)` milliseconds. If the time passed is greater than the polling interval, the thread does not sleep before re-querying.

The message reader reads the next batch of messages. Begin this procedure again at step 2.

The polling interval setting critically affects messaging performance. If the value is low, the message reader thread sleeps little time or even suppresses sleeping between rounds of querying the database for more messages.

To illustrate how this works, compare the following situations.

Suppose there are two messaging destinations, and both destinations use `Claim` as the primary entity. Also suppose the send queue contains 10 messages for each destination. For each destination, assume that there is no more than one message for each claim. In other words, for each destination, there are 10 total messages related to 10 different claims:

- Assuming the number of messages does not exceed the chunk size, each destination gets only one message for the claim for that destination from the database. In this case, every message can be sent immediately because each message for that destination is independent because they are for different claims. If the destination's **Number Sender Threads** setting is greater than 1, ClaimCenter distributes all claim-specific messages to multiple subthreads. The length of the destination queue never exceeds the number of messages queried in each round of sending. The message reader waits until all sending is complete before repeating.

In contrast, suppose the messages for one destination includes 10 claim-specific messages for the same claim and 5 non-claim-specific messages:

- Assuming the number of messages does not exceed the chunk size, ClaimCenter reads all 5 non-safe-ordered messages and sends them. However, ClaimCenter only gets one message for the claim for that destination from the database. If the destination's **Number Sender Threads** setting is greater than 1, ClaimCenter distributes all claim-specific messages in multiple threads per destination. In this case there is only message that is sendable. Compared to the previous example, ClaimCenter handles fewer messages for each polling interval.

To improve performance, particularly cases like the second example, change the following settings in the Messaging editor for your destinations:

- Lower the **Polling Interval**. The value is in milliseconds. Experiment with lower values perhaps as low as 1000 (which means 1 second) or even lower. Test any changes to see the real-world effects on your messaging performance. If your performance issues primarily relate to many messages per primary entity per destination, then the polling interval is the most important messaging performance setting.
- Increase the value for **Number Sender Threads**. This permits more worker threads to operate in parallel on the batch server only for sending safe-ordered messages. Again, test any changes to see the real-world effects on your messaging performance. If your performance issues primarily relate to many messages but few messages per primary entity for each destination, then the sending threads number is the most important messaging performance setting.

To get maximum performance from multiple threads, keep your message transport plugin implementation's `send` method as quick as possible, with little variation in duration. The application does not load the next chunk of messages until the application passes all messages in the chunk to the message transport `send` method. If your `send` method sometimes takes a long time, convert your code to asynchronous sending. With asynchronous sending, the `send` method completes quickly and the reply is handled later. See "Message Destination Overview" on page 311.

## Thread-Safe Plugins

You must write all messaging plugin implementation code as *thread-safe code*. This means that you must be extremely careful about static variables and any other shared memory structures that multiple threads might access running the same (or related) code.

For more information, see "Concurrency" on page 369 in the *Gosu Reference Guide*.

You must write your messaging plugin code as thread-safe even if the **Number Sender Threads** setting is set to 1.

## How Destination Settings Affects Ordering

For messages without a primary object (sometimes called non-safe-ordered messages), there are settings in the Messaging editor for each messaging destination that configures how to order them.

The **Message Without Primary** setting has three choices:

- Single thread** – Messages without a primary object send in a single thread, and do not wait for an acknowledgement before proceeding to other messages.
- Multi thread** – Messages without a primary object send in multiple threads, and do not wait for an acknowledgement before proceeding to other messages. The precise order of sending of messages without a primary object is non-deterministic.

- **Strict Mode** – If Strict Mode is enabled, messages without a primary object send in a strict order, and wait for an acknowledgement before proceeding to other messages.

Carefully choose the value for each messaging destination.

---

**WARNING** If you use either **Single thread** or **Multi thread** options, errors for messages without a primary object, for example new/changed catastrophes, do not hold up other messages. This means that errors in such messages may cause errors at the destination if the system is unprepared for this situation. For instance, suppose a new claim links to a new catastrophe. If the new catastrophe's add message fails, the catastrophe is unknown to the destination and may cause an error.

---

If Strict Mode is enabled, if errors occur other messages for that destination stop until an administrator resolves the problem. For example, an administrative user can resync that message. Resyncing the messages allows other messages to send for that primary object for that destination after resynchronizing that primary object. For more information about resynchronizing, see “Resynchronizing Messages for a Primary Object” on page 350.

With the value to **Single thread** or **Multi thread**, which means Strict Mode is disabled:

- Each destination sends non-safe-ordered messages in one or more threads, depending on whether you choose **Single thread** or **Multi thread**. Next, the application sends safe-ordered messages.

**Note:** Optionally, sending safe-ordered messages is multi-threaded. For more, see “Message Ordering and Multi-Threaded Sending” on page 334.

- Non-safe-ordered messages send in order but never wait for acknowledgement. Non-safe-ordered message errors never block other messages.

With Strict Mode on, the behavior is:

- Non-safe-ordered messages require an acknowledgement before sending future next message (of either type) to that destination. This option reduces throughput of non-safe-ordered messages.
- Delay sending safe-ordered messages until all non-safe-ordered messages are sent.
- Errors block all future messages for that destination, independent of the message type.

For each messaging destination, carefully consider the data integrity, error handling, and performance needs for your system before deciding on the value of the Message without Primary option.

## Late Binding Data in Your Payload

In your Event Fired messages, in general it is best to use the current state of entity data to create the message payload. In other words, generate the entire payload when the Event Fired rule set runs. For example, for **ClaimChanged** events, messages typically contain the latest information for the claim as of the time the messaging event triggers. This includes any changes in this database transaction (entity instance additions, removals, or changes). Generally this is the best approach for messaging. If you waited until messaging sending time (rather than creation time), the data might be partially different or even data removed from the database. This could disrupt the series of messages to a downstream system. Downstream systems typically need messages that match with data model changes as they happen. Creating the entire payload at Event Fired time is the standard recommended approach. This is called *early binding*.

However, this prevents *later* changes to an object appearing in an *earlier* message about that object. This is relevant if there is a large delay in sending the message for whatever reason. Sometimes you need the latest possible value on an entity as the message leaves the send queue on its way to the destination. For example, imagine sending a new claim to an external system. As part of the acknowledgement, the external system might send back its ID for the new claim. You can set the *public ID* in ClaimCenter to that external system's ID for the claim.

Continuing this example, suppose the next message send a *reserve* for that claim to the external system. In this message, include the claim's new public ID received from the external system in the acknowledgement. This lets

the external system knows to which claim this reserve belongs. If you exported the original public ID value during the original Event Fired rule, the original message cannot contain the new value. In that case, you cannot tell external system which claim belongs with this reserve.

ClaimCenter solves this problem by permitting *late binding* of properties in the message payload. You can designate certain properties for late binding so you re-calculate values immediately before the messaging transport sends the message.

---

**IMPORTANT** Guidewire recommends exporting all data in the Event Fired rules into a message payload (early-bound) unless you have a specific reason why late binding is critical for that situation.

---

For newly-created entity instances, some customers send the entity instance's *public ID property* as a late bound property. A message acknowledgement or external system (using web service APIs) could change the public ID between creating (submitting) the message and sending it.

For other properties, decide whether early binding or late binding is most appropriate.

At message creation time in Event Fired rules, add your own marker text within the message, for example "<AAAAAA>". Your `MessageRequest` plugin code or `MessageTransport` plugin code can directly find the message root object or primary object and substitute the current value. If ClaimCenter calls your `MessageRequest` plugin, the current value of the property is a late bound value and you can replace the marker with the new value.

For example, a Gosu implementation of the `MessageRequest` plugin interface might do this using the following code in the `beforeSend` method. In this example, the transport assumes the message root object is a `Claim` and replaces the special marker in the payload with the value of extension property `SomeProperty`. This example assumes that the message contains the string "<AAAAAA>" as a special marker in the message text:

```
function beforeSend(m : Message) {
    var c = m.MessageRoot as Claim
    var s = org.apache.commons.lang.StringUtils.replace(m.getPayload(), "<AAAAAA>", c.SomeProperty)
    return s
}
```

#### See also

- "Primary Entity and Primary Object" on page 301
- "Setting a Message Root Object or Primary Object" on page 332

## Reporting Acknowledgements and Errors

ClaimCenter expects to receive acknowledgements back from the destination. In most cases, the destination submits a *positive acknowledgement* to indicate success. However, errors can also occur, and you must tell ClaimCenter about the issue.

### Message Sending Error Behaviors

Sometimes something goes wrong while sending a message. Errors can happen at two different times. Errors can occur during the *send* attempt as ClaimCenter calls the message sync transport plugin's `send` method with the message. For asynchronous replies, errors can also occur in negative acknowledgements.

Error conditions during the destination `send` process:

- **Exceptions during `send()` causes automatic retries** – Sometimes a message transport plugin has a `send` error and expects it to be temporary. To support this common use case, if the message transport plugin throws an exception from its `send` method, ClaimCenter retries after a delay time, and continues to retry multiple times. The delay time is an exponential wait time (*backoff time*) up to a wait limit specified by each destination. For safe-ordered messages, ClaimCenter halts sending messages all messages for that combination of primary object and destination during that retry delay. After the delay reaches the wait limit, the retryable error becomes a non-automatic-retry error.

- **For errors during send() and you do not want automatic retry** – If the destination has an error that the application does not expect to be temporary, do not throw an exception. Throwing an exception triggers automatic retry. Instead, call the message's `reportError` method with no arguments. See “Submitting Ack, Errors, and Duplicates from Messaging Plugins” on page 341.

The destination suspends sending for all messages for that destination until one of the following is true:

- An administrator retries the sending manually, and it succeeds this time.
- An administrator removes the message.
- It is a safe-ordered ClaimCenter message and an administrator resynchronizes the claim.

Errors conditions that occur later:

- **A negative acknowledgement (NAK)** – A destination might get an error reported from the external system (database error, file system error, and delivery failure) and human intervention might be necessary. For safe-ordered messages, ClaimCenter stops sending messages for this combination of primary object and destination until the error clears through the administration console or automated tools.
- **No acknowledgement for a long period** – ClaimCenter does not automatically time out and resend the message because of delays. If the transport layer guarantees delivery, delay is acceptable whereas resending results in message duplicates. External system may not be able to properly detect and handle duplicates.

For safe-ordered messages, ClaimCenter does not send more messages for the combination of primary object and destination until it receives an acknowledgement (ACK) or some sort of error (NAK).

For ClaimCenter administrative tools that monitor and recover from messaging errors, see “The Administration User Interface” on page 355.

## Submitting Ack, Errors, and Duplicates from Messaging Plugins

To submit an acknowledgement or a negative acknowledgement from a messaging plugin implementation class, use the following APIs. Refer to the following table based on the success status, the type of failure, and the location of your code.

Situation	Do this	Description
<b>Report acknowledgement</b>		
Success	<code>message.reportAck()</code>	Submits an ACK for this message, which may permit other messages to be sent. For detailed logic of message ordering, see “Message Ordering and Multi-Threaded Sending” on page 334.
Errors within the send method of your <code>MessageTransport</code> plugin implementation, and the error is presumed temporary.	Throw an exception within the send method, which triggers automatic retries potentially multiple times.	Automatically retries the message potentially multiple times, including the backoff timeout and maximum tries. After the maximum retries, the application ceases to automatically retry it and suspends the messaging destination.  An administrator can retry the message from the <b>Administration</b> tab. Select the message and click <b>Retry</b> . For details of automatic retry, see “Message Sending Error Behaviors” on page 340.
<b>Report error</b>		
Errors in any messaging plugins and no automatic retry is needed	<code>message.reportError()</code>	The no-argument version of the <code>reportError</code> method reports the error and omits automatic retries. An administrator can retry the message from the <b>Administration</b> tab. Select the message and click <b>Retry</b> .  For more information about the retry schedule, see “Reporting Acknowledgements and Errors” on page 340.

Situation	Do this	Description
Errors in any messaging plugins and scheduled retry is needed	<code>message.reportError(date)</code>	Reports the error and schedules a retry at a specific date and time. An administrator can retry the message from the user interface before this date. This is equivalent to using the application user interface in the <b>Administration</b> tab at that specified time, and select the message and click <b>Retry</b> . You can call this method from the <code>MessageTransport</code> or when handling errors in your <code>MessageReply</code> plugin implementation.
Errors in any messaging plugins and scheduled retry is needed	<code>message.reportError(category)</code>	Reports the error and assigns an error category from the <code>ErrorCategory</code> typelist. The <b>Administration</b> tab uses this error category to identify the type of error in the user interface. You can extend this typelist to add your own meaningful values.  In the base configuration of ClaimCenter, this typelist is empty.  You can call this method from the <code>MessageTransport</code> or when handling errors in your <code>MessageReply</code> plugin implementation.
<b>Report duplicate</b>		
Message is a duplicate	<code>msgHist.reportDuplicate()</code>	<p>Reports a duplicate. This is a method on the message history object, not the message object. A message history object is what a message becomes after successful sending. A message history (<code>MessageHistory</code>) object has the same properties as a message (<code>Message</code>) object but has different methods.</p> <p>If your duplicate detection code runs in the <code>MessageTransport</code> plugin (typical only for synchronous sending), use standard database query builder APIs to find the original message. Query the <code>MessageHistory</code> table.</p> <ul style="list-style-type: none"> <li>For asynchronous sending with the <code>MessageReply</code> plugin implementation, The <code>MessageFinder</code> interface has methods that a reply plugin uses to find message history entities. The methods use either the original message ID or the combination of sender reference ID and destination ID:  <code>ID:findHistoryByOriginalMessageID(original MessageId)</code> - find message history entity by original message ID           </li> <li><code>findHistoryBySenderRefID(senderRefID, destinationID)</code> - find message history entity by sender reference ID</li> </ul> <p>After you find the original message in the message history table, report the duplicate message by calling <code>reportDuplicate</code> on the original message. For related information about asynchronous replies, see "Implementing a Message Reply Plugin" on page 346</p>

## Using Web Services to Submit Ack and Errors From External Systems

If you want to acknowledge the message directly from an external system, use the web service method `IMessagingTools.acknowledgeMessage(ack)`. First, create an `Acknowledgement` SOAP object.

If there are no problems with the message (it is successful), pass the object as is.

If you detect errors with the message, set the following properties:

- **Error** – Set the `Acknowledgement.error` property to `true`
- **Retryable** – For all errors other than duplicates, always set the `Acknowledgement.Retryable` property to `true` and `Acknowledgement.Error` to `true`. Set this property to `false` (the default) only if there is no error.
- **Duplicate** – If you detect the message is a duplicate, set the `Duplicate` and `Error` properties to `true`.

### Using Web Services to Retry Messages from External Systems

The web service interface `MessagingToolsAPI` contains methods to retry messages.

Review the documentation for the `MessagingToolsAPI` methods `retryMessage` and `retryRetryableErrorMessages`. The method `retryRetryableErrorMessages` optionally limits retry attempts to a specified destination. You can only use the `MessagingToolsAPI` interface if the server's run mode is set to `multiuser`. Otherwise, all these methods throw an exception.

As part of an acknowledgement, the destination can update properties on related objects. For example, the destination could set the `PublicID` property based on an ID in the external system.

## Tracking a Specific Entity With a Message

If desired, you can track a specific entity at message creation time in your Event Fired rules. You can use this entity in your messaging plugins during sending or while handling message acknowledgements. To attach an entity to a message in Event Fired rules, use the `Message` method `putEntityByName`. to attach an entity to this message and associate it with a custom ID called a *name*. Later, as you process an acknowledgement, use the `Message` method `getEntityByName` to find that entity attached to this message.

The `putEntityByName` and `getEntityByName` methods are helpful for handling special actions in an acknowledgements. For example, if you want to update properties on a certain entity, these methods authoritatively find the *original* entity that triggered the event. These methods work even if the entity's public ID or other properties changed. This is particularly useful if the public IDs on some objects changed between the time you create the message and the time the messaging code acknowledges the messaged. In such cases, `getEntityByName` always returns the correct entity.

For example, Event Fired rules could store a reference to an object with the name “abc:expo1”. In the acknowledgement, the destination would set the `publicID` property. For example, set the public ID of object abc:expo1 to the value “abc:123-45-4756:01” to indicate how the external system thinks of this object.

Saving this name is convenient because in some cases, the external system's name for the object in the response is known in advance. You do not need to store the object type and public ID in the message to refer back to the object in the acknowledgement.

## Implementing Messaging Plugins

There are three types of messaging plugins. See the following sections for details:

- “[Implementing a Message Request Plugin](#)” on page 344
- “[Implementing a Message Transport Plugin](#)” on page 345
- “[Implementing a Message Reply Plugin](#)” on page 346

---

**WARNING** Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “[Important Entity Data Restrictions in Messaging Rules and Messaging Plugins](#)” on page 309.

---

## Getting Message Transport Parameters from the Plugin Registry

It may be useful in some cases to get parameters from the plugin registry in Studio. The benefit of setting parameters for messaging transports is that you can separate out variable or environment-specific data from your code in your plugin.

For example, you could use the Plugins editor in Studio for each messaging plugin to specify the following types of data for the transport:

- external system's server name
- external system's port number
- a timeout value

If you want to get parameters from the plugin registry, your messaging plugin must explicitly implement `InitializablePlugin` in the class definition. This interface tells the application that you support the `setParameters` method to get parameters from the plugin registry. Your code gets the parameters as name/value pairs of `String` values in a `java.util.Map` object.

For example, suppose your plugin implementation's first line looks like this:

```
class MyTransport implements MessageTransport {
```

Change it to this:

```
class MyTransport implements MessageTransport, InitializablePlugin {  
    function setParameters(map: java.util.Map) { // this is part of InitializablePlugin  
        // access values in the MAP to get parameters defined in plugin registry in Studio  
        var myValueFromTheMap = map["servername"]  
    }  
}
```

## Implementing a Message Request Plugin

A destination can optionally define a message request (`MessageRequest`) plugin to prepare a `Message` object before a message is sent to the message transport. It may not be necessary to do this step. For example, if textual message payload contains strings or codes that must be translated for a specific remote system. Or perhaps a textual message payload contains simple name/value pairs that must be translated into a XML before sending to the message transport. Or, perhaps you need to set data model extension properties on `Message`.

To prepare a message before sending, implement the `MessageRequest` method `beforeSend`. ClaimCenter calls this method with the message entity (a `Message`).

The main task for this method is to generate a modified payload and return it from the method. Generally speaking, do not modify the payload directly in the `Message` entity. The result from this method passes to the messaging transport plugin to send in its `transformedPayload` parameter. This transformed payload parameter is separate from the `Message` entity that the message transport plugin gets as a parameter.

You can modify properties within the `Message` or within the message's root object such as a `Claim` object as needed. However, as mentioned before, to transform the payload just return a `String` value from the method rather than modifying the `Message`.

You can also use the `MessageRequest` plugin to perform post-sending processing on the message. To perform this type of action, implement the `MessageRequest` method `afterSend`. ClaimCenter calls the method with the message (a `Message` object) as a parameter. ClaimCenter calls this method *immediately* after the transport plugin's `send` method completes. If you implement asynchronous callbacks with a message reply plugin, be sure to understand that the server calls `afterSend` in the same thread executing the plugin's `send` method. However, it might be a separate thread from any asynchronous callback code.

Also, if the `send` method acknowledges the message with any result (successful ACK or an error), then be aware that the ACK does not affect whether the application calls `afterSend`. Assuming no exceptions trigger during calls to `beforeSend` or `send`, ClaimCenter calls the `afterSend` method.

If you want to get parameters from the plugin registry, your messaging plugin must explicitly implement `InitializablePlugin` in the class definition. This interface tells the application that you support the `setParameters` method to get parameters from the plugin registry. Your code gets the parameters as name/value pairs of `String` values in a `java.util.Map` object.

## Implementing a Message Transport Plugin

A destination must define a message transport (`MessageTransport`) plugin to send a `Message` object over some physical or abstract transport. This might involve submitting a message to a message queue, calling a remote web service API, or might implement a complex proprietary protocol specific to some remote system. The message transport plugin is the only required plugin interface for a destination.

To send a message, implement the `send` method, which ClaimCenter calls with the message (a `Message` object). That method has another argument, which is the *transformed payload* if that destination implemented a message request plugin. See “[Implementing a Message Request Plugin](#)” on page 344.

In the message transport plugin’s simplest form, this method does its work synchronously entirely in the `send` method. For example, call a single remote API call such as an outgoing web service request on a legacy computer. Your `send` method optionally can immediately acknowledge the message with the code `message.reportAck(...)`. If there are errors, instead use the message method `reportError`. To report a duplicate message, use `reportDuplicate`, which is a method not on the current message but on the `MessageHistory` entity that represents the original message.

If you must acknowledge the message synchronously, your `send` method may optionally update properties on ClaimCenter objects such as `Claim`. If you desire this, you can get the message root object by getting the property `theMessage.MessageRoot`. Changes to the message and any other modified entities persist to the database after the `send` method completes. Changes also persist after the `MessageRequest` plugin completes work in its `afterSend` method. To visualize how these elements interact, see the diagram in “[Message Destination Overview](#)” on page 311.

If your message transport plugin `send` method does not synchronously acknowledge the message before returning, then this destination must also implement the message reply plugin. The message reply plugin to handle the asynchronous reply. See the “[Implementing a Message Reply Plugin](#)” on page 346 for details.

You must handle the possibility of receiving duplicate message notifications from your external systems. Usually, the receiving system detects duplicate messages by tracking the message ID, and returns an appropriate status message. The plugin code that receives the reply messages can call the `message.reportDuplicate()` method. Depending on the implementation, the code that receives the reply would be either the message transport plugin or the message reply plugin. Your code that detects the duplicate must skip further processing or acknowledgement for that message.

Your implementations of messaging plugins must explicitly implement `InitializablePlugin` in the class definition. This interface tells ClaimCenter that you support the `setParameters` method to get parameters from the plugin registry. Even if you do not need parameters, your plugin implementation must implement `InitializablePlugin` or the application does not initialize your messaging plugin.

Your implementation of this plugin must explicitly implement `InitializablePlugin` in the class definition and then also add a `setParameters` method to get parameters. Implement this interface even if you do not need the parameters from the plugin registry.

The following example in Java demonstrates a basic message transport.

### Example Basic Message Transport For Testing

```
uses java.util.Map;
uses java.plugin;
```

```
class MyTransport implements MessageTransport, InitializablePlugin {  
    function setParameters(map: java.util.Map) { // this is part of InitializablePlugin  
        // access values in the MAP to get parameters defined in plugin registry in Studio  
    }  
    // NEXT, define all your other methods required by the MAIN interface you are implementing...  
    function suspend() {}  
    function shutdown() {}  
    function setDestinationID(id:int) {}  
    function resume() {}  
    function send(message:entity.Message, transformedPayload:String) {  
        print("=====  
        print(message)  
        message.reportAck()  
    }  
}
```

### More Examples

Several example message transport plugins ship with ClaimCenter in the product. Refer to these directories:

`ClaimCenter/java-api/examples/src/examples/plugins/messaging`  
`ClaimCenter/java-api/examples/src/com/guidewire/cc/plugin/messaging`

### Exception Handling

For details of exceptions and handling suspect/shutdown in messaging plugins, see “Error Handling in Messaging Plugins” on page 348.

## Implementing a Message Reply Plugin

A destination can optionally define a message reply (`MessageReply`) plugin to asynchronously acknowledge a `Message`. For instance, this plugin might implement a trigger from an external system that notifies ClaimCenter that the message send succeeded or failed.

ClaimCenter requires a special step in this process to setup the ClaimCenter database transaction information appropriately so that any entity changes commit to the ClaimCenter database. To do this properly, Guidewire provides the types `MessageFinder`, `PluginCallbackHandler`, and inner interface `PluginCallbackHandler.Block`.

A message finder (`MessageFinder`) object is built-in object that returns a `Message` entity instance from its `messageID` or `senderRefID`, using its `findById` or `findBySendRefId` method. During the message reply plugin initialization phase, ClaimCenter calls the message reply plugin `initTools` method with an instance of `MessageFinder`. Save this instance of `MessageFinder` in a private variable within your plugin instance.

A plugin callback handler (`PluginCallbackHandler`) is an object provided to your message reply plugin. The callback handler safely executes the message reply callback code block. The callback handler sets up the callback’s server thread and the database transaction information so entity changes safely and consistently save to the database.

If you want to get parameters from the plugin registry, your messaging plugin implementation must explicitly implement `InitializablePlugin` in its class declaration. The `InitializablePlugin` interface declares that the class supports the `setParameters` method to get parameters from the plugin registry. The application calls the plugin method `setParameters` and passes the parameters as name-value pairs of `String` values in a `java.util.Map` object.

For important information about using message finders to submit errors and duplicates, see “Error Handling in Messaging Plugins” on page 348.

**IMPORTANT** ClaimCenter includes multi-threaded inbound integration APIs that you can optionally use in conjunction with MessageReply plugins. For example, listen for JMS messages or process text files that represent message replies. If you want to use input data other than JMS messages or text files for message replies, write a custom integration by implementing the `InboundIntegrationMessageReply` plugin. `InboundIntegrationMessageReply` is a subinterface of `MessageReply`. See “Multi-threaded Inbound Integration Overview” on page 279 and “Custom Inbound Integrations” on page 289.

## Message Reply Plugin Initialization

During initialization, the message reply plugin must get and store a reference to the message finder (`MessageFinder`) and callback handler (`PluginCallbackHandler`) objects, since it needs to use them later.

Implement the simple `MessageReply` interface and include the `initTools` method, which gets these objects as parameters. Your plugin implementation must store these values in private properties, for example in private properties `_pluginCallbackHandler` and `_messageFinder`.

## Message Reply Callbacks

To acknowledge the message asynchronously, your message reply plugin uses its reference to the `PluginCallbackHandler`. To run your code, you plugin must pass a specially-prepared block of Java code to the `PluginCallbackHandler` method called `execute`.

The `execute` method takes a *message reply callback block*, which is an instance of `PluginCallbackHandler.Block`. The `Block` is a simple private interface to encapsulate the block of code that you write.

The `PluginCallbackHandler` object has an `add` method that marks a ClaimCenter entity as modified so the application commits changes to the database with the message acknowledgement. Call the `add` method and pass an entity instance. This method adds the entity to the correct *bundle*. For more about bundles, see “Bundles and Database Transactions” on page 331 in the *Gosu Reference Guide*. This process ensures that changes to the entity instance commit to the database in the correct database transaction with related changes.

Your code in `PluginCallbackHandler.Block` must perform the following steps:

1. Find a `Message` object. Use methods on the plugin’s `MessageFinder` instance, described earlier in this section.

2. Call methods on the `Message` to signal acknowledgement or errors:

- To report success, call the `reportAck` method on the message.
- To report errors, call the `reportError` method on the message. There are multiple method signatures to accommodate automatic retries and error categories.
- To report duplicates, call the `reportDuplicate` method on the message history (`MessageHistory`) entity instance that corresponds to the original message.

For details, see “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 341.

3. Optional post-processing such as property updates or triggering custom events. If any objects must be modified other than the objects originally attached to the message, the callback block must call

`PluginCallbackHandler.add(entityReference)`. This call to the `add` method ensures all changes on the other objects properly commit to the database as part of that database transaction. You must use the return result of the `add` method and only modify that return result. The return result is a clone of the object that is now writable. Do not continue to hold a reference to the original object you passed to the `add` method. This is equivalent to the `bundle.add(entityReference)` method, which adds an entity instance to a writable bundle. For more information, see “Adding Entity Instances to Bundles” on page 334 in the *Gosu Reference Guide*.

For example, the following example demonstrates creating a `PluginCallbackHandler.Block` object and executing the callback block.

```
...
PluginCallbackHandler.Block block = new PluginCallbackHandler.Block() {
    public void run() throws Throwable {
        Message message = _messageFinder.findById(messageID);
        message.reportAck();
    }
};

\PluginCallbackHandler.execute(block);
...
```

You can call `EntityFactory` as necessary in your callback handler block to create or find entities. The application properly sets up the thread context so that it supports `EntityFactory`. For more information about `EntityFactory`, see “Accessing Entity and Typecode Data in Java” on page 631.

For details of exceptions and handling suspend/shutdown in messaging plugins, see “Error Handling in Messaging Plugins” on page 348.

## Error Handling in Messaging Plugins

For a summary of message error APIs, see “Reporting Acknowledgements and Errors” on page 340.

Several methods on messaging plugins execute in a strict order for a message. Consult the following list to design your messaging code, particularly error-handling code:

1. ClaimCenter selects a message from the send queue on the batch server.
2. If this destination defines a `MessageRequest` plugin, ClaimCenter calls `MessageRequest.beforeSend`. This method uses the text payload in `Message.payload` and transforms it and returns the transformed payload. The message transport method uses this transformed message payload later.
3. If `MessageRequest.beforeSend` made changes to the `Message` entity or other entities, ClaimCenter commits those changes to the database, assuming that method threw no exceptions. The following special rules apply:
  - Committing entity changes is important if your integration code must choose among multiple pooled outgoing messaging queues. If errors occur, to avoid duplicates the message must always resend to the same queue each time. If you require this approach, add a data model extension property to the `Message` entity to store the queue name. In `beforeSend` method, choose a queue and set your extension property to the queue name. Then, your main messaging plugin (`MessageTransport`) uses this property to send the message to the correct queue.
  - If exceptions occur, the application rolls back changes to the database and sets the message to retry later. There is no special exception type that sets the message to retry (an un-retryable error).
  - In all cases, the application never explicitly commits the transformed payload to the database.
4. ClaimCenter calls `MessageTransport.send(message, transformedPayload)`. The `Message` entity can be changed, but the transformed payload parameter is read-only and effectively ephemeral. If you throw exceptions from this method, ClaimCenter triggers automatic retries potentially multiple times. This is the only way to get the automatic retries including the backoff multiplier and maximum retries detection.
5. If this destination defines a `MessageRequest` plugin, ClaimCenter calls `MessageRequest.afterSend`. This method can change the `Message` if desired.
6. Any changes from `send` and `afterSend` methods (step 4 and step 5) commit if and only if no exceptions occurred yet. Any exceptions roll back changes to the database and set the message to retry. Be aware there is no special exception class that sets the message not to retry.

7. If this destination defines a `MessageReply` plugin, its callback handler code executes separately to handle asynchronous replies. Any changes to the `Message` entity or other entities commit to the database after the code completes, assuming the callback throws no exceptions.

If there are problems with a message, you do not necessarily need to throw an exception in all cases. For example, depending on the business logic of the application, it might be appropriate to *skip* the message and notify an administrator. If you need to resume a destination later, you can do that using the web services APIs.

All Gosu exceptions or Java exceptions during the methods `send`, `beforeSend`, or `afterSend` methods imply retryable errors. However, the distinction between retryable errors and non-retryable errors still exists if submitting errors later in the message's life cycle. For example, you can mark non-retryable errors while acknowledging messages with web services or in asynchronous replies implemented with the `MessageReply` plugin.

### Submitting Errors

If there is an error for the message, call the `reportError` method on the message. There is an optional method signature to support automatic retries. See “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 341.

### Handling Duplicates

Your code must handle the possibility of receiving duplicate messages at the plugin layer or at the external system. Typically the receiving system detects duplicate messages by tracking the message ID and returns an appropriate status message. The plugin code that receives the reply messages can report the duplicate with `message.reportDuplicate()` and skip further processing.

Depending on the implementation, your code that receives the reply messages is either within the `MessageTransport.send()` method or in your `MessageReply` plugin. For more information about acknowledgements and errors, see “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 341.

### Saving the Destination ID for Logging or Errors

Each messaging plugin implementation must have a `setDestinationID` method, which allows the plugin to find out its own destination ID and store it in a private variable. The destination can use the destination ID within code such as:

- Logging code to record the destination ID.
- Exception-handling code that works differently for each destination.
- Other integrations, such as sending destination ID to external systems so that they can suspend/resume the destination if necessary.

### Handling Messaging Destination Suspend, Resume, Shutdown

The standard messaging plugins have methods that perform special actions for the administrative commands for destination suspend, destination resume, and before messaging system shutdown.

Typically, suspend and resume is the result of an administrator using the **Administration** tab in the user interface to suspend or resume a messaging destination. Alternatively, suspend and resume could be the result of a web service API call to suspend or resume destinations.

Messaging shut down occurs during server shutdown or if the configuration is about to be reread. After message sending resumes again, ClaimCenter reuses the same *instance* of the plugin. ClaimCenter does not destroy the existing messaging plugin instance (or recreate it) as part of shutdown.

To trap these actions, implement the `suspend`, `shutdown`, and `resume` methods of your messaging plugin.

You can implement these methods just for logging and notification, if nothing else. For example, a message transport plugin's `suspend` or `shutdown` methods could log the action and send notifications as appropriate. For example:

```
public void suspend() {  
    ...  
  
    if(_logger.isDebugEnabled()) {  
        _logger.debug("Suspending message transport plugin.")  
    }  
  
    MyEmailHelperClass.sendEmail(".....")  
}
```

During the `suspend`, `shutdown`, and `resume` methods of the plugin, the plugin must not call any ClaimCenter `IMessageToolsAPI` web service APIs that suspend or resume messaging destinations. Doing so creates circular application logic, so such actions are forbidden.

It is unsupported for a messaging plugin `suspend`, `shutdown`, and `resume` method to use messaging web service APIs that suspend or resume messaging destinations.

## Resynchronizing Messages for a Primary Object

ClaimCenter implementations can use the messaging system to synchronize data with an external system.

In rare cases some messaging integration condition might fail, such as failure to enforce an external validation requirement properly. If this happens, an external system might process one or more ClaimCenter messages incorrectly or incompletely. If the destination detects the problem, the external system returns an error. The error must be fixed or there may be synchronization errors with the external system.

However, suppose the administrator fixes the data in ClaimCenter and improves any related code. It still might be that the external system has incorrect or incomplete data. ClaimCenter provides a programming hook called a *resync event* (a resynchronization event) to recover from such messaging failures.

To trigger this manually, an administrator navigates to the Administration tab, views any unsent messages, clicks on a row, and clicks **Resync**.

The resync can be triggered from the administration user interface or programmatically using the `MessagingToolsAPI` web service method `resyncClaim`.

As a result of a resync request, ClaimCenter triggers the resync event (`ClaimResync`). Configure your messaging destination to listen for this event. Then, implement Event Fired business rules that handle that event.

ClaimCenter supports resynchronizing a Contact from the user interface. However, in the default configuration, ClaimCenter does not support resynchronizing a contact from web services.

Afterwards, ClaimCenter marks all messages that were pending as of the resync as *skipped*. You must implement Guidewire Studio rules that examine the and generate necessary messages. You must bring the external system into sync with the current state of the primary object related to those messages.

Design your Gosu resync Event Fired rules to how your particular external systems recover from such errors.

There are two different basic approaches for generating the resync messages.

In the first approach, your Gosu rules traverse all claim data and generate messages for the entire primary object and its subobjects that might be out-of-sync with the external system.

Depending on how your external system works, it might be sufficient to overwrite the external system's claim with the ClaimCenter version of this data. In this case, resend the entire series of messages. To help the external system track its synchronization state, it may be necessary to add custom extension properties to various objects with the synchronization state. If you can somehow determine that you only need to resend a subset of messages, only send that minimal amount of information. However, one of the benefits of resync is the opportunity to send

all information might conceivably be out of sync. Think carefully about how much data is appropriate to send to the external system during resync.

Your Event Fired rules that handle the resync can examine the failed message and all queued and unsent messages for the claim for a specific destination. Your rules use that information to determine which messages to recreate. Instead of examining the entire claim's history you could consider only the failed and unsent messages. Because a message with an error prevents sending subsequent messages for that claim, there may be many unsent pending messages. To help with this process, ClaimCenter includes properties and methods in the rules context on `messageContext` and `Message`.

From within your Event Fired rules, your Gosu code can access the `messageContext` object. It contains information to help you copy pending `Message` objects. To get the list of pending messages from a rule that handles the resync event, use the read-only property `messageContext.PendingMessages`. That property returns an array of pending messages. After your code runs, the application skips all these original pending messages. This means that the application permanently removes the messages from the send queue after the resync event rules complete. If there are no pending messages at resync time, this array has length zero.

**WARNING** If you create new messages, the new messages send in creation order independent of the order of original messages. This might be a different order than the original messages. Think carefully about how this may or not affect edge cases in the external system.

There are various properties within any message you can get in pending messages or set in new messages:

- `payload` - A string containing the text-based message body of the message.
- `user` - The user who created the message. If you create the message without cloning the old message, the user by default is the user who triggered the resync. If you create the message by cloning a pending message, the new message inherits the original user who creates the original message. In either case, you can choose to set the `user` property to override the default behavior. However, in general Guidewire recommends setting the `user` to the original user. For financial transactions, set the `user` to the user who created the transaction.

There are also read-only properties in pending messages returned from `messageContext.PendingMessages`:

- `EventName` - A string that contains the event that triggered this message. For example, "ClaimAdded".
- `Status` - The message status, as an enumeration. See the following table for values and their meanings.
- `ErrorDescription` - A string that contains the description of errors, if any. This may or may not be present. This is set within a negative acknowledgement (NAK).
- `SenderRefID` - A sender reference ID set by the destination to uniquely identify the message. Your destination can optionally set the `message.senderRefID` field in any of your messaging plugins during original sending of the message. Only the *first* pending message has this value set due to *safe ordering*. You only need to use the sender reference ID if it is useful for that external system. The `SenderRefID` property is read-only from resync rules. This value is `null` unless this message is the first pending message and it was already sent (or pending send) and it did not yet successfully send. As long as the `message.status` property does not indicate that it is *pending send*, the message could have the sender reference ID property populated by the destination.

The following table describes the message status values and the contexts they can appear for `Message` and `MessageHistory` objects:

Message status value	Meaning	Valid in Message	Valid in MessageHistory	Can appear during resync
1	Pending send	●		●
2	Pending ACK	●		●
3	Legacy non-retryable error. Provided for legacy use and is no longer used.			
4	Retryable error	●		●

Message status value	Meaning	Valid in Message	Valid in MessageHistory	Can appear during resync
10	Acknowledged		●	
11	Error cleared (and skipped)		●	
12	Error retried. This is the original message as represented as a MessageHistory object. Another message in the Message table represents the clone of this message.		●	
13	Skipped		●	

## Cloning New Messages From Pending Messages

As mentioned earlier, during resync you can clone a new message from a pending message that you got from `messageContext.PendingMessages`. To clone a new a new message from the old message, pass the old message as a parameter to the `createMessage` method:

```
messageContext.createMessage(message)
```

This alternative method signature (in contrast to passing a `String`) is an API to copy a message into a new message and returns the new message. If desired, modify the new message's properties within your resync rules. All new messages (whether standard or cloned) submit together to the send queue as part of one database transaction after the resync rules complete.

The cloned message is identical to the original message, with the following exceptions:

- The new message has a different message ID.
- The new message has status of pending send (`status = PENDING_SEND`).
- The new message has cleared properties for ACK count and code (`ackCount = 0; ackCode = null`).
- The new message has cleared property for retry count (`retryCount = 0`).
- The new message has cleared property for sender reference ID (`senderRefID = null`).
- The new message has cleared property for error description (`errorDescription = null`).

ClaimCenter marks all pending messages as skipped (no longer queued) after the resync rules complete. Because of this, resync rules must either send new messages that include that information, or manually clone new messages from pending messages, as discussed earlier.

---

**IMPORTANT** All pending messages skip after the Event Fired rules for the resync event complete. You must create equivalent new messages for all pending messages.

---

## Resync and ClaimCenter Financials

Your resync code must handle resending any unprocessed financial transactions, taking care not to resend any successfully processed financial transactions.

---

**WARNING** Be careful not to resend processed financial transactions.

---

## How Resync Affects Pre-Update and Validation

Be aware that pre-update and validation rule sets do not run solely because of a triggered event. A claim's pre-update and validation rules run only if actual entity data changes. In cases where triggered events do not

correspond to already-changed entities, the event firing alone does not trigger claim pre-update and validation rules.

This does not affect most events because almost all events correspond to entity data changes. However, for the events related to resync, no entity data inherently changes due to this event.

This also can affect other custom event firing through the `addEvent` entity method.

## Resync in ContactManager

If you license Guidewire ContactManager for use with ClaimCenter, be aware that ContactManager supports `resync` features for the `ABContact` entity.

To detect resynchronization of an `ABContact` entity, set your destination to listen for the `ABContactResync` event.

Your Event Fired rules can detect that event firing and then resend any important messages. Your rules generate messages to external systems for this entity that synchronize ContactManager with the external system. For more information, see “*ContactManager Messaging Events by Entity*” on page 280 in the *Contact Management Guide*.

## Message Payload Mapping Utility for Java Plugins

A messaging plugin may need to convert items in the payload of a message before sending it on to the final destination. A common reason for this is mapping typecodes. For many properties governed by typelists, the typecode might have the same meaning in both systems. However, this does not work for all situations. Instead, you might need to map codes from one system to another. For example, convert code A1 in ClaimCenter to the code XYZ for the external system.

If you implement your plugin in Java, you can use a utility class included in the ClaimCenter Java API libraries that map the message payload using text substitution. This class scans a message payload to find any strings surrounded by delimiters that you define and then substitutes a new value. The class is `com.guidewire.util.StringSubstitution`. Refer to the *Java API Documentation* for reference.

### To use the String substitution Java class from Java plugin code

1. Choose start and end delimiters for the text to replace. For example, the 2-character string “\*\*” could be used as the start delimiter and end delimiter.
2. Put these delimiters around the Gosu template text that needs to be mapped and replaced. For example, in a Gosu template you might have “Injury=\*\*exposure.InjuryCode\*\*”. This might generate text such as “Injury=\*\*A1\*\*” in the message payload.
3. Implement a class that implements the inner interface called `StringSubstitution.Exchanger`. This exchanger class might use its own look-up table or look in a properties file and substitute new values. The Exchanger interface has one method called `exchange` that translates the token. This method takes a `String` object (the token) and translates it and returns a new `String` object.
4. Instantiate your class that implements Exchanger, and then instantiate the `StringSubstitution` class with the constructor containing your delimiters and your Exchanger instance.

```
MyExchanger myExchangerInstance = new MyExchanger()  
StringSubstitution mySub = new StringSubstitution("/**", "**", myExchangerInstance)
```

5. Call the `substitute` method on the `StringSubstitution` instance to convert the message payload string.

Refer to the *Java API Documentation* for more API details of this class.

# Monitoring Messages and Handling Errors

ClaimCenter provides tools for handling errors that occur with sending messages:

- Automatic retries of sending errors
- Ability for the destination to request retrying or skipping messages in error
- User interface screens for viewing and taking action on errors
- Web service APIs for taking action on errors
- A command line tool for taking action on errors

## Error Handling Concepts

Before describing the tools, it is useful to think about the kinds of errors that can occur and the actions that can be taken on these errors:

- **Pending send** – The message has not been sent yet:
  - If the message is related to a claim, this is a safe-ordered message. The messaging destination may be waiting for an acknowledgement on the previous message for that same claim.
  - The destination may be *suspended*, which means it is not processing messages
  - The destination may be simply not fast enough to keep up with how quickly the application generates messages. ClaimCenter can generate messages very quickly. For more information about how ClaimCenter retrieves messages from the send queue, see “Message Ordering and Multi-Threaded Sending” on page 334.
- **Errors during the Send method** – ClaimCenter attempted to send the message but the destination threw an exception. If the exception was *retryable*, ClaimCenter automatically attempts to send the message again some number of times before turning it into a failure. If it is a *failure*, ClaimCenter suspends the destination automatically until an administrator restarts it. Failures include a retryable send error reaching its retry limit, or unexpected exceptions during the `send` method.

The destination also resumes (un-suspends) if the administrator removes the message or resynchronizes (resyncs) the message’s primary object.

- **In-flight** – ClaimCenter waits for an acknowledgement for message it sent. If errors occur such that the external system does not receive or properly acknowledge the message, ClaimCenter waits indefinitely. If the message has a related primary object for that destination, it is *safe-ordered*. This type of error blocks sending other messages for that primary object for that destination. For more information, see “Safe Ordering” on page 302.

In this case, you can intervene to process the message (*skip* the unfinished message) or retry the message. Be very careful about issuing retry or skip instructions:

- A *retry* could cause the destination to receive a message twice.
- A *skip* could cause the destination to never get the intended information.

In general, you must understand the actual status of the destination to make an informed decision about which correction to make.

- **Error** – The destination indicates that the message did not process successfully. Again, the error blocks sending subsequent messages. In some cases, the error message indicates that the error condition may be temporary and the error is retryable. In other cases, the message indicates that the message itself is in error (for example, bad data) and resending does not work. In either case, ClaimCenter does not automatically try to send again.
- **Positively Acknowledged (ACK)** – The message successfully processed. These messages stay in the system until an administrator purges them. However, since the number of messages is likely to be very large, Guidewire recommends that you purge completed messages on a periodic basis.

- **Negatively Acknowledged (NAK)** – Message sending for that message failed at the external system (not a network error) either because the message had an error or was a duplicate.

If ClaimCenter retries a failed message, it marks the original message as failed/retried and creates a new copy of that message (with a new message ID) to send. The assumption behind this behavior is that a destination tracks messages received and does not accept duplicate messages. To do this, the retry must have a new message ID. If ClaimCenter retries an in-flight message because it never got an ACK, then it resends the original message with the same ID. If the destination never got the message, then there is no problem with duplicate message IDs. If the destination received the message but ClaimCenter never got an acknowledgement, then this prevents processing the message twice. The destination can send back an error (mark it as a duplicate) or send back another acknowledgement.

If ClaimCenter receives an error, it holds up subsequent messages until the error clears. If the destination sends back duplicate errors, you can filter out duplicates and warn the administrator about them. However, you can choose to simply issue a positive acknowledgement back to ClaimCenter.

ClaimCenter could become sufficiently out of sync with an external system such that simply skipping or retrying an individual message is insufficient to get both systems in sync. In such cases, you may need special administrative intervention and problem solving. Review your sever logs to determine the root cause of the problem.

ClaimCenter makes every attempt to avoid this problem. However, it provides a mechanism called resynchronizing (resyncing) to handle this case. All related pending and failed messages drop and resend. For more information, see “Resynchronizing Messages for a Primary Object” on page 350.

If you also use Guidewire ContactManager, note that it also supports resync.

For more information about acknowledgements, see “Reporting Acknowledgements and Errors” on page 340.

## The Administration User Interface

ClaimCenter provides a simple user interface to view the event messaging status for claims. This helps administrators understand what is happening, and might give some insight to integration problems and the source of differences between ClaimCenter and a downstream system.

For example, if you add an exposure to a claim and it does not appear in the external system, you need to know the following information:

- As far as ClaimCenter knows, have all messages been processed? (“Green light”) If the systems are out of sync, then there is a problem in the integration logic, not an error in any specific message.
- Are messages pending, so you simply need to wait for the update to occur. (“Yellow light”)
- Is there an error that needs to be corrected? (“Red light”) If it is a retryable error, you can request a retry. This might make sense if the external system caused the temporary error. For example, perhaps a user in the external system temporarily locked the claim by viewing it on that system’s screen. In many cases, you can simply note the error and report it to an administrator.

This status screen is available from the claim screen by selecting **Claim Actions:Sync Status** from the **Claim** menu.

Administrators have extensive errors across the system. In the Administration section of ClaimCenter, you can select the Event Messages console from the administration page. There are three levels of detail for viewing events and messaging status:

- **Destinations List** – This shows a list of destinations, its sending status (for example, started or suspended), and counts of failed or in-process messages. At this level an administrator can only suspend or resume the entire destination. If suspended, ClaimCenter just holds all newly generated messages until the destination resumes.
- **Destination Status** – This provides a list of claims that have failed messages or messages in-process for a single destination. Different filters can help you find different kinds of problems. You can also search for a particular claim. This opens the detail view for that claim. You can then select one or more claims and indicate what to do with the failed or in-flight messages for each claim. Choose to skip, retry, or resync the claim.

- **Claim Details** – This shows a list of all failed or in-process messages for a claim (for all destinations). You can select messages that are in error or in-process and skip or retry them.

Within the **Destination Status** screen, there is a special row for non-safe-ordered messages (cross-claim messages) if any are in-process or failed. Most ClaimCenter messages relate to claims, so all rows except that one row show problems related to claims. Selecting the special row opens a list of all the non-safe-ordered messages.

From the **Administration** tab, you can force ClaimCenter to stop and restart the messaging sending queue without restarting the entire server. In rare situations, this is useful if you suspect that the sending queue became out of sync with messages waiting in the database. For example, perhaps your a network interruption in the cluster might cause such an issue. If the administrator restarts the messaging system, ClaimCenter shuts down each destination, then restarts the queue process, then reinitializes each destination.

Messaging tool actions such as **suspend**, **resume**, **retry**, and others can trigger from the **Administration** interface and also from the **MessagingToolsAPI** web service. These messaging tools require the server's run level to be **multiuser**.

## Web Services for Handling Messaging Errors

In addition to monitoring and responding to errors with the administration user interface, ClaimCenter provides some other interfaces for dealing with errors. This section describes these tools.

First, ClaimCenter provides a web service called **MessagingToolsAPI**, which lets an external system remotely control the messaging system. See “**Messaging Tools Web Service**” on page 356.

## Messaging Tools Web Service

ClaimCenter provides a web service called **MessagingToolsAPI**, which lets an external system remotely control the messaging system.

These API methods (except for `getClaimMessageStatistics`) are available using the `messaging_tools` command line tool. See “**Messaging Tools Command**” on page 185 in the *System Administration Guide*. Within the administrative environment, the following command shows the syntax of each command:

```
messaging_tools -help
```

### Retry a Message

To retry a message, call the `retryMessage` method. The behavior is the same as in the ClaimCenter user interface.

### Skip a Message

To skip a message, call the `skipMessage` method. The behavior is the same as in the ClaimCenter user interface.

### Retry Messages

To retry all *retryable* messages for a destination, call the `retryRetryableErrorMessages` method.

For example, call this method if the destination was temporarily unavailable and is now back on-line.

You can specify a maximum number of times to retry each message. This maximum prevents messages from retrying forever.

### Purge Completed Messages

To purge completed messages, call the `purgeCompletedMessages` method. This method deletes completed messages in the messaging history table (`MessageHistory`) from the ClaimCenter database for all messages older than the given date.

Since the number and size of messages may be very large, Guidewire recommends you use this method periodically to purge old messages. Purging messages in the message history table prevents the database from growing unnecessarily large.

Always purge completed (inactive) messages before upgrading to a new version of ClaimCenter. Purging completed messages reduces the complexity of your upgrade.

Additionally, periodically use this command to purge old messages to avoid the database from growing unnecessarily.

### Suspend Destination

To suspend a destination, call the `suspendDestination` method. Suspending a destination means that ClaimCenter stops sending messages to a destination.

ClaimCenter suspends the destination so that it can also release any resources such as a message batch file. Use this method to shut down the destination system to halt sending during processing of a daily batch file.

### Resume Destination

To resume a destination, call the `resumeDestination` method. Resuming a destination means that ClaimCenter starts trying to send messages to the destination again.

If a previous suspend action released any resources, resuming the destination reclaims those resources. For example, the destination might reconnect to a message queue.

### Get Messaging Statistics for a Safe Ordered Object

To get messaging statistics for a safe-ordered object, call the `getMessageStatisticsForSafeOrderedObject` object.

This method returns information that is similar to the user interface for a claim:

- the number of messages failed
- retryable messages
- in-flight messages
- unsent messages

Messaging tool actions such as `suspend`, `resume`, `retry`, and others can be triggered from the Administrator interface and also from the web services `MessagingToolsAPI` interface. These messaging tools can only be used if the server run mode is `multiuser`.

### Change Messaging Destination Configuration Parameters

To change messaging destination configuration parameters on a running server, call the `configureDestination` method. This restarts the destination with the change to the configuration settings. The command waits for the destination to stop for the configured stop time. The method returns nothing,

The arguments are:

- `destID` – The destination ID of the destination to suspend
- `maxretries` – maximum retries
- `initialretryinterval` – initial retry interval
- `retrybackoffmultiplier` – additional retry backoff
- `pollinterval` – how often to poll, from start to start
- `numsenderthreads` – number of sender threads for multi-threaded sends
- `chunksize` – number of messages to read in a chunk
- `timeToWaitInSec` – the number of seconds to wait for the shutdown before forcing it

### Get Configuration Information from a Destination

To get some configuration information from a messaging destination, call the `getConfiguration` method. This information is read from files on disk during server startup, however can be modified by web services and command line tools. See earlier in this topic for the `configureDestination` method.

The `getConfiguration` method takes only one argument, the destination ID. The method returns a `ExternalDestinationConfig` object, which contains properties matching the properties in the `getConfiguration` method, such as the polling interval and the chunk size. See the `getConfiguration` method documentation for details of each property.

## Batch Mode Integration

Most integrations using the messaging system are real-time integrations between ClaimCenter and one or more destination systems. However, an external system might only be able to support batch updates. For example, some external server might need regular data from ClaimCenter and retrieves it only at night because it is resource intensive. In this case, ClaimCenter generates system events in real-time but sends updates in one batch to the external system.

There are essentially two approaches to handle batch messaging:

- **Approach 1: suspend a destination, then resume it later** – Using the SOAP API or command line tools, you can suspend sending messages to a destination during most of the day. If it is time to generate the batch file, you can resume (un-suspend) the destination so that ClaimCenter drains its queue of messages to generate the batch file. If there are no more messages (or after some period of time), you can suspend the destination again while you process the batch file or until a pre-defined time. This is a very safe method because it uses the messaging transaction and acknowledgement model to track each message.
- **Approach 2: append messages to a batch file and send all later** – ClaimCenter sends messages to a destination, which appends the messages to a batch file and immediately acknowledges the message. Periodically, the batch file is sent to the destination and processed. For example, after a certain number of messages are in the queue, then the message transport plugin can send them. Or, a separate process on the server can send these batch files. This approach allows you to send more than one message in a single remote call to the external system.

With both approaches, the message acknowledgement would come from the message transport plugin (or the message reply plugin) immediately, and not from the external system. With both approaches, this means that the messaging system built-in retry logic and ordered message sending cannot be used to deal with errors as gracefully as with a real-time integration. If any errors are found after sending an acknowledgement to ClaimCenter, your integration code must deal with these issues outside of ClaimCenter (or by issuing a resync request).

## Included Messaging Transports

### The Built-in Email Transport

ClaimCenter includes a built-in transport that can send standard SMTP emails. See “[Sending Emails](#)” on page 135 in the *Rules Guide* for details of configuration and APIs to send emails.

By default, the `emailMessageTransport` plugin use the *system user* to retrieve a document from the external system. You can chose to retrieve the document on behalf of the user who generated the email message. To do this, set the `a UseMessageCreatorAsUser` property in the `emailMessageTransport` plugin. In Studio, navigate to **Configuration** → **config** → **Plugins** → **registry** → **emailMessageTransport**. In the parameters area of the pane, click the **Add** button. Add the parameter `UseMessageCreatorAsUser` and set it to `true`.

## Enabling the Built-in Console Transport

ClaimCenter includes a built-in console message transport example, which is an extremely simple messaging transport that writes the message text payload to the ClaimCenter console window. Enable this transport in the plugin registry in Studio to debug integration code that creates and sends messages. This is particularly useful if you are working on financials to see if the events look reasonable. For more detail of how to do this, see the *Studio Guide*.

If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor; see “Using the Plugins Registry Editor” on page 113 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 137 in the *Configuration Guide*.

In the plugin editor, use the plugin name `consoleTransport`, the interface name `MessageTransport`, the Java class `examples.plugins.messaging.ConsoleMessageTransport` and the plugin directory `messaging`.

In the messaging editor, use the following settings:

- set the plugin name to "Console Message Logger"
- set transport plugin name to `consoleTransport`
- set initial retry interval set to 100
- set max retries set to 3
- set `retrybackoffmultiplier` to 2
- set event name to "\w\*", which means the destination wants notification of all events

This tells ClaimCenter to send all events to this destination, and trigger Event Fired rules accordingly. write Event Fired rules that create messages for this destination

After redeploying the server, watch the console window for messages.



---

part V

## Financials



# Financials Integration

This topic explains how to integrate external systems with ClaimCenter to track financial transactions, including details of financials events and how to track and control transaction status changes.

This topic includes:

- “Financial Transaction Status and Status Transitions” on page 363
- “Claim Financials Web Service Data Transfer Objects (DTOs)” on page 368
- “Claim Financials Web Services (ClaimFinancialsAPI)” on page 369
- “Check Integration” on page 372
- “Payment Transaction Integration” on page 381
- “Recovery Reserve Transaction Integration” on page 387
- “Recovery Transaction Integration” on page 388
- “Reserve Transaction Integration” on page 390
- “Bulk Invoice Integration” on page 391
- “Deduction Plugins” on page 407
- “Initial Reserve Initialization for Exposures” on page 409
- “Exchange Rate Integration” on page 409

**See also**

- Financials integration depends heavily on the messaging system, which includes events, messages, destinations, messaging plugins, and acknowledgements. See “Messaging and Events” on page 299.

## Financial Transaction Status and Status Transitions

The most important thing for you to understand about financials integration is the status value of a check or transaction. The transaction status value represents the current lifecycle state of a financial transaction or check in ClaimCenter. For example, a check status might have the value `issued` or `pendingvoid`.

The status value tracks and controls the flow of a transaction or check through the ClaimCenter check creation and approval process, and the transaction's subsequent submission to an external system. In the case of checks, the status value also affects what actions are possible after submission to an external system. For each type of financial transaction object (a check, a reserve, and so on), status codes have specific meanings explained later in this section for each kind of financial transaction.

In most cases, the status of a transaction can escalate in only one direction for two specific status values. For example, reserves can transition from `null` → `pendingapproval`, but not from `pendingapproval` → `null`.

To detect status changes to financial transaction, create a messaging destination that listens for entity status changed events such as `CheckStatusChanged`, `PaymentStatusChanged`, `RecoveryStatusChanged`, and so on. You can write business rules in the Event Fired rule set to trigger custom actions, perhaps to log each change, or to send notifications to external systems. For more information about the messaging system, see “Messaging and Events” on page 299.

The `EntitynameStatusChanged` events trigger after any code creates a financial transaction entity. For example, if you create a new check in ClaimCenter, ClaimCenter triggers a `CheckAdded` and a `CheckStatusChanged` event.

Messages to external systems could represent things like the following:

- Sending a check to a check printing service
- Voiding a check in an accounting system
- A brief user notification email
- Sending a message to a mainframe that records changes to all financial transactions.

Some status transitions always trigger requests and notifications to an external system. If a financial transaction or a transaction change is ready to send to an external system, ClaimCenter changes the financial transaction's status to a status that indicates the pending change. As the transaction's status changes, ClaimCenter generates an event. In the Event Fired rule set you can handle this event. You can generate new messages to external systems in your Event Fired rules. As the main changes that triggered the issue commit to the database, any new `Message` objects also commit to the database.

In a separate thread, ClaimCenter sends the message with the messaging plugins.

After the external system acknowledges the message, you must call special methods on the financial objects that indicate completed status transition to another status. See “How Do Status Transitions Happen?” on page 364 and “Message Acknowledgement-based Status Transitions” on page 366.

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309

## How Do Status Transitions Happen?

The status of a financial transaction can change for the following reasons:

- **Message acknowledgements from an external system** – Some status transitions happen as a consequence of successful acknowledgement of a special message by an external system. This does not happen automatically. You must call a special financials acknowledgement API for the transition to complete. For more information on this process, see “Message Acknowledgement-based Status Transitions” on page 366.
- **User action** – User actions trigger some status transitions, such as initiating a void on a check. If you want to customize PCF files to generate different user actions or current actions in different contexts, you must restrict changes to valid status transitions documented in this topic. Discuss any special needs with Guidewire Customer Support for any edge cases. For the list of valid check status transitions, see “Check Integration” on page 372. Other status changes happen if a user edits, deletes, approves, or rejects a transaction, all of which can occur only in certain circumstances listed in this topic by transaction type.

- **Financials escalation batch processes** – Some transitions occur as a consequence of batch processes that move transactions from one status to another, such as from awaiting submission to submitting. For more information about configuring these batch processes, see “Batch Processes Related to Checks and Payments” on page 616 in the *Configuration Guide*.
- **External systems update check or bulk invoice status with a web service API** – Some transitions happen using the `updateCheckStatus` method, which directly affects checks and indirectly affects associated payments. For example, suppose a bank informs the accounts payable system that a check clears. The accounts payable system in turn can use a web service API to update the ClaimCenter check status to the status `cleared`.

---

**WARNING** Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine. There are various problems if you call SOAP APIs that modify entities that are currently in use. Additionally, messaging plugins must avoid calling other APIs that might change the message root entity in another bundle. For more details, refer to “Messaging Plugins Must Not Call SOAP APIs on the Same Server” on page 311. If there are SOAP APIs related to financials you want to call from plugin code and you do not know of a workaround, contact Guidewire Customer Support.

---

- **Messaging plugin code calling methods on check or bulk invoice entity** – Messaging plugin code from the same server can call domain methods on the check or bulk invoice.

For example, your plugin code could use code like the following.

To set the check status to `voided` from Gosu

```
(Message.MessageRoot as Check).updateCheckStatus(checkNumber, issueDate, TransactionStatus.VOIDED)
```

To set the check status to `voided` from Java

```
((Check) Message.getMessageRoot()).updateCheckStatus(checkNumber, issueDate, TransactionStatus.VOIDED)
```

To set the bulk invoice status to `voided` from Gosu:

```
(Message.MessageRoot as BulkInvoice).updateBulkInvoiceStatus(checkNumber, issueDate, TransactionStatus.VOIDED)
```

To set the bulk invoice status to `voided` from Java:

```
((Check) Message.getMessageRoot()).updateBulkInvoiceStatus(checkNumber, issueDate, TransactionStatus.VOIDED)
```

For the previous examples, be aware that the `updateBulkInvoiceStatus` method does not update the status of the placeholder checks. If you call `updateBulkInvoiceStatus`, you must immediately also call the `check.updateCheckStatus(status)` method to update the status of each check.

---

**IMPORTANT** From plugin code, including messaging plugins, do not call SOAP APIs on the same server. To change the status of an item from plugin code, use the check or bulk invoice methods discussed in the preceding paragraphs. For other types of transactions, such as Recovery, there is no supported API to directly change the status.

---

You can only use these methods in very specific contexts:

- Only from messaging plugins. Never use these methods from rule sets or other Gosu or Java contexts.
- Only after submitting the acknowledgement for a message. If this message is associated with status transitions that occur only using the special financials acknowledgement APIs, call those APIs first before attempting other status changes. This would be either in your `MessageTransport` plugin after a synchronous send and acknowledgement or in your `MessageReply` plugin after your asynchronous acknowledgement of the message.

---

**WARNING** Do not attempt to use `check.updateCheckStatus()` or `bulkinvoice.updateBulkInvoiceStatus()` from any other context than those listed above. It is dangerous to do so.

---

Notice that changing the `Status` property directly is not mentioned in the preceding list. Under no circumstances ever change the `Check.Status` property or any other `transaction.Status` property directly in business rules or

any other Gosu. Only use the appropriate message acknowledgements or check status update API. Even in those cases, be careful only to make transitions that this documentation identifies as valid status transitions for each transaction type.

---

**WARNING** You must never directly update any *transaction.Status* property. Doing so is unsupported and dangerous. Only use the supported APIs discussed in this section, and only for the supported status transitions documented for that type of transaction.

---

There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309

## Message Acknowledgement-based Status Transitions

As mentioned in “How Do Status Transitions Happen?” on page 364, some status transitions must happen as a consequence of handling a message to an external system. For this reason, it is critical for you to understand how events and messaging relate to check and transaction status values.

For some status transitions, the successful acknowledgement of the message completes some larger process. For example, during a standard check transfer, the status must transition from changing from the status `pendingtransfer` → `transferred`. In these cases, the transitions are not automatic as part of acknowledging the message. Instead, your messaging plugins must call a special API immediately after acknowledging the message.

The steps happen in the following order for status transitions that happen through message acknowledgements:

1. An action occurs that would reflect a status change. For example, suppose a user is transferring a check to another claim.
2. This action changes the status to a pending status. In the check transfer example, the status would be `pendingtransfer`.
3. ClaimCenter generates a `CheckStatusChanged` event.
4. Your business rules catch this event and generate a message.
5. At a later time, in another thread on the batch server, the messaging destination (your messaging plugins) send the message to an external system. For example, one messaging destination might represent your check printing service or your accounting system.
6. After the external system responds that it processed that message, the messaging plugins that represent the messaging destination submit an acknowledgement (Ack) for that message using the code:  
`message.reportAck()`.
7. Immediately after submitting the Ack, your messaging plugins must tell ClaimCenter that financial entity's status must complete the action. For example, for a check transfer the check's status must change from `pendingtransfer` to `transferred`. Get a reference to the original financials entity instance and call the special method whose name begins with "acknowledge" (see the table after this numbered list). For a check transfer, the special API to call is `check.acknowledgeTransfer()`. For the other APIs for other financials action, refer to the table after this numbered list. Be careful to call the check or transaction submission methods no more than once for each message Ack.

If you fail to call the special acknowledgement method in the financial entity instance, the financial object inappropriately remains in its status without proceeding.

---

**IMPORTANT** Some financials status transitions happen as part of message acknowledgment. This is not automatic. You must call a special API in your messaging plugins as part of message acknowledgement.

---

If the current status is not the previous status it expects, these APIs throw an exception. For example, The `check.acknowledgeSubmission()` method throws an error if the check is not in requesting status. To avoid exceptions, message code such as asynchronous reply plugins can get the status before calling the API. For example, confirm that a check is still in requesting status before calling `check.acknowledgeSubmission()`.

8. In some cases, as a consequence of calling the financials acknowledge method, a separate but associated financial transaction changes status. For example, if a check changes status, an associated payment may change status too. This status is sometimes identical to the check's status, for example both have the status `voided`. In other cases, they are different statuses, for example the check status is `issued` and the payment status is `submitted`.

The following table defines which acknowledgement methods to use in your messaging plugins.

Action	Call this method at message acknowledgement time in your messaging plugins	Description
Submit a check	<code>check.acknowledgeSubmission()</code>	Updates the check's status to requested if it was requesting, or issued if it was notifying. Updates its payments to submitted. Throws an exception if this check is not in requesting or notifying status
Transfer a check	<code>check.acknowledgeTransfer()</code>	Updates the check's status to transferred. Updates its pendingtransfer payments to transferred. For each transferred payment, updates its onset and offset to submitted. Throws an exception if the check is not in pendingtransfer status.
Void a recovery	<code>recovery.acknowledgeVoid()</code>	Acknowledges a message that this recovery was voided. Updates its status to voided. Throws an exception if the recovery is not in pendingvoid status.
Recode a payment	<code>payment.acknowledgeRecode()</code>	Acknowledges a message that this payment was recoded. Updates its status to recoded. Updates its onset and offset to submitted.
Submit a reserve, recovery, or recovery reserve	<code>transaction.acknowledgeSubmission()</code>	Acknowledges a message that this transaction was submitted. Updates its status to submitted. Throws an exception if this transaction is not in submitting status.
Submit a bulk invoice	<code>bulkInvoice.acknowledgeSubmission(message)</code>	Acknowledges a message that this bulk invoice was submitted. Updates its status to requested. For each line item, updates its status to submitted if it was submitting. Throws an exception if this invoice is not in requesting status.

There is no requirement for there to be a one-to-one correspondence between the number of messages and the number of affected financials objects. For example, one message can be a submission message for more than one transaction. In that case, during message acknowledgment, you must call the special acknowledgement domain method on each relevant financial transaction.

If you add reserves and checks to an exposure that has a too-low validation level, you might want to suppress (omit) messages to an external system. In most cases, you must not send messages to an external system about an exposure that the mainframe does not know about yet. Later, if the exposure passes a higher validation level, the Event Fired rules must send information about all financial transactions on the exposure already entered. As a result, one message may be associated with multiple events or transactions.

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309

## In Event Fired Rules, Treat Status Transition as Final

In your Event Fired business rules that catch changes to the status of a check or transaction, treat the current status as final to avoid race conditions. Specifically, assume the status transition is final as soon as the event triggers, even if your messaging code did not yet sent this information to an external system. You must consider the status fully complete and unchangeable within the Event Fired rule sets that generate financials-related messages to external systems.

On a related note, never directly change the `check.status` or any other `transaction.status` property.

**WARNING** It is dangerous to directly change a check's or transaction's `status` property. For more on this subject, see "How Do Status Transitions Happen?" on page 364.

However, strictly speaking the change of the financial entity's status does not commit to the database until all related code finishes, including all Event Fired rules and any related validations. If there is a major error in the transaction (for instance, an unhandled exception), the entire transaction rolls back. Thus, any new messages did not commit to the Send Queue, just as the change the check or financial transaction did not change nor commit to the database.

In all cases, the Event Fired rules must consider the transaction final. Any messages that you create commit if and only if the change that triggered the rules commits. This is the correct behavior to assure ClaimCenter stays in sync with other systems.

If there were no errors and everything commits to the database, any new messages commit to the send queue. The batch server delivers messages one by one to each destination (set of messaging plugins) as separate database transactions. Refer to the diagram in "Messaging Flow Details" on page 305 for more information.

## Types of Transactions That Track Status

The following sections in this topic discuss specific types of transactions with status codes:

- "Check Integration" on page 372
- "Payment Transaction Integration" on page 381
- "Recovery Reserve Transaction Integration" on page 387
- "Recovery Transaction Integration" on page 388
- "Reserve Transaction Integration" on page 390

## Debugging Financials Messaging

ClaimCenter includes a built-in console message transport example, which is an extremely simple messaging transport that writes the message text payload to the ClaimCenter console window. Enable this transport in `config.xml` to debug integration code that creates and sends messages. This is particularly useful if you are working on financials to see if the events look reasonable. See "Enabling the Built-in Console Transport" on page 359 for details.

## Claim Financials Web Service Data Transfer Objects (DTOs)

It is important to understand that ClaimCenter WS-I web services do not support entity data directly as method arguments or return values. Instead, web service APIs use a combination of:

- a *public ID* value as a `String`, which refers to an object of a specific type by its unique `PublicID` property
- a *data transfer object* (DTO) to encapsulate entity data, and implemented as a Gosu class

For example, instead of passing an `BulkInvoice` entity directly as a method argument, the API might pass a Gosu class called `BulkInvoiceDTO`, which acts as the DTO. The DTO class contains only the properties in an `BulkInvoice` entity instance that are necessary for the web service.

---

**IMPORTANT** If you extend the entity data model with properties that must exist in the web service for sending or receiving data, you must also extend the data transfer object. Add to the set of properties in the corresponding Gosu class with the DTO suffix. For example, if you add an important property to the `BulkInvoice` entity, also add the property to the Gosu class `BulkInvoiceDTO`. Before editing these files, carefully review the introductory comments at the top of each DTO file.

---

## Claim Financials Web Services (ClaimFinancialsAPI)

The claim financials (`ClaimFinancialsAPI`) web service manipulates checks, transaction sets, and other financial information attached to claims. To manipulate claims and exposures from external systems in general ways, use the `ClaimAPI` WS-I web services. See “Claim-related Web Services” on page 151.

This web service uses Data Transfer Objects to represent entity data. See “Claim Financials Web Service Data Transfer Objects (DTOs)” on page 368.

These web service methods are for active claims only. If an external system posts any new updates on an archived claim, `ClaimAPI` methods throw the exception `EntityStateException`.

---

**WARNING** Do not call the SOAP APIs from inside ClaimCenter code to the same computer. From your rules code or plugin code, instead call the local Gosu APIs of any APIs.

---

### Check Status SOAP APIs

An important method in the `ClaimFinancialsAPI` web service is the `updateCheckStatus` method. The `updateCheckStatus` updates the status of a payment based on information coming back from the check processing system or bank. For example, an external system could indicate that a check has cleared.

To update check status from SOAP APIs, see “Update Check Status” on page 378.

### Check Void, Stop, Reissue, Deny SOAP APIs

To void, stop, or reissue a check from SOAP APIs, see “Voiding, Stopping, Denying, and Reissuing Checks” on page 379.

### Add and Import Claim Financials SOAP APIs

There are two general ways to use the `ClaimFinancialsAPI` web service for adding claim financials to a claim:

- **importing financials** – Import financials that were already processed in another system. Although validation is run at commit time, there is no programmatic access to validation results other than exception messages.
- **creating financials** – Process, approve, and runs validation just like the user interface would do for new transactions. There is full programmatic access to validation results.

#### Caution About Archived Claims

These methods work only with active claims. If an external system tries to post updates on an archived claim, ClaimCenter throws the exception `BadIdentifierException` to the web service client.

To check whether a claim is archived before calling these methods, call the `ClaimAPI` web service method `getClaimInfo` and get the archived state. See “Getting Information from Claims/Exposures from External

Systems” on page 154.

```
archiveState = myClaimAPI.getClaimInfo("ABC:12345").getArchiveState();
```

If needed, you can use the `reopenClaim` method of the `ClaimAPI` web service to reopen the claim, which would allow you to add financials. See “Archiving and Restoring Claims from External Systems” on page 156.

## Importing Processed Financials from External Systems

If financials entry happens outside ClaimCenter, you can import the financials. Use the `ClaimFinancialsAPI` web service method `importClaimFinancials`. Import financials only if financials entry happens outside ClaimCenter. This method executes Validation rules when committing to the database, but does not submit items for approval or run Transaction Post-Setup rules after the data is approved.

**IMPORTANT** With `importClaimFinancials`, validation is run at commit time, but there is no programmatic access to validation results other than exception messages.

Never attempt to import financials that would fail validation. This method does not perform the additional steps of duplicate checking, approval processing, or submission procession.

You can only import transaction sets with status `approved`. All checks in the transaction set must have a status of `requested`, `pendingvoid`, `voided`, `pendingstop`, `stopped`, `issued`, or `cleared`. All other types of transactions in the transaction set must have a status of `submitted`, `pendingvoid`, `voided`, `pendingstop`, `stopped`, or `recoded`.

For checks, if a claim or exposure is not at the Ability to Pay validation level, a built-in approval rule prevents check approval and sets check status to Pending Approval.

The transaction set can attach a set of `Document` objects to the claim:

- If the documents are already in the database, populate the document public ID String values in the `TransactionSetDTO.DocumentIDs` property.
- If the documents are not already in the database, populate a set of `DocumentDTO` objects and include them in the `TransactionSetDTO.NewDocuments` property.

## Creating New Financials from External Systems

To create new financial transactions in an automated way just as they would from the user interface, use the `ClaimFinancialsAPI` web service methods that start with the prefix `create`. For example, `createCheckSet`. These methods run approval and Transaction Post-Setup rules, and provide full programmatic access to validation results.

Some additional requirements for financials creation methods:

- The items must correspond to a single claim.
- All of the transactions must have the same currency in the same request. If the reserving currency is not provided on the item, ClaimCenter sets it to the claim currency.

Some notes about the behavior of these methods:

- Creation methods that create financials run approval and Transaction Post-Setup rules.
- Creation methods run Validation rules twice: once before submitting for approval, and once during database commit.
- All the methods encapsulate the main data in a transaction set data transfer object (`TransactionSetDTO`). The `TransactionSetDTO.Subtype` property defines what type of transaction set it is.

The following table summarizes the methods.

To create a transaction set of this type	Use this ClaimFinancialsAPI web service method	Description
Check set	createCheckSet	Creates a check set from an external system
Recovery set	createRecoverySet	Creates a recovery set from an external system
Recovery reserve set	createRecoveryReserveSet	Creates a recovery reserve set from an external system
Reserve set	createReserveSet	Creates a reserve set from an external system

All these methods return a validation result object of type `TransactionSetApprovalResult`. That result object has a `State` property that specifies the result state. The rest of the details in the `TransactionSetApprovalResult` depend on the value of the `State` property enumeration:

- `INVALID` – The result contains a list of validation errors. Check the `TransactionSetApprovalResult.ValidationErrors` property for details.
- `VALID_UNAPPROVED` – The result contains the public ID of the imported transaction set as well as a list of reasons for approval. Check the `TransactionSetApprovalResult.ApprovalReasons` property for details.
- `VALID_APPROVED` – The result contains the public ID of the imported transaction set.

If you want approval to run, set the financial transaction status for the financials to the draft status (`TransactionStatus.TC_draft`). For checks and transactions, in general set the financial transaction status for incoming financials to the draft status (typecode `TransactionStatus.TC_draft`).

You can also choose to use `AwaitingSubmission` or `PendingApproval` status to mark the object for Rules to transition a recently-imported check to a later status such as `issued`. Do not set status to a status value that implies committed actions, such as the status `submitted`, `requesting`, `requested`, or `issued`. Those values will throw exceptions on import.

For checks, if a claim or exposure is not at the Ability to Pay validation level, a built-in approval rule prevents check approval and sets check status to Pending Approval.

The transaction set can attach a set of `Document` objects to the claim:

- If the documents are already in the database, populate the document public ID `String` values in the `TransactionSetDTO.DocumentIDs` property.
- If the documents are not already in the database, populate a set of `DocumentDTO` objects and include them in the `TransactionSetDTO.NewDocuments` property.

## Multicurrency with New Financials

Both the importing financials and creating financials web service methods support setting of custom exchange rates. Within the `TransactionSetDTO` data transfer object, set the `NewExchangeRate` and `NewExchangeRateDescription` properties before passing the data transfer object to ClaimCenter. Note that you cannot override each transaction with its own exchange rate in the `TransactionDTO` object. All the transactions in the transaction set must have the same exchange rate. See “Multiple Currencies” on page 335 in the *Application Guide* for more information about multicurrency and “Exchange Rate Integration” on page 409 for more information about specifying exchange rates.

There is internal validation that the passed-in amount is the same as the calculated value of the product (multiplication) of transaction amount and exchange rate. If they are not equal, ClaimCenter logs a warning message (logging level `WARN`), but still saves such changes.

---

**WARNING** Never call locally-hosted web services from within a plugin or the rules engine. See “Messaging Plugins Must Not Call SOAP APIs on the Same Server” on page 311.

---

## Multicurrency Foreign Exchange Adjustment SOAP APIs

There are two `ClaimFinancialsAPI` web service methods to perform foreign exchange adjustments using web services. There are two different methods, one to apply to a check, and one to apply to a payment.

**IMPORTANT** ClaimCenter does not support foreign exchange adjustments for multi-payee checks or payments. However, joint-payee checks can have adjustments.

For a check, use the method `applyForeignExchangeAdjustmentToCheck`. For a payment, use the method `applyForeignExchangeAdjustmentToPayment`.

These methods take the following arguments:

- a check or payment public ID
- a new claim amount, as a `BigDecimal`. If `null`, the claim amount is not adjusted.
- a new reporting amount, as a `BigDecimal`. This is the value to set for the check's reporting amount. If `null`, the reporting amount is not adjusted.

They both apply a foreign exchange adjustment to the payments on the indicated check or payment, once the final converted amount is known. This method can only be called on a check/payment that has already been escalated and sent downstream, but has not been canceled or transferred.

This method is only for the case of where the insurer's deployment does not support multiple base currencies. That is, use this if the claim and reporting currencies are always the same. For example, suppose you create a check with two payments in the amounts of \$60.00 and \$40.00 in the transaction currency. Initially, the payments have the same amounts in the claim currency, which means the exchange rate at the time of check creation is 1:1. If the check eventually cashes, suppose the exchange rate changes to be 1:1.1. The two payments now total \$110.00 in the claim currency. If this happens, you can call this method and pass \$110 as the new claim currency amount. The additional \$10 divides up between the payments proportionally, resulting in one payment for \$66.00, and one for \$44.00, which is a 10% increase of each. If either payment had multiple line items, the additional monies divide up proportionally across line items. The Total Incurred and Total Payments calculated values change to reflect the foreign exchange rate adjustment.

See “Multiple Currencies” on page 335 in the *Application Guide* for more information about multicurrency and “Exchange Rate Integration” on page 409 for more information about specifying exchange rates.

**WARNING** Never call locally-hosted web services from within a plugin or the rules engine. See “Messaging Plugins Must Not Call SOAP APIs on the Same Server” on page 311.

## Check Integration

There are two types of checks within ClaimCenter:

- **Standard checks** – These are the normal checks requested from within the ClaimCenter application within the **Check Wizard** user interface, but you can also trigger them from business rules. Typically, you generate checks from within the user interface. This triggers an approval process of automated approval rules and human approvers. Eventually, ClaimCenter sends checks to external systems to print and process. You can track check status, and request a void or stop if necessary.
- **Manual checks** – Use manual checks to record checks written from a check book or another system outside the ClaimCenter application. In this case, ClaimCenter tracks the check for completeness only. Processing manual checks within ClaimCenter allows you to track checks in a central location with optional notification of other users or external systems. You can track check status, and request a void or stop if necessary. Manual checks a status code only used by manual checks (`notifying`), which indicates printing the check is unnecessary.

Technically, checks are not *transactions*. However, in most ways ClaimCenter treats checks like other financial transactions entities, such as tracking them with a status code.

The following table includes check status codes and their meanings. The rightmost two columns indicate whether the status exists for standard checks (“Std”), manual checks (“Man”), or both.

Check status	Meaning	Std	Man
null	An internal initial status.	Yes	Yes
pendingapproval	The check request saved in ClaimCenter but is not yet approved. Manual checks do not need approval, however you can customize this behavior if necessary.	Yes	Yes
rejected	The approver did not approve the check request. In the reference implementation, manual checks do not need approval but you can customize this.	Yes	Yes
awaitingsubmission	The check is held in this status until the date reaches the check's issuance date and a background processing task prepares the check for submission.	Yes	--
requesting	The standard check request is ready to be sent to an external system.	Yes	--
requested	The standard check successfully sent to an external system.	Yes	--
pendingvoid	The check void request is ready to be sent to an external system.	Yes	Yes
pendingstop	A check stop request is ready to be sent to an external system.	Yes	Yes
pendingtransfer	A check transfer request is ready to be sent to an external system.	Yes	Yes
issued	The check issued in the external system. For manual checks, this means specifically that the manual check notification was sent to an external system and successfully acknowledged.	Yes	Yes
cleared	The check cleared in the external system.	Yes	Yes
notifying	The manual check notification is ready to be sent to an external system. This indicates that printing the check is unnecessary.	--	Yes

To detect standard check requests or manual checks, you can write event business rules that listen for the `CheckStatusChanged` event and check for changes to the `check.status` property. The following table includes the possible status code transitions and how this transition can occur. The rightmost two columns indicate whether the transition exists for standard checks (“Std”), manual checks (“Man”), or both.

Check status	Can become status	How it changes	Std	Man
null	→ pendingapproval	For a new check, either approval rules trigger approval, or the check exceeded authority limits. The ClaimCenter reference implementation auto-approves manual checks, but you can customize this with approval rules.	Yes	Yes
	→ awaitingsubmission	A standard check reaches the end of the new check wizard and approval rules determine that no approval is necessary.	Yes	--
	→ notifying	A manual check would make this initial transition if the check does not require approval. The ClaimCenter reference implementation auto-approves manual checks, but this can be customized with approval rules.	--	Yes
pendingapproval	→ awaitingsubmission	The approver approves the standard check	Yes	--
	→ rejected	The approver rejects the check	Yes	Yes
	→ object deleted	A user deletes a check in the user interface.	Yes	Yes
	→ notifying	The approver approves the manual check.	--	Yes
rejected	→ awaitingsubmission	A user's edit to a check does not trigger approval.	Yes	--

Check status	Can become status	How it changes	Std	Man
awaitingsubmission	→ pendingapproval	A user's edit to a check triggers approval. The ClaimCenter reference implementation auto-approves manual checks, but this can be customized with approval rules.	Yes	Yes
	→ notifying	A user edits a rejected check and the updated check does not require approval.	--	Yes
	→ <i>object deleted</i>	A user deletes a check in the user interface.	Yes	Yes
	→ requesting	Automatically transitioned to the new status using the scheduled financials escalation batch process.	Yes	--
	→ pendingapproval	A user's edit to a check triggers approval.	Yes	--
	→ <i>object deleted</i>	A user deletes a check in the user interface.	Yes	--
requesting	→ requested	For a standard check only, this transition indicates ClaimCenter received an acknowledgement for the associated message for the requesting status. This transition requires you to call <code>check.acknowledgeSubmission()</code> in your message Ack code in your messaging plugins.	Yes	--
pendingtransfer	→ pendingtransfer	A user attempts to transfer a check.	Yes	--
	→ pendingvoid	A user attempts to void a check in the application user interface.	Yes	--
	→ pendingstop	A user attempts to stop a check in the application user interface.	Yes	--
	→ denied	For single standard checks, this transaction works from the check method <code>denyCheck</code> (either SOAP API method or domain method).  For single manual checks, this transition works only from the check method <code>denyCheck</code> called from messaging plugins. It does not work from the SOAP API version of this method.	Yes	--
		This transaction only works for single checks. The check cannot be part of a multipayee/grouped check, nor a <code>BulkInvoiceItem</code> check.		
pendingvoid	→ issued	<i>This transition is disallowed.</i> The check must first have the status requested.	--	--
	→ cleared	<i>This transition is disallowed.</i> The check must first have the status requested.	--	--
	→ voided	From SOAP API or domain method, calling <code>updateCheckStatus</code> for a check status change that indicates the check voided.	Yes	Yes
voided	→ issued	From SOAP API or domain method, calling <code>updateCheckStatus</code> for a check status change that indicates the check issued. The void was unsuccessful.	Yes	Yes
	→ cleared	From SOAP API or domain method, calling <code>updateCheckStatus</code> for a check status change that indicates the check cleared. The void was unsuccessful.	Yes	Yes

Check status	Can become status	How it changes	Std	Man
	→ stopped	<p><i>This transition is disallowed.</i></p> <p>To simulate this transition:</p> <ol style="list-style-type: none"> <li>1. first transition from PendingVoid → Issued using the method updateCheckStatus (either SOAP API method or domain method).</li> <li>2. Request stop using the stopCheck() method (again from either SOAP API or domain method).</li> <li>3. Finally use updateCheckStatus again to transition from PendingStop → Stopped.</li> </ol>	Yes	Yes
pendingstop	→ stopped	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check stopped.	Yes	Yes
	→ issued	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check issued. The stop was unsuccessful.	Yes	Yes
	→ cleared	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check cleared. The stop was unsuccessful.	Yes	Yes
	→ voided	<p><i>This transition is disallowed.</i></p> <p>To simulate this transition:</p> <ol style="list-style-type: none"> <li>1. first transition from PendingStop → Cleared using the method updateCheckStatus (either SOAP API method or domain method).</li> <li>2. Request void using the voidCheck() method (again from either SOAP API or domain method).</li> <li>3. Finally use updateCheckStatus again to transition from PendingVoid → Voided.</li> </ol>	Yes	Yes
pendingtransfer	→ transferred	ClaimCenter received an acknowledgement for the associated message for the pendingtransfer status. This transition requires you to call check.acknowledgeTransfer() in your message Ack code in your messaging plugins.	Yes	Yes
issued	→ pendingtransfer	A user requests a check transfer.	Yes	Yes
	→ pendingvoid	A user attempts to void a check in the application user interface.	Yes	Yes
	→ pendingstop	A user attempts to stop a check in the application user interface.	Yes	Yes
	→ cleared	SOAP API for a check status change that indicates the check cleared.	Yes	Yes
cleared	→ pendingtransfer	A user attempts to transfer a check.	Yes	Yes
	→ pendingvoid	A user attempts to void a check in the application user interface.	Yes	Yes
requested	→ pendingtransfer	A user attempts to transfer a check.	Yes	--
	→ issued	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check issued.	Yes	--
	→ cleared	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check cleared.	Yes	--
	→ pendingvoid	A user requests a check void.	Yes	--
	→ pendingstop	A user requests a check stop.	Yes	--

Check status	Can become status	How it changes	Std	Man
	→ denied	For single standard checks, this transaction works from the check method denyCheck (either SOAP API method or domain method). This transition works only for single standard checks. The check cannot be part of a multipayee/grouped check, nor a BulkInvoiceItem check.	Yes	--
notifying	→ pendingtransfer	A user requests a check transfer.	--	Yes
	→ issued	For a manual check only, this transition indicates ClaimCenter received an acknowledgement for the associated message for the requesting status. This transition requires you to call check.acknowledgeSubmission() in message acknowledgement code in your messaging plugins.	--	Yes
	→ cleared	The check must first have the status issued. For manual checks, this transition happens if, from the SOAP API or domain method, calling updateCheckStatus for a check status change indicates that the check cleared.  For standard checks, this transition is disallowed.	--	Yes
	→ pendingvoid	A user requests a check void.	--	Yes
	→ pendingstop	A user requests a check stop.	--	Yes
	→ denied	For single standard checks, this transition happens through the check method denyCheck, either through SOAP APIs or the domain method.  For single manual checks, this transition works only from the check method denyCheck called from messaging plugins. It does not work from the SOAP API version of this method.	--	Yes
		This transition only works for single checks. The check cannot be part of a multipayee/grouped check, nor a BulkInvoiceItem check.		
voided	→ issued	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check issued.	Yes	Yes
	→ cleared	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check cleared.	Yes	Yes
stopped	→ issued	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check issued.	Yes	Yes
	→ cleared	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check cleared.	Yes	Yes

### Required Message Acknowledgements for Checks

Message acknowledgements trigger certain automatic status transitions. The messaging plugin representing the receiving system must call special financials acknowledge methods for the transition to occur. For checks, acknowledgement-driven status transitions apply in these cases:

- Standard check status: `requesting` → `requested`
- Manual check status: `notifying` → `issued`
- Standard check status: `pendingtransfer` → `transferred`

For more information, see “Message Acknowledgement-based Status Transitions” on page 366.

---

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309

---

## Check Status Transitions

As mentioned earlier, some status transitions occur after an external system calls web service APIs to update the transaction status. To do this, use the `ClaimFinancialsAPI` web service method `updateCheckStatus`.

**Note:** For more information about `ClaimFinancialsAPI` web service interface, see “Claim Financials Web Services (`ClaimFinancialsAPI`)” on page 369.

For example, an external system could use this API to update the status of an issued check to the status `issued`. Alternatively, change the status from your messaging code after you acknowledge a message using the domain method `check.updateCheckStatus(...)`.

However, do not call the SOAP API version from messaging plugin code from the same server. Refer to the discussion in “How Do Status Transitions Happen?” on page 364 and “Messaging Plugins Must Not Call SOAP APIs on the Same Server” on page 311 for more information.

---

**WARNING** Never call locally-hosted web services from within a plugin or the rules engine. See “Messaging Plugins Must Not Call SOAP APIs on the Same Server” on page 311. Additionally, messaging plugins must avoid calling other APIs that might change the message root entity in another bundle.

---

Changing the check status is particularly important for triggering these status transitions:

- Check status: `requested` → `issued`
- Check status: `issued` → `cleared`
- Check status: `pendingvoid` → `voided`
- Check status: `pendingstop` → `stopped`

However, those are not the only legitimate status transitions for checks using this API. For example, if the user requested a void, the external system using this API could respond that the check has already been issued and thus the void request was unsuccessful. For the full list of valid check status transitions, see “Check Integration” on page 372.

The valid status values for use with the `updateCheckStatus` API are `issued`, `cleared`, `voided`, and `stopped`. If the check status changes to `voided` or `stopped`, the status on the associated `Payment` entities updates to the appropriate `Payment` status code. The `Payment` status code is not necessarily the same as the `Check` status code. For details of how these transitions affect `Payment` entities, see “Payment Transaction Integration” on page 381.

### Updating Status from SOAP

The `ClaimFinancialsAPI` web service method `updateCheckStatus` takes the following:

- Check ID
- Check number
- Transaction date
- A new status for the transaction
- An optional list of name/value pairs of properties to update on the `Check` object at the same time the `Check` transaction status commits to the ClaimCenter database.

ClaimCenter uses the check number and date only if updating the check status to `issued`. Otherwise, just pass `null` for those parameters.

You can also call the `voidCheck` method to void the check and the `stopCheck` method to stop the check.

**Note:** For more information about using the `IClaimFinancials` web service interface, see “Claim Financials Web Services (ClaimFinancialsAPI)” on page 369.

### Updating Status from Entity Domain Method

Instead of using a SOAP call from an external system, your messaging plugins can set the status.

Message acknowledgements trigger certain automatic status transitions. The messaging plugin representing the receiving system must call special financials acknowledge methods for the transition to occur. For those transitions, call the special acknowledgement method on any related financials objects before attempting any other status transitions.

Keeping in mind the warning in the previous paragraph, from your messaging plugin you can call the `check.updateCheckStatus(....)` method to change the check status. For additional important warnings and limitations, see “How Do Status Transitions Happen?” on page 364.

### Changing Check Status Using Either Approach

The following information applies to changing status of a check or transaction with SOAP APIs or using domain methods.

If a check is `pendingstop` or `pendingvoid` and the new status is `issued` or `cleared`, the status values of the check and its payments update. Next, ClaimCenter creates a warning activity. Next, ClaimCenter assigns the activity to the user who attempted to void or stop the check. This activity lets the user know that the check did not void successfully. Finally, ClaimCenter generates any reserve that is necessary to keep open reserves from becoming negative.

## Update Check Status

The most important method in the `ClaimFinancialsAPI` web service is the `updateCheckStatus` method. It updates the status of a payment based on information coming back from the check processing system or bank. For example, an external system could indicate that a check has cleared.

**IMPORTANT** This method is also available as a domain method on the `Check` entity. Use the SOAP version only from external systems. See the warning later in this topic about avoiding SOAP callbacks to the same server.

To use `updateCheckStatus`, pass as arguments:

- Check public ID
- Check number (optional, ignored if `null`)
- Issue date (optional, ignored if `null`)
- Status of the check, as a `TransactionStatus` typecode

To omit any optional arguments, pass `null` for that argument.

For example:

```
claimfinancialsAPI.updateCheckStatus("abc:1234" , "1024", null, TransactionStatus.TC_REQUESTED);
```

To void or stop a check, see “Voiding, Stopping, Denying, and Reissuing Checks” on page 379.

For more information about how a check’s status affects check and payment status, see the following topics:

- “Check Integration” on page 372
- “Payment Transaction Integration” on page 381,

- “Check Status Transitions” on page 377

---

**WARNING** Never call locally-hosted web services from within a plugin or the rules engine. See “Messaging Plugins Must Not Call SOAP APIs on the Same Server” on page 311. Additionally, messaging plugins must avoid calling other APIs that might change the message root entity in another bundle.

---

## Voiding, Stopping, Denying, and Reissuing Checks

ClaimCenter includes APIs to change status of a check in specific ways. They are available in two ways:

- The `ClaimFinancialsAPI` web service interface, which external systems can call.
- Domain methods on `Check` entity instances.

### Voiding a Check

To void a check, call the method `voidCheck`. This is available as a method in the `ClaimFinancialsAPI` web service and also as a domain method on a `Check` entity instance. Both fundamentally have the same behavior, although the arguments are slightly different due to how web services work.

The web service variant takes a check public ID and a reason (which you can set to `null` if unneeded):

```
claimfinancialsapi voidCheck("abc:12345", "this is the reason");
```

The domain version takes no arguments:

```
check voidCheck();
```

Both versions void a check. This changes the status of the check and creates offsetting payments to offset each payment on the check. In addition, this API creates offsetting reserves if a payment on the check is eroding and either of the following is true:

- The payment's exposure is closed. Closed exposures always have zero open reserves.
- Open reserves on the payment's `ReserveLine` would become negative without an offsetting reserve.

Offsetting reserves are included in this check's `CheckSet`.

The status of the check and the original payments on the check are set to `pendingvoid`. The offsetting payments are set to `submitting` status. The status of any offsetting reserves is set to `submitting`.

This method works for both single-payee and multi-payee checks. However, if this is a multi-payee check, then the void request happens for all checks in the check group.

This action does not require approval.

It throws an exception if the check status is not one of the following: `notifying`, `requested`, `requesting`, `issued`, `cleared`.

**Note:** See related method `voidAndReissueCheck` that applies only to multi-payee checks “Void and Reissue (Only for Multi-Payee Checks)” on page 379.

### Void and Reissue (Only for Multi-Payee Checks)

There is a similarly-named method called `voidAndReissueCheck` that applies only to multi-payee checks and does not affect the payments on the check. It does not create any new transaction entities. (Contrast this with the other method called simple `voidCheck`, which creates offsetting payments.) This is available as a method in the `ClaimFinancialsAPI` web service and also as a domain method on a `Check` entity instance. Both fundamentally have the same behavior, although the arguments are slightly different due to how web services work.

This method voids the check and reissues it. ClaimCenter creates a new replacement check. The status of the original check becomes `pendingvoid`.

Compared to the regular `void` method, it takes an extra parameter for the description of the reason for stopping the check. For example, the following code calls the SOAP API version of `voidAndReissueCheck`:

```
claimfinancialsapi_voidAndReissueCheck("abc:12345",
                                         "check was accidentally sent twice, voiding this one");
```

This action does not require approval.

The API throws an exception if the check status is not one of the following: `notifying`, `requested`, `requesting`, `issued`, `cleared`.

Reissuance proceeds as follows:

1. If the original check was the primary Check for the CheckGroup, the new check becomes the primary Check. The original check converts to a secondary Check (still in the same CheckGroup). All of the Payments and Deductions move to the new Check.
2. Regardless of whether the original Check already had a `CheckPortion`, ClaimCenter creates a new, fixed-amount `CheckPortion` for it. In case it does not already specify a fixed portion, its amount does not fluctuate (for example, if it previously used a percentage portion).
3. On the new Check:
  - `CheckNumber` and `IssueDate` are `null`.
  - `ScheduledSendDate` is set to today.
  - The status is `awaitingsubmission`.

You can configure how ClaimCenter initializes the new check.

## Stopping a Check

To stop a check, call the method `stopCheck`. This is available as a web service (in the `ClaimFinancialsAPI` web service) and also as a domain method on a `Check` entity instance. However, the arguments are slightly different due to how web services work.

The `voidCheck` and `stopCheck` methods fundamentally have the same behavior except that the check transitions into `pendingstop` status instead of `pendingvoid` status. The check status requirements are the same as for `voidCheck`. For details, see “Voiding a Check” on page 379.

The SOAP API version takes a check public ID and a reason `String` (set to `null` if unused):

```
claimFinancialsApi_stopCheck("abc:12345", "this is the reason");
```

The domain method version takes no arguments:

```
check.stopCheck()
```

## Stop and Reissue (Only for Multi-Payee Checks)

Similar to the pair of methods called `voidCheck` and `voidAndReissueCheck`, there is another method called `stopandReissueCheck` that applies only to multi-payee checks. The `stopandReissueCheck` method does not affect the payments on the check. It does not create any new transaction entities. It is just the same as `voidAndReissueCheck` except that transitions into `pendingstop` status instead of `pendingvoid` status.

Compared to the `stopCheck` method, it takes an extra parameter for the description of the reason for stopping the check.

For example:

```
claimfinancialsapi_stopAndReissueCheck("abc:12345", "fraud was detected after check was created");
```

## Denying Checks

Denial of a check supports automated processes in downstream systems that catch an invalid check and then deny it immediately.

**IMPORTANT** Only single-payee checks can be denied.

An example for denial is catching an invalid payee because the payee is on a watch list. After the check transitions to the status `issued`, `cleared` or further statuses such as `voided`, it is too late to deny the check. Denying the check must happen almost immediately, if needed.

During development, add ClaimCenter rules to enforce any check-related requirements that could result in the denial of a check by a downstream system. For example, in the earlier example with a watch list, you could make a web service that checks the validity of the payee before creating the check. Next, incorporate that verification into the Check Wizard user interface or associated rules. However, be sure that you test such a system to ensure that it does not impact performance beyond your requirements or create other complications.

To deny a check from web services, call the `ClaimFinancialsAPI` method `denyCheck`. It takes a check public ID.

The domain method version takes no arguments:

```
check.denyCheck()
```

## Check Scheduled Send Date Only Modifiable in Special Situations

The check property for the schedule send date, `ScheduledSendDate`, is only modifiable in Transaction Presetup Rules or from Gosu called from the application user interface (PCF) code. The date determines whether to include the check amount in one of the following:

- **Future Payments** – tomorrow or later
- **Awaiting Submission** – today, which is included in Total Payments and similar financials calculations

If the date is inappropriately updated, ClaimCenter throws an exception with message:

```
Check contains a payment whose LifeCycleState is inconsistent with the Check's Scheduled Send Date.
```

## Payment Transaction Integration

The following table includes the status codes and meanings for payments. The rightmost two columns indicate whether the status exists for standard checks (“Std”), manual checks (“Man”), or both.

Payment Status	Meaning	Std	Man
<code>null</code>	An internal initial status.	Yes	Yes
<code>pendingapproval</code>	The associated check saved in ClaimCenter but is not yet approved. The ClaimCenter reference implementation auto-approves manual checks, but this can be customized with approval rules.	Yes	Yes
<code>rejected</code>	The approver did not approve the associated check.	Yes	Yes
<code>awaitingsubmission</code>	The associated check is held in this status until the date reaches the check's issuance date and a background processing task prepares the check for submission. See “Check Integration” on page 372 for more details.	Yes	--
<code>submitting</code>	The payment is ready to be sent to an external system. This corresponds to the associated standard check being in the requesting status or the associated manual check being in the notifying status.	Yes	Yes
<code>submitted</code>	The payment sent to an external system. This corresponds to the associated standard check being in the requested, issued, or cleared status or the associated manual check being in the issued or cleared status.	Yes	Yes
<code>pendingvoid</code>	Void request for the associated check is ready to be sent to an external system.	Yes	Yes

Payment Status	Meaning	Std	Man
pendingstop	A stop request for the associated check is ready to be sent to an external system.	Yes	Yes
pendingtransfer	The associated check is ready to be transferred to another claim, and is ready to send appropriate notification to an external system.	Yes	Yes
pendingrecode	The payment recode notification is ready to be sent to an external system.	Yes	Yes
recoded	The payment recode notification sent to an external system.	Yes	Yes
transferred	The check transferred to another claim.	Yes	Yes
voided	The associated check voided.	Yes	Yes
stopped	The associated check stopped.	Yes	Yes

To detect new or changed payments, you can write event business rules that listen for the `PaymentStatusChanged` event and check for changes to the `payment.Status` property. The following table includes the possible status code transitions and how this transition can occur. The rightmost two columns indicate whether the transition exists for associated standard checks (“Std”), manual checks (“Man”), or both.

Payment status	Can change to status	How it changes	Std	Man
null	→ pendingapproval	The user creates a new check and it requires approval. The ClaimCenter reference implementation auto-approves manual checks, but this can be customized with approval rules.	Yes	Yes
	→ awaitingsubmission	The user creates a standard check, and approval rules specify that the check does not require approval.	Yes	--
	→ submitting	A user creates a manual check, and approval rules specify that the check does not require approval. The ClaimCenter reference implementation auto-approves manual checks, but this can be customized with approval rules.	--	Yes
pendingapproval	→ awaitingsubmission	The approver approves the associated standard check and it does not need further approval.	Yes	--
	→ rejected	The approver rejects the associated check.	Yes	Yes
	→ submitting	The approver approves the associated manual check.	--	Yes
	→ object deleted	A user deletes the object in the user interface.	Yes	Yes
rejected	→ awaitingsubmission	A user edits the associated check and approval rules specify that the check does not require approval.	Yes	--
	→ pendingapproval	A user edits the associated check, and approval rules specify that the check requires approval.	Yes	Yes
	→ submitting	A user edits the associated manual check, and approval rules specify that the check does not require approval.	--	Yes
	→ object deleted	A user deletes the object in the user interface.	Yes	Yes
awaitingsubmission	→ submitting	Automatic escalation batch process.	Yes	--
	→ object deleted	If the user deletes the object in the user interface.	Yes	--
	→ pendingapproval	If the user edits the associated check, and one or more payments on the check changes and the check requires reapproval after that change	Yes	--
submitting	→ submitted	ClaimCenter received an acknowledgement for the associated message for the submitting status. This transition requires you to call <code>check.acknowledgeSubmission()</code> in your message Ack code in your messaging plugins.  The check and the associated payment update with new statuses. Also, the status of associated payments for onset or offset payments created for a recode or transfer automatically update to match the new payment status.	Yes	Yes

Payment status	Can change to status	How it changes	Std	Man
	→ pendingvoid	<p>Either of the following:</p> <ul style="list-style-type: none"> <li>A user attempts to void an associated check in the user interface.</li> <li>Some code calls the voidCheck method (available as a SOAP API or as a domain method on a check). See “Voiding, Stopping, Denying, and Reissuing Checks” on page 379</li> </ul>	Yes	Yes
	→ pendingstop	<p>Either of the following:</p> <ul style="list-style-type: none"> <li>A user attempts to stop an associated check in the user interface.</li> <li>Some code calls the stopCheck method (available as a SOAP API or as a domain method on a check). See “Voiding, Stopping, Denying, and Reissuing Checks” on page 379</li> </ul>	Yes	Yes
	→ pendingtransfer	A user requests an check transfer to another claim for the associated check.	Yes	Yes
	→ pendingrecode	A user requests a payment recoding. Unlike most other payment status transitions, a recode is about the payment, not simply a mirror or translation of an associated check status. See “Integration Events For Payment Recoding” on page 385.	Yes	Yes
submitted	→ pendingvoid	<p>Either of the following:</p> <ul style="list-style-type: none"> <li>A user attempts to void an associated check in the user interface.</li> <li>Some code calls the voidCheck method (available as a SOAP API or as a domain method on a check). See “Voiding, Stopping, Denying, and Reissuing Checks” on page 379</li> </ul>	Yes	Yes
submitted	→ pendingstop	<p>Either of the following:</p> <ul style="list-style-type: none"> <li>A user attempts to stop an associated check in the user interface.</li> <li>Some code calls the stopCheck method (available as a SOAP API or as a domain method on a check). See “Voiding, Stopping, Denying, and Reissuing Checks” on page 379</li> </ul>	Yes	Yes
submitted	→ pendingtransfer	A user requests a check transfer to another claim.	Yes	Yes
submitted	→ pendingrecode	A user requests a payment recoding. Unlike most other payment status transitions, a recode is primarily about the payment, not simply a mirror or translation of an associated check status. For more information, see “Integration Events For Payment Recoding” on page 385.	Yes	Yes
pendingvoid	→ voided	SOAP API for a check status change that indicates successful voiding.	Yes	Yes
pendingvoid	→ stopped	SOAP API for a check status change that indicates the check stopped. The void was unsuccessful.	Yes	Yes
pendingvoid	→ submitted	SOAP API for a check status change that indicates the check submitted. The void was unsuccessful. This happens if the associated check's status changed to issued or cleared.	Yes	Yes
pendingstop	→ stopped	SOAP API for a check status change that indicates stopping.	Yes	Yes
pendingstop	→ voided	SOAP API for a check status change that indicates the check voided. The stop was unsuccessful.	Yes	Yes

Payment status	Can change to status	How it changes	Std	Man
	→ submitted	SOAP API for a check status change that indicates the check submitted. The stop was unsuccessful. This happens if the associated check's status changed to issued or cleared.	Yes	Yes
pendingtransfer	→ transferred	ClaimCenter received an acknowledgement for the associated message for the pendingtransfer status. This transition requires you to call <code>check.acknowledgeTransfer()</code> in your message Ack code in your messaging plugins.  The check and the associated payment update to the transferred status. See "Integration Payment Events for Check Transfer" on page 386	Yes	Yes
pendingrecode	→ recoded	ClaimCenter received an acknowledgement for the associated message for the pendingrecode status. This transition requires you to call <code>check.acknowledgeRecode()</code> in your message Ack code in your messaging plugins.  For more information, see "Integration Events For Payment Recoding" on page 385.	Yes	Yes
recoded	→ pendingvoid	Either of the following: <ul style="list-style-type: none"><li>• A user attempts to void an associated check in the user interface.</li><li>• Some code calls the <code>voidCheck</code> method (available as a SOAP API or as a domain method on a check). See "Voiding, Stopping, Denying, and Reissuing Checks" on page 379</li></ul>	Yes	Yes
	→ pendingstop	Either of the following: <ul style="list-style-type: none"><li>• A user attempts to void an associated check in the user interface.</li><li>• Some code calls the <code>stopCheck</code> method (available as a SOAP API or as a domain method on a check). See "Voiding, Stopping, Denying, and Reissuing Checks" on page 379</li></ul>	Yes	Yes
transferred	<i>no transitions possible</i>	<i>no transitions possible</i> . For related information, see "Integration Payment Events for Check Transfer" on page 386.	--	--
voided	→ submitted	SOAP API for a check status change that indicates that the check issued. The void request failed.	Yes	Yes
stopped	→ submitted	SOAP API for a check status change that indicates that the check issued. The stop request failed.	Yes	Yes

### Required Message Acknowledgements for Payments

Message acknowledgements trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. The messaging plugin representing the receiving system must acknowledge the message for the transition to occur. For payments, acknowledgement-driven status transitions apply in these cases

- Payment status: `pendingrecode` → `recoded`
- Payment status: `submitting` → `submitted`
- Payment status: `pendingtransfer` → `transfer`

For more information, see "Message Acknowledgement-based Status Transitions" on page 366.

For more information about using the `IClaimFinancials` web service interface (for example for changing status of a payment's check), see "Claim Financials Web Services (ClaimFinancialsAPI)" on page 369.

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See "Important Entity Data Restrictions in Messaging Rules and Messaging Plugins" on page 309

## Integration Events For Payment Recoding

ClaimCenter supports payment *recoding*, which is a type of accounting change that does not issue a new check. These offsets connect with the original check. Every payment has a *reserve line*, which is a ClaimCenter encapsulation of a specific claim, a specific exposure, a specific cost type, and a specific cost category. This reserve line coding of payments provides critical data for accounting departments to track an insurance company's many payments. Insurance companies often generate a check, record the payment, and later recode the payment to maintain the payment amount but reassign it to a different reserve line for accounting purposes.

**Note:** Within a reserve line definition, you can set the following to `null` as appropriate to the exposure component, the cost type component, or the cost category component. The claim is the only non-nullable component.

During payment recoding, ClaimCenter generates the following events:

- Events on the original payment that indicate that the original payment recoded.
- Events on a new *offset payment*, a `Payment` that effectively negates the original payment.
- Events on a new *onset payment*, a `Payment` that encodes a new payment in the correct reserve line.

For the new offset and onset payments, there are Gosu APIs that allow integration developers to easily access related objects. The following APIs are domain methods on the `Payment` entity:

- `RecodingOffset` – Returns `true` if the payment in question is an offset created for a recoded payment, and returns `false` otherwise.
- `RecodingOnset` – Returns `true` if the payment in question is an onset created for a recoded payment, returns `false` otherwise.
- `PaymentBeingOffset` – If the payment is an offset, this property returns the payment being offset by the payment in question. If the payment is not an offset, returns `null`.
- `OriginalPayment` – returns the payment for which ClaimCenter created the onset. So, if you call this method on a payment created as the onset for a recoded payment, it returns the original recoded payment.
- `Onset` – For a transferred or recoded payment, returns the new onset payment on the target (destination) claim.
- `Offset` – For a transferred or recoded payment, returns its offsetting payment.

For more about these methods and related APIs, see "Payment Integration Domain Methods (For Event Fired Rules)" on page 386.

The following table lists events that trigger during recoding:

Root object	Event name	Meaning in this context
Original payment	<code>PaymentChanged</code>	The payment status property changed. You can catch this event, but you can also use the special <code>PaymentStatusChanged</code> event listed later in this table.
	<code>PaymentStatusChanged</code>	The payment status property changed, specifically <code>payment.status = pendingrecode</code> . Use the <code>Payment</code> domain properties <code>Onset</code> and <code>Offset</code> to get the associated payments.

Root object	Event name	Meaning in this context
New offset payment	PaymentAdded	New offset payment. Within business rules, check <code>payment.RecodingOffset</code> to determine if this is the offset payment. Call <code>payment.PaymentBeingOffset</code> to get the original payment.
	PaymentStatusChanged	New financial transaction entities also receive a status changed event after their creation.
New onset payment	PaymentAdded	New onset payment. Within business rules, call <code>payment.RecodingOnset</code> to determine if this is the onset payment. Call <code>payment.OriginalPayment</code> to get the original payment.
	PaymentStatusChanged	New financial transaction entities also receive a status changed event after their creation.

It is important to understand that the order events trigger are not necessarily the order listed in this table. The event ordering is the event name ordering within each messaging destination. See “Message Destination Overview” on page 311 and “Using the Messaging Editor” on page 137 in the *Configuration Guide*.

**IMPORTANT** The event firing order is not pre-defined. Event order is the event name ordering within each destination.

After your messaging destination processes the `pendingrecode` event and submits message acknowledgements, it must call the special financials acknowledgement APIs on the original payment. Call the method `payment.acknowledgeRecode`. See in “Message Acknowledgement-based Status Transitions” on page 366 for more on these APIs. These change the status on the original payment from `pendingrecode` to `recoded`. This also updates the offset and onset payments to the `submitted` status.

**Note:** You only need to use this for recoding a payment, not for a regular payment associated with a check. For more information on submitting checks and check status changes, see “Check Integration” on page 372.

## Integration Payment Events for Check Transfer

ClaimCenter users can transfer a check from one claim to another. Catch this event by listening for the `CheckStatusChanged` event and check for the status value `pendingtransfer`. Similarly, you can listen for the `PaymentStatusChanged` event and check for the status value `pendingtransfer`.

At the time the status change event triggers, all the status changes and new payment creation has finished and must be considered final. To get values about the pre-transfer and post-transfer values and entities, use the domain methods on `Check` and `Payment` entities.

Your Event Fired rules that handle check transfers can use helpful domain methods on the `Payment` entity. See “Payment Integration Domain Methods (For Event Fired Rules)” on page 386. Use these methods to handle either `CheckStatusChanged` events or `PaymentStatusChanged` events.

After the destination’s messaging plugins send the associated message for the `pendingtransfer` event, the plugins get an acknowledgement back from the external system. After acknowledging the message itself, your messaging plugins must call a special acknowledgement API to complete this status transition: `check.acknowledgeTransfer()`. For more information, see “Message Acknowledgement-based Status Transitions” on page 366. If you use those APIs, the status on the original payment changes from `pendingtransfer` to `transferred`. This also updates the offset and onset payments to the `submitted` status.

## Payment Integration Domain Methods (For Event Fired Rules)

The following table lists payment object methods that Gosu exposes as properties. Use these properties in Event Fired rules to test what type of payment generated the event. Use the results to determine how to generate new messages to external systems.

In some cases if voiding and recoding a payment, ClaimCenter creates multiple Payment entities. To prevent duplicate messages, your rules that handle payment events such as PaymentAdded and PaymentStatusChanged must be very careful not to cause duplicate messages. To avoid this problem, check whether a payment is onset payment or is an offset payment.

Property to read	Description
payment.Onset	For a transferred or recoded payment, returns the new onset payment on the target (destination) claim.
payment.Offset	For a transferred or recoded payment, returns its offsetting payment.
Payment.OriginalPayment	If a payment is an onset payment that is the result of a check transfer or a payment recoding, this method returns the payment for which it is an onset. In other words, this method returns the transferred (or recoded) payment. If the payment is not an onset, returns null.
Payment.PaymentBeingOffset	For a payment that is acting as the offset for a transferred payment (or for a recoded payment), returns the payment being offset (the transferred or recoded payment). If the payment is not an offset, returns null.
check.TransferredCheck	If the check is the result of a transfer, this property returns the original check.
check.TransferredToCheck	If the check transferred, returns the new check that created on the target (destination) claim.
payment.TransferOffset	Returns true if the payment is the offsetting payment for a transferred payment; otherwise, returns false.
payment.TransferOnset	Returns true if the payment is the new payment created on the target claim for a transferred payment; otherwise, returns false
payment.Transferred	Tests whether this payment is transferring or transferred. In other words, returns true if its status is pendingtransfer or transferred
payment.RecodingOffset	Tests whether this payment is an offset payment that is the result of a payment recoding.
payment.RecodingOnset	Tests whether this payment is an onset payment that is the result of a payment recoding.

## Recovery Reserve Transaction Integration

The following table lists the status codes and meanings for recovery reserve transactions:

Recovery reserve status	Meaning
null	An internal initial status.
pendingapproval	The associated recovery reserve saved in ClaimCenter but is not yet approved.
rejected	The approver did not approve the associated recovery reserve.
submitting	The approver approved the recovery reserve and it is ready to be send to an external system.
submitted	The recovery reserve was sent to an external system and acknowledged.

To detect new or changed recovery reserves, you can write event business rules that listen for the `RecoveryReserveStatusChanged` event and check for changes to the `recoveryreserve.status` property. The following table includes the possible status code transitions and how this transition can occur.

Recovery reserve status	Can change to status	How it changes
null	→ <code>submitting</code>	New recovery reserves make this transition if they are auto-approved. This is the behavior in the reference implementation of ClaimCenter, but can be customized with approval rules.
	→ <code>pendingapproval</code>	New recovery reserves make this transition if they require approval. This is not the behavior in the reference implementation of ClaimCenter, but can be customized with approval rules.
<code>pendingapproval</code>	→ <code>submitting</code>	The approver approves the recovery reserve and no additional approval is necessary.
	→ <code>rejected</code>	The approver rejects the recovery reserve.
<code>rejected</code>	→ <code>object deleted</code>	A user deletes the recovery reserve in the user interface.
	→ <code>object deleted</code>	A user deletes the recovery reserve in the user interface.
<code>submitting</code>	→ <code>submitted</code>	ClaimCenter received an acknowledgement for the associated message for the <code>submitting</code> status. This transition requires you to call <code>transaction.acknowledgeSubmission()</code> in your message Ack code in your messaging plugins.
<code>submitted</code>	<i>no transitions possible</i>	<i>no transitions possible</i>

### Required Message Acknowledgements for Recovery Reserves

Message acknowledgements in some cases trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. The messaging plugin representing the messaging transport must call a special financials acknowledgement API for the status transition to complete. For recovery reserves, acknowledgement-driven status transitions apply in these cases:

- Recovery reserve status: `submitting` → `submitted`

For more information, see “Message Acknowledgement-based Status Transitions” on page 366

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309

## Recovery Transaction Integration

The following table lists the status codes and meanings for recovery transactions:

Recovery status	Meaning
<code>null</code>	An internal initial status.
<code>submitting</code>	The recovery was approved and ready is to send to an external system. In the reference implementation of ClaimCenter, there is no approval step for recoveries, but this can be customized with approval rules.
<code>submitted</code>	The recovery sent to an external system (and acknowledged).
<code>pendingvoid</code>	The check void request is ready to be sent to an external system.
<code>voided</code>	The void request was sent and acknowledged.

Recovery status	Meaning
pendingapproval	The recovery saved in ClaimCenter but is not yet approved. In the reference implementation of ClaimCenter, there is no approval step for recoveries, but this can be customized with approval rules.
rejected	The approver did not approve the check request.

To detect new or changed changes recoveries, you can write event business rules that listen for the `RecoveryStatusChanged` event and check for changes to the `recovery.Status` property. The following table includes the status codes and meanings for recovery reserve transactions: The following table includes the possible status code transitions and how this transition can occur.

Recovery status	Can change to status	How it changes
null	→ <code>submitting</code>	New recoveries make this transition if they are auto-approved. This is the behavior in the reference implementation of ClaimCenter, but can be customized with approval rules.
	→ <code>pendingapproval</code>	New recoveries make this transition if they require approval. This is not the behavior in the reference implementation of ClaimCenter, but can be customized with approval rules.
<code>submitting</code>	→ <code>submitted</code>	ClaimCenter received an acknowledgement for the associated message for the <code>submitting</code> status. This transition requires you to call <code>recovery.acknowledgeSubmission()</code> in your message Ack code in your messaging plugins.
	→ <code>pendingvoid</code>	A user attempts to void a recovery.
<code>submitted</code>	→ <code>pendingvoid</code>	A user attempts to void a recovery.
<code>pendingvoid</code>	→ <code>voided</code>	ClaimCenter received an acknowledgement for the associated message for the <code>pendingvoid</code> status. This transition requires you to call <code>recovery.acknowledgeVoid()</code> in your message Ack code in your messaging plugins.
<code>voided</code>	<i>no transitions possible</i>	<i>no transitions possible</i>
<code>pendingapproval</code>	→ <code>submitting</code>	The approver approves the recovery and no additional approval is necessary.
	→ <code>rejected</code>	The approver rejects the recovery
<code>rejected</code>	→ <code>object deleted</code>	A user deletes the recovery in the user interface.

### Required Message Acknowledgements for Recoveries

Message acknowledgements in some cases trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. Immediately after acknowledging the financials message itself, your messaging plugins must call a special financials acknowledgement API to complete the status transition. For recoveries, acknowledgement-driven status transitions apply in the following cases:

- Recovery status: `submitting` → `submitted`
- Recovery status: `pendingvoid` → `voided`

For more information, see “Message Acknowledgement-based Status Transitions” on page 366.

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309.

## Reserve Transaction Integration

To detect new or changed reserves, you can write event business rules that listen for the `ReserveStatusChanged` event and check for changes to the `reserve.status` property. The following table includes the status codes and meanings for reserve transactions:

Reserve status	Meaning
<code>null</code>	An internal initial status.
<code>submitting</code>	The approver approved the reserve and it ready is to be send to an external system. Or it is an offset (positive or negative) and the associated payment moved to the <code>submitting</code> status.
<code>submitted</code>	The reserve was sent to an external system (and acknowledged).
<code>pendingapproval</code>	The reserve saved in ClaimCenter but is pending approval.
<code>rejected</code>	The approver did not approve this reserve.
<code>awaitingsubmission</code>	The reserve is either a non-eroding offsetting reserve or a zeroing offsetting reserve for a payment. The payment that it is offsetting now awaits submission.

To detect new or changed reserves, you can write event business rules that listen for the `ReserveStatusChanged` event and check for changes to the `reserve.Status` property. The following table includes the possible status code transitions and how this transition can occur.

Reserve status	Can change to status	How it changes
<code>null</code>	→ <code>submitting</code>	New reserves make this transition if approval rules and authority limits indicate it does not require approval and the reserve is part of a reserve set (not a check set).
	→ <code>pendingapproval</code>	New reserves make this transition if they require approval due to approval rules and authority limits.
	→ <code>awaitingsubmission</code>	The reserve acts as either a non-eroding offsetting reserve or a zeroing offsetting reserve for a payment.
<code>submitting</code>	→ <code>submitted</code>	ClaimCenter received an acknowledgement for the associated message for the <code>submitting</code> status. This transition requires you to call <code>reserve.acknowledgeSubmission()</code> in your message Ack code in your messaging plugins.
<code>submitted</code>	<i>no transitions possible</i>	<i>no transitions possible</i>
<code>pendingapproval</code>	→ <code>submitting</code>	The approver approves the recovery and it does not require further approval.
	→ <code>rejected</code>	The approver rejects the reserve.
<code>rejected</code>	→ <code>object deleted</code>	A user deletes the recovery in the user interface.
	→ <code>submitted</code>	A user deletes the recovery in the user interface.
<code>awaitingsubmission</code>	→ <code>submitting</code>	If the reserve is a non-eroding offsetting reserve or zeroing offsetting reserve and the payment that it offsets transitions to the <code>submitting</code> state.
	→ <code>object deleted</code>	Deletion automatically occurs if something deletes the associated payment or if the reserve payment changes such that the offsetting reserve is unnecessary. Examples of the latter case: <ul style="list-style-type: none"> <li>• A non-eroding payment changes to be an eroding payment.</li> <li>• A current day payment that exceeds reserves has its owning check rescheduled to be a future check.</li> <li>• The payment amount changes such that it no longer exceeds reserves, or exceeds reserves by a different amount.</li> </ul>

### Required Message Acknowledgements for Reserves

Message acknowledgements in some cases trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. Immediately after acknowledging the financials message itself, your messaging plugins must call a special financials acknowledgement API to complete the status transition. For reserves, acknowledgement-driven status transitions apply in the following cases:

- Reserve status: `submitting` → `submitted`

For more information, see “Message Acknowledgement-based Status Transitions” on page 366.

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309.

## Bulk Invoice Integration

Use bulk invoices to create and submit a list of payments for many claims quickly without using the New Check wizard user interface.

This can simplify an insurance company’s relationship with some large vendors or partners. For example, suppose an auto insurance company insures a large rental car company. At the end of the month, the rental car company could invoice the insurance company for all payments owed. The insurance company could use the ClaimCenter user interface to quickly enter payments by claim number, validate payment amounts and claim numbers, then finally submit payments for approval and processing.

The user interface provides rows of text properties for quick entry of each payment on the invoice. Because there is lots of data, users can make and save draft versions of an invoice that persist to the ClaimCenter database before submitting the final version. This is a feature that the regular check wizard in ClaimCenter does not have.

ClaimCenter also provides an optional web service (`BulkInvoiceAPI`) that an external system can use to submit bulk invoices to ClaimCenter. This avoids entirely the need to use the web user interface to submit bulk invoices.

### Bulk Invoice Validation

If you enter data into the **Bulk Invoice** screen and you are ready to complete the bulk invoice, click the **Submit** button to validate the invoice. This validate step ensures that the bulk invoice does not violate your business logic requirements for bulk invoices.

**Note:** Bulk invoice *validation* is separate from bulk invoice *approval*. In general, validation determines whether the data appears to be correct, such as checking whether only certain vendors can submit any bulk invoices. In contrast, submitted invoices go through the approval process with business rules or human approvers to approve or deny an invoice before final processing.

Some types of validation happen automatically. Do not duplicate the validation logic for the following items in your validation plugin (`IBulkInvoiceValidationPlugin`):

- **Basic claim number validity** – In the user interface, the user interface code checks basic claim number validity. The web service method `BulkInvoiceAPI` method `addItemsToInvoice` verifies this automatically.
- **Claim and exposure validation levels** – In the user interface, the user interface code checks claim and exposure validation levels to see if they reach the validation level Ability to Pay. The web service method `BulkInvoiceAPI` method `addItemsToInvoice` verifies this automatically.
- **Check if payments exceed reserves** – In the user interface, the user interface code checks whether payments exceed reserves for those claims and exposures. The web service `BulkInvoiceAPI` method `submitBulkInvoice` verifies this automatically.

After a bulk invoice passes validation successfully, users can submit the invoice for approval and final processing. Final processing includes system-level validation tests on each approved invoice item, followed by creation of an individual check for each invoice item. ClaimCenter creates these checks as placeholders for accounting purposes on each affected claim. Bulk invoices have their own approval rule set called Bulk Invoice Approval. For details, see “[BulkInvoice Approval Rule Set Category](#)” on page 45 in the *Rules Guide*.

A plugin interface called `IBulkInvoiceValidationPlugin` controls bulk invoice validation. Because the logic of this particular plugin usually is simple and typically does not need to connect to an external system, Guidewire recommends implementing this plugin in Gosu.

ClaimCenter provides a demonstration bulk invoice validation plugin implementation in Gosu. Use this plugin implementation for testing purposes only. You can access the sample Gosu plugin by navigating in Studio in the Resources pane to `Classes → gw → plugin → bulkinvoice → impl`.

Your own implementation of the bulk invoice validation plugin must implement the `IBulkInvoiceValidationPlugin` interface, which contains one simple method:

```
public BIValidationAlert[] validateBulkInvoice(BulkInvoice invoice)
```

In this method, run your own validations on the `BulkInvoice` parameter. Call scriptable domain methods of the bulk invoice entity to access or calculate whatever necessary to test validity:

- If all tests pass, this method must return `null` or an empty array.
- If one or more tests fail, construct one or more instances of the error message entity `BIValidationAlert` and return all the instances in a single array. Each instance must have an alert type and a message indicating the reason for the test failure. The default type is `Unspecified`. However, this method can use a more specific type in the extendible typelist `BIValidationAlertType`. The user interface PCF files can use the alert type of any alerts to generate different behaviors compared to the alert type `Unspecified`.

If ClaimCenter receives `null` or an empty array as the return value for the `validateBulkInvoice` method, ClaimCenter marks the bulk invoice as Valid. Otherwise, ClaimCenter marks the bulk invoice as Not Valid and commits the returned `BIValidationAlerts` to the database so the Bulk Invoice details page can display alerts.

If you do not register a bulk invoice validation plugin and you click the `Validate` button in the bulk invoice user interface, ClaimCenter immediately marks the bulk invoice as Valid.

## Bulk Invoice Web Service APIs

The ClaimCenter bulk invoice (`BulkInvoiceAPI`) web service allows external systems to submit bulk invoices directly. For example, an associated rental car company could directly submit bulk invoices to ClaimCenter from other systems using these web service APIs. `BulkInvoiceAPI` methods can create and submit `BulkInvoice` objects, as well as add, update, and delete `BulkInvoiceItem` objects. Additionally, other methods do things like void, stop, and place holds on a bulk invoice.

This web service uses Data Transfer Objects to represent entity data. See “[Claim Financials Web Service Data Transfer Objects \(DTOs\)](#)” on page 368. In this case, the main DTO objects are Gosu classes called `BulkInvoiceDTO` and `BulkInvoiceItemDTO`.

---

**WARNING** Do not call the SOAP APIs from inside ClaimCenter code to the same computer. From your rules code or plugin code, instead call the local Gosu APIs of any APIs.

---

Many returned entities contain foreign key IDs rather than direct subobject links to associated entities. For example, calling any of the methods with names that begin with `getItems` return an array of bulk invoice item DTO (`BulkInvoiceItemDTO`) objects. Each bulk invoice item DTO has a `BulkInvoiceID` property, which is the public ID of the bulk invoice (`BulkInvoice`) entity instance that contains it. To get a reference to the `BulkInvoice` entity instance to read its properties, make a separate call to `BulkInvoiceAPI` method `getBulkInvoice`.

Similarly, each bulk invoice item links to the related claim and exposure with public ID values in the properties `ClaimID` and `ExposureID`.

### Add Invoice Items

To add items to an invoice, from an external system call the `BulkInvoiceAPI` method `addItemsToInvoice`. The method takes as arguments an invoice public ID (a `String`) and an array of bulk invoice item DTOs (`BulkInvoiceItemDTO[]`). The method returns an array of public IDs for the newly-added `BulkInvoiceItem` objects.

The following `BulkInvoiceItemDTO` properties are required at minimum:

- `Amount`, but only if the property `BulkInvoice.SplitEqually` is set to `false`
- `ClaimID`
- `CostCategory`
- `CostType`
- `PaymentType`

If any `BulkInvoiceItemDTO` object has invalid data, the method throws an exception, in which case no items add to the invoice.

### Get Bulk Invoice

To get a bulk invoice from an external system, call the `BulkInvoiceAPI` method `getBulkInvoice`. The method takes invoice public ID (a `String`) and returns a bulk invoice DTO (`BulkInvoiceDTO`).

### Get Invoice Items

To get invoice items, there are multiple `BulkInvoiceAPI` methods you can call from an external system:

Method name	Description
<code>getItemsForInvoice</code>	Gets all invoice items on a specific invoice.
<code>getItemsForInvoiceAndClaim</code>	Gets invoice items on a specific invoice, filtered by a specific claim public ID ( <code>String</code> )
<code>getItemsForInvoiceAndClaimAndAmount</code>	Gets invoice items on a specific invoice, filtered by a specific claim public ID ( <code>String</code> ) and amount ( <code>BigDecimal</code> ) of the invoice items

All these methods return an array of bulk invoice item DTO objects (`BulkInvoiceItemDTO[]`).

### Create Bulk Invoice

To create a bulk invoice, from an external system, call the `BulkInvoiceAPI` method `createBulkInvoice`. Populate a `BulkInvoiceDTO` object and pass it as a method argument. The following `BulkInvoiceDTO` properties are required at minimum:

- The payee property `PayeeID` on the bulk invoice
- `BulkInvoiceTotal`, but only if the property `BulkInvoice.SplitEqually` is set to `true`

The new bulk invoice has the status `draft`.

Adding invoice items is optional. Each invoice item must have all the required properties mentioned earlier for the `addItemsToInvoice` method. Additionally, be aware that if the server is configured in multicurrency mode, ClaimCenter selects a default market rate for the property `TransToReportingExchangeRate`. If you wish to provide your own custom rates, provide appropriate values for properties `NewExchangeRate` and `NewExchangeRateDescription` on the corresponding `BulkInvoiceDTO` object.

For more examples and more instructions, see “Typical Bulk Invoice API Usage” on page 395.

### Update Items on Invoice

To update invoice items on a bulk invoice, from an external system call the `BulkInvoiceAPI` method `updateItemsOnInvoice`. Pass an invoice public ID, and an array of invoice item DTOs that represent items to change.

Each invoice item must at minimum have the `PublicID` property to identify the item to change. Additionally, set additional properties that you want to change, such as `Amount`.

By default, all `null` property values are ignored. As a consequence, by default you cannot set a value to `null` using this web service method. However, this is configurable. Go to the WS-I web service implementation class. At the end of the `updateItems` method, find the line:

```
itemDTO.writeTo(itemFromDB)
```

Change it to say:

```
itemDTO.writeTo(itemFromDB, false)
```

After this change, all values are set on the entity instance, including `null` values.

**WARNING** If you make the change to reverse the behavior of `null` values, be careful to always populate all invoice items properties to prevent data loss.

### Delete Items

To delete invoice items on a bulk invoice from an external system, call the `BulkInvoiceAPI` method `deleteItemsFromInvoice`. Pass an invoice public ID, and an array of invoice item public IDs (`String[]`) for the items to delete.

### Validate an Invoice

To validate invoice items on a bulk invoice, from an external system, call the `BulkInvoiceAPI` method `validateBulkInvoice`. Pass the invoice public ID as an argument. Any validation failure messages are returned in an array of `BIValidationAlertDTO` objects. If successful, the method returns an empty array. ClaimCenter also attaches and persists validation errors with the bulk invoice. Refer to the property `BulkInvoice.ValidationAlerts`, which contains an array of `BIValidationAlert` entity instances.

### Submit an Invoice

To submit an invoice for approval, from an external system, call the `BulkInvoiceAPI` method `submitBulkInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of `Draft`, `Rejected`, `OnHold`, or `PendingItemValidation`. The invoice must have at least one `BulkInvoiceItem`.

For more examples and more instructions, see “Typical Bulk Invoice API Usage” on page 395.

### Request Downstream Escalation

To request downstream escalation for an invoice, from an external system, call the `BulkInvoiceAPI` method `requestBulkInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of `AwaitingSubmission`.

This API is only for highly unusual situations. Typically, ClaimCenter handles escalation automatically using the scheduled `bulkinvoicesescalation` batch process. Use this API only if you want to manually escalate a `BulkInvoice` without waiting for the batch process to run.

### Void Bulk Invoice

To void a bulk invoice from an external system, call the `BulkInvoiceAPI` method `voidBulkInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of `Requesting`, `Requested`,

Issues, Cleared, or OnHold. This method sets the invoice and all its items to status PendingVoid. The method also voids any related bulk checks.

#### Stop Bulk Invoice

To stop a bulk invoice from an external system, call the BulkInvoiceAPI method `stopBulkInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of Requesting, Requested, Issues, or OnHold. This method sets the invoice and all its items to status PendingStop. The method also stops any related bulk checks.

#### Place Downstream Hold On Invoice

To place a hold on a bulk invoice from an external system, call the BulkInvoiceAPI method `placeDownstreamHoldOnInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of Requesting, Requested. This method sets the invoice to status OnHold.

#### Update Invoice Status

To update invoice status from an external system, call the BulkInvoiceAPI method `updateBulkInvoiceStatus`. You can optionally update the check number and issue date.

Possible status transitions and requirements:

- Issued – old status must be Requesting, Requested, PendingVoid, PendingStop, Voided, or Stopped.
- Cleared – old status must be Requesting, Requested, Issued, PendingVoid, PendingStop
- Voided – old status must be PendingVoid or PendingStop
- Stopped – old status must be PendingVoid or PendingStop

As arguments to method:

- the invoice public ID
- optional new check number (pass `null` to ignore)
- optional new issue date of the bulk check (pass `null` to ignore)
- new status

---

**IMPORTANT** The BulkInvoiceAPI web service method `updateBulkInvoiceStatus` does not update the status of the placeholder checks. If you call `updateBulkInvoiceStatus`, you must immediately also call the ClaimFinancialsAPI web service method `updateCheckStatus` to update the status of each check.

---

### Typical Bulk Invoice API Usage

Suppose you want to create a new bulk invoice and submit it.

The following is a typical workflow for your integration code:

1. Populate a new BulkInvoiceDTO object to represent the new bulk invoice.
2. Set the following properties on the BulkInvoiceDTO object:
  - CheckNumber – The check number
  - RequestingUser – The requesting user
  - SplitEqually – (Boolean) Whether to split items equally. If `SplitEqually` has the value `true`, you must also set the `BulkInvoiceTotal` property on the DTO.
  - PublicID – Bulk invoice public ID
  - InvoiceNumber – Invoice number

- `InvoiceItems` – Invoice items (see following note).

---

**IMPORTANT** Guidewire recommends you add invoice items directly on the DTO when you create the bulk invoice DTO. It is not strictly required to add the invoice items at this step. You could choose to add the invoice items separately at a later step.

---

3. Carefully set the `PayeeID` property on the `BulkInvoiceDTO` object. Set `PayeeID` to the public ID for the payee contact entity. Using Bulk Invoices requires using a contact system and the payees must be in the contact system, such as ContactManager. For the `PayeeID` property, ClaimCenter needs the public ID of the contact. To safely get the public ID:
  - a. Ensure that the contact is in the contact system. If you use ContactManager, use the ContactManager web services `ABContactAPI` to add the contact. Whether you are using an already-existing contact or creating a new one, remember to save the *address book link ID*. An address book link ID is the native address book ID for the contact. It is critical to recognize that this is different from the public ID.
  - b. Call the ClaimCenter web service API that takes the link ID and returns a contact entity: the `BulkInvoiceAPI` method `createContactByLinkId`.
  - c. Get the public ID from the result of that method, and set the invoice `PayeeID` property to that value.
4. Call the `createBulkInvoice` method with your `BulkInvoiceDTO` object to create a new draft unapproved bulk invoice in the database.
5. If you did not add the invoice items yet, use the `addItemsToInvoice` method to add bulk invoice items on the bulk invoice.
6. Validate the bulk invoice with the `validateBulkInvoice` method
7. If there were no validation errors, call the `submitBulkInvoice` method to submit the bulk invoice.

After creating the bulk invoice, you can optionally call the `updateItems` or `updateBulkInvoiceStatus` to change the existing data.

## Post-submission Actions On the Bulk Invoice

The bulk invoice lifecycle is not necessarily complete after it submits to a downstream system. Possible updates can occur either through the ClaimCenter user interface, or through web services.

### Updating the Bulk Invoice to Issued or Cleared

Typically, after paying the bulk payment associated with the invoice, you want to update the business status of the invoice to `issued` and then `cleared`. You can use multiple APIs to do this, depending on whether you are calling from an external system or from your own messaging plugins. For more information, see “Bulk Invoice Status Changes Using SOAP or Domain Method” on page 404

### Downstream System Holds on a Bulk Invoice

Occasionally, there may be problems with a bulk invoice that can only be detected on your server after submission. For this reason, you can place a hold on a submitted bulk invoice. Do this with the `BulkInvoiceAPI` method `placeDownstreamHoldOnInvoice` method. Refer to the Java API Javadoc Documentation for more information about this method.

After a downstream hold is on an invoice, only the following actions are possible:

- You can void the bulk invoice
- You can stop the bulk invoice
- You can resubmit the bulk invoice as is

### Stopping a Bulk Invoice

You can stop a bulk invoice if it is considered stoppable by the system. Stopping a bulk invoice has the following effects:

- The status of the invoice transitions to `pendingstop`.
- For every invoice item that has an associated check, the invoice item status changes to `pendingstop`.
- ClaimCenter stops all associated checks. This is the same effect as stopping the checks individually through the user interface.

Once a bulk invoice transitions to `pendingstop`, it can only transition to `stopped` status using a call to the `BulkInvoiceAPI` method `updateBulkInvoiceStatus`. Any attempt to use this method to transition a bulk invoice to `stopped` if it has not already stopped (through the user interface or the API) results in an error.

If the stop on the bulk invoice is unsuccessful, use the method `updateBulkInvoiceStatus` on the bulk invoice in Gosu, or with the web service `BulkInvoiceAPI`. Those methods transition the invoice back to `issued` or `cleared` status with the following effects:

- The status of the invoice transitions to either `issued` or `cleared` status respectively.
- Every bulk invoice item that has an associated check transitions back to `submitted` status.
- Every associated check is un-stopped. This is the same behavior as if a regular un-bulked check stops but then the stop attempt fails.

The `BulkInvoiceAPI` web service method `updateBulkInvoiceStatus` does not update the status of the placeholder checks. If you call `updateBulkInvoiceStatus`, you must immediately also call the `ClaimFinancialsAPI` web service method `updateCheckStatus` to update the status of each check.

### Voiding a Bulk Invoice

You can void a bulk invoice if it is considered voidable by the system. Voiding a bulk invoice has the following effects:

- The status of the invoice transitions to `pendingvoid`.
- Every invoice item that has an associated check transitions to status `pendingvoid`.
- ClaimCenter voids all associated checks. This has the same effect as voiding the checks individually through the user interface.

Once a bulk invoice transitions to `pendingvoid`, it can only transition to `voided` status using one of the following:

- A call from an external system to the `BulkInvoiceAPI` method `updateBulkInvoiceStatus`. For important warnings about usage and limitations with calling SOAP APIs, refer to “How Do Status Transitions Happen?” on page 364.
- A call from your messaging plugins to the bulk invoice domain method `updateBulkInvoiceStatus`. For important warnings about usage and limitations with calling status-updating domain methods, refer to “How Do Status Transitions Happen?” on page 364.

In both cases, the bulk invoice must already be at the status `pendingvoid`. The `updateBulkInvoiceStatus` method (in either variant) throws an exception if the status is not `pendingvoid`.

If the call to `updateBulkInvoiceStatus` to void a bulk invoice fails for any reason, you can use the `updateBulkInvoiceStatus` method to transition the invoice to a `issued` or `cleared` status. This has the following effects:

- The status of the invoice transitions to either `issued` or `cleared` status respectively.
- Every bulk invoice item that has an associated check transitions back to `submitted` status.
- Every associated check is un-voided. This is the same behavior as if a regular un-bulked check voided but then the void attempt failed.

The BulkInvoiceAPI web service method `updateBulkInvoiceStatus` does not update the status of the placeholder checks. If you call `updateBulkInvoiceStatus`, you must immediately also call the ClaimFinancialsAPI web service method `updateCheckStatus` to update the status of each check.

## Bulk Invoice Processing Performance

Bulk invoices printed on paper (recorded by hand) usually have few line items. However, for automated creation and processing of bulk invoices using web services, incoming electronic invoices might be large. For example, a large favored supplier (for rental cars, medical billing adjustment services, and so on) might have thousands of line items. To handle one such huge electronic invoice, they may need to create multiple bulk invoices in ClaimCenter. Converting extremely large bulk invoices into smaller ClaimCenter invoices might be necessary to create, fix, and finally submit the bulk invoices in a reasonable amount of time.

Bulk invoice processing performance varies greatly. Performance depends on factors that include but are not limited to the following:

- Server hardware
- Rule Set code
- Server configuration settings

Before you deploy your final production configuration of your ClaimCenter server, you must test your ClaimCenter configuration using the same hardware as your production server. Use either the same physical hardware or an exact copy. You must test performance for submitting a Bulk Invoice. Experiment to determine your settings for the following settings:

- **Maximum time per invoice** – The acceptable amount of time for ClaimCenter to process and fully submit any bulk invoice. As part of submitting, ClaimCenter might mark some bulk invoice items as Not Valid. You must plan on allocating necessary time to manually edit some items to fix issues and resubmit bulk invoices.
- **Maximum items per invoice** – Based on the maximum amount of time per invoice (see previous item), establish the maximum number of items on each bulk invoice. This value will depend on your particular combination of server hardware, rule code, and server configuration.

Depending on the volume of invoices, you might need to design special procedures to maximize performance from large vendors. For example, you may want to do SOAP API handling and inevitable manual cleanup and resubmission on a dedicated ClaimCenter server in the cluster. That way, this process minimally affects regular ClaimCenter users.

It is important to distinguish Bulk Invoice Item processing from Bulk Invoice Escalation. The performance challenges are with bulk invoice item processing. Bulk invoice processing happens immediately after bulk invoice approval. The bulk invoice item processing step creates a placeholder check for each bulk invoice item on each item's claim. By the end of the process, ClaimCenter creates and approves all checks and the bulk invoice has the status `AwaitingSubmission`.

The following steps occur for submitting a bulk invoice:

1. The user clicks the **Submit** button or an external system calls the `submitBulkInvoice` Soap API method.
2. If the bulk invoice passes Bulk Invoice Approval rules, proceed to the next step in this list. However, it may be that the Bulk Invoice Approval specifies additional approval is necessary. Eventually, a supervisor signs in and approves the Bulk Invoice Approval activity. At this point, no further approvals are necessary.
3. After final approval, bulk invoice item processing starts on that same machine.
4. If everything is valid, no checks require approval by TransactionSet Approval Rules, so the bulk invoice has the status `AwaitingSubmission`. ClaimCenter now creates the checks and runs all rules on them. This step is what takes so long during bulk invoice processing.
5. ClaimCenter starts bulk invoice processing by adding the bulk invoice to a single task queue (`TaskQueue`) per machine. Each task queue processes each bulk invoice and all its items in order in a single thread. This

process starts on the same machine on which Bulk Invoice Approval rules run. This thread is not a ClaimCenter batch process.

This is a single thread and not a batch process. Because of this, Guidewire recommends for large vendors to perform bulk invoice operations on a separate machine. This includes both the web service APIs and the user actions.

Whether bulk invoice processing begins due to the user interface **Submit** button or the resolution of final approval, bulk invoice processing always takes place on that same machine. Thus, it is best to segregate these tasks to a separate machine so that it does not interfere with the CPU, disk, or network resources of regular application web users.

6. After the bulk invoice reaches the status `AwaitingSubmission`, the Bulk Invoice Escalation batch process finds the bulk invoice. Specifically, the escalation batch process finds bulk invoices in status `AwaitingSubmission` that reached their scheduled send date. The batch process moves any such bulk invoices to the status `Requesting`. The batch process sets the status of all the related bulk invoice items and their placeholder checks to the status `Requesting`. This status change happens mainly to update the status values and create an opportunity to create event messages in Event Fired rules. Like all message sending code, the message processing happens asynchronously as part of the send queue processing on the batch server. The asynchronous messaging sending is typically not performance intensive.

## Bulk Invoice (Top Level Entity) Status Transitions

The following table lists the status codes and meanings for bulk invoice transactions:

Bulk invoice status	Meaning
<code>null</code>	An internal initial status for a new <code>BulkInvoice</code> entity.
<code>draft</code>	Status for initial editing of an invoice, or after invalidation due to changes.
<code>inreview</code>	Bulk Invoice requires approval and awaits action by the assigned approver.
<code>pendingbulkinvoiceitemvalidation</code>	The approver approved the bulk invoice. Individual bulk invoice items are now pending validation and undergoing final processing.
<code>invalidbulkinvoiceitems</code>	One or more invoice items failed validation or failed during check creation, or the check associated with one or more items submitted for approval and then the approver rejected it.
<code>awaitingsubmission</code>	Bulk invoice awaits submission to the downstream system.
<code>requesting</code>	The bulk invoice queued for submission to the downstream system.
<code>requested</code>	The bulk invoice sent successfully to the downstream system.
<code>issued</code>	The bulk invoice's associated bulk check issued.
<code>cleared</code>	The bulk invoice's associated bulk check cleared.
<code>rejected</code>	The assigned approver rejected the bulk invoice.
<code>pendingvoid</code>	The bulk invoice voided, Confirmation of the void is pending.
<code>pendingstop</code>	The bulk invoice stopped. Confirmation of the void is pending.
<code>voided</code>	The bulk invoice successfully voided.
<code>stopped</code>	The bulk invoice successfully stopped.
<code>onhold</code>	The downstream system placed the bulk invoice on hold.

To detect new or changed reserves, you can write event business rules that listen for the `BulkInvoiceStatusChanged` event and check for changes to the `bulkinvoice.Status` property. The following table includes the possible status code transitions and how this transition can occur.

<b>Bulk invoice status</b>	<b>Can change to status</b>	<b>How it changes</b>
null	→ draft	New BulkInvoice entities created.
draft	→ inreview → pendingbulkinvoiceitemvalidation	This transition occurs if both of the following occur: <ul style="list-style-type: none"><li>• A bulk invoice submits either through the ClaimCenter user interface or using the BulkInvoiceAPI method <code>submitBulkInvoice</code>.</li><li>• The defined bulk invoice approval rules require it to escalate for approval.</li></ul>
		This transition occurs if both of the following occur: <ul style="list-style-type: none"><li>• A bulk invoice submits either through the ClaimCenter user interface or using the BulkInvoiceAPI method <code>submitBulkInvoice</code>.</li><li>• The defined bulk invoice approval rules do not require it to escalate for approval.</li></ul>
inreview	→ pendingbulkinvoiceitemvalidation → rejected	This transition occurs if you approve a bulk invoice through the ClaimCenter user interface.  This transition occurs if you reject a bulk invoice through the ClaimCenter user interface.
	→ draft	This transition occurs if you edit the bulk invoice in such a way that it no longer passes validation.
pendingbulkinvoiceitemvalidation	→ awaitingsubmission → invalidbulkinvoiceitems	This transition occurs after all approved bulk invoice items successfully validate, and a check automatically creates for them and the approver approves the check.  This transition occurs if the processing of the bulk invoice's items complete and one or more items are now <code>notvalid</code> or the approver rejects one or more associated checks.
invalidbulkinvoiceitems	→ pendingbulkinvoiceitemvalidation → draft	This transition occurs if the bulk invoice is resubmitted without first changing such that it becomes <code>draft</code> status again and does not require approval.  This transition occurs if you edit the bulk invoice in such a way that it no longer passes validation.
	→ inreview	This transition occurs if the bulk invoice resubmits and requires approver (and if it does not become <code>draft</code> status).
awaitingsubmission	→ draft → requesting	This transition occurs if you edit the bulk invoice such that it no longer passes validation.  This transition occurs for bulk invoices with a scheduled send date of the current day if the <code>bulkinvoicesescalation</code> batch process runs. The process runs either automatically at a scheduled time or manually from the command-line. Alternatively, the SOAP APIs can escalate the bulk invoice.

Bulk invoice status	Can change to status	How it changes
requesting	→ requested	ClaimCenter received an acknowledgement for the associated message for the submitting status. This transition requires you to call <code>bulkInvoice.acknowledgeSubmission()</code> in your message Ack code in your messaging plugins.
	→ issued	This transition can occur through the calling <code>updateBulkInvoiceStatus</code> method from SOAP API or domain method to change to status issued.
	→ cleared	This transition can occur through the calling <code>updateBulkInvoiceStatus</code> method from SOAP API or domain method to change to status cleared.
	→ pendingstop	This transition occurs if the bulk invoice stops either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>stopBulkInvoice</code> .
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>voidBulkInvoice</code> .
requested	→ onhold	This transition occurs if the Bulk Invoice API places a downstream hold on the invoice, using a call to the <code>placeDownstreamHoldOnInvoice()</code> method.
	→ issued	This transition can occur through the calling <code>updateBulkInvoiceStatus</code> method from SOAP API or domain method to change to status issued.
	→ cleared	This transition can occur through the calling <code>updateBulkInvoiceStatus</code> method from SOAP API or domain method to change to status cleared.
	→ pendingstop	This transition occurs if the bulk invoice stops either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>stopBulkInvoice</code> .
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>voidBulkInvoice</code> .
cleared	→ onhold	This transition occurs if the Bulk Invoice API places a downstream hold on the invoice, using a call to the <code>placeDownstreamHoldOnInvoice()</code> method.
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>voidBulkInvoice</code> .
issued	→ cleared	This transition can occur through the calling <code>updateBulkInvoiceStatus</code> method from SOAP API or domain method to change to status cleared.

Bulk invoice status	Can change to status	How it changes
	→ pendingstop	This transition occurs if the bulk invoice stops either through the ClaimCenter user interface, or using a call to the BulkInvoiceAPI method stopBulkInvoice.
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the BulkInvoiceAPI method voidBulkInvoice.
rejected	→ draft	This transition occurs if you edit the bulk invoice in such a way that it no longer passes validation.
	→ inreview	This transition occurs if both: <ul style="list-style-type: none"> <li>the bulk invoice resubmits either through the ClaimCenter user interface or using the BulkInvoiceAPI method submitBulkInvoice</li> <li>bulk invoice approval rules require escalation for approval.</li> </ul>
	→ pendingbulkinvoiceitemvalidation	This transition occurs if both: <ul style="list-style-type: none"> <li>the bulk invoice resubmits in the user interface or through the BulkInvoiceAPI method submitBulkInvoice</li> <li>the bulk invoice approval rules do not require it to escalate for approval.</li> </ul>
pendingvoid	→ voided	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status voided.
	→ stopped	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status stopped.
	→ issued	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status issued. This means the void attempt was unsuccessful.
	→ cleared	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status cleared. This means the void attempt was unsuccessful.
pendingstop	→ stopped	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status stopped.
	→ voided	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status voided.
	→ issued	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status issued. This means the stop attempt was unsuccessful.

Bulk invoice status	Can change to status	How it changes
	→ cleared	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status cleared. This means the stop attempt was unsuccessful.
stopped	→ issued	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status issued. This means the stop attempt was unsuccessful.
	→ cleared	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status cleared. This means the stop attempt was unsuccessful.
voided	→ issued	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status issued. This means the void attempt was unsuccessful
	→ cleared	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status cleared. This means the void attempt was unsuccessful.
onhold	→ pendingstop	This transition occurs if the bulk invoice stops either through the ClaimCenter user interface, or using a call to the BulkInvoiceAPI method stopBulkInvoice.
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the BulkInvoiceAPI method voidBulkInvoice.
	→ requesting	This transition occurs if the invoice resubmits either through the ClaimCenter user interface or using the BulkInvoiceAPI method submitBulkInvoice.

### Required Message Acknowledgements for Bulk Invoices

Message acknowledgements in some cases trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. Immediately after acknowledging the financials message itself, your messaging plugins must call a special financials acknowledgement API to complete the status transition. For bulk invoices, acknowledgement-driven status transitions apply in the following cases:

- Bulk invoice status: `requesting` → `requested`

For more information, see “Message Acknowledgement-based Status Transitions” on page 366

After you acknowledge the submission message, the following updates occur:

1. The bulk invoice’s status changes to `requested`.

2. The status of every invoice items with a current status of **submitting** changes to **submitted**.

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309.

### Bulk Invoice Status Changes Using SOAP or Domain Method

Typically, after paying the bulk payment associated with the invoice, you want to update the business status of the invoice to **issued** and then **cleared**. You can use either of the following approaches:

- From an external system, call the SOAP API `BulkInvoiceAPI` method `submitBulkInvoice`. The method `updateBulkInvoiceStatus` does not update the status of the placeholder checks. If you call `updateBulkInvoiceStatus`, you must immediately also call the `ClaimFinancialsAPI` web service method `updateCheckStatus` to update the status of each check.
- From your messaging code, call the bulk invoice domain method `updateBulkInvoiceStatus`. The `updateBulkInvoiceStatus` method does not update the status of the placeholder checks. If you call `updateBulkInvoiceStatus`, you must immediately also call the `check.updateCheckStatus(status)` method to update the status of each check.

This method allows you to set the bulk check number and the issue date on the bulk invoice. For important information and warnings about both approaches, see “How Do Status Transitions Happen?” on page 364.

The `updateBulkInvoiceStatus` method can make these changes:

- Update a bulk invoice from **requested** to **issued** status, which also sets the Issued Date. Typically this occurs after the bulk payment is made from the insurer to the vendor who submitted the bulk invoice.
- Update a bulk invoice to **cleared** status.
- Update a **pendingvoid** bulk invoice to **voided** status.
- Update a **pendingstop** bulk invoice to **stopped** status.
- Update a bulk invoice with a canceled status (the statuses **pendingstop**, **stopped**, **pendingvoid**, **voided**) to the **issued** or **cleared** status. This effectively means that the void or stop attempt failed.

From your messaging plugin, you can call the `bulkInvoice.updateBulkInvoiceStatus(....)` method to change the status, and only after your messaging code acknowledges a message. For sample code and additional important warnings and limitations, see “How Do Status Transitions Happen?” on page 364.

If a check is **pendingstop** or **pendingvoid** and the new status is **issued** or **cleared**, the status values of the check and its related payments change to the new value. (This is true both for changes from SOAP APIs or domain methods.) Next, ClaimCenter creates a warning activity. Next, ClaimCenter assigns the activity to the user who attempted to void or stop the check. This activity lets the user know that the check did not stop/void successfully. Finally, ClaimCenter generates any reserve that is necessary to keep open reserves from becoming negative.

### Bulk Invoice Item Status Transitions

The following table lists the status codes and meanings for bulk invoice item transactions:

Bulk invoice item status	Meaning
<code>null</code>	An internal initial status for a new <code>BulkInvoiceItem</code> entity not yet committed to the database.
<code>draft</code>	Bulk invoice item commits to the database but its owning invoice it is not yet submitted for approval. This is the initial status for every new invoice item.
<code>approved</code>	The approver approved the bulk invoice item and is valid for processing.

Bulk invoice item status	Meaning
rejected	The assigned bulk invoice approver did not approve the bulk invoice item and ClaimCenter must not process it further.
submitting	Bulk invoice item is pending submission to the downstream system.
submitted	The bulk invoice item successfully submitted to the downstream system
inreview	Bulk invoice item requires action before it can be paid. This is the status during bulk invoice approval.
notvalid	The bulk invoice item failed validation or a problem occurred while creating its associated check.
pendingvoid	The bulk invoice item's owning bulk invoice voided and confirmation of the void is pending.
pendingstop	The bulk invoice item's owning bulk invoice stopped and confirmation of the void is pending.
voided	The bulk invoice item's owning bulk invoice successfully voided.
stopped	The bulk invoice item's owning bulk invoice successfully stopped.
pendingtransfer	The bulk invoice item's associated check is pending transfer.
transferred	The bulk invoice item's associated check successfully transferred.

To detect new or changed reserves, you can write event business rules that listen for the `BulkInvoiceItemStatusChanged` event and check for changes to the `bulkinvoiceitem.Status` property. The following table includes the possible status code transitions and how this transition can occur:

<b>Bulk invoice item status</b>	<b>Can change to status</b>	<b>How it changes</b>
null	→ draft	A new BulkInvoiceItem
draft	→ approved	This transition occurs in a few cases: (1) Upon submit of the owning bulk invoice if the defined bulk invoice approval rules do not require escalation for approval. (2) if the bulk invoice is in review and the assigned approver either approves the entire bulk invoice, or does not mark the invoice item as rejected or inreview.
	→ rejected	This transition occurs if the bulk invoice is In review and either: (1) The approving user rejects the entire bulk invoice. (2) The approving user approves the bulk invoice, but specifically marks this invoice item as rejected on the approval activity details worksheet.
	→ inreview	This transition occurs if the bulk invoice is In review and the approving user approves the bulk invoice, but specifically marks this invoice item as inreview
approved	→ draft	This transition occurs if the bulk invoice item changes in such a way that the owning bulk invoice no longer passes validation.
	→ submitting	This transition occurs if the owning bulk invoice escalates for submission to the downstream system.
	→ notvalid	This transition occurs if an approved bulk invoice item fails a system validation check during the final processing phase. Also occurs if there is a problem creating and approving the associated check for a validated invoice item.
rejected	→ draft	This transition occurs if the bulk invoice item changes in such a way that the owning bulk invoice no longer passes validation.
submitting	→ submitted	ClaimCenter received an acknowledgement for the associated message for the submitting status. This transition requires you to call <code>bulkInvoice.acknowledgeSubmission()</code> in your message Ack code in your messaging plugins.
	→ pendingvoid	This transition occurs if the owning bulk invoice stops either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>stopBulkInvoice</code> .
	→ pendingstop	This transition occurs if the owning bulk invoice voids either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>voidBulkInvoice</code> .
submitted	→ pendingtransfer	This transition occurs if the check associated with the bulk invoice item transfers from the ClaimCenter user interface.
	→ pendingvoid	This transition occurs if the owning bulk invoice voids either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>voidBulkInvoice</code> .
	→ pendingstop	This transition occurs if the owning bulk invoice stops either through the ClaimCenter user interface, or using a call to the <code>BulkInvoiceAPI</code> method <code>stopBulkInvoice</code> .
inreview	→ pendingtransfer	This transition occurs if the check associated with the bulk invoice item transfers from the ClaimCenter user interface.
	→ draft	This transition occurs if the bulk invoice item changes in such a way that the owning bulk invoice no longer passes validation.
	notvalid	This transition occurs if the bulk invoice item changes such that the owning bulk invoice no longer passes validation or the bulk invoice resubmits while in <code>pendingbulkinvoiceitemvalidation</code> status.

Bulk invoice item status	Can change to status	How it changes
pendingvoid	→ voided	This transition occurs if the updateBulkInvoiceStatus method gets a value of Voided to confirm a successful void of the owning bulk invoice.
	→ stopped	This transition occurs if the updateBulkInvoiceStatus gets a value of stopped to confirm a successful stop of the owning bulk invoice.
	→ submitted	This transition occurs if the updateBulkInvoiceStatus method gets a value of issued or cleared to indicate a failed void attempt for the owning bulk invoice.
pendingstop	→ stopped	This transition occurs if the updateBulkInvoiceStatus method gets a value of Stopped to confirm a successful stop of the owning bulk invoice.
	→ voided	This transition occurs if the updateBulkInvoiceStatus method gets a value of Voided to confirm a successful void of the owning bulk invoice.
	→ submitted	This transition occurs if the updateBulkInvoiceStatus method gets a value of issued or cleared to indicate a failed stop attempt for the owning bulk invoice.
pendingtransfer	→ transferred	This transition occurs if the associated check transitions from pendingtransfer to transferred status.  This happens if ClaimCenter received an acknowledgement for the associated message for the check changing to the pendingtransfer status. This transition requires you to call <code>check.acknowledgeTransfer()</code> in your message Ack code in your messaging plugins.
	→ submitted	This transition occurs if the updateBulkInvoiceStatus method gets a value of issued or cleared to indicate a failed void attempt for the owning bulk invoice.
stopped	→ submitted	This transition occurs if the updateBulkInvoiceStatus method gets a value of issued or cleared to indicate a failed stop attempt for the owning bulk invoice.
transferred	n/a	This is an ending status.

**WARNING** There are important restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule set triggers and concurrent data exceptions are disabled in some messaging code. See “Important Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 309

## Bulk Invoice Batch Processes

ClaimCenter has several bulk invoice batch processes: `bulkinvoicesworkflow`, `bulkvoicesescalation`, and `BulkInvoiceWF`. For more information, see “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide*.

## Deduction Plugins

### Deduction Calculations for Checks

You can customize the logic that generates a list of deductions from a new primary check by implementing the backup withholding plugin (`IBackupWithholdingPlugin`) in ClaimCenter. It provides an integration point for

automatically creating deductions from a payment. The plugin implementation must determine whether the deduction type applies and, if so, to create any applicable deductions.

Your plugin implementation must implement the `getDeductions` method on the plugin interface. This method takes a `Check` entity. Your implementation must generate zero or more deductions (`Deduction` entities) to submit with the primary payment. The method must return an array of zero or more `Deduction` entities.

The properties you must set on each `Deduction` entity are:

- `Amount` – The deduction amount.
- `DeductionType` – The type of deduction being applied. This is a value in the `DeductionType` typelist. The type determines how to account for the deduction in your external financial system. For example, withholding for taxes.

ClaimCenter automatically sets the value of the `Check` property to refer to the current check, so you can leave this property `null`.

The built-in implementation of this plugin simply calls the backup withholding utility class `gw.util.BackupWithholdingCalculator` to do the work. You can view and edit this Gosu class if you want to understand or modify the behavior.

ClaimCenter calls this plugin before the last step in the new check wizard. Your plugin implementation must identify deductions to a check, similar to taking taxes or benefits out of an employee paycheck. The net amount of the check must be less than payments charged to the claim file.

The most important example of this is backup withholding, which causes taxes to be withheld from the amount of the payment. This plugin implements the default behavior and gives you an opportunity to modify as appropriate.

## Handling Other Deductions

ClaimCenter also includes another plugin definition for a similar purpose as the backup withholding plugin. This plugin interface is called `IDeductionAdapter`. This plugin is similar to the `IBackupWithholding` plugin interface. Differences include:

- The `IDeductionAdapter` handles deductions in a generic way, rather than for backup withholding for checks.
- `IDeductionAdapter` requires use of special template to generate parameters into a large `String` data object. In contrast, the `IBackupWithholding` plugin takes a typesafe `Claim` object.

The default implementation of the `IDeductionAdapter` plugin simply calls the registered version of the `IBackupWithholding` plugin interface. In turn, that class calls the backup withholding utility class `gw.util.BackupWithholdingCalculator` to do all of its work. You can view and edit this file in Studio.

The plugin takes some data about the new primary check from a plugin template (`Deduction_Check.gs`) with root object (`check`) and returns a list of deductions. A deduction is a much simpler object to generate than a reserve or a check. The template just needs enough information about the check and the payees to determine whether a deduction is necessary, and if so for how much.

**Note:** For more information about using and deploying plugin templates, see “Writing Plugin Templates For Plugins That Take Template Data” on page 172.

Your plugin implementation must implement the `getDeductions` method on the plugin interface. Your implementation must generate zero or more deductions (`Deduction` entities) to submit with the primary payment. The method must return an array of zero or more `Deduction` entities.

The properties you must set on each `Deduction` entity are:

- `Amount` – The deduction amount.
- `DeductionType` – The type of deduction being applied. This is a value in the `DeductionType` typelist. The type determines how to account for the deduction in your external financial system. For example, withholding for taxes.

ClaimCenter automatically sets the value of the `Check` property to refer to the current check, so you can leave this property `null`.

The deduction object just requires an amount and a type, which is a value in the `DeductionType` typelist. The type determines how to account for the deduction in your external financial system. For example, withholding for taxes. ClaimCenter automatically sets the value of the `Check` property to refer to the current check, so you can leave this property `null`.

## Initial Reserve Initialization for Exposures

If you use single currency mode, you can customize the logic that generates a list of initial reserves for an exposure by implementing the initial reserve plugin (`IInitialReserveAdapter`) in ClaimCenter. ClaimCenter calls this plugin after creation of a new exposure and claim is set up. The plugin receives some information about the new exposure using a plugin template, specifically `InitialReserve_Exposure.gs` that runs with root object with symbol `exposure`. This plugin must return a list of initial reserves for the exposure. If no reserve is necessary, then the plugin must return a zero length list rather than `null`.

If you use multicurrency features of ClaimCenter, you must use Initial Reserves rule set instead of using the `IInitialReservesAdapter` plugin due to multicurrency support in the rule set.

ClaimCenter takes care of setting claim and exposure IDs and the status of the new reserves automatically. Do not bother to set these properties in your returned objects. However, you must modify the plugin template file to pass enough information in the template to make a decision about the category and amount of reserve. See “Writing Plugin Templates For Plugins That Take Template Data” on page 172.

## Exchange Rate Integration

To make financial transactions in multiple currencies, ClaimCenter needs a way of describing current currency exchange rates around the world. Do this using the exchange rate set plugin (`IExchangeRateSetPlugin`) interface, whose main task is to create `ExchangeRate` entities encapsulated in a `ExchangeRateSet` entity.

**Note:** For more information about multicurrency, see “Multiple Currencies” on page 335 in the *Application Guide*.

This plugin interface has one method, `createExchangeRateSet`, which takes no arguments and returns a `ExchangeRateSet`. This method must populate the `ExchangeRateSet` with `ExchangeRate` entities, with a total of at least the total number of currencies minus one. Each one must describe the exchange rate from the reporting currency (the system main/default currency) to each non-reporting currency (secondary currency). You must provide each of these conversions. For example, if there are 20 registered currencies, your plugin implementation must populate at least 19 entities (20 minus 1) to describe the rate changes. In other words, create one entity for each non-reporting currency.

If ClaimCenter commits an `ExchangeRateSet` entity to the database, ClaimCenter automatically creates `ExchangeRate` entities for all remaining permutations of currencies, including all combinations of two currencies, in both directions.

For example, suppose the application has three currencies: USD, EUR, and GBP. Suppose USD is the application default. To create an `ExchangeRateSet`, specify two rates because two is one fewer than the three currencies. Suppose you specify the following set:

```
{USD→EUR=0.5, USD→GBP=0.33}
```

The final `ExchangeRateSet` that commits to the database has  $n^2=9$  rates, with the following approximate values:

```
{USD→EUR=0.5, EUR→USD=2.0, USD→GBP=0.33, GBP→USD=3.0, EUR→GBP=0.66, GBP→EUR=1.5,  
USD→USD=1.0, EUR→EUR=1.0, GBP→GBP=1.0}
```

You may specify more ExchangeRate values, but the set of currency conversions from the reporting currency to each non-reporting currency are required at a minimum. If you specify additional conversions and the ExchangeRateSet commits, ClaimCenter does not automatically compute rates that you explicitly specified. For example, if you specify both USD→EUR and EUR→USD, it does not automatically compute the rate for EUR→USD. Instead, it uses the rate you explicitly specified.

#### See also

- “Multiple Currencies” on page 335 in the *Application Guide*

## Sample Exchange Rate Plugin Implementation

ClaimCenter provides a sample Gosu implementation of the exchange rate plugin that illustrates useful new features including outbound SOAP calls (to a third-party service) and XML processing.

---

**IMPORTANT** The sample is an example only and may not work with all currencies. ClaimCenter does not guarantee that the service it queries works at any given moment, either now or in the future.

---

The sample plugin implementation connects to a service provided by the Federal Reserve Bank of New York. The service provides a method for retrieving an exchange rate between two currencies. The response from the service is an XML document, and its exchange rate is within the <frbny:OBS\_VALUE> element.

Additional calculation must be done because the value may be the actual rate you want, or it may be the reciprocal. Note the creation of ExchangeRates added to the newly created ExchangeRateSet, and the number of them is one fewer than the registered number of currencies.

Real implementations follow this general pattern:

```
package gw.plugin.exchangerate.impl;
uses java.util.Date;
uses gw.plugin.exchangerate.IExchangeRateSetPlugin
uses gw.api.util.CurrencyUtil
uses soap.ExchangeRateService.api.FXWS

class SampleExchangeRateSetPlugin implements IExchangeRateSetPlugin
{
    public override function createExchangeRateSet() : ExchangeRateSet {
        var erSet = new ExchangeRateSet();
        // the sample ExchangeRateService uses newyorkfed.org, which only provides
        // exchange rates to and from USD
        var defaultCurrency = typekey.Currency.TC_USD;
        var api = new FXWS();
        for (var currency in typekey.Currency.TypeKeys) {
            if (currency != defaultCurrency) {
                var er = new ExchangeRate();
                er.BaseCurrency = currency;
                er.PriceCurrency = defaultCurrency;
                if( currency == typekey.Currency.TC_RUB ) {
                    // newyorkfed.org doesn't provide exchange rates in Rubles,
                    // so for example only use an arbitrary sample value here...
                    er.Rate = .04156103;
                } else {
                    er.Rate = extractRate(api,defaultCurrency,currency);
                }
                erSet.addToExchangeRates(er);
            }
        }
        erSet.Name = "Test ExchangeRateSet";
        erSet.Description = "From SampleExchangeRateSetPlugin.";
        erSet.MarketRates = true;
        erSet.EffectiveDate = gw.api.util.DateUtil.currentDate();
        return erSet;
    }

    /**
     * @return the number in units of defaultCurrency per unit of currency
     */
    private function extractRate(api : FXWS, defaultCurrency : Currency, currency : Currency) : float {
        var response = api.getLatestNoonRate(currency.Code)

```

```
// The response from the external service is an XML document.  
// The value you want is the number within the frbny:OBS_VALUE element.  
// If the UNIT attribute of the frbny:Series element is "USD", the conversion rate  
// you want is the reciprocal of the value you got.  
// Otherwise, just return the value.  
var node = gw.api.xml.XMLNode.parse(response);  
var match = node.findFirst(x -> x.ElementName == "frbny:OBS_VALUE") as gw.api.xml.XMLNode;  
var value = java.lang.Float.valueOf(match.Text);  
var unit = node.findFirst(x -> x.ElementName == "frbny:Series").Attributes["UNIT"];  
if (unit == defaultCurrency.Code.toUpperCase()) {  
    return value;  
} else {  
    return 1/value;  
}  
}  
}
```

## Invoking the Exchange Rate Plugin

ClaimCenter exposes the plugin functionality with the Gosu static method:

```
gw.api.util.CurrencyUtil.invokeMarketExchangeRateSetPlugin();
```

This method invokes the plugin and commits the newly created `ExchangeRateSet`. Use this method if you need to update the exchange rate data.

After calling this method, any subsequent financial transactions that require exchange rate information by default use the new `ExchangeRateSet` that the plugin creates.

In the user interface, some transaction screens allow optional overriding of the exchange rate.

### Batch Process

You can trigger the related batch process using the command line:

```
maintenance_tools.bat -password gw -startprocess exchangerate
```

For more information about running batch process, see “Batch Processes and Work Queues” on page 123 in the *System Administration Guide*. For information about running batch processes through a web service, see “Maintenance Web Services” on page 146.



---

part VI

# Importing Claims Data



# Importing from Database Staging Tables

ClaimCenter supports high-volume bulk data import by using database staging tables. This would be used typically for large-scale data conversions, particularly after migrating claims from a legacy system into ClaimCenter. This topic describes how database staging table import works and how to use it as part of a larger conversion process.

This topic includes:

- “Introduction to Database Staging Table Import” on page 415
- “Overview of a Typical Database Import” on page 419
- “Database Import Performance and Statistics” on page 424
- “Table Import Tools” on page 424
- “Populating the Staging Tables” on page 426
- “Data Integrity Checks” on page 431
- “Table Import Tips and Troubleshooting” on page 432
- “Staging Table Import of Encrypted Properties” on page 433

## Introduction to Database Staging Table Import

Database staging table import provides significantly higher performance than importing individual records with entity-specific web services APIs. Staging table import avoids intermediate data formats such as XML and avoids the need to parse and transform data into internal Java objects.

Database staging table import further improves performance by using bulk SQL `Insert` and `Select` statements that operate on entire tables at a time. Staging table import does not operate on single rows individually to import data, unlike entity-specific web services APIs that insert one row at a time.

The sections in this topic explain important concepts of ClaimCenter database staging table import. Review the concepts and terminology in these sections before proceeding to the topics that follow.

## Staging Tables

*Staging tables* are database tables that are near-duplicates of specific operational tables in the ClaimCenter database. To prepare data for import into ClaimCenter using bulk import features, you load data into these database staging tables. In general, staging tables correspond one-to-one with ClaimCenter operational tables. For most operational tables with names that start with a cc\_ prefix, there are corresponding staging tables with a ccst\_ prefix.

For example, the table for the `Claim` entity is `cc_claim`. This table has a corresponding staging table, `ccst_claim`. Similarly, ContactManager has an address table `ab_address`, with a corresponding staging table `abst_address`.

There are important differences between columns of staging tables and operational tables, which are discussed further in “Populating the Staging Tables” on page 426. However, columns with basic data are the same in staging tables and their corresponding operational tables. For a detailed list of tables, refer to the ClaimCenter Data Dictionary documentation.

Any loadable data model extension entities have table names that start with a ccx\_ prefix rather than cc\_ in operational tables. In these cases, the staging table name prefix is the same as for built-in entities. For example, a custom entity called ABC would have regular table name `ccx_abc` and the staging table `ccst_abc`.

Staging tables are created during the database upgrade subroutines as the server is loading.

During upgrade, the server creates staging tables for every entity defined as `loadable` and has at least one property defined as `loadable`.

## Zone Data

ClaimCenter uses zone data for the following features:

- Assignment by location
- Address auto-fill
- Setting regional holidays
- Defining catastrophes

### Steps to Import Zone Data

The process to import zone data comprises these main steps:

1. **Get or create your zone data files** – Create zone data files in comma separated value (CSV) format that contain columns for the postal code, state, city, and county of specific geographic zones. Each line in the file must use the following format:

```
postalcode,state,city,county
```

For example,

```
94114,CA,San Francisco,San Francisco
```

Guidewire provides a few zone data files for you to use. See “Zone Data Files Supplied by Guidewire” on page 417.

2. **Run the zone import command to add zone data to the zone staging table** – Run the command line tool `zone_import` in the directory `ClaimCenter/admin/bin`. You can also use the web service `IZoneImportAPI` to add your zone data to the staging tables.

For example with the command line tool, your command might look like this:

```
zone_import -clearstaging -import myzonedata.csv -server http://myserver:8080/pc -user myusername
```

For more information, see “Zone Import Command” on page 194 in the *System Administration Guide*.

3. Run the **table import command to bulk load the zone operational table** – Run the command line tool `table_import` in the directory `ClaimCenter/admin/bin`.

For more information, see “Table Import Command” on page 191 in the *System Administration Guide*.

## Zone Data Files Supplied by Guidewire

The base configuration of ClaimCenter includes zone data files for several countries. These files are located in the following directory:

```
configuration.config.geocode
```

The directory also includes smaller sets of zone data for development and testing purposes.

Guidewire provides the `US-Locations.txt` and similar files for testing purposes to support autofill and autocomplete when users enter addresses. This data is provided on an as-is basis regarding data content. For example, the provided zone data files are not complete and may not include recent changes.

Also, the formatting of individual data items in these files might not conform to your internal standards or the standards of third-party vendors that you use. For example, the names of streets and cities are formatted with mixed case letters but your standards may require all upper case letters.

The `US-Locations.txt` file contains information that does not conform to United States Postal Service (USPS) standards for bulk mailings. You can edit the `US-Locations.txt` file to conform to your particular address standards, and then import that version of the file.

## Your Conversion Tool

A critical component of this process is a custom conversion tool that you write. The conversion tool converts legacy data into the ClaimCenter format and places the converted data into the staging tables to await staging table import. Any conversion tool must map a legacy data format, which might be a flat file format, into a format almost identical to ClaimCenter operational tables. If the legacy format is dissimilar to the ClaimCenter format, which is most often the case, this tool must support complex internal logic.

## Integrity Checks

Before loading staging table data into operational database tables, ClaimCenter runs many ClaimCenter-specific data integrity checks. These checks find and report problems that would cause import to fail or might put ClaimCenter into an inconsistent state. Integrity checks are a large set of auto-generated database queries (SQL queries) built into the application.

You can check if any integrity checks failed at `Server Tools → Info Pages → Load Errors`. For more information, see “Load Errors” on page 178 in the *System Administration Guide*.

## Logical Units of Work, LUW, and LUWIDs

The data related to a claim is spread out across many tables and potentially many rows. You must identify self-contained units of data that must load together or fail together if something is wrong within that claim.

ClaimCenter uses the generic term *logical unit of work* (LUW) to refer to all rows across all tables as a single unit for integrity checks and loading. For example, if a claim fails an integrity check, all associated records such as related Address records in that logical unit of work fail the integrity check with the claim.

Each staging table row has a *logical unit of work ID* property, called `LUWID`, which identifies the LUW grouping of this data. This topic sometimes refers to this logical unit of work ID as an LUWID. After your conversion tool populates the staging tables, your conversion tool must set the `LUWID` property to something useful for each row. For example, a pre-defined legacy claim number/ID.

An LUWID is used in the following ways:

- LUWIDs help identify which data failed. See “Load Error Tables” on page 418.
- LUWIDs identify which data to exclude from future integrity checks or load requests. See “Exclusion Table” on page 418.

## Load Error Tables

The *load error tables* hold data from failed data integrity checks. Do not directly read or write these tables. Instead, examine them using the **Load Errors** interface. You can view these errors in ClaimCenter at **Server Tools** → **Info Pages** → **Load Errors**. For more information about using this screen, see “Load Errors” on page 178 in the *System Administration Guide*.

Most errors relate to a particular staging table row, so the **Load Errors** page shows:

- the table
- the row number
- the logical unit of work ID
- the error message
- the data integrity check (also called the query) that failed.

In some cases, ClaimCenter cannot identify or store a single LUWID for the error. For example, this may happen for some types of invalid ClaimCenter financials imports.

## Exclusion Table

The *load exclusion table* is a table of logical unit of work IDs to exclude from staging table processing during the next integrity check or load request. Do not directly read from or write to these tables. Instead, use SOAP APIs or command line tools to populate exclusion tables from logical units of work IDs (LUWIDs) in the load error tables.

## Load History Tables

Load history tables store results for import processes, including rows for each integrity check, each step of the integrity check, and row counts for the expected results. Use these to verify that the table-based import tools loaded the correct amount of data. You can view this information in ClaimCenter at **Server Tools** → **Info Pages** → **Load History**. For more information about using this screen, see “Load History” on page 178 in the *System Administration Guide*.

## Load Commands and Loadable Entities

ClaimCenter creates staging tables during the upgrade process when the server starts. If a ClaimCenter table is loadable and has at least one loadable property, ClaimCenter creates a corresponding staging table for it.

During staging table import, all loadable entities are copied from the staging tables to the operational tables. After an entity imports successfully, the application sets each entity’s load command ID property (`LoadCommandID`) to correspond to the staging table conversion run that brought the row into ClaimCenter.

An entity’s `LoadCommandID` property is always `null` for rows that were created in some other way, in other words new entities that did not enter ClaimCenter through staging table import.

The `LoadCommandID` property persists even after performing additional import jobs. The presence of the `LoadCommandID` property does not guarantee that the current data is unchanged since the row was imported. If the user, application logic, or integration APIs change the data, the `LoadCommandID` property stays the same as when the row was first imported.

You can use this feature to test whether an entity was loaded using database staging tables or some other way. From your business rules or from a Java plugin, test an entity’s `LoadCommandID`. From Gosu, check

`entity.LoadCommandID`. From Java, check the `entity.getLoadCommandID` method. If the load command is non-null, the entity was imported through the staging table import system. All entities with that same load command ID loaded together in one import request. If the load command is null, the entity was created in some way other than database table import.

These load command IDs correspond to results of programmatic load requests to import staging tables.

For example, use the command line `table_import` tool in the `ClaimCenter/admin/bin` directory. The tool returns the `LoadCommandID`. Alternatively, you can call the SOAP API to load database tables:

```
result = ITableImport.integrityCheckStagingTableContentsAndLoadSourceTables(...);
```

The result of that method is a `TableImportResult` entity instance, which contains a `LoadCommandID` property, which is the load command ID. Call `result.getLoadCommandID()` to get the load command ID for that load request. Save that value and test specific entities to see how they were loaded. Compare that value against that saved `LoadCommandID`. Similarly, if you used the command line tools to trigger database table import, those tools return the `LoadCommandID`.

You can also track load import history using the database load history tool user interface at `Server Tools → Info Pages → Load History`. For more information about using this screen, see “Load History” on page 178 in the *System Administration Guide*.

Remember that even if an object has a load command ID that matches a known load request, its data may have changed after loading due to user actions or APIs.

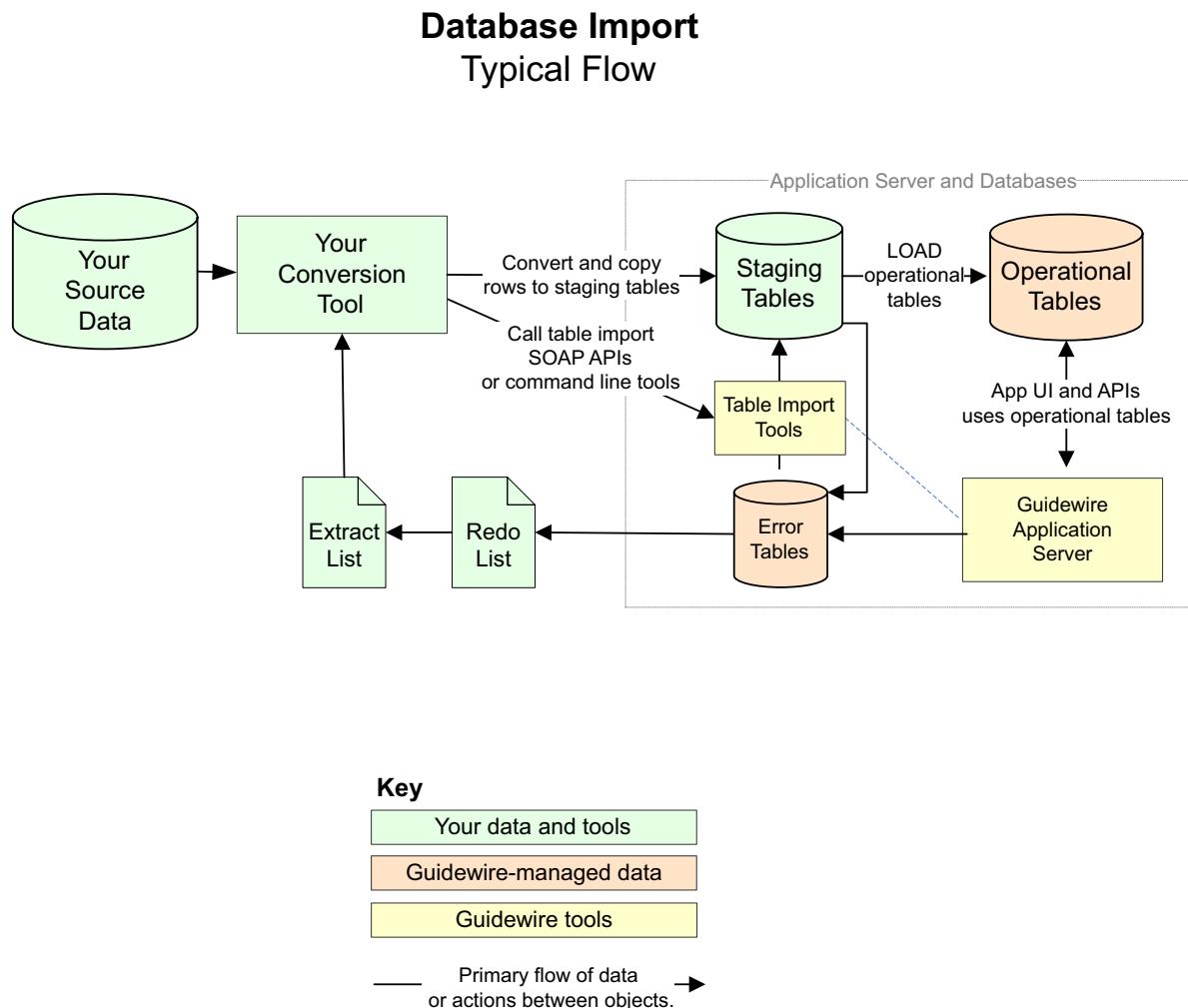
## Overview of a Typical Database Import

As an example of how the table-based import tools can be used, consider a typical migration process for converting claims from a legacy system into ClaimCenter. As mentioned in previous sections, the conversion tool is a tool that you write to convert legacy data into the ClaimCenter format.

The high-level steps in a typical database import are:

1. A conversion tool converts source data to ClaimCenter format in staging tables.
2. Run integrity checks on staging table data using SOAP APIs or command line tools. To view the integrity checks, use the `Load Integrity Checks` page. See “Load Integrity Checks” on page 178 in the *System Administration Guide*.
3. View load errors on the `Load Errors` page. See “Load Errors” on page 178 in the *System Administration Guide*.
4. Fix errors and run integrity checks again, repeating some or all steps listed earlier in this list. Optionally, you can exclude some data specifying records by LUWID. This optional exclusion is not shown in the diagram that follows.
5. After you are sure all integrity checks succeed, load the data into operational tables using SOAP APIs or command line table import tools. ClaimCenter always runs the integrity checks again during a load request to ensure data integrity.
6. Check the load history on the `Load History` page. See “Load History” on page 178 in the *System Administration Guide*.

The following diagram shows the major steps in a typical database staging table import:



Migrating data from the source system into ClaimCenter typically happens in the following steps:

1. **If you changed the data model, run the server to perform upgrades** – The staging table import tools expect that the data is already in the same format as the server data.

**IMPORTANT** If you add encryption or change encryption settings with data in operational tables, perform the upgrade successfully before running staging table import. For more information, see “Encryption Features for Staging Tables” on page 257.

2. **Back up the operational tables in database** – It is important to back up operational tables before import.

**WARNING** As with any major database operation, back up all operational tables before importing.

3. **Put the server in maintenance run level** – Set each ClaimCenter server to the **maintenance run level** using the `system_tools` command line tool or the web services API `SystemToolsAPI.setRunLevel` method. At the maintenance run level, the system prevents new user connections, halts existing user sessions, and releases all database resources. This prohibits access to the database while you are backing up the database.

**4. Clear the staging tables, the error tables, and the exclusion tables** – Your conversion tool would typically do this manually, but you can also do this using web service APIs described in “Table Import Tools” on page 424.

**5. Run ClaimCenter database consistency checks** – Database consistency checks verify that the data in your operational tables does not contain any data integrity errors. Run database consistency checks using the web services API `SystemToolsAPI` method `checkDatabaseConsistency`, or using the command line:

```
system_tools -password password -checkdbconsistency
```

See “System Tools Command” on page 187 in the *System Administration Guide* for more information about the command line tool.

If you do not run consistency checks regularly, run these a long time before converting your data. For example, plan several weeks to correct any errors you may encounter.

**6. Determine which records to convert** – Your conversion tool would determine which records to convert using some criteria. For example, it might generate an extraction list of all the claim numbers of claims to convert.

**7. Transform legacy data and populate staging tables** – Your conversion tool reads the desired claims from the source system and transforms and maps the data. Then, your tool inserts each claim into the staging tables along with associated subobjects in related tables. This step represents the bulk of your effort. Once completed, populate ClaimCenter staging tables with claim data ready to load into ClaimCenter operational tables. For more information about populating staging tables, see “Populating the Staging Tables” on page 426.

**8. Request integrity checks from SOAP APIs or command line tools** – These tools are available as web service (SOAP) APIs, and also as command line tools, all described in “Table Import Tools” on page 424. The command line tool is as follows:

```
table_import -integritycheck
```

Your conversion tool might automatically trigger the integrity check using the SOAP APIs or command line tools after converting and loading the data into staging tables. At the time you request an integrity check, you can optionally clear error tables and exclusion tables using optional parameters. Also, you can choose to perform the integrity check as part of a load request, and if so the load proceeds only if no integrity check errors occur.

There is an optional command line flag to allow references to existing non-admin rows in the database: `-allreferencesallowed` (corresponding to the SOAP API boolean parameter `allowRefsToExistingNonAdminRows`). Only use this flag (or for SOAP API, set it to `true`) if absolutely necessary. For example, in the rare case a policy period overlaps between the existing operational data and the data you are loading. This option flag can cause performance degradation during the check and load process.

**9. ClaimCenter performs the integrity check** – An integrity check ensures the data meets ClaimCenter basic data integrity requirements. If problems are found, those records are saved in an error table. ClaimCenter runs all integrity checks and reports all errors before stopping.

**10. Check load errors using the Load History page** – This page is available at `Server Tools → Info Pages → Load History`. For more information about using this page, see “Load History” on page 178 in the *System Administration Guide*.

**11. Fix the errors and/or exclude records** – If there are errors, the conversion tool must correct the errors or remove the bad data and try again. Use the data viewable in the user interface to generate a redo list, which is a list of records to fix and rerun. Or, use this data to find records to skip by adding to the exclusion table. If you want to exclude records with errors and you know the corresponding LUWIDs, add the LUWIDs to the exclusion table using web service APIs and command line tools. See “Table Import Tools” on page 424. Some errors are not associated with specific LUWIDs, in which case nothing can be moved to the exclusion table.

For example, some types of financials errors are not associated with specific LUWIDs.

**12. Repeat integrity checks until they succeed** – Repeat integrity checks until there are no errors.

13. **Optionally, remove excluded records from staging tables** – Remove all rows from all staging tables for records whose LUWIDs are listed in the exclusion table. To do this, use the `deleteExcludedRowsFromStagingTables` tool described in “Table Import Tools” on page 424.
14. **Handle encrypted properties** – If you have any entities with properties that support property-level encryption, your staging table data must have encrypted properties before importing them into ClaimCenter. For more information, see “Encryption Features for Staging Tables” on page 257.
15. **Load staging tables into operational tables** – Eventually, integrity checks succeed for all records except for records in the exclusion table. At this point, you can load data from the staging tables into the operational tables used by ClaimCenter. Do this using one of the various web service (SOAP) APIs or command line tools for loading. See “Table Import Tools” on page 424. For example, use the table import web service method `integrityCheckStagingTableContentsAndLoadSourceTables`. As ClaimCenter inserts rows into the operational tables, it also inserts results into the load history table.

There is an optional command line flag to allow references to existing non-admin rows in the database: `-allreferencesallowed` (corresponding to the SOAP API boolean parameter `allowRefsToExistingNonAdminRows`). Only use this flag (or for SOAP API, set it to true) if absolutely necessary. For example, in the rare case a policy period overlaps between the existing operational data and the data you are loading. This option flag can cause performance degradation during the check and load process.

If you use Oracle databases, Guidewire recommends you use the Boolean parameter `updateDBStatisticsWithEstimate` set to `true`. This indicates to update database statistics on any table loading with estimated changes based on contents of the associated staging table. This corresponds to the optional `-estimateorastats` option for the command line tool. If you load large amounts of data, the source table grows significantly. The optimizer could choose a bad query plan based on the existing state of the database statistics. Avoid this situation by updating the database statistics to reflect the expected size of the table after the load completes.

In other words, for non-Oracle databases, load staging tables with the command:

```
table_import -integritycheckandload
```

For Oracle databases, load staging tables with the command:

```
table_import -estimateorastats -integritycheckandload
```

---

**WARNING** For Oracle databases, failing to add the `-estimateorastats` option (or the equivalent SOAP API parameter set to `true`) extremely reduces database performance. Remember to always add this additional option for Oracle databases.

---

The server automatically removes all data from staging tables at completion. If there were errors, data remains in the staging tables.

16. **ClaimCenter populates user and time properties, as well as other internal or calculated values** – During import, ClaimCenter sets the value of the `CreateUserID` and `UpdateUserID` properties to the user ID of the user that authenticated the SOAP API or command line tool. To detect converted records in queries, create a ClaimCenter user called `Conversion` (or something like that) to detect these records. Creating a separate user for conversion helps diagnose potential problems later on. This user must have the SOAP Administration permission to execute the SOAP and command line APIs. Do not give this user additional privileges or access to the user interface or other portions of ClaimCenter. At this step, ClaimCenter sets the `CreateTime` and `UpdateTime` properties to the start time of the server transaction. All rows now have the same time stamp for a single import run.
17. **Review the load history** – Review the load history at **Server Tools** → **Info Pages** → **Load Errors**. Ensure that the amount of data loaded is correct. For more information, see “Load Errors” on page 178 in the *System Administration Guide*.
18. **Update database statistics** – Particularly after a large conversion, update database statistics. You can use the following command to update statistics:

```
maintenance_tools -password password -startprocess dbstats
```

Or, you can use the following command to update database statistics for tables exceeding the change threshold. The change threshold is defined by the `incrementalupdatethresholdpercent` attribute of the `<databasestatistics>` element within the `<database>` block in `config.xml`.

```
maintenance_tools -password password -startprocess incrementaldbstats
```

You can use the following command to get the list of SQL statements to update the statistics for all tables.

```
maintenance_tools -password password -getdbstatisticssstatements
```

Finally, you can use the following command to get the list of SQL statements to update database statistics for tables exceeding the change threshold.

```
maintenance_tools -password password -getincrementaldbstatisticssstatements
```

See “Configuring Database Statistics” on page 44 in the *System Administration Guide*.

**19. Update database consistency checks again** – Assuming there were no consistency errors before importing, new consistency errors after imports indicate something was wrong in the staging table data uncaught by integrity checks. See step 5 for the data consistency check options.

**20. Convert more records if necessary** – If you have more data to convert, begin again at step 5.

**21. Run batch processes** – Run the following batch processes in the following order:

- a. `ClaimContactsCalc` – Claim Contacts Calculations
- b. `DBConsistencyCheck` - Database Consistency Check
- c. `ClaimHealthCalc` – Claim Health Calculations
- d. `AggLimitCalc` – Aggregate Limit Calculations

However, this is the minimal recommendation. You may want to run other batch processes, depending on what other features you need, such as:

- `DashboardStatistics` – Dashboard Statistics Recalculation
- `DataDistribution` – Data Distribution

In general, never interact with the ClaimCenter database directly. Instead, use ClaimCenter APIs to abstract access to entity data, including all data model changes. Using APIs removes the need to understand many details of the application logic governing data integrity. However, the staging tables are exceptional because conversion tools that you write can read/write staging table data before import.

---

**WARNING** You can directly manipulate the staging tables. Do not directly read or write the load error tables or the exclusion tables. The only supported access to these tables is the Server Tools user interface. See “Using the Server Tools” on page 159 in the *System Administration Guide*.

---

ClaimCenter attempts to perform the integrity check and then load all data in the staging tables, so each import attempt must start with a clean set of tables. Therefore remove the staging data after successfully loading the data. This is also why claims that cannot pass the integrity check must be fixed or removed before the import proceeds.

As part of doing a conversion from a source system into ClaimCenter, there are several ways in which errors can be handled. During the development of the conversion tool, errors frequently occur due to incorrect mapping data from the source system into the staging tables. Errors also occur if you do not properly populating all necessary properties. Typically, fix these errors by adjusting algorithms in your conversion tool. Typically, you run many trial conversions and integrity checks with iterative algorithm changes before finally handling all issues.

If the server flags an error that is simply a problem with the source data, then correct it directly in the staging tables using direct SQL commands. In contrast, never modify operational tables with direct SQL commands.

Alternatively, you may want to correct errors in the source system. In that case, remove records that fail the integrity check from the staging tables. Correct the errors in the source system. Then, rerun the entire conversion process.

## Database Import Performance and Statistics

Guidewire strongly encourages you to design database import code in such a way as to isolate your code performance from ClaimCenter database import API performance:

- If using Oracle, take statistics snapshots (statspack snapshots at level 10 or AWR snapshots) at the beginning and end of `integritycheck` commands and `integritycheckandload` commands.
- Generate the statistics reports (statspack reports or AWR reports) and debug any performance problems. Guidewire recommends you download the Oracle AWR/Oracle Statspack from [Server Tools → Info Pages](#) along with [Load History Info](#) to debug check and load performance problems.
- Save a copy of the [DatabaseCatalogStatistics](#) page and archive it before each database import attempt.

**IMPORTANT** Designing appropriate statistics collection into all database import code dramatically improves the ability for Guidewire to advise you on performance issues related to database import.

ClaimCenter has an API to update database statistics for the staging tables. Use the `TableImportAPI` interface method `updateStatisticsOnStagingTables` or use the command line tool command:

```
table_import -updatedatabasestatistics
```

## Table Import Tools

ClaimCenter provides a set of staging table import functions to help you in the process described in “Overview of a Typical Database Import” on page 419.

ClaimCenter exposes the staging table import tools in the following ways:

- **Table import web service (SOAP) APIs** – You can use table import SOAP APIs defined in the `TableImportAPI` interface. All tools are provided as synchronous API methods, which do not return until the command completes. Some tools have an additional asynchronous method, which returns immediately and then performs the command as a batch process. The asynchronous versions have method names with that end in “`AsBatchProcess`”. For example, `deleteExcludedRowsFromStagingTablesAsBatchProcess`.

For general information about logging into and using web services, see “Web Services Introduction” on page 31.

- **Table import command line tools** – You can use table import command line tools within the developer admin `bin` directory with options that define which functions to perform. For a detailed help, go to the file `ClaimCenter/admin/bin` and view its built-in help with the command:

```
table_import -help
```

For more information about the `table_import` command, see “Table Import Command” on page 191 in the *System Administration Guide*.

All servers in your ClaimCenter cluster must be at the maintenance run level to call any of these import functions. The maintenance run level ensures that no end users are on the system. This prevents new data added through the user interface from interfering with bulk data imports from the staging tables. Use the following command (for each server in the cluster) to set this run level:

```
system_tools -maintenance -server url -password password
```

**IMPORTANT** All table import commands require all ClaimCenter servers in a cluster to be at the maintenance run level or the table import command fails.

Alternatively, set the run level using web service APIs in the `SystemToolsAPI` web service method `setRunLevel`.

The following sections briefly describe the most useful table-based import tools.

### Integrity Check Tool

This tool performs the integrity check only. It does not attempt to load the operational tables. Optionally, the tool clears the error table before starting and load the exclusion table with the distinct list of logical unit of work IDs in the error table. This is the default behavior.

Also by default, the method only allows references from data in the staging tables to administrative tables in the operational tables. For example, it allows links to an existing user or group but not loading additional exposures, or notes for an existing claim.

This simplifies the integrity check and increases the performance of the operation. If you cannot accept this limitation, then set the `allowRefsToExistingNonAdminRows` option.

- API method: `integrityCheckStagingTableContents`
- Command line option: `integritycheck`

### Integrity Check And Load Tool

This tool runs the integrity check process and, if no errors are detected, proceeds with loading the operational tables. The tool has the same options as listed earlier in the section for the error and exclusion tables and limiting to references to admin tables.

In addition, if you are using Oracle only, you can instruct the system to update database statistics on each table after data is inserted into it. This helps the database avoid bad queries caused by large changes in the size of tables which occur during import.

- API method: `integrityCheckStagingTableContentsAndLoadSourceTables`
- API method: `integrityCheckStagingTableContentsAndLoadSourceTablesAsBatchProcess`
- Command line option: `-integritycheckandload`

This tool updates estimated row and block counts on the table and indexes. This helps avoid potential optimizer issues if you reference tables with a size that qualitatively changed in the same transaction.

The Boolean parameter `updateDBStatisticsWithEstimate` updates database statistics on any table loading with estimated changes based on contents of the associated staging table. This corresponds to the optional `-estimateorastats` option for the command line tool. If you load large amounts of data, the source table grows significantly. The optimizer could choose a bad query plan based on the existing state of the database statistics. Avoid this situation by updating the database statistics to reflect the expected size of the table after the load completes. Use this feature only with Oracle databases. Otherwise, ClaimCenter ignores this parameter.

### Populate Exclusion Table Tool

Populates the exclusion table with all of the distinct logical unit of work IDs that are in the error table.

- API method: `populateExclusionTable`
- API method: `populateExclusionTableAsBatchProcess`
- Command line option: `-populateexclusion`

### Delete Excluded Rows From Staging Tables Tool

Deletes all rows from all staging tables that match a logical unit of work ID in the exclusion table.

- API method: `deleteExcludedRowsFromStagingTables`
- Command line option: `-deleteexcluded`

### Clear Exclusion Table Tool

Deletes all rows (logical unit of work IDs) from the exclusion table.

- API method: `clearExclusionTable`
- Command line option: `-clearexclusion`

### **Clear Error Table Tool**

Deletes all rows from the error table, typically in preparation for a new integrity check.

- API method: `clearErrorTable`
- Command line option: `-clearerror`

### **Clear Staging Table Tool**

Deletes all rows from the staging table, typically in preparation for a new integrity check.

- API method: `clearStagingTables`
- Command line option: `-clearstaging`

### **Update Statistics**

This tool updates the database statistics on all staging tables.

- API method: `updateStatisticsOnStagingTables`
- Command line option: `-estimateorastats`

## **Other Import-Related Tools**

The following tools are available for system-wide settings during table import. For more information about the command line tools, see the “ClaimCenter Administrative Commands” on page 181 in the *System Administration Guide*. For more information about logging into and using web service APIs, see “Web Services Introduction” on page 31

### **Setting Run Level**

Set the run level to `maintenance` before performing a table import.

- API method: `SystemToolsAPI.setRunLevel(SystemRunlevel.GW_MAINTENANCE);`
- Command line option: `system_tools -password password -maintenance`

### **Checking Operational Table Consistency**

Check the operational database consistency before running a table import.

- API method: `SystemToolsAPI.checkDatabaseConsistency(returnAllResults);`
- Command line option: `system_tools -password password -checkdbconsistency`

---

**IMPORTANT** If you do not run consistency checks regularly, run them long before starting conversion. Allow several weeks to correct any errors you may encounter.

---

## **Populating the Staging Tables**

The first step in importing data using the staging tables is mapping external data in the tables. As described earlier, there is a one-to-one correspondence between ClaimCenter operational tables (prefix: “`cc_`”) and the staging tables (prefix: “`ccst_`”).

For example, the ClaimCenter `cc_claim` table would load from the `ccst_claim` table.

Similarly, the ContactManager `ab_address` table would load from the `abst_address` table.

Not all operational tables have a corresponding staging table. For example, ClaimCenter does not import group hierarchies, roles, and system parameters. Also, ClaimCenter excludes internal tables for the message Send Queue, the internal SMTP email queue, and statistics information. However, it does include most data, including user data using the staging tables.

Even between corresponding tables there are differences in the columns. For example, compare the `cc_claim` and `ccst_claim` tables. There are a number of differences:

- The staging version of the table (starts with `ccst_`) contains a row number column and a LUWID column that do not exist in the `cc_` tables. The purpose of these is described earlier.
- The following properties do not exist in the `ccst_` versions of the tables: `CreateUserID`, `UpdateUserID`, `LoadCommandID`, `BeanVersion`, `CreateTime`, and `UpdateTime` properties. ClaimCenter sets these properties automatically during import into the operational tables.
- Some properties that track internal system state are not included in staging tables because ClaimCenter itself sets these properties at run time.
- There are internal properties (for example, `State`) that ClaimCenter cannot be certain of the value of the property upon import. Mark the claim's `state` as either open or closed upon import using the appropriate typelist codes.

In the ClaimCenter Data Dictionary, you can tell which properties are included in the `ccst_` table by checking if a property is loadable in the Data Dictionary. Notice that some properties are identified as `loadable`.

For example, in the Data Dictionary, look at the table `cc_claim`. For example, next to the property `AccidentType`, it says “`(loadable)`”, which indicates that this property is included in the staging tables.

**Note:** Use load command IDs to tell whether an entity loaded from database staging tables. See “Load Commands and Loadable Entities” on page 418.

During database import planning, use the `Conversion View` in the Data Dictionary. Get to the conversion view from the main `Home` page of the Data Dictionary, with the links that say:

- [Data Entities \(Conversion View\)](#)
- [Typelists \(Conversion View\)](#)

The `Conversion View` hides all properties that are not backed by actual database columns or are not importable in staging tables. For example, all virtual properties, which are generated (calculated) from other properties at run time.

## Important Properties and Concepts for Database Import

Most business data properties are straightforward to populate in the staging tables. You can look at the Data Dictionary to understand the meaning of the properties on the operational tables. Put the right values into the properties by the same name on the staging tables. However, there are some general guidelines for how to populate the staging tables, described in the following subsections.

### Public IDs and the Retired Property

ClaimCenter relies on `PublicID` strings to identify rows in the staging tables. Every row you insert must have a `PublicID`, which must be unique within both the staging table and its corresponding operational table. For tables with a `retired` property, the combination of `PublicID` and `Retired` must be unique.

In the operational tables, logical deletes are done by setting the `Retired` property to something other than 0 (in practice, it is set to the ID of the row). This allows more than one row to have the same `PublicID` as long as only one is currently not retired. ClaimCenter uses this to make sure that you are not inserting duplicate rows and to resolve foreign keys (references between tables).

You must have an algorithm for generating unique `PublicID` strings in the conversion tool. For example, the `PublicID` for each claim could be the company name, then a colon character, then claim number. The `PublicID` for exposures could be the claim number's public ID, a colon character, and the exposure's claim order incremental counter.

Set the retired property (if any) to the value 0, indicating an active row in the operational tables. To retire a row, set the `retired` property to the object's public ID. Once an object is retired, it can never revert to active.

**WARNING** Guidewire recommends that you not load retired data during staging table import.

Integration code must never explicitly set a public IDs to a string that starts with a two-character ID and then a colon because Guidewire reserves all such IDs. If you used the `PublicIDPrefix` configuration parameter to change the prefix of auto-created public IDs, integration code that sets explicit public IDs also must not conflict with that namespace. Additionally, plan your public ID naming carefully to support large (long) record numbers. Ensure your system can support a significant number of records over time and stay within the 20 character public ID limit. For detailed discussion on these issues, see “Public IDs and Integration Code” on page 27.

For `Transaction` and `TransactionLineItem`, the character length limit for public IDs is 18, not 20.

### Foreign Key Fields

Foreign key fields (for example, the `ClaimID` property on `ccst_exposure`) must contain the public ID (`publicID`) of the entity, such as the claim. To resolve the `ClaimID` foreign key, ClaimCenter looks in both the `ccst_claim` and `cc_claim` tables – another reason why the claim's `PublicID` must be unique across both tables.

For `ContactManager`, foreign key fields must contain the public ID (`publicID`) of the target entity.

During the load phase, ClaimCenter looks up the foreign key by public ID and insert its internal ID into the operational table.

### All Columns in the Staging Tables Are Nullable

Most columns in the staging tables are nullable, even if the corresponding column in the operational table is not nullable. This is designed to allow you to insert rows and then set default values afterwards, if necessary, while loading the staging tables.

However, if the corresponding column in the operational table has a default value (check the Data Dictionary), then you can leave the staging table column `null`. In this case, ClaimCenter sets the default value. If the property does not have a default value, you must set some value in the staging table even if merely a default value that your conversion tool chooses.

### Type Key Columns

Type key columns must be loaded by a valid `Code` value from the type list. During population, ClaimCenter converts this to an internal integer type code ID.

For example, this would apply to the `AccidentType` column on the `cc_claim` table.

### Subtyped Tables

If the operational table is subtyped, then there is a `subtype` column on the staging table. This property must load by the text name of the subtype. Look in the Data Dictionary for the subtype names.

For example, for `cc_contact` the subtypes are listed as `Person` and `Company`.

### Ignore the Row Number Column

Ignore the row number column. This column is automatically populated by ClaimCenter by the database on inserting the row or by the integrity check process, depending on which database management system (DBMS) you are using. The row number property is only used for reporting errors.

### Populating the Logical Unit of Work ID (LUWID) Column

Populate the `LUWID` column with something useful. Typically this is the claim number, but you could use other values. For example, if you insert users, it might appropriate to use the user's login name (for example, `ssmith`)

since the user is probably related to more than one claim. This way, you can delete all the data related to a user (in other words, ccst\_user, ccst\_credential, ccst\_contact) if the user already exists in ClaimCenter.

#### Overwritten Tables and Columns

There are also tables that have staging tables but are always overwritten during import. These are marked within the data model files as `overwritteninstagingtables` set to `true`.

Likewise, some columns are defined with the `loadable` attribute and also the attribute `overwriteinstagingtables`. These columns in the staging table are usually cleared and populated by the staging table loader subroutines.

#### Virtual Properties

Some Guidewire entities contain properties called virtual properties (virtual fields). They act like regular properties in many ways from Gosu such as reading property values. However, virtual properties are not backed from columns in the database. Values of these properties generate dynamically if Gosu code requests the property value. In many cases, virtual properties are read-only from Gosu code, from Java plugins, and from Java classes called from Gosu. Some virtual properties are also writable. Refer to “Properties” on page 193 in the *Gosu Reference Guide* for related information.

Generally speaking, ignore all virtual properties for conversion projects using staging tables.

To determine where to load data for some virtual property (virtual field), there is no one answer. In some cases the virtual property calls a method that performs a complex calculation that pull from many different properties or even other entities.

As a general rule, to find out what properties the virtual property pulls information from, see the *ClaimCenter Data Dictionary*. Sometimes the Data Dictionary contains the necessary information for staging table database import for how that virtual property is calculated. However, you are only responsible for populating loadable properties and columns. Use the Conversion View of the Data Dictionary to show only loadable entities and loadable columns that your conversion tool must populate.

## Indexes in Staging Tables

All staging tables must contain a unique index on each table before loading the staging tables. The system applies an integrity constraint during load that there is a unique index on all staging tables. If the table has a retired property, the unique index must be on the combined columns (`publicID`, `retired`). If the table does not have a retired property, the unique index must be on the `publicID` column.

## Special Requirements for Some ClaimCenter Entities

In addition to general guidelines, you must understand the ClaimCenter data model enough to correctly generate rows in all the needed tables and set any important properties.

The following table provides requirements for specific ClaimCenter entities:

Table	Column	Requirement
cc_claim	State	This property governs the claim's open or closed status. Set this to <code>open</code> or <code>closed</code> for all claims being imported. If <code>closed</code> , then also set the <code>ClosedDate</code> property.
	PolicyID	Every claim must have its own copy of policy information. Do not share this value across claims even if more than one claim is associated with the same policy revision. The policy table can contain only minimal information, such as the policy number, however a policy information record must exist for each claim.

Table	Column	Requirement
cc_transaction	Status	ClaimCenter only supports insert of financial transactions that have already been processed. This means that the status must be a post-submission state, such as submitted for a reserve. See the following discussion for more restrictions on loading financial transactions.
	CheckID (payments only)	Every payment must be part of a check. Therefore, if importing a payment from an external system, you also need to create a row in the ccst_check table to hold check information, such as the check number.
	TransactionSetID	In ClaimCenter, you can enter more than one transaction as part of a set that gets approved as a batch (for example, a set of reserve changes). All imported transactions must be part of a transaction set, even if there is only a single transaction in the set.
cc_contact	ABContactID	ClaimCenter can store multiple versions of a person or company's address info. The different versions link together because they share the same ABContact. As you load data, you can provide a public ID of an ABContact to link to an existing record in ClaimCenter. You can also create a new ABContact so that a loaded contact exists both locally for the claim and in the central address book. You can also leave this property null. In that case, the contact is local to the claim and does not appear in the central address book nor is available to other claims.

## Policy Location Data Import Warning

As you load policy location and building data from staging table, you must add risk units to show policy location information from the user interface. Specifically, you must load `LocationBasedRU` entity data, which is a subtype of the `RiskUnit` entity. If you do not load this data, ClaimCenter hides the policy location information in the user interface.

## Financial Transaction Importing Restrictions

The ClaimCenter web application enforces certain requirements on the data that you enter. For example, you cannot add an additional payment to an existing check. In general, it only makes sense to import entire financial occurrences as a unit. For example, adding a new `TransactionLineItem` to an existing `Transaction` or adding another reserve to a reserve set would change the meaning of the original action.

ClaimCenter enforces these restrictions by preventing certain types of imports:

- No inserting new financials data for a claim that already has financials data on it at all. Through the web-based user interface, checks are recoded by adding onset and offset payment transactions that change the categorization of the payments. Since loading of payments on existing checks is not allowed, an external system could not send recoding transactions to ClaimCenter using table-based import
- No inserting additional transactions into an existing transaction set. For example, do not add another reserve into an approved reserve set. Do not add another check or reserve into a check set. Do not add another recovery into a recovery set. Do not add another recovery reserve into a recovery reserve set.
- No inserting an additional payment into an existing check.
- No inserting an additional add-on or deduction into an existing check.
- No inserting an additional `TransactionLineItem` into an existing `Transaction`.

You must also follow the following rules for financials data:

- Do not insert transactions and checks that have not yet been processed because no events would be generated for these, so they would never get processed.
- Do not insert recoded payments.
- Do not insert transferred checks.

- Do not insert multi-payee checks.
- Only insert approved TransactionSets.
- For supplemental payments, you must ensure that you do not violate the rules of the system. For example, if the system configuration parameter AllowNoPriorPaymentSupplement is set to `false`, then do not attempt to load supplemental payments on claims with no other financial transactions. Additionally, never load supplemental payments into open claims or exposures.
- Never include more than one Recovery in a RecoverySet.
- Financial entities of types Reserve, Recovery Reserve, or Recovery must have the status `submitted`.
- A Payment entity must have the status `submitted`, `voided`, `stopped`, or `recoded`.
- A Check entity must have the status `requested`, `issued`, `cleared`, `voided`, or `stopped`.
- A Check entity must have at least one payee.
- If the `AllowMultipleLineItems` configuration parameter is `false`, then each Transaction can have only one TransactionLineItem.
- If the `AllowMultiplePayments` configuration parameter is `false`, then each Check can have only one Payment.
- For defining transactions, there are special rules for multicurrency support.

If you import transactions through the web service APIs, then you can pass a set of exchange rates. The application creates the complete set of all permutations of ExchangeRate and ExchangeRateSet entities for them. Next, it links Transaction entities in each TransactionSet to them.

In contrast, if you import transactions using staging tables, create an ExchangeRate and ExchangeRateSet entity for every Transaction. Create an ExchangeRate and ExchangeRateSet for every Check, noting that the payments for the check share the same ExchangeRateSet entity. Every Transaction imported through staging tables has a custom exchange rate in the ClaimCenter user interface.

For importing transactions, there are staging table integrity checks for

- Checking that the payments in a check all point to the same ExchangeRate entity.
- In relation to the previous item, also that check payments have all `null` or all non-`null` transaction to claim exchange rates.
- The reporting amount equals the claim amount.
- The transaction amount's sign (positive or negative) matches the claim amount.
- The claim to reporting exchange rate is `null`.
- The transaction to claim exchange rate has the correct currencies. In other words, the ExchangeRate entity has its transaction currency as `ExchangeRate.BaseCurrency` and the claim/default currency is `ExchangeRate.PriceCurrency`.
- The check payments have the same currency.

See “Multiple Currencies” on page 335 in the *Application Guide* for more information about multicurrency. See “Exchange Rate Integration” on page 409 for more information about specifying exchange rates.

- Separate from the transaction requirements for multicurrency, each transaction's `PublicID` property must be two or more characters shorter than the maximum property length.

## Data Integrity Checks

Before loading staging table data into the operational database tables, ClaimCenter runs a broad set of ClaimCenter-specific data integrity checks. These checks find and report problems that would cause the import to fail or put ClaimCenter into an inconsistent state.

ClaimCenter requires that data integrity checks succeed as the first step in the load process. This means that even if errors are found and these rows were removed, ClaimCenter still requires rerunning integrity checks before your data is reloaded.

Contrast data integrity checks with other validations:

- Integrity checks check different things from requirements that the user interface (PCF) code enforces. For example, a property that is nullable in the database may be a property that users must set in the ClaimCenter user interface. Importing a `null` value in this property is acceptable for database integrity checks. However, if you edit the object containing the property in the ClaimCenter interface, ClaimCenter requires you to provide a non-`null` value before saving because of data model validation.
- Integrity checks are different from validation rule sets and the validation plugin.

### Examples of Integrity Checks

The following list is a partial list of data integrity checks that ClaimCenter enforces during database import:

- No duplicate `PublicID` strings within the staging tables or in the corresponding operational tables
- No unmatched foreign keys
- No missing, required foreign keys, for example every exposure must be tied to a claim
- No invalid codes for type key properties
- No invalid subtypes, for example `BI` is not a valid exposure subtype
- No `null` values in non-`null` (operational) properties that do not provide a default. Empty strings and text containing only space characters are treated as `null` values in data integrity checks for non-nullable properties.
- No duplicate values for any unique indexes, for example, `ClaimNumber` on `cc_claim`.

You might notice that this list does not include enforcing property formats for properties that use a field validator in the user interface. For example, enforcing a standard format of claim number.

For a full list of integrity checks, see the [Load Integrity Checks](#) page within the ClaimCenter Server Tools tab. You can view all integrity check SQL queries. For more details, see “Load Integrity Checks” on page 178 in the *System Administration Guide*

### Why Integrity Checks Always Run Before Loading

There are many reasons for ClaimCenter to rerun integrity checks during any load request, even in situations in which the conversion tool believes that it fixed all load errors. For example:

- If the logical unit of work IDs were not populated correctly, removing a claim could leave extra rows in the staging tables that were not properly tied to the claim.
- If errors were found during population of the operational tables, the entire process must roll back the database. Rolling back the database changes typically is slow and resource-intensive. It is much better to identify problems initially rather than trigger exceptions during the load process that require rolling back changes.
- Even if you remove all error rows, integrity check violations can occur for certain errors that cannot be tied to a single row. Because some errors cannot be tied to a particular row, there is no associated logical unit of work ID

## Table Import Tips and Troubleshooting

Some things to know about importing using the staging tables and safely and successfully running the process:

- All staging table import commands require the servers to be in maintenance mode, formally known as the maintenance run level. This prevents users from logging into the ClaimCenter application.

- ClaimCenter runs the load process inside a single database transaction to be able to roll back if errors occur. This means that a large rollback space may be required. Run the import in smaller batches (for example, a few thousand records at time) if you are running out of rollback space.
- As with any major database change, make a backup of your database prior to running a major import.
- After loading staging tables, update database statistics on the staging tables. Update database statistics on all the operational tables after a successful load. After a successful load, ClaimCenter provides table-specific update database statistics commands in the `cc_loaddbstatisticscommand` table. You can selectively update statistics only on the tables that actually had data imported, rather than for all tables.
- Always run database consistency checks on the operational database tables both before and after table imports. Assuming there were no consistency errors before importing, consistency errors after an import indicates that there is something wrong with the data in the staging tables uncaught by integrity checks. See “Overview of a Typical Database Import” on page 419 for example commands.

For example, this can happen with financials totals that are denormalized and stored as totals for performance reasons. ClaimCenter does not validate that the denormalized amount in the staging tables is correct. It would be extremely slow to do so. Also, problems can be corrected after the data is loaded by recalculating denormalized values using the following tool:

```
maintenance_tools -password pw -startprocess financialscalculations
```

- During ClaimCenter upgrade, the ClaimCenter upgrader tool may drop and recreate the staging tables. This occurs for any staging table in which the corresponding operational table changed and requires upgrade.

---

**WARNING** Never rely on data in the staging tables remaining across an upgrade. Never perform a database upgrade during your import process.

---

- Integrity checks during staging table import do not use field validators. For example, these would not be checked to validate a claim number using the field validator. If you need to ensure that a field fits a certain pattern, be sure to include that logic in your conversion tool before the data gets to the staging tables

## Staging Table Import of Encrypted Properties

If you are importing records using staging table import and some data contains encrypted properties, you can use a ClaimCenter tool that encrypts encrypted properties in your staging tables. See “Encryption Features for Staging Tables” on page 257.



# FNOL Mapper

The FNOL mapper is a ClaimCenter integration tool that imports First Notice of Loss (FNOL), or initial claim, reports. The FNOL mapper can import reports in the standard XML format known as ACORD XML for Property & Casualty. In addition, you can configure the FNOL mapper to import reports from other, custom XML formats.

This topic includes:

- “FNOL Mapper Overview” on page 435
- “FNOL Mapper Detailed Flow” on page 436
- “Structure of FNOL Mapper Classes” on page 437
- “Example FNOL Mapper Customizations” on page 442

**See also**

- For more information on the ACORD XML for Property & Casualty standard, see:  
<http://ftp.acord.org/Standards/propertyxml.aspx#about>

## FNOL Mapper Overview

The FNOL mapper is a ClaimCenter integration tool that imports First Notice of Loss (FNOL), or initial claim, reports. Typically, the FNOL mapper imports FNOL report data from XML documents in the standard XML format known as ACORD XML for Property & Casualty. However, you can configure the FNOL mapper to import report data in other, custom XML formats.

For ACORD XML data, the external system calls the following `IClaimAPI` web service method:

```
importAcordClaimFromXML
```

For custom XML data, the external system calls the following `IClaimAPI` web service method:

```
importClaimFromXML
```

With either method, the external system passes the XML data to ClaimCenter as one large `String` object. With both methods, the external system calls the `IClaimAPI` web service once for each FNOL report. External systems

cannot pass a batch of reports in a single call. The `importClaimFromXML` method has an additional parameter for the special custom mapper class that you must right to handle the custom format.

## Import ACORD XML Data with the FNOL Mapper

With the FNOL mapper tool, you can import FNOL reports in the ACORD XML for Property & Casualty format. ACORD XML is a standard XML format for FNOL reports, but variants of the format and claim and exposure data models vary widely. It is impossible to provide a default mapping that works for all cases.

For example, if reports have additional properties on exposures, you must specify how to map the additional properties to the ClaimCenter data model. The names of properties or typecodes might vary between the reports and ClaimCenter. If you have custom exposure types, you might need to change the exposure type based on other fields in the reports.

### Custom Mapping of ACORD XML Data

To customize your ACORD XML mapping, modify the built-in Gosu classes or write new ones to handle each report object specially. The included classes are modular already to keep exposure mapping code separate from contact mapping code and from address mapping code.

### Special Handling of ACORD XML Data

The XSD for ACORD XML allows multiple FNOL reports to be included in one XML document. However, the FNOL mapper tool requires that ACORD XML documents contain a single report. External systems must call either method on the `IClaimAPI` web service once for each FNOL report.

An ACORD XML document can have many `<MClaimSvcRq>` subelements. Each subelement can have many `<CLAIMSSVCRQMGs>` subelements with many `<xsd:CHOICE>` subelements. One of the `<xsd:CHOICE>` subelements must be a `<ClaimsNotificationAddRq>` element. The default implementation of the FNOL mapper uses the first valid `<ClaimsNotificationAddRq>` element in the XML document. If the document has no valid `<ClaimsNotificationAddRq>` elements, Gosu throws an exception.

## Import Custom XML Data with the FNOL Mapper

With the FNOL mapper tool, you can import FNOL reports in non-ACORD XML formats. To import from custom XML formats, write new Gosu classes that implement special FNOL mapper interfaces. The mapper interfaces indicate that your classes can map the incoming XML.

### See also

- For more information on the FNOL mapper interfaces, see “Structure of FNOL Mapper Classes” on page 437.

## FNOL Mapper Detailed Flow

Of all the server files for the FNOL mapper tool, the most important top-level component is the interface `FNOLMapper` in the `gw.api.fnolmapper` package. The `FNOLMapper` interface defines the contract between ClaimCenter and a class that can map XML for FNOL reports to a `Claim` entity and its subobjects. The `FNOLMapper` interface has a single method, `populateClaim`.

Another important component of the FNOL mapper tool is the `IClaimAPI` web service interface. For ACORD XML data, external systems call the interface method `importAcordClaimFromXML`. For a custom mapper class for non-ACORD XML data, external systems call the method `importClaimFromXML`. The `importClaimFromXML` method takes an extra parameter for the name of your custom mapper class.

You can invoke the FNOL mapper for ACORD XML data from Java web services client code like the following:

```
String myClaimPublicID = claimAPI.importAcordClaimFromXML(myXMLData);
```

For both methods on the `IClaimAPI` web service interface, the web service implementation itself is relatively simple, because the FNOL mapper classes do most of the work.

The general flow of the FNOL mapper web service is as follows:

1. Find the appropriate implementation class for the mapper and instantiate it. This class must implement the `FNOLMapper` interface. For the `importAcordClaimFromXML` method, ClaimCenter always uses the built-in mapper implementation class. For the `importClaimFromXML` method, the second argument is the name of your custom mapper implementation class.
2. Create a new `Claim` entity.
3. Create a new `Policy` entity and attach it to the new claim.
4. Call the `populateClaim` method on the mapper class and pass the new `Claim` entity. The method signature is:  

```
function populateClaim(Claim claim, String xml) : void
```

**IMPORTANT** The mapper must throw `FNOLMapperException` if it cannot map the claim.

5. The mapper class reads the XML data to set fields on the `Claim` entity and add subobjects as appropriate.
6. After the `populateClaim` method completes, the web service persists the new claim and its subobjects.
7. The web service returns the public ID (a `String` value) for the imported and persisted claim. If errors occur, the web service API throws an exception to the web service client.

#### See also

- For more information about ClaimCenter web services, see “Web Services Introduction” on page 31 and “Claim Web Service APIs and Data Transfer Objects” on page 152.

## Structure of FNOL Mapper Classes

To support basic ACORD files, ClaimCenter includes a built-in implementation of the `gw.api.fnolmapper.FNOLMapper` interface. This implementation is a customer-viewable Gosu class called `gw.fnolmapper.acord.AcordFNOLMapper`. It maps a basic ACORD file starting at the `<ClaimsNotificationAddRq>` subelement to the default ClaimCenter data model for claims, exposures, including built-in typecodes. A real-world implementation likely requires some customization to one or more of these files.

You can use the default implementation and modify it directly. However, the `AcordFNOLMapper` class delegates most of its important work to other mapping interfaces to handle specific element objects. The following table lists the component, the interface class that manages this mapping, and the name of the built-in implementation class that implements the default behavior.

To map this data	Delegate to implementation of this interface	Built-in implementation class of the interface
address	<code>gw.fnolmapper.acord.IAddressMapper</code>	<code>gw.fnolmapper.acord.impl.AcordAddressMapper</code>
contact	<code>gw.fnolmapper.acord.IContactMapper</code>	<code>gw.fnolmapper.acord.impl.AcordContactMapper</code>
exposure	<code>gw.fnolmapper.acord.IExposureMapper</code>	<code>gw.fnolmapper.acord.impl.AcordExposureMapper</code>
incident	<code>gw.fnolmapper.acord.IIncidentMapper</code>	<code>gw.fnolmapper.acord.impl.AcordIncidentMapper</code>
policy	<code>gw.fnolmapper.acord.IPolicyMapper</code>	<code>gw.fnolmapper.acord.impl.AcordPolicyMapper</code>

To make significant changes to the default mappings, instead of modifying the built-in files you could create new implementations of these interfaces. Each one of these interfaces is fairly simple. Each interface contains one method for each variant of this object possible in the ACORD XSD.

For example, for contacts the ACORD format supports two contact types: `InsuredOrPrincipal_Type` and `ClaimsParty_Type`. In ClaimCenter, both of these correspond to `ClaimContact` entities. The `IContactMapper` interface defines a method for each one of the ACORD types. The method in the mapper must know how to convert the ACORD XML encoding of that type into a ClaimCenter `ClaimContact` object, which it returns. The built-in `AcordContactMapper` class implements this interface. However, you could provide an alternative implementation.

### Gosu Native XML Support

From Gosu, the mappers do not manipulate the XML as raw text. Gosu includes native XML support and the ability to read and write a Gosu representation of XML data as nodes as native Gosu objects. In cases where an XSD is available, which is the case for ACORD, properties on node objects have the correct compile-time type from Gosu. For example, a claim party type in the ACORD spec appears in Gosu as an XML node object as the type `xsd.acord.ClaimsParty_Type`. From Gosu you can work with these objects naturally in a typesafe way. You do not need any direct parsing of XML as text data.

**Note:** For information about manipulating XML objects from Gosu, see “Gosu and XML” on page 267 in the *Gosu Reference Guide*.

### If You Make New Address, Contact, Exposure, or Incident Mapping Classes

To instantiate the implementation classes for the specialized mappers for addresses, contacts, exposures, and incidents, the ACORD mapper calls the `AcordMapperFactory` class, which implements the `IMapperFactory` interface.

If you want to replace the implementation for any of the ACORD-specific mapping classes (for address, contact, exposure, or incident), perform one and only one of the following:

- Modify the simple built-in `AcordMapperFactory` class to instantiate your own versions of mapping classes.
- Alternatively, provide a new `IMapperFactory` implementation class based on the default one. However, if you do this you must modify `AcordFNOLMapper` to instantiate your new mapper factory class instead of the default mapper factory. However, in the built-in implementation of the ACORD mapper, the root `AcordFNOLMapper` class does not directly create the mapper factory. Instead, it delegates this work to the configuration utility class `AcordConfig`. Modify that file and look for these lines:

```
protected function createMapperFactory() : IMapperFactory {  
    return new AcordMapperFactory(this)  
}
```

Change it to instantiate your own mapper factory:

```
protected function createMapperFactory() : IMapperFactory {  
    return new abc.claimcenter.fnolmapper.ABCMapperFactory(this)  
}
```

## Configuration Files and Typecode Mapping

ClaimCenter includes a utility class that allows you to store mapping configuration information in a configuration directory and access that information with convenience methods.

The core configuration file class is called `gw.api.fnolmapper.FNOLMapperConfig`. You can use that with your own mappers. For the built-in ACORD implementation, ClaimCenter includes another configuration utility class specific to ACORD called `AcordConfig`. It gets an instance of the `FNOLMapperConfig` class and gets methods on it. The built-in ACORD implementation then calls `AcordConfig` directly in most cases, and that file adds additional convenience method (discussed later in this section).

### Mapper Properties file

The `mapper.properties` file specifies the following basic properties:

- `mapper.alias.default` – A default alias name that is used when no mapping is found between an external code and a Guidewire TypeKey name in the mapping XML file. For more information about the standard typecode mapping files format, see “Mapping Typecodes to External System Codes” on page 143. If you do

not specify this property, there is no default alias. This means that unmapped typecodes cause Gosu to throw an exception eventually. ClaimCenter returns this exception back to the web services client as a SOAP fault.

- **mapper.file** – The path of the XML mapping file for mapping Guidewire typecodes to codes used in the XML to be mapped. If you do not specify this property, the mapper uses the default location:  
`MODULE_ROOT/config/typelists/mapping/typecodemapping.xml`.

To modify the mapper properties, make your own version at the location:

```
ClaimCenter/modules/configuraiton/config/fnolmapper/acord/mapper.properties
```

The default **mapper.properties** files contains the following:

```
# The default key used to look up a "catch-all" typecode, e.g. "Other"
mapper.alias.default=default

# The typecode mapping XML file: relative to config root or absolute path
# Comment out this property to use the default mapping file in ${CC}/config/typelists/mapping
mapper.file=typecodemapping.xml
```

You can add additional properties to this file. You can get properties from it by calling methods on the class `gw.api.fnolmapper.FNOLMapperConfig`. The methods include:

- **getFile** – returns a file (File object) by the given name.
- **getProperties** – return a **Properties** object for a file with name passed as an argument
- **getDefaultKey** – gets the default key parameter (see earlier in this topic)
- **getTypecodeMapper** – returns the typecode mapper to use. The return type is **TypecodeMapper**.

### Typecode Mapping

If you used the **TypecodeMapper** object natively, you might use the following methods in FNOL mapper:

- **getInternalCodesByAlias** – For use during imports, returns an array of strings representing typecodes given a typelist, a namespace, and an alias. The ACORD mapper uses the namespace `acord`. If no typecodes are found, returns a zero-length, non-null array. A namespace generally corresponds to an external integration source, but multiple namespaces per source are allowed. This method allows multiple typecodes to use the same namespace-alias tuple. If you require a namespace-alias to resolve to a single typecode, please use the `getInternalCodeByAlias` method instead. The method signature is:  
`getInternalCodesByAlias(String typelist, String namespace, String alias) : String[]`
- **getInternalCodeByAlias** – return an internal code from an alias. For use during imports, returns a `String` corresponding to a typecode in the given typelist that matches the given namespace and alias. The ACORD mapper uses the namespace `acord`. If no match is found, returns `null`. If more than one match is found, throws a `NonUniqueTypecodeException` exception. The method signature is:  
`getInternalCodeByAlias(String typelist, String namespace, String alias) : String`

However, that manipulates the typekeys as `String` values. To avoid writing much code in a non-typesafe way, ClaimCenter includes the Gosu class `TypeKeyMap`. The `TypeKeyMap` class provides a typesafe wrapper around the `TypecodeMapper` for performing alias-to-typekey conversions for a specific typelist. The `AcordConfig` class defines a series of `TypeKeyMap` getter methods for each typelist. The built-in implementation uses this to map enumerations in the ACORD XML to Guidewire typekeys in a typesafe and easy-to-read way.

For example, the `AcordConfig` class defines a method called `getContactRoleMap` to map contact roles. This returns a map of `ContactRole` typecodes so you can map the external alias to the internal code.

The contact mapping class `AcordContactMapper` uses the typecode map to convert a role name to the internal typecode for contact roles with easy-to-read Gosu code:

```
private function getRole(roleName:String) : ClaimContactRole {
    var claimRole = new ClaimContactRole()
    claimRole.Role = config.getContactRoleMap().get(roleName)
    return claimRole
}
```

If you write or customize your own mappers, you might want to follow this approach to keep your mapping code as easy to understand as possible.

## ACORD Utility Class with Constant Definitions

The included simple class `AcordUtil` contains simple constants for processing ACORD XML, such as role IDs. Refer to the implementation in Studio for details.

## Contact Manager

The included `ContactManager` class tracks of the roles and contacts on the current claim. Refer to the implementation in Studio for details.

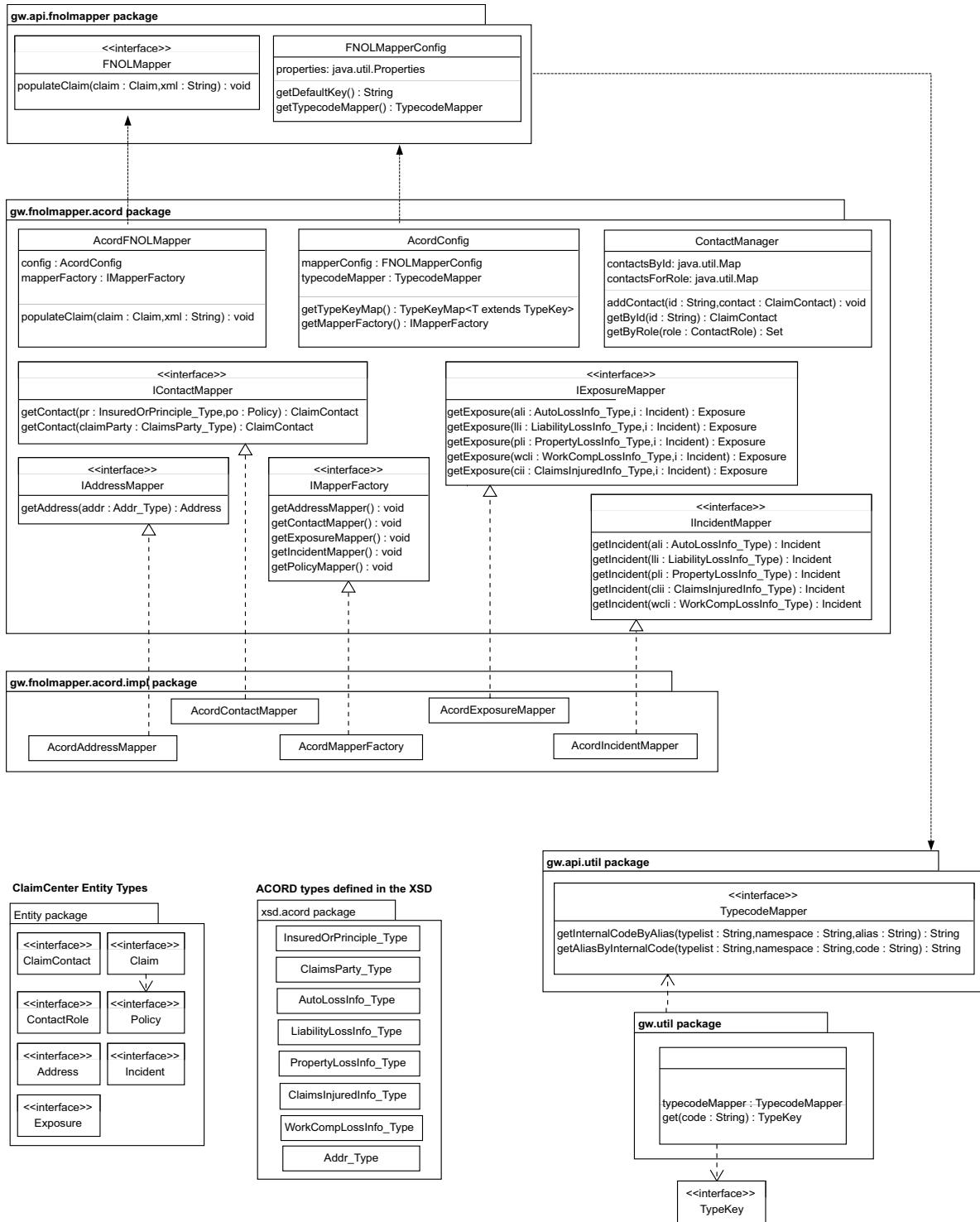
## ACORD XSD Type Enhancements

ClaimCenter includes special Gosu enhancements for dates, times, and addresses to make it easier to manipulate ACORD XSD types. For details, refer to the source for the following Gosu enhancement files:

- `DateEnhancement` – enhances the type `xsd.acord.Date` to add two methods `toDate` and `toDateTime` to convert it to a `java.util.Date` object. The `toDateTime` version takes an argument, which is the time of day in a `gw.xml.xsd.types.XSDTime` object.
- `DateTimeEnhancement` – enhances the type `xsd.acord.DateTime` to add a `toDate` method to convert it to a `java.util.Date` object
- `DetailAddressEnhancement` – enhances the type `xsd.acord.DetailAddr_Type` to add a `DisplayName` property that contains the display name
- `ClaimsPartyEnhancement` – enhances the type `xsd.acord.ClaimsParty_Type` to add the `RoleCodes` property to return role codes (`ClaimsPartyRoleCd`) for this `ClaimsParty`.

## FNOL Mapper and Built-in ACORD Class Diagram

The following diagram shows the relationships of different classes in the built-in ACORD mapper.



## Example FNOL Mapper Customizations

This topic includes examples of what files to modify to make certain types of simple customizations.

### Add Mapping for New Exposure Type

Suppose you want to specially map a general liability incident to one of two different exposure types in ClaimCenter based on other custom properties in the ACORD file.

To add mappings for a new exposure type, you might follow steps similar to the following:

1. In Studio, create a new implementation of the `IExposureMapper` interface based on the contents of the built-in `AcordExposureMapper.gs` file. For this example, we assume your company is called ABC and you name your class `ABCExposureMapper.gs` in the package `abc.claimcenter.fnolmapper`.
2. In Studio, create a new implementation of the `IMapperFactory` interface based on the contents of the built-in `AcordMapperFactory.gs` file. For this example, we assume your company is called ABC and you name your class `ABCMapperFactory.gs` in the package `abc.claimcenter.fnolmapper`.
3. Modify class `ABCMapperFactory.gs` to create your own exposure mapper:

```
override function getExposureMapper(contactManager:ContactManager) : IExposureMapper {
    return new abc.claimcenter.fnolmapper.ABCExposureMapper(acordConfig, contactManager)
}
```

4. In Studio, modify the built-in classes so that the `FNOLMapper` class call your new mapper factory. In the built-in implementation of the ACORD mapper, the root `AcordFNOLMapper` class does not directly create the mapper factory. Instead, it delegates this work to the configuration utility class `AcordConfig`.

Look for these lines:

```
protected function createMapperFactory() : IMapperFactory {
    return new AcordMapperFactory(this)
}
```

Change it to say:

```
protected function createMapperFactory() : IMapperFactory {
    return new abc.claimcenter.fnolmapper.ABCMMapperFactory(this)
}
```

5. In your `ABCExposureMapper` class, modify the method that maps the `PropertyLossInfo_Type` XML object:

```
//returns an exposure for a General Liability incident
override function getExposure(c:Claim, glLossInfo:LiabilityLossInfo_Type,
    incident:Incident) : Exposure {
    var exposure = new Exposure()
    exposure.ExposureType = ExposureType.TC_GENERALDAMAGE
    exposure.PrimaryCoverage = CoverageType.TC_GL
    exposure.Incident = incident
    exposure.LossParty = LossPartyType.TC_THIRD_PARTY
    return exposure
}
```

Change the line that says `exposure.ExposureType` and set the value based on other fields in the `glLossInfo` object.

The first argument to all `getExposure` method variants is a reference to the claim. Use that link to access properties and objects that are already mapped by previous-run FNOL mapper code. For example, suppose you already mapped a `ClaimCenter Incident` entity. Code that runs later can access the new mapped incident. For example, if wanted to get or set the vehicle incident property `VehicleIncident.Driver` you will want to reference the incident.

**Note:** The incident mapper interface has the claim as the first argument to all its methods.

### Add Additional Properties to an Exposure

Suppose that for general liability incidents in the ACORD file, you want to set additional data model extensions in ClaimCenter based on other custom properties in the ACORD file.

To do this, follow the steps in “Add Mapping for New Exposure Type” on page 442 except for the last step (step 5).

Instead of that last step, in your ABCExposureMapper class, modify the method that maps the `PropertyLossInfo_Type` XML object:

```
//returns an exposure for a General Liability incident
override function getExposure(c:Claim, glLossInfo:LiabilityLossInfo_Type,
    incident:Incident) : Exposure {
    var exposure = new Exposure()
    exposure.ExposureType = ExposureType.TC_GENERALDAMAGE
    exposure.PrimaryCoverage = CoverageType.TC_GL
    exposure.Incident = incident
    exposure.LossParty = LossPartyType.TC_THIRD_PARTY
    return exposure
}
```

Before the return statement at the end of that method, set additional properties value based on other fields in the `glLossInfo` object.

The first argument to all `getExposure` method variants is a reference to the claim. Use that link to access properties and objects that are already mapped by previous-run FNOL mapper code. For example, suppose you already mapped a ClaimCenter `Incident` entity. Code that runs later can access the new mapped incident. For example, if wanted to get or set the vehicle incident property `VehicleIncident.Driver` you will want to reference the incident.

**Note:** The incident mapper interface has the claim as the first argument to all its methods.

## Create an Entirely New Mapper For Non-ACORD Data

Suppose you want to use the basic FNOL Mapper structure to map XML data for an FNOL but the data is not in the ACORD format. The basic two steps are as follows:

- The most important thing is to create a new implementation of the `FNOLMapper` interface. For this example, we assume your company is called ABC and you name your class `ABCFNOLMapper.gs` in the package `abc.claimcenter.fnolmapper`.
- To invoke the mapper, instead of your client code calling the `IClaimAPI` web service interface method `importAcordClaimFromXML`, instead call `importClaimFromXML`. The `importClaimFromXML` method takes an extra parameter for the name of the mapper class you want to use:

```
String myClaimPublicID;
myClaimPublicID = claimAPI.importClaimFromXML(myXMLData, "abc.claimcenter.fnolmapper.ABCFNOLMapper");
```

Your `ABCFNOLMapper.gs` file does not need to follow the pattern of the built-in ACORD mapper. However, you might want to review the structure of the ACORD mapper for useful patterns you can duplicate. For example:

- Encapsulate your mapping code for addresses, contacts, exposures, incidents, and policies into separate files
- You can add Gosu enhancements on XSD-specific types. Use this to encapsulate the implementation of type-specific code together and make your mapping code look high-level and easy to read.
- Use the configuration files and the `TypeKeyMap` utility class, as used in the ACORD mapper. For more information, see “Configuration Files and Typecode Mapping” on page 438.



---

part VII

# ISO and Metropolitan



# Insurance Services Office (ISO) Integration

The insurance service organization ISO provides a service called ClaimSearch that helps detect duplicate and fraudulent insurance claims. If an insurance company enters a claim, the company can send details to the ISO ClaimSearch service, and get reports of potentially similar claims from other companies.

The full ISO XML data hierarchy is in the ISO-provided documentation the *ISO XML User Manual* with file-name *XML User Manual.doc*. ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations. If you want to send ISO optional properties that ClaimCenter does not send by default, use the ISO XML User Manual to find the properties. To add optional properties to the ClaimCenter generated XML payload, see “ISO Payload XML Customization” on page 478.

You can refer to the ISO ClaimSearch web site at:

<http://www.iso.com/Products/ISO-ClaimSearch/ISO-ClaimSearch-Facts-and-Figures.html>

This topic includes:

- “ISO Integration Overview” on page 448
- “ISO Implementation Checklist” on page 454
- “ISO Network Architecture” on page 456
- “ISO Activity and Decision Timeline” on page 460
- “ISO Authentication and Security” on page 466
- “ISO Proxy Server Setup” on page 467
- “ISO Validation Level” on page 468
- “ISO Messaging Destination” on page 469
- “ISO Receive Servlet and the ISO Reply Plugin” on page 471
- “ISO Properties on Entities” on page 471
- “ISO User Interface” on page 472
- “ISO Properties File” on page 473
- “ISO Type Code and Coverage Mapping” on page 476

- “ISO Payload XML Customization” on page 478
- “ISO Match Reports” on page 480
- “ISO Exposure Type Changes” on page 481
- “ISO Date Search Range and Resubmitting Exposures” on page 482
- “ISO Integration Troubleshooting” on page 482
- “ISO Formats and Feeds” on page 483

## ISO Integration Overview

ClaimSearch helps detect duplicate and fraudulent insurance claims. After an adjuster enters a claim, the carrier can send details to the ISO ClaimSearch service and get reports about potentially similar claims from other companies. ClaimCenter includes built-in integration to this service.

After an insured customer calls an insurance company with a claim, the insurance company sends details to the ISO ClaimSearch service to save the claim’s basic information. ISO returns a response with reports describing potential duplicate claims. ISO might also send match reports later if another company enters a matching claim into their database. Consequently, a single submission to ISO may result in several responses: the initial matches and then additional matches, possibly much later.

### Claim-based Messaging, Legacy Support for Exposures

Both Guidewire and ISO recommend that you configure ClaimCenter to send ClaimCenter claims, not exposures, to ISO. For compatibility with older versions of ClaimCenter that supported sending exposures, ClaimCenter supports optionally sending exposures to ISO.

Configure the type of object to send to ISO by editing the `ISO.properties` file property `ClaimLevelMessaging`. By default, this property is set to `true`. If you set this property to `false`, ClaimCenter uses exposure-based messaging.

If you enable exposure-level ISO integration, the ClaimCenter exposure detail page displays the `ISO Resend` button. If you enable claim-level ISO integration, the ClaimCenter claim detail page displays the `ISO Resend` button. These buttons resend a claim to ISO and check on the ISO status. If you use claim-level messaging, the exposure-level ISO information disappears from the user interface. As mentioned earlier, ClaimCenter makes an exception for legacy exposures that are already known to ISO. Those exposures continue to be treated at the exposure level in the user interface and the underlying messaging layer.

If you switch from exposure-based ISO messaging to claim-based ISO messaging, ISO continues to track your previous submissions of ClaimCenter exposures in the ISO system. ClaimCenter handles match reports that ISO sends later based on exposures submitted before the switch to claim-based ISO messaging. ClaimCenter continues to maintain the following exposure ISO-related properties:

- `ISOSendDate` – The ISO send date, which is set in the same database transaction as the new message to ISO.
- `KnownToISO` – The Boolean flag that indicates whether ISO has acknowledged the message. This property is set in the message acknowledgement database transaction.

If ClaimCenter attempted to send an exposure to ISO—the `ISOSendDate` is non-null—ClaimCenter handles ISO replies on that claim by using only exposure-based mode. Once a user attempts to send ISO a claim or exposure, that claim is fixed in that mode, claim-based or exposure-based. This behavior allows a smooth transition from exposure-based ISO messaging to claim-based ISO messaging while still handling exposures submitted to ISO before the switch to claim-based messaging.

The actual implementation of this algorithm is in a property called `ISOClaimLevelMessaging` defined on claims as a Gosu enhancement. It returns `true` to indicate that ClaimCenter is to use claim level messaging for this claim. It returns `false` to indicate exposure-based messaging.

The built-in implementation uses the algorithm described previously for changing modes between exposure-based and claim-based. However, you can configure this property for special business needs. For example, you could choose to send messages at the claim level for a particular loss type or line of business. To make modifications like this, edit the `GWClaimISOEnhancement` enhancement file. However, if you change the logic of the `ISOClaimLevelMessaging` property, you must follow these rules:

- The return result must be deterministic given the same claim. In other words, it must return the same result for that claim if called multiple times.
- However, once ClaimCenter sends a claim or exposure to ISO—the `ISOSendDate` is non-null—that claim always uses that same mode from then on. You cannot mix both ISO levels on a single claim. Even if you make other changes, the return result must preserve this logic.

Most ISO-specific properties and methods apply to both claims and exposures. From an implementation perspective, the similarities between exposures and claims are defined by the fact that `Claim` and `Exposure` entities both implement the interface `ISOReportable`. As you review the documentation or look through built-in code, such as payload generation Gosu code, you can generally think of references to `ISOReportable` as “a claim or an exposure”. However, whether ClaimCenter actually uses claim-level messaging is controlled primarily by the `ISOClaimLevelMessaging` property, which is dynamic, as discussed earlier in this topic.

Because you can send either claims or exposures to ISO, there are two different types of ISO reports. Therefore there are two separate entities: `ClaimISOMatchReport` for claims and `ExposureISOMatchReport` for exposures.

## Configuring Claim-Based Messaging

In the default configuration, claim-level ISO reporting does not report a claim to ISO until all exposures reach the ISO validation level. You can change this behavior to cause ClaimCenter to report claims when one or more exposures reach ISO validation level.

### To send a claim when one or more exposures reach ISO validation level

1. Edit the `ISO.gs` file.

2. In the `checkForClaimChanges` method, look for the following method call:

```
claim.isValid("iso", true)
```

The second argument indicates whether to check the validation levels of all exposures before proceeding. If you pass the value `true`, ClaimCenter skips ISO reporting if any exposure has not yet reached ISO validation level. Instead, change the second argument to `false`:

```
claim.isValid("iso", false)
```

3. Next, add a filter to remove exposures that are not yet at ISO validation level. To begin, edit the file `ISOClaimSearchRequestBase.gs`.

4. Find the `createClaimLevelSearchPayload` method, which iterates through exposures while inserting them into the ISO Claims Search payload.

5. Add a filter to remove non-ISO-ready exposures:

```
for (exposure in Claim.ISOOrderedExposures) {  
    if (exposure.isValid(ValidationLevel.TC_ISO)) {  
        addExposureToClaimLevelSearchRequest(exposure)  
    }  
}
```

## Match Reports

`ISOMatchReport` is a delegate that defines the interface for ISO match reports. The claim and exposure versions of ISO match reports implement this interface. The `ClaimISOMatchReport` and `ExposureISOMatchReport` entities contain the properties `ISOClaim` and `ISOExposure`. For `ClaimISOMatchReport`, the `ISOClaim` property points to the claim and the `ISOExposure` property is `null`. For `ExposureISOMatchReport`, the `ISOExposure` property points to the exposure and the `ISOClaim` property points to the claim that owns that exposure.

If you use exposure-based messaging, remember that the difference in terminology between ClaimCenter and ISO. For exposure-based messaging, ISO calls a *claim* is what ClaimCenter calls an *exposure*.

ClaimCenter has a special validation level for ISO. When a claim or exposure is ready to send to ISO, the validation level matches or exceeds this level. Once an exposure reaches this level, ClaimCenter sends the claim to ISO and records any ISO match reports associated with the exposure.

The relative position of the levels is defined by the **Priority** field on each validation level.

## Implementation Overview

ClaimCenter ISO integration components include:

- **ISO user interface in ClaimCenter** – The built-in user interface includes:
  - a button to manually send/resend a claim or exposure to ISO
  - a tab in the exposure details page for users to view ISO information on an exposure or claim
  - pages to view the details of a particular ISO match.
- **ISO validation level and related validation rules** – Once validation rules indicate a claim or exposure reached the ISO validation level, ClaimCenter knows the claim or exposure can be sent to ISO. For more information, see “ISO Validation Level” on page 468.
- **ISO messaging destination and related event rules** – After a claim or exposure reaches the ISO validation level, or if it changes after reaching this level, built-in messaging rules detect the change. The rules send a request to ISO built-in messaging destinations to generate messages to ISO. ClaimCenter includes built-in Event Fired rules written in Gosu that assemble the payload to send to ISO. The largest part of real-world ISO integrations are customizing these payload-generation rules for changes to your data model and special business requirements, such as custom exposure types. For more information about payload generation functions, see “ISO Messaging Destination” on page 469 and “ISO Payload XML Customization” on page 478.
- ClaimCenter provides payload generating Gosu rules for the most common payload types, and sends all required properties for those payloads. To generate different ISO payloads, to add ISO optional properties, or to add new properties that ISO requires, customize your payload generation functions. The built-in rules provide nearly all of what you probably need for initial deployment, not including your data model changes. You must modify the Gosu code to match your data model changes, to generate new types of payloads, or to add ISO optional properties unsent in ClaimCenter by default.
- **ISO receive servlet that waits for responses from ISO** – After the servlet gets an ISO response, it notifies the destination and updates the claim or exposure with the new response and match report. A match report from ISO describes data from other insurance companies about an exposure that seems similar to a claim or exposure in your system. ClaimCenter stores match reports as ClaimCenter *documents*. The most important details are also added to a new array of **ISOMatchReport** subtype entities within the ClaimCenter claim or exposure. For more information, see “ISO Receive Servlet and the ISO Reply Plugin” on page 471 and “ISO Match Reports” on page 480.
- **ISO properties on ClaimCenter claims, exposures, and vehicles** – For ISO support, ClaimCenter stores special properties on **Claim**, **Exposure** and **Vehicle** entities, plus certain ISO-specific typelists. For more information, see “ISO Properties on Entities” on page 471. Additionally, ClaimCenter stores ISO responses as match reports on the claim or exposure. Additionally, for incoming match reports, ClaimCenter generates documents on the claim or exposure as linked **Document** entities. For more details, see “ISO Match Reports” on page 480.
- **ISO properties files.** There are additional configuration options in the ISO property files (see “ISO Properties File” on page 473)
- **ISO mapping files.** There are additional configuration options in the mapping files for coverages and type-codes (see “ISO Type Code and Coverage Mapping” on page 476)

- **Administration pages to examine outgoing messages.** Although the Administration pages for messages are not ISO-specific, outgoing requests to ISO are handled using the messaging system. It is sometimes necessary for ClaimCenter administrators to track connectivity issues using this user interface. For more information about this administrative user interface, see “Monitoring and Managing Event Messages” on page 64 in the *System Administration Guide*.

## Reference Material for ISO Integration Work

The full ISO XML data hierarchy is in the ISO-provided documentation the *ISO XML User Manual* with file-name is *XML User Manual.doc*. ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations. If you want to send ISO optional properties not sent by ClaimCenter by default, use the ISO XML User Manual for the reference of all their properties. To add optional properties to the generated XML payload, see “ISO Payload XML Customization” on page 478.

For additional related information, refer to the ISO ClaimSearch web site::

[http://www.iso.com/index.php?option=com\\_content&task=view&id=701&Itemid=543](http://www.iso.com/index.php?option=com_content&task=view&id=701&Itemid=543)

## What To Know About ISO Mapping

The most important thing to know about ISO integration is that largest part of real-world ISO integrations are customizing payload-generation rules. You probably need to customize the rules for changes to your data model and special business requirements, such as custom exposure types. Additionally, you probably need to change typelists and update ISO payload rules accordingly. This process is called *mapping* the ClaimCenter view of your business data with ISO’s structure for similar data.

Secondly, you need to know that mapping is very important and that business analysts are critical for every ISO integration team to help with mapping. See “Step 2: Team Resources” on page 454.

It is critical that before ISO integration work begins, business analysts determine the relationship between custom exposure types and typecode values and their corresponding types in ISO. This information is used in mapping files that map ClaimCenter typecodes to their ISO equivalents as well as in the payload generation rule customizations.

You must allocate time in your integration projects to determining these relationships, customizing the ISO integration, and testing it. Additionally, even after you have pushed your system to production, if you make ongoing change your data model, you must make matching ISO changes. For example, if you add new typecodes to a typelist used by ISO integration, you must update mapping files that map the ClaimCenter new typecodes to the ISO equivalents.

Mapping is typically involves some estimation and research, and sometimes there is no clear answer to a mapping issue. For example, by default, vehicle damage in ClaimCenter maps to what ISO calls a property payload. For a variety of business reasons, this tends to lead to better ISO match result quality, even though it tends to lower the number of total returned matches. Give yourself adequate time for designing, finalizing, and testing your mapping. Allow at least three to five weeks.

You may additionally need to carefully consider how any user interface changes in ClaimCenter might affect your required properties. For example, it may seem like hiding some properties on exposures or subobjects you do not use would have no negative effects. However, if those properties are strictly required by ISO, hiding those properties in the user interface is typically not recommended due to the changes required. If you choose to do it anyway, you must make changes in the payload generation rules to add placeholder data in the XML payload. This causes ISO to accept the records as valid so that it does not return errors.

Set your expectations that you need more ISO integration mapping work every time you make any user interface changes on exposures or any of its subobjects. This is particularly true if you hide built-in properties. As a general rule, if you make changes after initial ISO deployment, you may require additional ISO-related mapping work.

Similarly, if you modify the data model on any typelists on ISO required properties, you might need to rewrite typecode mapping and possibly the payload generation rules. Set your expectations that you might need more ISO integration mapping work every time you make data model changes on exposures or any of its subobjects. If you make changes after initial ISO deployment, you may require additional ISO-related mapping work.

The mapping process is typically not extremely complex. However, it is time consuming and typically is an ongoing process because of ongoing data model and user interface changes. You must adapt the mapping process for your own business needs.

The most important files related to mapping are:

- the ISO payload generation Gosu rules, especially to accommodate new exposure types
- the ISO properties files, especially to map settings related to the `ClaimContact` entity. If you want more complex claim contact mapping than supported by the ISO property files, you can customize the Gosu that loads this properties file.
- the ISO typecode mapping file `TypeCodeMap.xml`
- the ISO coverage mapping file `ISOCoverageCodeMap.csv`

Important subtopics for mapping are the following:

- “ISO Payload XML Customization” on page 478
- “ISO Exposure Type Changes” on page 481
- “ISO Properties File” on page 473
- “ISO Type Code and Coverage Mapping” on page 476

## Implementation Technical Overview of ClaimSearch Web Service

ISO ClaimSearch is a web service accessed using the SOAP protocol over a secure sockets (SSL) connection. ISO provides two ClaimSearch systems: the *live system* and a *test system*. Both systems provide the same interface and functionality, but the test system allows clients to test and experiment with their ISO integrations without affecting the live system. The following table lists the two URLs that provide these services:

Type of system	URL
The live system	<a href="https://clmsrchwebsvc.iso.com/ClaimSearchWebService/XmlWebService.asmx">https://clmsrchwebsvc.iso.com/ClaimSearchWebService/XmlWebService.asmx</a>
The test system	<a href="https://clmsrchwebsvct.iso.com/ClaimSearchWebService/XmlWebService.asmx">https://clmsrchwebsvct.iso.com/ClaimSearchWebService/XmlWebService.asmx</a>

Each system provides two operations: `SubmitToISO` and `SubmitTestToISO`. Both take a single argument which is an XML string. The XML is the ACORD format with a few ISO extensions. The XML data describes the claim submitted to ISO or updates to a claim already known to ISO. The following table compares and contrasts the two operations:

Operation	Behavior
<code>SubmitToISO</code>	<code>SubmitToISO</code> checks the XML and queues it up for addition to the ISO database. After the request processes, ISO sends an asynchronous response.
<code>SubmitTestToISO</code>	<code>SubmitTestToISO</code> checks the XML but does not add it to the queue. The ISO database does not update and ISO does not send an asynchronous response. <code>SubmitTestToISO</code> is only useful for testing that you can connect to ISO and submit correctly formatted XML.

At the time you register as a new customer with ISO, you must specify a range of IP addresses from which you send requests. You must also specify a callback URL to which ISO sends asynchronous responses. ISO allocates a customer ID and password that you must put in the header of every XML request. All communication to and from ISO is done over secure sockets (SSL) so your ID and password cannot be intercepted. For more information about setting up your account, see “ISO Implementation Checklist” on page 454.

You submit ISO requests with the following steps:

1. ClaimCenter submits a request using `SubmitToISO`.
2. ISO receives a request and immediately checks all of the following:
  - The request contains a valid customer ID and password.
  - The source IP address of the request is within the range allocated to the customer.
  - The request contains correctly formatted XML.
3. ISO sends back the return value, which is a short XML message known as a *receipt*. If the ClaimCenter request passes the initial checks, then the receipt contains the status value `ResponsePending`. This means that the request queued up and processes soon. If the request did not pass the checks, then the receipt contains an error message.

---

**IMPORTANT** A good receipt does not guarantee that the request is correct. ISO performs most of its detailed error checking as it actually processes the request later.
4. ISO processes the request. First, the request is checked for errors. For instance: all required properties must be present; also, policy, coverage and loss types must be consistent. If the request is correct, it is added to or updated in the ISO database. Then, a query is run to find if there are other matching claims.
5. ISO sends a response to the your callback URL. If the request was valid, then the response contains a `Success` status value plus details of any matches. If the request was invalid, then the response contains a description of the problem.
6. At a later time, if another company submits a claim to ISO and it matches one you submitted, then ISO sends another response to your callback URL. The response contains the new list of matching claims.

There are some variations to this basic protocol. ClaimCenter requests to ISO fall into three major categories: *adding* a new claim, *replacing* the information about an existing claim, and *updating* an existing claim. In most cases, you can specify with a property in the XML data, whether or not you want a search to be done after the add/replace/update.

## Enabling ISO Integration

The ISO destination is a built-in destination that handles outgoing requests to ISO. This built-in destination includes a built-in message transport plugin to handle outgoing ISO communication and the a built-in message reply plugin to handle incoming ISO messages. Enabling the ISO messaging plugin and the messaging transport is the way to enable ISO support.

### To enable built-in ISO support

1. In Guidewire Studio, navigate to `configuration` → `config` → `Messaging`.
2. Double-click `messaging-config.xml` to open it in the editor.
3. Click the row with ID 66 and the name `Java.MessageDestination.ISO.Name`.
4. Select the `Enabled` check box.
5. Navigate to `configuration` → `config` → `Plugins` → `registry`.
6. Double-click `isoTransport.gwp` to open it in the editor.
7. If not checked already, select the `Enabled` check box.

For more information about messaging plugins, see “[Messaging and Events](#)” on page 299.

# ISO Implementation Checklist

Integrating Guidewire ClaimCenter with ISO ClaimSearch requires some planning and interactions with ISO. Guidewire recommends that project teams that plan ISO integration must include the following deployment steps in project planning timelines:

- “Step 1: ISO Membership” on page 454
- “Step 2: Team Resources” on page 454
- “Step 3: Requesting ISO Testing and Production Server Access” on page 454
- “Step 4: ISO Configuration and Integration Testing” on page 455
- “Step 5: ISO Integration Data Migration” on page 455
- “Step 6: ISO Production Testing, Including Connectivity Testing” on page 455
- “Step 7: ISO Production Deployment” on page 456

## Step 1: ISO Membership

If your company is new to the Insurance Services Office, you must purchase an ISO membership.

Purchasing a membership typically includes contract negotiation with ISO. This can take various lengths of time depending on the needs of both companies, corporate legal interaction, and final steps for approving and finalizing contracts. Guidewire recommends that projects begin this membership phase immediately to reduce delays to later ISO integration phases.

Once membership is complete, ISO provides you with an *ISO username*, an *ISO password*, and *ISO company code* (an ISO-specific identification code separate from the ISO username). For more information about passwords, see “ISO Authentication and Security” on page 466.

Be sure you get contact information for people at ISO, for both administrative and technical contacts.

**IMPORTANT** Begin the ISO membership process immediately.

## Step 2: Team Resources

You must ensure you have the appropriate set of team members for an ISO integration.

Your team must include the following:

- **A business analyst from the insurance company for mapping coverages and properties** – ClaimCenter provides payload generating Gosu rules for the most common payload types, and sends all required properties for those payloads. If you want to generate different ISO payloads, or add ISO optional properties, or add properties that ISO later makes required, customize the payload generation functions. The largest part of real-world ISO integrations are customizing these payload-generation rules for changes to your data model and special business requirements, such as custom exposure types. In practice, a business analyst from the insurance company is the best resource for understanding the business requirements and the coverage implications for their business needs.
- **Engineers to implement the integration** – Some combination of your engineers, perhaps including additional Guidewire Professional Services engineers.
- **Access to an experienced integration ISO consultant** – Due to the specialized nature of ISO integrations, each ISO integration team must have access to an experienced integration ISO consultant that they can contact for questions.

## Step 3: Requesting ISO Testing and Production Server Access

To properly test ISO integration, make a request to ISO to access the ISO ClaimSearch *test server*. ISO typically makes the test system ready to you within two business weeks of receiving a request. You must perform extensive testing with the ISO test server before moving any data to ISO final production with the *production servers*.

To integrate with production servers, you must make a request to ISO to access the ISO ClaimSearch *production server*, which is the public (real) database for your data. ISO typically makes the test system ready for you within two business weeks of receiving a request.

In your request to ISO to access to the ISO ClaimSearch test servers or production servers, you tell them your callback URL and an IP Address. The callback URL must be secured for HTTPS (SSL over HTTP) communication. An SSL certificate signed by a *trusted authority* must be used so that ISO can authenticate the URL. If a valid certificate is not already available, one must be purchased.

Whether or not a certificate is available yet, Guidewire recommends that you explicitly contact ISO to ensure that ISO recognizes the existing certificate or the about-to-be-purchased certificate.

For more information about passwords and ISO security, see “ISO Security with Customer Passwords” on page 466.

Remember to have a valid *agency ID* string, which is stored in the ISO data path `MiscParty/ItemID/AgencyID`.

You must also tell ISO the acceptable IP address ranges for outgoing requests to ISO. For more information, see “ISO Security with Customer IDs and IP Ranges” on page 466.

On the access form that ISO provides, there is an important property in the section “XML Request from Customer to ISO ClaimSearch”. There is a choice “Please indicate your preferred method of communication with ISO ClaimSearch.”. For the choice “HTTPS Post”, select *No*. For the choice “Web Services using SOAP”, select *Yes*.

#### **Step 4: ISO Configuration and Integration Testing**

Carefully read this entire topic for ISO configuration and integration information, particularly these sections:

- “ISO Properties File” on page 473
- “ISO Type Code and Coverage Mapping” on page 476
- “ISO Payload XML Customization” on page 478
- “ISO Match Reports” on page 480

After you are ready for testing, thoroughly test ISO integration using the ISO test servers. Because of the complexity of ISO, this process sometimes takes longer than initially planned. Guidewire recommends that you budget time for extensive testing with potentially delays as you refine your ISO configuration.

#### **Step 5: ISO Integration Data Migration**

If your legacy systems were integrated with ISO systems, you may need to update ISO data. Specifically, you must mirror changes that you make to claim data as you convert from a pre-ClaimCenter system to Guidewire ClaimCenter by updating ISO database records.

Along with ISO data format differences, any changes in ISO *feed types* affect the complexity of the data migration. For more information about ISO feed types, see “ISO Formats and Feeds” on page 483.

#### **Step 6: ISO Production Testing, Including Connectivity Testing**

Before the final transition to the production server, Guidewire strongly recommends a connectivity test to confirm the network configuration, including firewalls and proxies. At the minimum, be sure to do a minimal test with the validation URL with *Message 1 : Claims Sent* as described in “ISO Network Layout and URLs” on page 457.

Test connectivity with the validation URL before submitting claims to the production database. Be careful not to submit any claims to the production server by using the wrong URL. For URL information, see “ISO Network Layout and URLs” on page 457.

Additionally, you can submit claims to the production database to test validity and ensure they are rejected by purposely changing the address or city to be empty. ISO always rejects claims with an empty address or empty city.

If you want to temporarily cause this type of rejection, you can remove these properties by customizing the code that generates ISO payloads. ClaimCenter sends the claim to ISO and receives a reply to *Message 1 : Claims Sent*. Next it sends an additional *Message 2 : Claims Processed* message and Message 2 contains a rejection.

If you do not ever receive Message 2, your system has *at least* one connectivity problem to resolve before going live with your production system. If you temporarily modify the payload to generate a rejection as discussed earlier, be extremely careful. Ensure that you undo any temporary changes before going live.

Run thorough tests and submit a few records to the production database before final deployment. Final deployment requires you to set the ISO URL to the *submission URL* (not the *validation URL*).

Coordinate ISO production server tests carefully with ISO to reduce the chance of accidentally corrupting ISO's production database with invalid data.

To change the ISO URL that ClaimCenter uses, change the `ISO.ConnectionURL` setting within `ISO.properties`. You may need to change firewall and proxy settings in conjunction with this URL change. For more information about properties in this file, see "ISO Properties File" on page 473.

### Step 7: ISO Production Deployment

After completing production testing, your systems are ready for production deployment.

## ISO Network Architecture

ClaimCenter includes built-in support for communicating with ISO. However, ISO's asynchronous responses require special network configuration including a network proxy to safely and efficiently handle ISO responses.

There are two main issues involved:

1. **For security, a network proxy insulates internal networks from external messages** – ISO must send its responses to a your callback URL. This means you must expose the callback URL outside the your Internet firewall. If you do not want to expose your ClaimCenter URL outside the firewall, you must have another server acting as the proxy.
2. **For performance, off-load SSL to a server other than ClaimCenter** – Although ClaimCenter can use SSL for outgoing requests, it is most efficient to handle SSL encryption outside ClaimCenter. This is because Java-based SSL uses unnecessarily large processing resources. Instead, encode and decode SSL using a separate proxy that can implement SSL faster, including native Apache web server support or other proxy systems designed for this task.

Both issues are solved using an extra *proxy server*, sometimes called a *bastion host*, which lives in the area of the your network called the DMZ (De-Militarized Zone). The DMZ contains computers that are accessible from the outside world but partitioned off from your main network for network security. Network firewalls control access between the outside world and the DMZ, and also between the DMZ and the main network. The proxy server's job is to provide a secure gateway between an external service (in this case, ISO) and an internal server (in this case, ClaimCenter).

For outgoing messages from ClaimCenter, the proxy server forwards the request to ISO, and also typically wraps the request in an encrypted SSL/HTTPS connection. For incoming messages from ISO, the proxy server decodes the encrypted SSL/HTTPS request from ISO and forwards the request to the `ISOReceive` servlet, which is part of ClaimCenter. For more information about options for forwarding applications, see "ISO Proxy Server Setup" on page 467 and "Proxy Servers" on page 619.

## Basic ISO Message Types

There are three basic messages between ISO and ClaimCenter:

## Message 1: Claims Sent

After you create a new claim, ClaimCenter sends the claim to ISO for processing in what this documentation refers to as *Message 1: Claims Sent*. Also, certain types of claim changes cause ClaimCenter to resend the claim to ISO. In this message, ClaimCenter sends a SOAP request (over HTTPS) to ISO with an XML payload that contains information about a new claim or changes to an existing claim.

The reply to this message (HTTPS request result) contains an *ISO receipt* with the following information:

- indicates that data was received
- validates the well-formedness of the XML
- confirms a valid ISO customer ID
- confirms the ISO customer password
- confirms that the request came from a valid IP address for this ISO customer

Additional confirmation information is included in *Message 2: Claims Processed*.

For Message 1, ISO supports HTTPS POST format or SOAP format. However, ClaimCenter only supports SOAP format for Message 1. Inform ISO of your requirement for SOAP format during initial ISO registration on the form that they provide on the line that says:

**Please indicate your preferred method of communication with ISO ClaimSearch.**

---

**IMPORTANT** You must inform ISO of their requirement for the SOAP format during initial ISO registration on the form that they provide.

---

## Message 2: Claims Processed

After *Message 1: Sending Claims* completes, including its HTTPS reply, ISO starts to process the new claims or the claim changes. Some time later, ISO finishes processing the request and asynchronously notifies ClaimCenter that the ISO databases now contain the new claim information. ISO initiates an HTTPS POST message to your callback URL confirming the claim submission or if there are errors. Possible errors could include incorrectly defined or missing properties that prevented claim submission or claim changes. This documentation refers to this message as *Message 2: Claims Processed*.

For Message 2, ISO always uses the HTTPS POST protocol, not the SOAP protocol. The choice mentioned earlier about the ISO registration form does not affect this.

## Message 3: Matches Detected

At some future time, ISO might detect claims from other companies that match those submitted by your ClaimCenter implementation. This might occur soon after ClaimCenter sends the claims information, or it might occur much later, or it might not occur at all. If it happens, ISO initiates an HTTPS POST message to your callback URL with information about the matching claims. This documentation refers to this message as *Message 3: Matches Detected*.

For Message 3, ISO always uses the HTTPS POST protocol, not the SOAP protocol. The choice mentioned earlier about the ISO registration form does not affect this.

## ISO Network Layout and URLs

From the ISO perspective, there are two basic types of URLs: the URLs from you to ISO and the *callback URLs* from ISO back to you. However, Guidewire strongly encourages you to put a proxy server between your

ClaimCenter implementation and the ISO servers. Consequently, most proxy server configurations include the following basic network areas:

1. **Your intranet** – This internal network is not directly accessible from the Internet, and is the site of your ClaimCenter implementation. Your intranet is, however, accessible from your DMZ. The DMZ insulates your sensitive systems from hackers and other intrusions.
2. **Your DMZ** – This is the part of your network that isolates your intranet from the potentially dangerous Internet. Your DMZ is protected on both sides by firewalls. One firewall tightly controls access to and from the unsecured Internet. Another firewall tightly controls access to and from the DMZ to your intranet.
3. **The Internet** – Presume that the Internet is unsecured and dangerous. All connections over the Internet happen with secure sockets layer (SSL), which provide encryption and identity confirmation in HTTPS connections (SSL over HTTP).
4. **ISO's network** – This is the ISO network as viewed from the Internet. It is protected by a firewall and exposes only a handful of IP addresses to the outside world. One exposed IP address is the server that handles incoming requests. Another IP address (although theoretically it could be the same IP address) is the computer on ISO's network that performs callbacks to your ClaimCenter implementation.

With this in mind, note the four different URLs in ISO communication. The following list describes each one. After the list is a diagram of the network architecture showing each URL. The URLs include placeholder variables using a dollar sign (\$), which is the syntax used in the example Apache directive configuration files included with ClaimCenter.

The URLs to be configured are as follows:

- **URL #1** – The outgoing request for *Message 1: Claims Sent* as viewed from your intranet. The port number is configurable. The format of the URL is: `http://$DMZProxyDomainName:$DMZProxyPortA`. The proxy server translates this into URL#2. The ISO receipt in the response of URL#2 becomes the ISO receipt response for URL#1.
- **URL #2** – The outgoing request for *Message 1: Claims Sent* defined by ISO. This is one of several ISO-specified URLs that always uses HTTPS and always on port 443. Use different URLs depending on whether you are simply validating basic connections, testing submissions with the ISO testing server, or testing with the ISO production (real) server.

**Testing URL** – Use `https://clmsrchwebsvct.iso.com:443/ClaimSearchWebService/XmlWebService.asmx` if calling ISO for programmatic test submission. The XML message validates and the claim submits to the ISO test database. ISO returns *Message 2: Claims Processed* after the claim inserts into the database and one or more instances of *Message 3: Matches Detected* after matches are found. This URL always uses the test database, not the production ISO database.

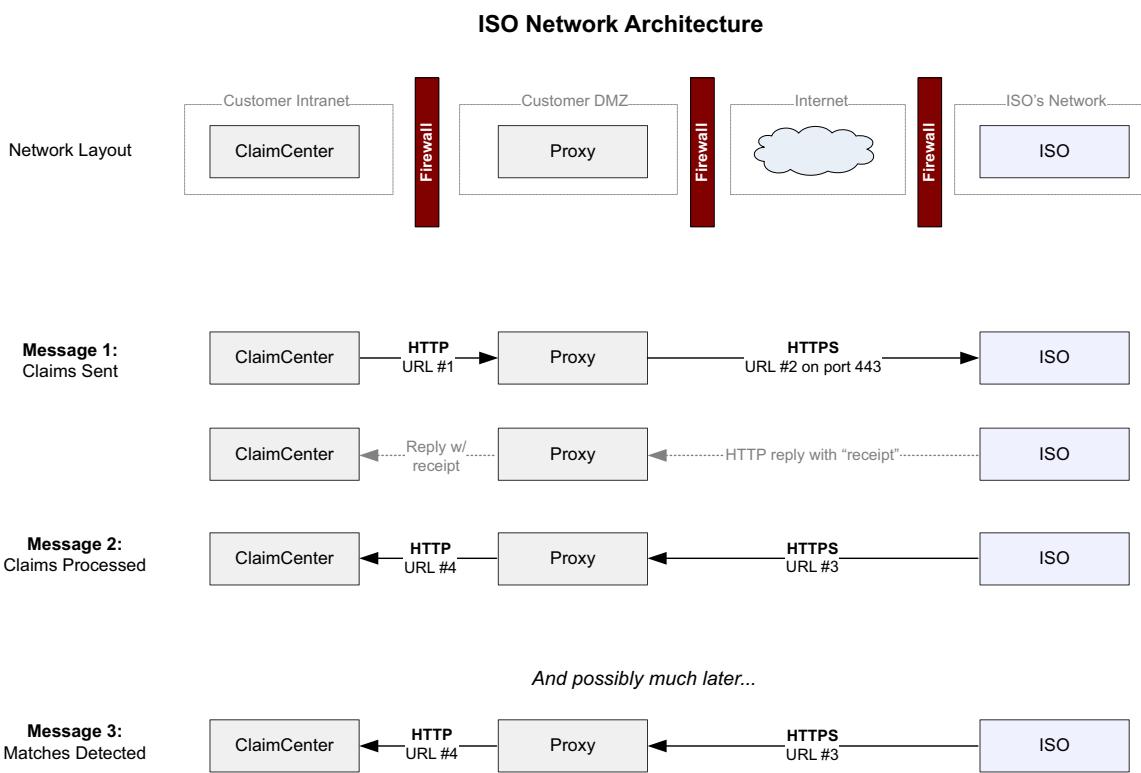
**Validation URL** – Use `https://clmsrchwebsvc.iso.com:443/ClaimSearchWebService/TestUtility.htm` if calling ISO for programmatic validation. ISO tests that validity of your XML message and authentication without submitting the claim to any ISO database. Although there is an immediate reply to the request containing a receipt, ISO never sends return messages as a consequence of calling the validation URL. This URL technically uses the production database but never affects the database content since no claim submits to the database.

**Submission URL**. Use `https://clmsrchwebsvc.iso.com:443/ClaimSearchWebService/XmlWebService.asmx` if calling ISO for programmatic submission to the production (real) database. The XML message validates and the ClaimCenter submits the claim to the ISO production database. ISO asynchronously returns *Message 2: Claims Processed* after the claim inserts into the database. ISO might later send one or more of *Message 3: Matches Detected* after finding matches.

In all URL variants listed, the HTTP/HTTPS reply receipt indicates the validity check success. The validity check includes XML well-formedness, ISO customer ID, ISO customer password, and the authorized IP address range. The complete ISO URL is referred to in the example Apache directive configuration files as `$ISO_URL`.

- **URL #3** – The ISO callback URL for *Message 2: Claims Processed* as viewed from the public Internet. This typically would point directly to your main external firewall. The port number is configurable. The format of the URL is `https://$DMZProxyDomainName:$DMZProxyPortB`. Ask ISO what IP address from which they send the callbacks. This IP address, which may not have a corresponding domain name, is referred to in example Apache directive configuration files as the IP address `$ISOCallbackIP`. The immediate HTTP reply to this request does not contain significant information.
- **URL #4** – The ISO callback URL for *Message 2: Claims Processed* to the ClaimCenter server as viewed from your proxy server in the DMZ. The port number is configurable. The format of the URL is `http://server:port/cc/ISOReceive`. The immediate HTTP reply to this request does not contain significant information.

The typical ISO network setup, the three ISO messages, and the four ISO URLs are illustrated in the following diagram. Solid black arrows in the diagram show the primary direction for the HTTP/HTTPS request, but in all cases there is a synchronous (immediate) HTTP/HTTPS reply. However, only for *Message 1: Claims Sent*, is the content of the reply used at all.



## ISO and ClaimCenter Clusters

If you use ClaimCenter clusters, the batch server is the only server in the cluster that interacts with ISO from a network architecture standpoint. That interaction occurs through your network proxy.

If a user makes a change on a claim or an exposure that triggers sending an exposure to ISO, that change submits a message to the messaging send queue. ClaimCenter stores the message in the database as a Message entity. The ISO messaging code that runs ISO-related rules runs on the server that triggered this change. However, after submitting the message to the send queue, ISO communication is managed only from the server designated the batch server.

There are two aspects to this:

- Because the messaging send queue and messaging plugins only run on the batch server, only the batch server sends outgoing messages to ISO (through your proxy server). Other servers in the cluster are not directly involved.
- The ClaimCenter ISO servlet runs only on the batch server, waiting for match requests from ISO through your proxy server. Other servers in the cluster are uninvolvement.

Remember to configure your firewall and proxy servers accordingly.

For ClaimCenter clusters, the batch server is the only server that attempts to interact with ISO's server (through the network proxy) in either direction.

## ISO Activity and Decision Timeline

This section describes the process of sending details of an exposure or claim to ISO and getting responses.

### Initial ISO Validation

Initial validation happens in the following steps:

1. The exposure is created.
2. The exposure is validated to level ISO. The ISO validation rules run and verify that all properties needed by ISO are present and correct. The ISO message rules examine the exposure and construct an appropriate message payload, which ClaimCenter puts on the central message send queue in the database.
3. On the batch server only, ClaimCenter gives the ISO messaging destination (the messaging plugins) one message to send from the message queue. The destination sets a few properties in the payload - for example, the message ID and the property indicating whether this is an *add* (initial search) or *replace*. Because this claim or exposure is not marked as *known to ISO*, the destination marks this message as an *add*.
4. The ISO destination sends the message to ISO, using SSL. It receives an immediate receipt indicating whether the message is correctly formatted and contains the correct authentication information. If the receipt is bad the destination adds a non-retryable error acknowledgement to indicate that resending the message does not help and the configuration probably needs to be fixed.
5. The ISO destination waits for a response from ISO. The initial receipt is **not** an acknowledgement (though it can be treated as an error acknowledgement if it is bad). Only the full response is considered to be an acknowledgement.
6. The ISO server processes the request and sends a response to the SSL callback URL, in other words, the *bastion host*.
7. The bastion host forwards the response on to the ISOReceiveServlet, running in the ClaimCenter server
8. The ISOReceiveServlet authenticates the response (it checks the password, message ID, and other properties), parses the response, and notifies the ISO destination. For most of the actual reply handling, however, the servlet delegates the work to the ISOReplyPlugin class. The ISOReplyPlugin class is a MessageReply plugin implementation responsible for the asynchronous message reply. For more information about message reply plugins and overall messaging flow, see “Messaging Overview” on page 300.
9. The ISOReplyPlugin message reply plugin marks the original message as *acknowledged*. If the response was good, the message marks the claim or exposure as “known to ISO”. This means that future requests are *replace* requests rather than *add* requests. If the response contained errors such as missing required properties, a non-retryable error acknowledgement is submitted. Again, resending the message does not help and you likely need to correct the configuration.
10. If the response was good, the ISOReplyPlugin sets the ISO receive date on the claim or exposure, adds the match report document and creates ISOMatchReport entities for any match reports

11. During message reply handling in the `ISOReplyPlugin` plugin, ClaimCenter adds any activities necessary to review the claim or exposure.

#### Simple Claim Change

1. User changes a property on exposure or an associated claim, policy or claim contact, causing a change event.
2. ISO business rules check if any property important to ISO changes. If so, constructs a message payload and adds it to the message queue.
3. Similar to the “Initial Validation” steps listed earlier as steps 3 through 11. However, ClaimCenter marks it as a replacement because the claim or exposure property for “known to ISO” is set to `true`. If any match reports are returned, ClaimCenter adds another document, but only adds new `ISOMatchReport` entities if they differ from the previously added `ISOMatchReport` entities. If they do not differ, ClaimCenter updates the existing entity’s received date, but does not create a new entity.

#### Simple Key Field Change

ISO uses a set of *key fields* to identify a claim. These key fields are the claim number, policy number, agency ID, and loss date (only the date matters, not the time of day). If the user changes any of these fields (*loss date* is the only property currently available through the user interface), the following occurs:

1. User changes a key field.
2. ISO destination rules detect that key field has changed, constructs special *key field update* message payload and adds it to the message queue.
3. ISO destination sends the special payload to ISO, handling the ISO receipt normally.
4. ISO does **not** send a response if a key field update message succeeds. So the ISO destination sets a timeout configurable using the `ISO.KeyFieldUpdateTimeout` property. If no error response is received within the timeout, the destination assumes the message has succeeded.

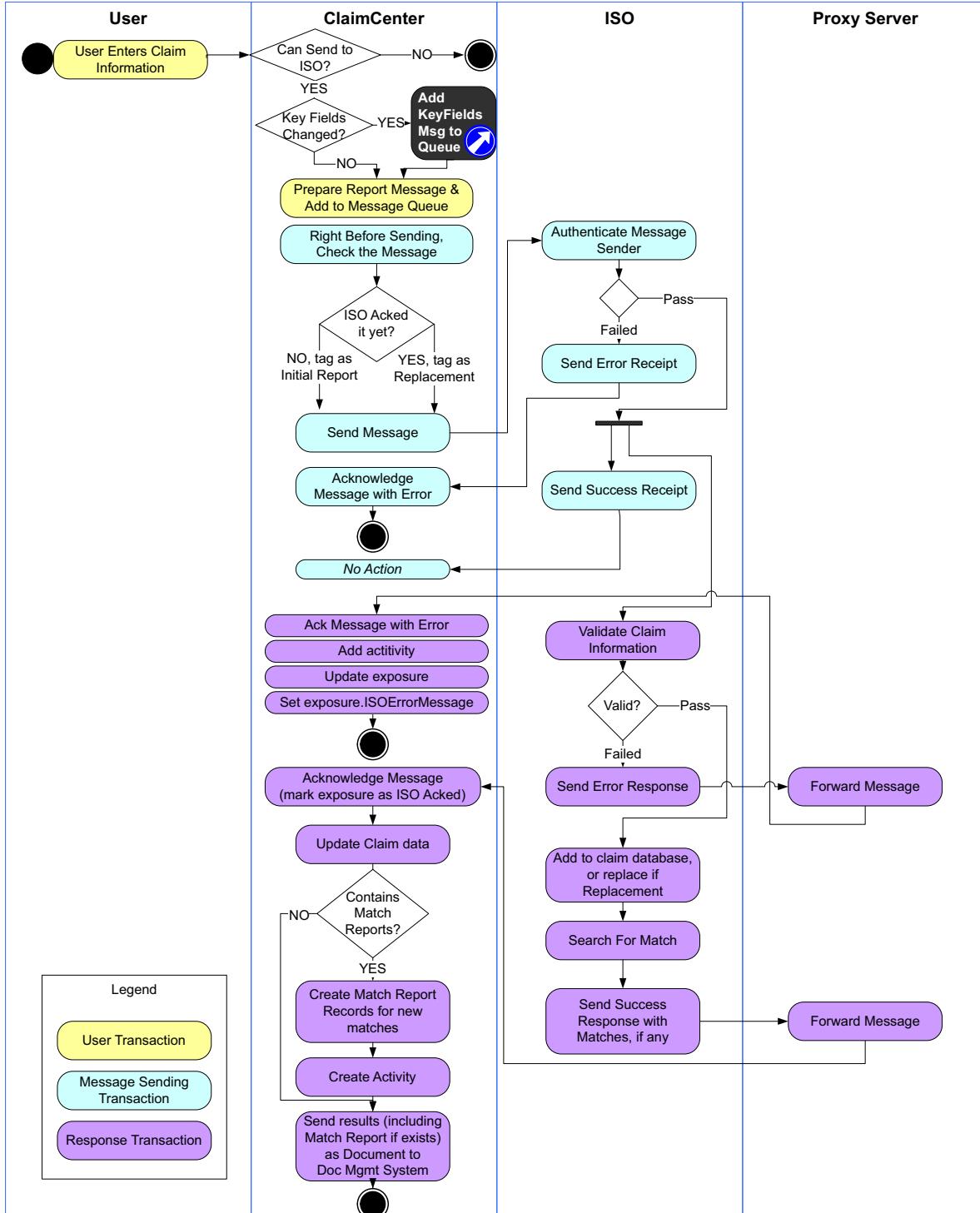
After the key field update message is handled another “replace” message is sent to get any new match results.

## ISO Activity and Decision Diagrams

The following diagram describes the ISO initial report and claim replacement activity. They are basically the same flow. The main difference is that immediately before the next message is sent from the message queue, the

ISO message-sending code checks if that exposure was acknowledged by ISO. If it was, it marks a flag in the ISO payload as a claim replacement instead of a new claim.

## ISO Initial Report or Claim Replacement Activity



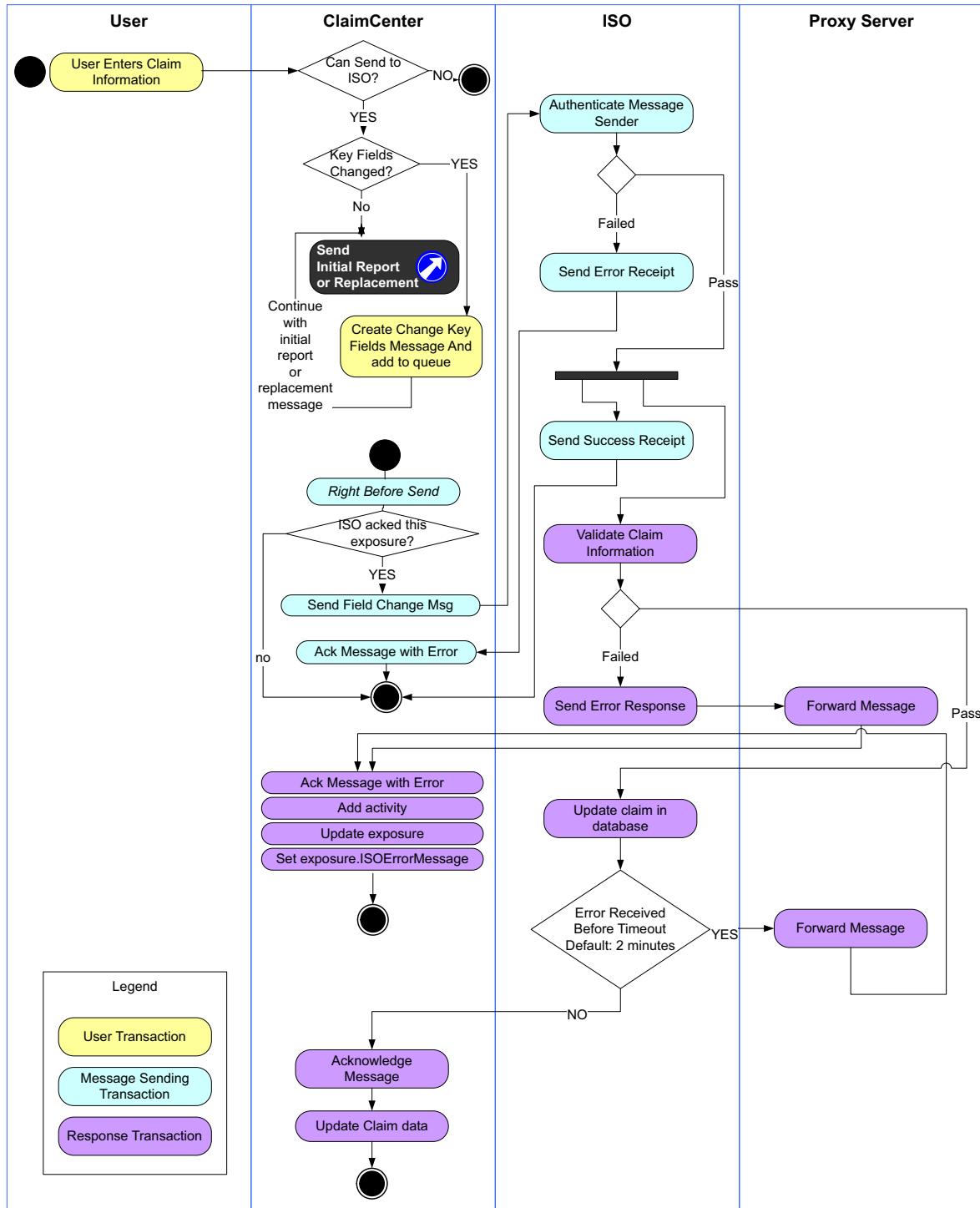
The following diagram describes the activity during a key fields change message. A key fields change message is a special type of message that ClaimCenter sends to ISO. For claim-based ISO messaging, ClaimCenter sends a

key fields message after any key fields change on a claim. For exposure-based ISO messaging, ClaimCenter sends a key fields message after any key fields change on an exposure.

Because ClaimCenter and ISO use these properties to track identity and uniqueness for the entities in ClaimCenter, ClaimCenter must tell ISO about the change. If ISO has not heard of the item yet, any changed properties are irrelevant to ISO. Immediately before the message is sent, ClaimCenter checks whether ISO acknowledged that exposure by checking the `KnownToISO` property on the claim or exposure. If ClaimCenter

thinks ISO knows about the claim or exposure, then it sends the key fields change message. Otherwise, ClaimCenter ignores the key fields message and does not send the message to ISO.

### ISO Key Fields Changed Activity

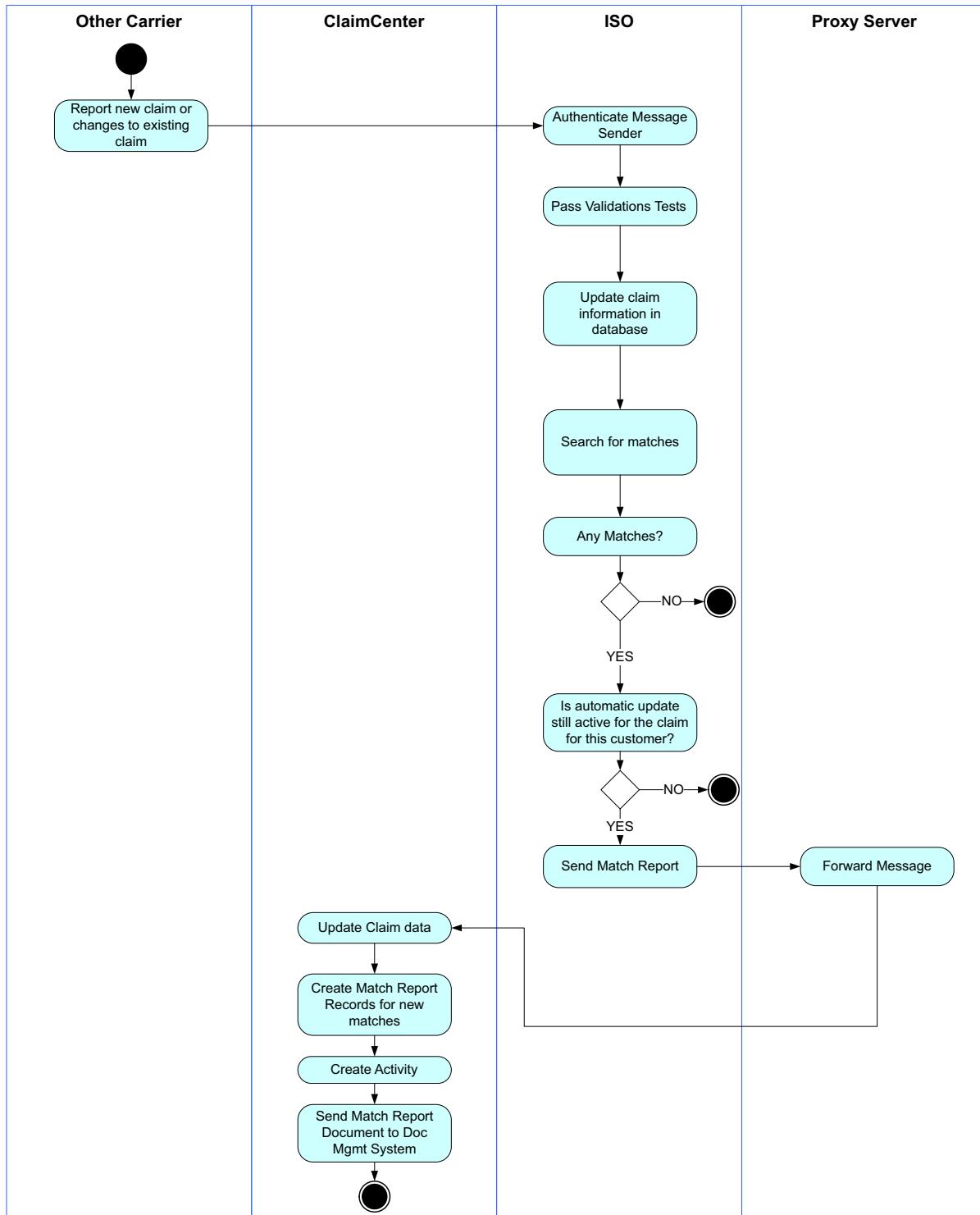


#### Legend

- User Transaction
- Message Sending Transaction
- Response Transaction

The following diagram describes the activity during a post-submit scan for matches after another carrier finds a match. This is analogous to Message 3 in the diagram in section “ISO Network Layout and URLs” on page 457.

### ISO Post-Submit Match Report Activity



## ISO Authentication and Security

All ISO communication across the unsecured Internet uses *secure sockets layer* (SSL) connections to protect claim data. SSL lets the message sender and receiver confirm the other's identity and guarantees that no one can eavesdrop on the connection. For example, the ISO destination can be sure it is communicating with ISO because only ISO has the private encryption key associated with the certificate of <https://clmsrchwebsvc.iso.com>. Similarly, ISO can ensure it is connecting to the correct callback URL because only you have the private key associated with the certificate of the callback URL. Both ISO and you have your own public/private encryption key pairs and associated certificates signed by trusted parties.

The proxy server (not ClaimCenter) is the server that must contain all the appropriate encryption keys and certificates to ensure safe and secure SSL connections. Therefore, the proxy server is the main component of the ISO architecture that must be hardened to avoid unauthorized access to the sensitive private keys.

In the setup described earlier, connections between ClaimCenter and the proxy server are not encrypted with SSL. This is the typical setup because intranet connections typically are presumed to be secure. Also, this approach reduces server processing load for encrypting SSL/HTTPS data on the ClaimCenter server. Performing SSL encryption within the Java virtual machine is particularly resource intensive. If you have some special reason for encrypting data between ClaimCenter and the proxy server, it is possible. However, SSL configuration is more complex and this approach is not fully discussed in the ClaimCenter Integration Guide.

### ISO Security with Customer IDs and IP Ranges

All requests to ISO must contain a valid *ISO customer ID*. Because ISO tracks a range of IP addresses for each ISO customer ID, ISO strictly requires requests contain a valid customer ID and comes from your appropriate IP address range. Because requests from ClaimCenter to ISO go through the firewall, ClaimCenter requests appear to come from the IP address of your firewall. As you register with ISO, you tell ISO an IP address range of IP addresses that are valid for proxy server outgoing requests.

Similarly, set up the proxy server so that it rejects responses that do not come from ISO's IP address. Confirm with ISO precisely what IP addresses are sources of the callback requests. These IP addresses do not need to be associated with externally-identifiable domain names (*DNS names*). You can hard code these IP addresses by number in your firewall configuration.

For the ISO *production servers*, confirm with ISO what IP address from which they send the ISO callbacks for *Message 2* and *Message 3*. See “ISO Network Architecture” on page 456. As of the publication of this documentation, the production IP addresses were 206.208.171.134, 206.208.171.63, and 206.208.171.89. These ISO IP addresses may not have corresponding domain names.

For the ISO *testing servers* (not the production servers), several different servers can send the callback messages. The IP addresses of the test servers are 206.208.170.244, 206.208.170.250, and 206.208.170.249. These ISO IP addresses may not have corresponding domain names.

Configure any of your firewalls, intermediate computers, or reverse proxies to allow incoming ISO messages from all of these servers.

If you have any problems with this type of configuration, or you think messages are coming in from other IP addresses, immediately contact ISO to confirm any changes in configuration. Do not simply allow incoming messages from other IP addresses that you might see, as they may be unauthorized requests from unauthorized IP addresses.

### ISO Security with Customer Passwords

All requests to ISO must contain a password. ISO checks that the password matches the customer ID and rejects the request if it does not match.

Similarly, the ISOReceive servlet, which runs as part of ClaimCenter to receive the ISO callback, rejects responses that do not contain the correct customer password. This means that it is vital for you to protect your password in a configuration file on your ClaimCenter server.

After you receive your initial username and password from ISO (typically using email), immediately change the password to avoid serious problems later on due to expiration of ISO-generated passwords. Once you change the password, then the ISO password never expires.

Change your ISO passwords using two different methods:

- Tell ISO what passwords to use. Contact ISO directly using phone or other secure system, not unsecured email.
- After you first log on to the ISO web site, specify a new password.

If you do not change the initial password immediately, then you must change it before it expires. Changing it immediately is better than waiting since the change itself can disrupt ISO service during the change. After you change the password, you might not receive some responses to some messages because the ClaimCenter password is out of sync with the initial password in the message.

ClaimCenter checks ISO responses to see if they contain the valid password. Because the message contains initial password instead of the new password, ClaimCenter rejects the responses. There is no easy workaround for this type of issue. Therefore, change your ISO password immediately to avoid more serious problems later on.

---

**WARNING** After you receive your initial password from ISO, immediately change the password to avoid serious problems related to password expiry.

---

Once changed, the password does not expire. Do not change the password again.

#### Other Notes About Passwords in Callbacks

As you request access to testing servers or production servers, the request form that ISO requires includes the question:

Do you wish to have ISO transmit an ID/Password/Domain back to your system for security?

This optional setting specifies whether to include additional information in the content of ISO-initiated messages. This relates to *Message 2* and *Message 3* in the diagram in “ISO Network Architecture” on page 456.

However, these three items are unrelated to your standard ISO account ID or account password. All three fields in the context of this question (ID, password, and domain) are arbitrary text fields that ISO offers to send with responses for extra authentication. The ClaimCenter ISO receive servlet checks those fields on incoming messages for authentication.

Answer “yes” to that question on the form. Supply the desired password to ISO over the phone, which is their preferred approach for this security information. Guidewire recommends that you prepare for the possibility of eventual support for this feature by supplying text for these items during initial account setup. Remember that these can be three arbitrary text data. They do not need to match any other ISO account information.

## ISO Proxy Server Setup

The *proxy server* (also called a *bastion host*) forwards ISO responses to the server that is implementing (or testing) ClaimCenter ISO integration. As described earlier, the proxy server also is usually an intermediary computer for outgoing messages from you to ISO. In both cases, the proxy server hosts a *forwarder application*. This application takes incoming and outgoing requests, writes logs, and forwards requests to a port on another computer. It might also handle SSL encryption and decryption. Several common applications handle this type of forwarding, including the following:

- Apache HTTP server, the popular open source web server that can be configured as a proxy.

- Squid, an open source dedicated proxy server.
- BorderManager, a commercial product from Novell.

From a technical standpoint, different proxy servers or applications can process different ISO messages, but do not do this unless you have a special requirement to do so.

For information about obtaining or setting up forwarding with these products, refer to the web sites and product documentation for those applications.

A common choice for a proxy server is the Apache HTTP Server. To simplify proxy server setup, this documentation includes “building blocks” of text to insert into Apache configuration file to facilitate proxy server setup.

For more details, see “Proxy Servers” on page 619.

## ISO Validation Level

The *ISO validation level* is used to confirm that a claim or exposure has all necessary information to submit it to ISO. ClaimCenter ensures that once a claim or exposure reaches this level, the validation level never drops below it at a later time. If the ClaimCenter user could remove properties needed by ISO or set to invalid values, it complicates the ClaimCenter ISO destination message plugin logic.

For claim-level messaging, a notable requirement in validation is that the claim description must be non-empty.

For exposure-level messaging, there is a notable requirement feature of the ISO exposure validation rules. The exposure validation rules strictly requires the claim to already be at the claim validation level called ISO. The built-in rules already enforce this. If you customize any of this code, you must continue to enforce this rule.

### Validation Timeline

After an exposure or a claim changes, ClaimCenter calls the validation rule set. In Studio, click **Rule Sets** → **Validation** → **Claim Validation Rules** → **ISO Validation** and review the built-in claim ISO rules. Similarly, in Studio, click **Rule Sets** → **Validation** → **Exposure Validation Rules** → **ISO Validation** and review the built-in exposure ISO rules.

The validation rules may detect validation issues and may change the validation level for the claim or for the exposure. For general information about how to write validation rules, see “Validation Rule Set Category” on page 74 in the *Rules Guide*.

After an exposure reaches the ISO validation level, ClaimCenter fires the Event Fired rule sets that detect this validation level change. These Event Fired rules send appropriate exposures to ISO. In practice, customizing these payload-generation rules for your data model and business requirements are the largest part of real-world ISO integrations. ClaimCenter provides payload generating Gosu rules for the most common payload types, and sends all required properties for those payloads. If you want to generate different ISO payloads, or add ISO optional properties, or add properties that ISO later makes required, customize the payload generation functions. In practice, the built-in rules provide nearly all of what most implementations need for initial deployment with the reference implementation data model and PCF files. Modify the rules to match your data model changes, generate new payloads, or add optional properties.

### Checking Priority of Validation Levels

In general if comparing typecode values, check the code of the typekey. However, validation levels are implicitly covered in a sequence. For example reaching one level means that you have reached all levels before it.

Because it is important to compare relative position in the sequence, always check the **priority** property of a typecode to determine the validation level and compare to another value. Do not simply check the equality of the **code** property of the typecode object because inequality does not indicate which level is higher.

For example, the exposure validation rules must confirm that the claim validation level is greater or equal the ISO level. Use the priority of the typecode, which is an integer that determines the ordering of the typecodes in

that typelist. Check the `claim.ValidationLevel.Priority` value and see if it is greater than or equal to as `ValidationLevel.TC_ISO.Priority`. If it is, the claim is at least at the ISO validation level, or higher.

#### What are Required Properties for ISO?

For more information about payload generation functions, see “ISO Payload XML Customization” on page 478. The full ISO XML data hierarchy is described in the ISO-provided documentation called the *ISO XML User Manual* (the filename is `XML User Manual.doc`), and ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations.

If you add additional properties to the payload, such as ISO optional properties not generated by the built-in rules, this may affect your validation rules. For example, check if a property is non-null in validation rules if it is a precondition to sending that property to ISO according to you or to ISO.

#### Validation Level Changes

You can add items to the validation level typelist or even switch values of validation levels by changing typecode `priority` properties in the typelist data model. For example, you could reverse the order of `iso` and `external` validation levels if necessary. If you make such changes, do so carefully and consider all possible other changes that you might need in your integration code or rules.

#### Make Validation Level or Rules Changes Before Deployment

Customize validation rules by changing the rules code or change validation levels by changing typecode `priority` properties in the typelist data model. As mentioned earlier, if you make such changes, do so very carefully and consider all possible other changes that you might need in your integration code or rules.

Be aware that making changes after deploying a production server is difficult. Only make changes after great consideration for the implications. For example, if you add a new validation requirement, some claims or exposures might not pass validation rules for the validation level as newly defined. This leads to commit failures if anyone (a real user, batch process, or web service request) makes minor changes to an object in the validation graph. For example, validation rule changes could cause activity batch processes to fail on activities by adding a new claim validation rule that causes even trivial changes to a related claim.

Carefully design your validation rules and validation level changes before initial deployment. You can change them later but it may require major engineering and testing to ensure that claims and exposures never fail their current validation level after your changes.

## ISO Messaging Destination

After you enable the ISO destination (see “Enabling ISO Integration” on page 453), the ISO destination requests notification for certain ClaimCenter events. Events are an abstraction of changes to Guidewire business data. The ISO destination listens for the messaging events `ClaimAdded`, `ClaimChanged`, `ExposureChanged`, and others. Event Fired rules create ISO-related messages as appropriate.

The ISO destination sends three subtypes of outgoing messages of type *Message 1: Sending Claims*:

- **Initial search request** – The initial search request is sent if the exposure becomes valid.
- **Replacement search requests** – Replacement search requests are sent if the exposure changes. These are nearly identical to the initial search request. A flag in the message indicates whether the information in this request is a new record or if it replaces previous information for the exposure.
- **Key field updates** – Key field updates are sent after changes to a key field such as the claim number, policy number, agency ID, or loss date. These update messages are handled specially by the destination because ISO uses a different protocol for changes to properties that uniquely identify an exposure. A key field update is immediately followed by a replacement search request because key field changes might result in new matches.

The rules use a lot of shared code in a Gosu ISO library (the `Libraries.ISO` class). You probably must customize this code for your data model, particularly the `createPayload` method. The `createPayload` method which creates the payload for a search request. The details of the payload depend on the your exposure types, policy types, coverage types and coverage sub types so they could not be hard coded in the destination. This is described more fully in “ISO Payload XML Customization” on page 478.

After the messaging destination sends a message to ISO, ClaimCenter gets an immediate *receipt* letting it know that ISO queued the request. If the receipt indicates a failure, then the destination adds an error acknowledgement immediately. Otherwise it waits for the following response which is either a success (possibly containing match reports) or an error response. If the ISO destination sees a successful response, it adds an acknowledgement. If it sees a failure, it adds an error acknowledgement.

Key field updates are more complicated because ISO does not send a response if a key field update is successful. Instead, the destination waits for 2 minutes (a configurable delay) and if no error response appears within that time, the ISO destination submits the acknowledgement.

The destination tracks whether a particular claim or exposure successfully added to the ISO database. In other words, did it ever had a successful initial search request? Once it gets a successful response from ISO, the ISO destination sets the claim or exposure field `ISOKnown` to true and marks all future search requests as replacement search requests.

There are several error cases:

- **ClaimCenter cannot contact ISO** – The inability for ClaimCenter to contact ISO for the send request is considered a temporary *retryable* error. The destination uses the standard destination *backoff delay* and retries with increasing delay time.
- **ISO sends a failure receipt** – Failure receipts indicate a *non-retryable* error. Something must be wrong with the configuration for ISO to reject the request. The ISO destination submits an error (negative) acknowledgement for the message, and then an administrator must diagnose what went wrong, fix the configuration problem, and send a new ISO message. If the problem is the *message payload* due to incorrect design of business rules, a common problem during development, it is important that you do **not** retry the message. Instead, delete the message and send it again by modifying the data or using the **Send to ISO** button within the ClaimCenter user interface.
- **ISO sends a failure response** – Failure response indicate a *non-retryable* error. Something must be wrong with the configuration for ISO to reject the request. The ISO destination submits an error (negative) acknowledgement for the message, and then an administrator must diagnose what went wrong, fix the configuration problem, and send a new ISO message. If the problem is the *message payload* due to incorrect design of business rules, a common problem during development, it is important that you do **not** retry the message. Instead, delete the message and send it again by modifying the data or using the **Send to ISO** button within the ClaimCenter user interface.
- **ClaimCenter is down, preventing ISO from responding** – If ISO cannot reach ClaimCenter, ISO usually retries for at least 12 hours. This typically is not a problem unless ClaimCenter is down for more than 12 hours.

The ISO destination and its associated classes have their own logger category. Setting `log4j.category.Integration.messaging.ISO` to `DEBUG` generates log messages whenever an ISO message is sent or received. There is also the ISO property `ISO.LogMessagesDir`, which causes all ISO messages and responses to be logged to a directory.

If you use ClaimCenter clusters, data changes may happen on any server in the cluster. Consequently, any server may trigger ISO rules that generate outgoing messages to ISO. Although any server can add a message to the messaging send queue, the actual outgoing communication between ClaimCenter and ISO happens only on the server designated the batch server. Messaging plugins for outgoing ISO messages and the ISO receive servlet for ISO responses run only on the batch server.

## ISO Receive Servlet and the ISO Reply Plugin

The *ISO receive servlet* on the ClaimCenter batch server handles all asynchronous responses from ISO to ClaimCenter. ISO sends these responses to the *bastion host*, and the bastion host forwards responses to the receive servlet. For discussion and a diagram of this architecture see “ISO Network Architecture” on page 456.

The servlet parses each response and then performs the following actions:

- The servlet checks that the response contains a valid request ID and the correct password. If not the response is rejected and an error is written to the log.
- The servlet notifies the ISO destination, in case it is waiting for an acknowledgement.
- The servlet updates the `ISOReceiveDate` field on the claim or exposure.
- The servlet adds the response XML as a document on the claim or exposure
- If the response contains any match reports, the servlet adds `ISOMatchReport` entities to the `ISOMatchReports` array on the exposure. The servlet checks to see if each `ISOMatchReport` is the same as an existing one. If it is the same as a previous one, ClaimCenter updates the existing entity’s `ReceivedDate` instead of creating a new entity. For example, suppose a claim or exposure changes several times and always gets the same two match reports back after each change. Afterwards, the claim or exposure contains only two `ISOMatchReport` entities.

- If ClaimCenter receives any match reports, the receive servlet calls the `ISOReplyPlugin` to add activities.

If you use ClaimCenter clusters, data changes may happen on any server in the cluster. Consequently, any server may trigger ISO rules that generate outgoing messages to ISO. Although any server can add a message to the messaging send queue, the actual outgoing communication between ClaimCenter and ISO happens only on the server designated the batch server. Messaging plugins for outgoing ISO messages and the ISO receive servlet for ISO responses run only on the batch server.

However, for most of the actual reply handling, however, the servlet delegates the work to the `ISOReplyPlugin` class. The `ISOReplyPlugin` class is a `MessageReply` plugin implementation responsible for the asynchronous message reply. For more information about message reply plugins and overall messaging flow, see “Messaging Overview” on page 300.

## ISO Properties on Entities

ClaimCenter includes ISO-specific properties and methods on both claims and exposures. From an implementation perspective, the similarities between exposures and claims are defined by the fact that `Claim` and `Exposure` entities both implement the new interface called `ISOReportable`. Be aware of this important change as you review the documentation or you are looking through built-in code such as payload generation Gosu code. Where you see a references to `ISOReportable`, you can generally think of this as “a claim or exposure”. Remember that whether ClaimCenter actually uses claim-level messaging is controlled by the `ISO.properties` file property `ClaimLevelMessaging`. (Additionally, for any legacy exposures that are known to ISO before switching to claim-based messaging, ClaimCenter continues to treat these as exposure-based messaging.)

The following table lists ISO-related properties on exposures, claims, or vehicle entities. Note that some properties are exceptions and exist only on some entities and not others.

Entity	Property	Type	Description
Claim, Exposure	<code>ISOSendDate</code>	date	The last time the exposure was sent to ISO.
Claim, Exposure	<code>ISOReceiveDate</code>	Boolean	The last time a response was received by ISO for this exposure.
Claim, Exposure	<code>ISOKnown</code>	Boolean	A flag whether this exposure been successfully received by ISO.

Entity	Property	Type	Description
Claim, Exposure	ISOStatus	ISOStatus	The status of exposure with ISO. Choices are TC_None, TC_NotOfInterest, TC_ResendPending, and TC_Sent. The ISO status is “not of interest” (TC_NotOfInterest) if the ISO payload generation rules for an exposure return null, thus indicating that is not appropriate to send it to ISO. In some cases an exposure’s ISO status is “not of interest” but you might think that is incorrect. If so, check that it is a custom exposure type and the ISO payload rules were appropriately customized to generate any payload for that subtype. For related information, see “ISO Exposure Type Changes” on page 481 and “ISO Payload XML Customization” on page 478. This property is viewable in the user interface for administrative users, as described further in “ISO User Interface” on page 472.
Claim, Exposure	ISOMatchReports	ISOMatchReport[]	An array of zero, one, or more ISO match reports for this exposure.

The following properties relate to ISO, but are not specific to ISO:

Entity	Property	Type	Description
Exposure only	LostPropertyType	LostPropertyType	For theft losses, the ISO category of lost property
Vehicle	SerialNumber	varchar 2	The vehicle serial number. This is only used if the Vehicle Identification Number (VIN) is not appropriate, such as for boats.
Vehicle	BoatType	BoatType	If vehicle style is boat, this is the type of boat.
Vehicle	ManufacturerType	VehicleManufacturerType	The company that manufactured the vehicle, based on NCIC 2000 vehicle codes. This code is required by ISO, so ClaimCenter generates a similar (unofficial) code with the first characters of the Make property if the code is not present. If you implement ISO integration, you must ensure the code is set correctly.
Vehicle	OffRoadStyle	OffRoadVehicleStyle	The type of off-road vehicle, for instance snowmobile, All Terrain Vehicle (ATV), or other vehicle with wheels, or wheels and tracks. This is also based on NCIC codes.

## ISO User Interface

If you enable ISO integration, the biggest change is on the claim detail or exposure detail page. There is an ISO tab in the claim or exposure detail screen. This tab shows the ISO status, send and receive dates, and any match reports. You can select items on the match reports list view to see details of a match.

Remember that whether ClaimCenter actually uses claim-level messaging is controlled by the ISO.properties file property `ClaimLevelMessaging`. Additionally, for any legacy exposures that are known to ISO before switching to claim-based messaging, ClaimCenter continues to treat these as exposure-based messaging.

Specific ISO views that are built-in to ClaimCenter and can be customized within the configuration module in the product directory `ClaimCenter/modules/configuration`:

Description	Location in configuration module
ISO detail view for claims or exposures	<code>config/webpcf/claim/exposures/iso/ISODetailsDV.pcf</code>
ISO list view for displaying a list of match reports	<code>config/webpcf/claim/exposures/iso/ISOMatchReportsLV.pcf</code>
ISO detail for displaying an individual match report	<code>config/webpcf/claim/exposures/iso/ISOMatchReportDV.pcf</code>

These are standard Guidewire PCF configuration files, so you can configure them like other PCF files or remove them entirely if you do not need them. By default, all user interface to these properties are read-only. The user interface displays status information and ISO information. In generally, you do not need to change these files.

If you have the Integration Admin permission, you see an extra property (*ISO known*) and can view and edit the *ISO known* and *ISO status* properties (`ISOKnown` and `ISOStatus`). This information can be useful as you correct configuration errors. For example, log in as an administrator with the Integration Admin permission. If you can see if the exposure status is incorrect, for example if the status is stuck at “Not of interest to ISO”. The application displays the status “Not of interest to ISO” if the payload generation function returns `null`. Typically the functions return `null` by accident because you added a custom exposure type and you did not yet modify the built-in payload code to handle it. To update ISO implementation for new exposure types, see “ISO Exposure Type Changes” on page 481 and “ISO Payload XML Customization” on page 478.

If the ISO destination is enabled, in the exposure detail toolbar there is a **Send to ISO** button. The button is enabled after the exposure was validated and sent to ISO. Use it to manually resend the exposure to ISO. Most changes to the exposure/claim/policy also cause a resend to ISO. You only need the button if you want to send the exposure to ISO but you are not changing any properties.

For more information about ISO properties on entities that you can access if you are customizing user interface PCF pages, see “ISO Properties on Entities” on page 471.

## ISO Properties File

The `ISO.properties` file is edited in `ClaimCenter/modules/configuration/config/iso/ISO.properties`. You must rebuild the ClaimCenter EAR file or WAR file to use the latest settings changes in your production application. For debugging-only modifications, note that ClaimCenter reads this file at the time ClaimCenter initializes the destination but rereads it if the ISO destination suspends then later resumes.

The name of the ISO property file is configurable using the `config.xml` file property `ISOPropertiesFileName`.

The following lists the most critical `ISO.properties` settings necessary for a typical ISO deployment:

Property	Typical setting
<code>ISO.AgencyId</code>	Set to the AgencyID provided by ISO. For claim-level ISO messaging, you can instead specify this at the claim level.
<code>ISO.ConnectionURL</code>	Set to the URL for your ISO connection, which would be ISO's direct URL if you were not using a proxy. If you use a proxy server between ClaimCenter and ISO, set this to use your proxy server and its port number, in other words <code>http://proxyDomainOrIP:portnumber</code> . Using the syntax of the Apache configuration file example file, use the URL <code>http://\$DMZProxyDomainName:\$DMZProxyPortA</code>
<code>ISO.CustLoginId</code>	Set to the login ID provided by ISO
<code>ISO.CustPswd</code>	Set to the ISO customer password provided by ISO
<code>ISO.RequireSecureServer</code>	Set to no.

The following table contains a reference for all standard properties in the ISO.properties file:

Property	Description
ISO.AgencyId	ID assigned by ISO to identify the company and office submitting a request. This is a required property with no default value, so it must be specified in ISO.properties. For claim-level ISO messaging, you can instead specify this at the claim level.
ISO.ClaimExposureNumberSeparator	The separator text used to separate the claim number from the exposure order number for building a unique identifier for an exposure. ClaimCenter sends this unique identifier to ISO. For example, if the separator is "exp", the unique identifier for exposure 1 on claim 123-45-6789 is the value 123-45-6789exp1. ISO strips out all non-alphanumeric characters from the unique identifier, so the example resolves to 123456789exp1 after processing by ISO. Your separator text must be alphanumeric.  This property only applies to exposure based ISO messaging. For claim-based messaging, ClaimCenter ignores this property.
ISO.ConnectionURL	The URL of outgoing requests to ISO server. For the recommended server architecture discussed and illustrated in the section "ISO Network Layout and URLs" on page 457, this would be URL#1. This URL is the URL from ClaimCenter to the proxy server. This property defaults to the test service's direct URL, https://clmsrchwebsvct.iso.com/ClaimSearchWebService/XmlWebService.asmx. For actual (non-test) requests, set to https://clmsrchwebsvc.iso.com/ClaimSearchWebService/XmlWebService.asmx.
ISO.CurrencyCode	Your currency code, defaults to en_US. Not usually changed
ISO.CustLangPref	Your language preference, defaults to en_US. Not usually changed.
ISO.CustLoginId	Your login id that ISO allocated for you. This is a required property with no default value, so you must store it in ISO.properties.
ISO.CustomerPhoneFormat	Regular expression used to parse phone numbers from your ClaimCenter implementation. ISO requires phone numbers in a special format: +1-650-3579100. The phone format is used to parse a ClaimCenter phone number, pull out the area code and remaining 7 digits, and then convert it to the ISO form. The regular expression must match three groups of numbers - the area code, then the remaining blocks of 3 and 4 digits. The default format is "( [0-9] {3} ) - ( [0-9] {3} ) - ( [0-9] {4} ) ( x [0-9] {0,4} ) ?". This format handles numbers of the form 650-357-9100 with an optional 0-4 digit extension.
ISO.CustPswd	Your customer password that ISO defines. This is a required property with no default value, so it must be specified in ISO.properties. ISO customer passwords are at most 8 characters long.
ISO.EncryptionTypeCd	Encryption mode, defaults to NONE. Not usually changed
ISO.ExpectReplies	This flag defines whether the destination expects replies from ISO. Set this to true for production servers. If false, the destination sends messages to ISO in test mode, which generate no responses nor adds anything to their database. However, the ISO destination gets an immediate receipt from ISO. This confirms that the connection to ISO works and that the XML syntax and authentication info were correct.
ISO.KeyFieldUpdateTimeout	Number of seconds to wait before assuming that a key field update request has succeeded so it can proceed with the next request. Defaults to 120.
ISO.LogMessagesDir	The name of a directory to log outgoing and incoming ISO XML messages. Defaults to null if unspecified. Messages are saved to files named after their message id, for example _request.xml, _receipt.xml and _response.xml.
ISO.MatchReportNameFormat	Simple date format used to generate the name for the match report document. Default is "ISOMatchReport-'yyyy-MM-dd-HH-mm-ss'.xml."
ISO.Name	Client application name. Set to XML_TEST during testing and but also during production. This is a required property with no default value. You must specify this value in ISO.properties.
ISO.NameSpace	URL namespace for the ISO claim search SOAP service. Defaults to http://tempuri.org/ and usually this does not need to change.
ISO.Org	Client application ISO Organization code, defaults to ISO. Not usually changed

Property	Description
ISO.RequireSecureReceive	Defines whether the receive servlet require that all incoming requests (from the point of view of the ClaimCenter server) are on a secure (HTTPS) connection. For the recommended server architecture, set to false. In this approach, the ClaimCenter server itself does not take on the burden of SSL processing within the Java virtual machine. Instead, the proxy server provides the SSL/HTTPS processing. The default of this property is true. For related information, see "ISO Network Layout and URLs" on page 457
ISO.SendMessages	This configures whether the destination sends messages to ISO. Set to true in production. If false, the ISO destination drops messages. Only set this property to false if debugging and logging all messages. Defaults to true.
ISO.SPName	Service provider name. Default is iso.com. This property is not usually changed.
ISO.TestSuffix	A number ClaimCenter appends to various properties on every request (the request id, claim number, policy number and insured's name) to make them unique in the ISO test database. Otherwise the results from one set of testing could interfere with the results from another set. Only for use during testing. During testing, start with this number at 1 and then increment it every time you drop a database and reimport data. If you do not, then ISO rejects all the sample data exposures because ISO thinks it has seen them already. Defaults to 0, which is the recommended number for production servers. For details, see "The ISO Prepare Payload Class" on page 479.
ISO.Version	Client application ISO version, defaults to 1.0. This property is not usually changed.

### Populating Match Reports from ISO's Response

A match report is information from ISO regarding other insurance companies with information about an exposure that seems similar to an exposure in your system.

You can customize how ISO match report properties in the incoming XML map to properties on the ISOMatchReport entity that stores the data on an exposure.

To do this, you can customize the ISOReplyPlugin class implementation in the populateMatchReportFromXML method. Perform any necessary logic there. For method arguments, this method gets the match report object to populate plus a Gosu object representing the match report XML. You can add as much additional post-reply logic you want.

For more information about how ClaimCenter stores match reports, see "ISO Match Reports" on page 480.

### Converting from ClaimContact (ClaimCenter) to ClaimParty (ISO)

ClaimCenter exports exposure contact data to ISO. What Guidewire calls ClaimContact entities is approximately what ISO calls ClaimParty XML elements. Similarly, ClaimCenter maps what Guidewire calls ClaimContact roles to what ISO calls a claim party role code (ClaimPartyRoleCd). ISO considers this ClaimParty contact data optional but strongly recommended for effective claim matching, so ClaimCenter provides built-in support to generate these elements.

The most important part of configuring this conversion is defining mapping codes in your ISO.properties file. The default ISO.properties file has two contact-related sections to configure.

ClaimCenter uses these lines to map and generate new claim party XML elements:

```
ISOClaimParty.BS = repairshop
ISOClaimParty.CO =
ISOClaimParty.CT =
ISOClaimParty.FM =
ISOClaimParty.IB = agent
ISOClaimParty.LC = attorney
ISOClaimParty.LP = checkpayee
ISOClaimParty.MD = doctor
ISOClaimParty.MF = hospital
ISOClaimParty.PA =
ISOClaimParty.TW =
```

```
ISOClaimParty.OW =
ISOClaclaimParty.PT =
ISOClaclaimParty.TN =
ISOClaclaimParty.WT = witness
```

The code on the left is the ISO claim party code, and the code on the right of the equals sign is the ClaimCenter `ClaimContact` entity role typecode. Add or change these mappings as needed for any changed or added contact role typecodes. Each line can map to one or more ClaimCenter typecode. To include more than one, you can separate them with commas. If ClaimCenter matches a `ClaimContact` with this role typecode, ClaimCenter creates a new ISO `ClaimParty` element with that code.

However, the `ClaimContact` conversion is not a direct one-to-one conversion. There are several important things to know about this translation of `ClaimContact` data:

- ClaimCenter permits a `ClaimContact` to have multiple roles, as defined by an array of claim contact roles on a claim contact. ISO does not permit this in their data model. So, in some cases ClaimCenter creates *multiple* (ISO) `ClaimParty` XML elements.
- Currently, there is limited support for multiple contacts with the same role. In the current release, if more than one contact has a certain role (such as a witness or attorney), additional contacts with that role may be omitted from conversion.

Additionally, the following two lines configure special parts of the ISO payload designed specifically for witness and driver identification:

```
ISOClaclaimSearch.ContactRole.Witness = witness
ISOClaclaimSearch.ContactRole.Driver = driver
```

The code on the right of the equals sign is the ClaimCenter `ClaimContact` entity role typecode. For example, this means that during ISO conversion, if a `ClaimContact` with the `driver` role typecode is found, use it as the ISO driver in the ISO `driver` XML element. The ISO driver corresponds to the `ClaimsDriverInfo` element in the `ClaimsParty` record. If a `ClaimContact` with the `witness` role typecode is found, ClaimCenter uses it for the `com.iso_AccidentWitnessedInd` element on the `ClaimsOccurrenceAggregate` element that describes this exposure/claim.

If more than one contact has a witness role or driver role, additional contacts with that role may be omitted from conversion. Also, as with the other `ClaimParty` mapping lines, each line can only map to a single ClaimCenter typecode.

For additional type code mapping information, see the following section, “ISO Type Code and Coverage Mapping” on page 476.

## ISO Type Code and Coverage Mapping

The ISO integration code includes an XML file to map ClaimCenter typecodes to typecodes that ISO understands. The type code mapping file is stored in `config/iso/TypeCodeMap.xml`. It uses the same format as the `typecodemapping.xml` file that is stored in the general-purpose file `config/typelists/mapping`. However, this file is used only by the ISO destination so its mappings must have their `namespace` attribute set to `iso`.

The type code mapping file is read by the ISO messaging destination as it starts. For debugging use only, be aware ClaimCenter rereads this file if the destination suspends and later resumes.

The important type codes are listed in the following table. Note that `PolicyType`, `CoverageType`, and `CoverageSubType` are not listed because those use a separate file, discussed later in this topic.

ClaimCenter typecode	Maps to ISO typecode
<code>LossType</code>	ISO Loss Type code
<code>ExposureType</code>	ISO Subject Insurance code. This indicates whether the damage is to property, contents, or loss of use. Used only for property exposures.

It is possible to add other typecodes to the file, but in practice it typically is not necessary.

## Coverage Mapping

The ISO Policy Type → Coverage → Loss Type hierarchy is very similar in concept to the ClaimCenter coverage subtype hierarchy (PolicyType → CoverageType → CoverageSubType). The full ISO XML data hierarchy is described in the ISO-provided documentation called the *ISO XML User Manual* (the filename is *XML User Manual.doc*), and ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations.

ISO rejects any requests if the loss type is not in the list of permitted loss types for the given coverage. This can make it challenging to map the ClaimCenter hierarchy to the ISO hierarchy, depending on how closely your data model hierarchy matches ISO's hierarchy. It is unnecessary for the map to cover all possible cases. You can override the ISO policy, coverage and loss type properties from business rules defined in Guidewire Studio.

The ISO integration uses a separate text file from the typecode mapping file to configure how ClaimCenter coverages map to ISO coverages. A comma-separated values (CSV) file called *ISOCoverageCodeMap.csv* controls this mapping.

ClaimCenter “coverage codes” represent the policy type, coverage type and coverage subtype. These map to the ISO policy type, coverage type and loss type.

The *ISOCoverageCodeMap.csv* file maps a ClaimCenter policy type, LOB code (optional), coverage type and coverage subtype to an ISO policy type, coverage type and loss type. The format is a comma-delimited row containing the following fields in this order:

- source ClaimCenter policy type
- source optional line of business code (can be blank)
- source coverage type
- source coverage subtype
- ISO policy type
- ISO coverage type
- ISO loss type

For example a couple of lines in this file might be:

```
auto_comm,,ABI,abi_bid,CAPP,BODI,BODI  
businessowners,pr,ADPERINJ,adpersinj_gd,CPBO,OTPR,OTPR
```

Notice that the first example line does not include a line of business. The second line includes a line of business.

This format allows for hierarchy differences between the ClaimCenter and ISO policy type hierarchies, which were not supported by *TypeCodeMap.xml*. For example, ClaimCenter might share the same coverage type, X, between two different policy types, A and B. But ISO might have two different coverage types ISOAX and ISOBX to handle this case.

In the new coverage mapping system, you can now do the following:

```
A,,X,CS,ISOA,ISOAX,ISOCS  
B,,X,CS,ISOB,IOSBX,ISOCS
```

This maps coverage type X to ISOAX if the policy type is A, but maps ISOBX if the policy type is B.

There are some special features to the *ISOCoverageCodeMap.csv* mapping:

- The ISO coverage code is empty in several mappings. These are usually mappings for first party property claims and exposures; ISO does not use a coverage code for such losses.
- The LOBCode. Normally ClaimCenter policy types map straight to ISO policy types, and entries for these policy types omit the LOBCode field. But there are some cases in which a single ClaimCenter policy type maps to multiple ISO policy types. In such cases the LOBCode can be added to the mapping, to narrow it down to a particular ISO policy type. For example, in the out of box configuration *businessowners,pr* (business-owners policy type, property LOB) maps to ISO's CPBO, while *businessowners,g1* maps to ISO's CLBO.

- In some cases the ClaimCenter coverage subtype is not very specific and the claim's loss cause gives a much better sense of the ISO loss type to use. The mapping file allows entries like `LossCause/OTPR` in the ISO loss type column. This tells ClaimCenter to try mapping the claim's loss cause field to an ISO value using the `TypeCodeMap.xml` file. If no mapping is found default to the ISO loss type `OTPR`. This extended mapping is mainly used for homeowner's at the moment.

The ISO code uses a slightly higher-level lookup operation based on a ClaimCenter policy type, LOB code, coverage type, coverage subtype and loss cause using a two-step full mapping lookup:

1. First ClaimCenter looks for appropriate mapping lines in `ISOCoverageCodeMap.csv`
2. If that lookup returns a loss type of the form `LossCause/OTPR`, ClaimCenter looks up the loss cause in the `TypeCodeMap.xml` mapping.

You can request a full lookup from Gosu using the `ISOTranslate.getCoverageCodes()` method.

The full ISO XML data hierarchy is described in the ISO-provided documentation called the *ISO XML User Manual* (the filename is `XML User Manual.doc`), and ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations. If you want to send ISO optional properties not sent by ClaimCenter by default, use the ISO XML User Manual for the reference of all their properties. To add optional properties to ClaimCenter generated XML payload, see “ISO Payload XML Customization” on page 478.

For additional type code mapping information relating to the ClaimCenter `ClaimContact` entity, see “Converting from `ClaimContact` (ClaimCenter) to `ClaimParty` (ISO)” on page 475.

## ISO Payload XML Customization

The largest part of real-world ISO integrations are customizing ISO payload-generation rules for changes to your data model and special business requirements, such as custom exposure types. Ensure you allocate adequate time for this process. Be sure to have business analysts on your integration team as discussed in “Step 2: Team Resources” on page 454 to advise on the business requirements for each customization.

Within Guidewire Studio business rule sets, the ISO rules perform the following steps:

1. Validate a claim or exposure (and associated policy) to see if it is valid for ISO.
2. Detect changes to a claim or exposure to see if it needs to be resent to ISO.
3. Construct an ISO message payload.

Due to data model changes, you probably need to customize ISO rules. ISO claim search requests can have various several types and each request type requires slightly different properties in the ISO XML message payload. Though the ISO destination has functions to create the different ISO claim search types, you need to tell ClaimCenter which combinations to use for a particular coverage or exposure. The ISO Gosu classes specify how to construct the appropriate message payload for an exposure and how to check that a particular exposure type is valid for ISO.

If you create new exposure types or change existing ones, you must change ISO payload generation rules to detect the new exposure type and generate appropriate XML for that subtype. If you fail to modify the payload, the payload generation functions returns `null` instead of an XML payload, and ClaimCenter sends nothing to ISO. You probably need to modify the injury, vehicle, property, or workers' compensation payload generation rules. In particular, the following functions in that library check for exposure types: `exposureFieldChanged()` and `createSearchPayload()`.

If you create new exposure types and do not customize ISO payload generation rules for this subtype, the ISO payload rules likely return `null`. This causes ClaimCenter to set the exposure's ISO status (the exposure's `ISOStatus` property) to the typecode for “`Not of interest to ISO`”. For related information, see “ISO Exposure Type Changes” on page 481.

ClaimCenter generates ISO payloads using functions in the ISO library files in Studio within **Classes → Libraries → iso**. This file contains functions that ClaimCenter calls to generate each type of ISO payload. However, this file is no longer the main location for payload generation code. The new payload generation system uses specialized classes. Each specialized class generates payloads for a certain type of ISO *loss section*. A loss section is the ISO term for a type of loss such as a property loss, a vehicle loss, and so forth.

To customize how ISO generates the payload XML for a loss section, modify the ClaimCenter class as defined in the following table.

Type of loss	ISO loss section	Customer class to modify in ClaimCenter package gw.api.iso
auto loss	AutoLossInfo	ISOAutoLossSection
information about injured people (standard, including third-party property)	ClaimsInjuredInfo	ISOInjuryLossSection
information about injured people (Workers' Comp)	ClaimsInjuredInfo	ISOWCInjuryLossSection
property loss insurance information for a person or insured party	PropertyLossInfo/ClaimsSubjectInsuranceInfo	ISOPropertyLossSection
property loss insurance information for an object	PropertyLossInfo/ItemInfo	ISOMobileEquipmentLossSection
property loss insurance information for a water craft	PropertyLossInfo/Watercraft	ISOWatercraftLossSection

All of these files in the previous table represent pairs of implementation classes. ClaimCenter organizes files like this to simplify code merges for ISO payload generation during every upgrade.

For each section, the implementation class for your changes is the class listed in the previous table. However, these classes extend another base class that contains Guidewire core code for default behaviors.

This base file's file name has the suffix “Base”. For example, the `ISOPropertyLossSection` class extends from the `ISOPropertyLossSectionBase` class, which contains the bulk of the Guidewire code. If you want to see the existing implementation, refer to the base file. However, if you want to modify the behavior, Guidewire strongly recommends simply overriding methods in the non-base file rather than modifying the base class directly.

The code in the ISO library file determines which loss section class to use by getting the exposure enhancement property called `ISOLossSectionType`. The Gosu enhancement file `GWExposureISOEnhancement` implements this dynamic property. Customize that file if you want to change the mapping of exposure type to loss section type.

The full ISO hierarchy is in the ISO documentation called the *ISO XML User Manual (XML User Manual.doc)*. ISO strictly enforces this hierarchy. Request this manual from ISO and refer to it during any customizations.

## ISO Payload Generation Properties Reference

The full ISO hierarchy is described in the ISO documentation *ISO XML User Manual (XML User Manual.doc)*. ISO strictly enforces this hierarchy. Request this manual from ISO and refer to it during customizations.

### The ISO Prepare Payload Class

ClaimCenter includes a Gosu class called `ISOPreparePayload`. Immediately before ClaimCenter sends the message to ISO, ClaimCenter calls this class to add some additional fields. At this point in time, the message is committed in the database and ClaimCenter has processed acknowledgements for any previous ISO messages related to the current claim.

Because of the rules of messaging and database transactions, the `ISOPreparePayload` class reliably knows two things that were unknown at the time that Event Fired rules created the message:

- **Message entity ID** – The database ID property on the message entity. This is important because ClaimCenter uses it to construct a unique ISO request ID.
- **Known to ISO** – This class knows that all previous ISO messages are complete. Thus, it knows definitively whether the claim or exposure is known to ISO already. ClaimCenter uses this information to determine whether the message is an initial search or a replacement request for an item that ISO already knows about.

The main job of `ISOPreparePayload` main job is to construct a unique request ID based on:

- the message ID
- the claim/exposure ID
- the customer password.

Next, it sets this value for the `ClaimsSvcRq` element's `RqUID` field and the `ClaimInvestigationAddRq` element's `RqUID` field to this unique ID.

If the claim or exposure is already known to ISO, this class sets the `ClaimInvestigationAddRq` element's `ReplacementInd` field to 1. If this is the first claim search request, it sets this value to 0.

Additionally, the `ISOPreparePayload` class has code to help testing. This code is not used in production. When testing ISO, it is common to run into problems. Tests typically send the same test data to ISO repeatedly. This will not work because the first time the test runs, the test data is not known to ISO. ClaimCenter must send it as an initial request. However, the next time the test runs, ISO already knows about the test data, so the test must send only *replacement* requests. This makes it hard to test the normal flow: first send an initial message, then send a replacement message. To help this, `ISO.properties` has a property called `ISO.TestSuffix`. For production servers, always set this property to 0. For testing, you can set it to a unique positive integer.

`ISOPreparePayload` uses this integer to append a unique suffix to identifiers in ISO requests. This convinces ISO that it sees different data during each test.

## ISO Match Reports

A match report from ISO describes information about an exposure from another company that seems similar to an exposure in your system. For more information about ISO messages that can return match reports, see “ISO Network Layout and URLs” on page 457

After ISO receives a match report, ISO extracts selected properties from the ISO match report and attach it to the claim or exposure as an `ISOMatchReport` entity subtype. `ISOMatchReport` is a delegate that defines the interface for ISO match reports. The claim and exposure versions of ISO match reports implement this interface. Both `ClaimISOMatchReport` and `ExposureISOMatchReport` entities now contain the properties `ISOClaim` and `ISOExposure`. For `ClaimISOMatchReport`, the `ISOClaim` property points to the claim and the `ISOExposure` properties is `null`. For `ExposureISOMatchReport`, the `ISOExposure` property points to the exposure and the `ISOClaim` properties points to the claim that owns that exposure.

ClaimCenter stores the reports in the `ISOMatchReports` property on the claim or exposure entity. This property contains an array of one or more match reports. You can customize how ISO match report properties populate properties on the `ISOMatchReport` entity. For details, see “Populating Match Reports from ISO’s Response” on page 475.

In addition to the `ISOMatchReport` entities, the complete match report is also saved as a document on the claim in raw XML format. The name of the document is configurable and includes a date stamp, configured by the `ISO.MatchReportNameFormat` property in `ISO.properties`.

To display the match report, click the `ISO` tab on the claim or exposure. To display the document in the user interface, click the `Documents` tab on the claim or exposure. To get the match report document data, ClaimCenter requests the document from the document management system. For more about document management integra-

tion, see “Document Management” on page 199.

Each time you view a match report document, ClaimCenter renders the XML into HTML using a stylesheet that ISO provides. The stylesheet takes raw XML and formats it in HTML. The `ISOMatchReport.gosu.xml` document template and `ISOMatchReport.gosu.xml` descriptor implement this stylesheet conversion.

The default template and descriptor works for most implementations. The only thing you might need to change is the context object for the URL of the stylesheet within in the descriptor file. You may need to update the URL to match the location of the XSL transform file that converts the XML to HTML:

```
<ContextObject name="xsl_file" type="string">
  <DefaultObjectValue>http://customerserver/cc/modules/configuration/config/
    iso/xsl/CS_Xml_Output.xsl</DefaultObjectValue>
</ContextObject>
```

This object must change to match the actual URL of ClaimCenter server or your proxy server.

The match report XML contains only supported ISO properties and does not include any custom extensions. Only update the XSL file if ISO changes their match report XML format to provide additional (or different) data in the future.

If you are customizing the ISO match report code, it might be useful to know the API for creating a new match report. From a claim or an exposure (that is, for all `ISOReportable` entities), to create the correct type of match report subtype, call the `ISOReportable.addNewISOMatchReport()` method.

To customize how ClaimCenter creates new ISO match reports from XML, modify the class in the method `populateMatchReportFromXML` in the class `ISOReply`. If you override this method, remember to call the super-class version first.

## ISO Exposure Type Changes

If you enable ISO, by default ClaimCenter sends exposures of the following types to ISO for fraud detection: `BodilyInjuryDamage`, `VehicleDamage`, `PropertyDamage`, `LossOfUseDamage`, and `WCInjuryDamage`. The `WCInjuryDamage` type covers all workers’ compensation claims, so there is no need to also submit `TimeLoss` exposures for those exposures.

You can send additional exposure types to ISO, but this requires several types of changes. Be sure to allocate sufficient engineering time (including testing) to these integration elements described in the following table.

Task	Description	To perform this task
Update exposure type to loss section mapping	Update the exposure enhancement file <code>GWExposureISOEnhancement</code> to customize the <code>ISOLossSectionType</code> enhancement property. This defines the mapping of exposure type to loss section Gosu class.	See “ISO Payload XML Customization” on page 478
Update ISO payload rules	You must change ISO payload generation rules to detect the new exposure type and generate XML appropriate for that subtype. If you fail to modify the payload, the payload generation functions return <code>null</code> instead of an XML payload, and nothing sends to ISO. You probably need to modify the injury, vehicle, property, or workers compensation payload generation rules. Review the loss section class files to understand or customize the logic for each loss section.	See “ISO Payload XML Customization” on page 478.
Update ISO user interface	If you use exposure-level messaging only, change the PCF page definition (user interface definition) for the <code>Exposure Details</code> view of that type to add the ISO responses subpage.  If you use claim-level messaging only, you do not need this step. However, if you have legacy exposures that are known to ISO, you must perform this step so that the user interface is appropriate for exposures in those special cases.	See “ISO User Interface” on page 472.

Task	Description	To perform this task
Update typecode mapping	Update the typecode mapping file with your data model changes.	See "ISO Type Code and Coverage Mapping" on page 476.
Update coverage mapping	Update the coverage mapping file with your data model changes.	See "ISO Type Code and Coverage Mapping" on page 476.
Update validation rules	Update your validation rules to include new rules for any required properties on your exposure types.	See "ISO Validation Level" on page 468.

If the payload generation functions return `null`, ClaimCenter displays the ISO status as “Not of interest to ISO”. Typically this happens if the exposure type is a custom subtype not handled by the default ISO payload generation rules. To update ISO implementation for new exposure types, see “ISO Payload XML Customization” on page 478.

## ISO Date Search Range and Resubmitting Exposures

Once a claim is entered in the ISO database, ISO searches for claim matches for a well-defined time period:

This period is 30 days for auto insurance, 60 days for property insurance, and 1 year for casualty insurance. There is no way to extend that time period nor to automatically resubmit the claim for a longer total search period.

ISO stops sending automatic update reports according to the initial received date. This means there is no point in sending periodic additional updates on that claim to try and extend the automatic update window. Do not attempt to resubmit claims or exposures to ISO for this purpose.

## ISO Integration Troubleshooting

If your computer is not getting responses back from ISO, first start *logging* and *message logging*.

Enable the following logs:

- To start ClaimCenter logging, set category `log4j.category.Integration.messaging.ISO` to `DEBUG` in `logging.properties`.
- To start message logging, set `ISO.LogMessagesDir` to a valid directory in your `ISO.properties`.
- On your proxy server, enable all logs for the forwarding application on the server such as Apache or Squid. Refer to the application documentation for details on this process.
- On your firewall, enable all logs.
- On your proxy server, enable any other logs (such as general system logs).

Once message logging is on, you can see which messages sent, whether you received receipts, and whether you received responses.

If there were no ISO requests, check if the exposure ISO status is not “Not of interest”. If it does have that status, check the ISO rules to see if the rules handle this type of exposure. If they do not handle the custom exposure type, and thus return `null` for the ISO payload, the exposure gets this ISO status. You can reset the ISO status property by logging in as an administrator and viewing the exposure in the **Exposure Detail** page. For related information, see “ISO Exposure Type Changes” on page 481.

If you see a request but did not receive a receipt, the message may not have been sent to ISO.

Confirm all of the following:

- Check that the `ISO.SendMessages` property in `ISO.properties` is set to `true`.

- Check that the ISO.ConnectionURL property in ISO.properties is set to the proxy server URL. (This assumes that you are using a proxy server, which is the recommended setup.)
- Check that the ISO server is up. Test the ISO server URL in your web browser.
- If you see a receipt, open it and see if it contains the correct MsgStatusCd value ResponsePending. If it does not, look at the error message. Probably the authentication information in ISO.properties is incorrect. Possibly you are sending the request from an IP address outside the range registered with ISO.
- If the receipt looks good but you do not see a response then check that your proxy server is active and set up correctly. Check if the request arrived at the proxy server and check the logs of the proxy server (for example, Apache or Squid). If not, the ISO server may be down, or the bastion host/forwarder is not listening on the correct URL. If the request arrived at the forwarder, check the connection between the forwarder and your ClaimCenter batch server. Carefully check your proxy server configuration files.
- If you see a response but it contains an error code, look at the associated error message. Most errors are configuration problems such as missing required properties or incorrect policy/coverage/loss types. Sometimes the ISO server thinks an exposure is a duplicate of an exposure that it knows already. Or, ISO does not know about an exposure that ClaimCenter thinks it knows. In these cases, login as an administrator and change the ISO Known flag on the **Exposure Detail** page.

Although the Administration pages for messages are not ISO-specific, outgoing requests to ISO are handled using the messaging system. It is sometimes necessary for ClaimCenter administrators to track connectivity issues using this user interface. For more information about this administrative user interface, see “Monitoring and Managing Event Messages” on page 64 in the *System Administration Guide*.

## ISO-Specific Error Codes

The most up to date and complete list of ISO errors is in the ISO documentation called the *ISO XML User Manual (XML User Manual.doc)*.

For further detail on errors returned from ISO, immediately contact your ISO customer support representative. ISO’s customer support can help you learn about the error. Encourage them to add additional new error information to the ISO documentation as needed.

## Testing Automatic Updates

After ClaimCenter sends an exposure to ISO and ISO responds, ISO keeps the exposure in its database. Afterwards, as other companies send claims to ISO, ISO may detect matches to one of your exposures. If this happens within a certain time range, ISO sends a match report to the ClaimCenter callback URL. The time ranges are 1 year for casualty, 60 days for property and 30 days for auto. ISO refers to these delayed responses as *automatic updates*.

Always test your system’s responses to *automatic updates*, but only after normal ISO receipts and responses are working reliably. However, testing automatic updates is difficult because automatic updates are sent only after *another* company adds a matching claim, so you cannot cause an automatic update yourself. To test automatic updates, contact your ISO representative. They can manually submit a matching claim marked with a different (test) customer ID.

## ISO Formats and Feeds

You can access the ISO Claim Search service using two different format types. ISO uses the same database for all protocols and feed methods. However, the data format is different within the ISO database for the universal format compared to the legacy FTP format.

## ISO ClaimSearch Universal Format

ISO's *universal format* supports multiple types of claims, such as casualty claims, property claims, and auto claims. You can choose one of three different *feeds types*, which indicate basic transport types.

- **universal format XML feed** – Upon creating or changing a claim, a claims management system sends an XML-formatted message to ISO, which ISO translates to (and from) universal format. Matches are sent back asynchronously, but immediately upon detection of the match. The XML messages can be submitted as a *simple HTTP POST* request or as a *HTTP SOAP web service* request, however ClaimCenter requires the SOAP version for initial queries. Replies from these ISO requests are always in universal format XML feed HTTP POST format. Notify ISO about a related configuration setting to ensure that ISO is expecting the SOAP request. See "Basic ISO Message Types" on page 456.
- **universal format FTP feed** – Daily claims are aggregated in one flat file sent daily as a batch file to ISO. ISO sends matches back as batch results only once per day, in contrast to the immediate (although asynchronous) response of the XML feed type.
- **universal format MQ feed** – ISO supports sending and receiving using the WebSphere MQ messaging system.

## ISO Legacy FTP Format

Prior to ISO's universal format, ISO required claims in legacy mono-line FTP formats provided by INDEX (casualty insurance), PILR (property insurance), and NICB (auto insurance). Like the universal format FTP feed, daily claims are aggregated into one flat file, which is uploaded daily to ISO using FTP.

Although the database is the same for all access formats and feed types, the legacy FTP format uses a different data format within the ISO database. Be careful not to confuse *legacy FTP format* with the *universal format access with FTP feed*.

## ISO Web Interface

ISO provides a web site to enter claims manually. You can log on at a later time to view found matches from queries you enter manually or queries you submit using other feeds. However, the HTML web interface is **not** a separate data format type or feed type.

## Migrating Old Claims Systems to ClaimCenter

If ClaimCenter is replacing a claim management system that already feeds the legacy ISO databases, all legacy claims must be converted to the ISO Master Claims Database. The basic process is to resubmit the claim in universal format using the identical claim number using the conversion tag. This prompts a process at ISO which converts the legacy record to universal format. After that you can submit updates using standard processes.

If you need to change any information such as converting claim numbers to the format expected by ClaimCenter, you can send a key field update transaction. Custom logic can be put in to trigger this transaction. Remember also that the old system must stop sending claims as soon as the new system goes live to prevent duplicate records in ISO's production database.

**IMPORTANT** You must ensure that the old system stops sending claims with the old system as soon as the new system goes live to prevent duplicate records in ISO's production database.

If ClaimCenter is replacing a claim management system that uses the universal format FTP feed, claims need no conversion because the original claims are already stored in the universal format. Because of this, ClaimCenter deployment for this type of migration is relatively straightforward with respect to ISO data.

As mentioned earlier in this section, ISO supports the following input methods at the same time:

- one system with universal format FTP feed using the *production database*
- another system using the universal format XML feed with the *testing database*.

This feature makes it easy to migrate from universal format FTP feed to the universal format web service feed.

If the claim was never sent to ISO in the legacy system, ISO was unaware of it. There is no need to run the conversion process for that claim. It can proceed normally in ClaimCenter.



# Metropolitan Reporting Bureau Integration

Metropolitan Reporting Bureau provides a nationwide police accident and incident reports service in the United States. Many insurance carriers use this system to obtain police accident and incident reports to improve record-keeping and to reduce fraud. ClaimCenter built-in support for this service decreases deployment time for Metropolitan Reporting Bureau integration projects, particularly for personal lines carriers.

This topic includes:

- “Overview of ClaimCenter-Metropolitan Integration” on page 487
- “Metropolitan Configuration” on page 490
- “Metropolitan Report Templates and Report Types” on page 492
- “Metropolitan Entities, Typelists, Properties, and Statuses” on page 494
- “Metropolitan Error Handling” on page 498

**See also**

- For more information about services that the Metropolitan Reporting Bureau provides, see:  
<http://www.metroreporting.com>

## Overview of ClaimCenter-Metropolitan Integration

ClaimCenter integrates with Metropolitan Reporting Bureau (hereafter, called *Metropolitan*) by using their XML Gateway for requesting police accident and incident reports. You enter all the necessary data in ClaimCenter and then send the information through the gateway. You do not need to visit the Metropolitan web site to request reports or to view reports.

Metropolitan integration includes the following features:

- **Ordering a report during claim intake** – Suppose an adjuster or customer service representative is on the phone with an insured customer taking in a First Notice of Loss (FNOL) report through the ClaimCenter New Claim Wizard. The adjuster can also submit a request for a police report.
- **Ordering a report on an established claim** – If a police report was not requested originally during claim intake, or First Notice of Loss (FNOL), the adjuster can order one later from the claim file user interface.
- **Multiple reports on the same claim** – Sometimes an adjuster requests a police report for a claim but has some data incorrect, such as the police department details. An adjuster can change the appropriate information and submits a request for another, new report.
- **Many report types** – Metropolitan has approximately 30 report types. ClaimCenter supports most of the report types.
- **Attaching a report to a claim file as a document** – After adjusters request reports, the reports are retrieved later, asynchronously. After Metropolitan returns the report, ClaimCenter matches it to a specific claim and attaches it as a document to the claim file. Users can view or print the report like any other document.
- **Report completion notification** – ClaimCenter notifies the adjuster assigned to a claim if a requested report successfully attaches to the claim file and is available for review. Alternatively, if the report request fails for some reason, ClaimCenter notifies the adjuster. This notification is implemented as an activity.

There are two ways to view the **Metropolitan Reports** detail page in ClaimCenter. Both require that you open a claim and click **Loss Details** in the left sidebar menu to open that screen. After the **Loss Details** screen is open:

- If reports have been received, you can scroll to the **Metropolitan Reports** section and click the link in the **Type** column.
- Click **Edit** to put the **Loss Details** page in edit mode, scroll to the **Metropolitan Reports** section, and click the **Add** button.

The **Metropolitan Reports** list looks like the following:

Metropolitan Reports					
	<b>New</b>	<b>Remove</b>			
	<b>Type</b>	<b>Status</b>	<b>Order Date</b>	<b>Document</b>	<b>Actions</b>
<input type="checkbox"/>	MV-104 (NY Only)	InsufficientData	06/29/2006		<a href="#">Re-Submit</a>
<input type="checkbox"/>	Auto Accident	Received	06/29/2006	AutoAccident3OCQYMO000.tif	<a href="#">View Document</a>
<input type="checkbox"/>	Driving History	InsufficientData	06/30/2006		<a href="#">Re-Submit</a>
<input type="checkbox"/>	Auto Theft	WaitForInquiry	06/30/2006		

The columns in the **Metropolitan Reports** list view are as follows:

- **Type** – The first column in the list view is the type of the report request. After you click on the report type link, ClaimCenter displays a page with details about this report request.
- **Status** – The current status of the report request. The possible values are listed in “Metropolitan Report Status and How It Changes” on page 495.
- **Order date** – The date that ClaimCenter sent the report request to Metropolitan.
- **Document** – The document column shows the name of the document if the report status is Received. The user can view the document by clicking the **View Document** button in the **Actions** column.
- **Actions (View Document, Resubmit)** – The last column is for buttons to view a document or to resubmit the report request. If the original report request has insufficient data, you can update the claim with all required properties. You can return to the loss detail page and click the **Resubmit** button to rerun preupdate rules. ClaimCenter can send a new report request if it passes preupdate rules.

In the **Metropolitan Report** list, the user can click the report type to open the details screen and then enter any required data for that report. Rather than replicate the Metropolitan form for every report type, ClaimCenter extracts relevant data from entities on the claim as appropriate. ClaimCenter copies this information to a new **MetroReport** entity in the property **claim.MetroReports** to track the report.

After a user requests a Metropolitan report, ClaimCenter returns to the **Loss Details** page. You see a new row in the Metropolitan reports list with a report **Status** of **New**.

The following table shows the current report types and their report type codes from the `MetroReportType` type-list:

Request code	Request description	Request code	Request description
A	Auto Accident	N	OSHA
B	Auto Theft	O	Other
C	Auto Theft Recovery	P	Activities Fixed Rate
D	Driving History	Q	Property and Judgements
E	Coroner Reports	R	Registration Check/DMV
F	Fire - Home	S	Insurance Check
G	Burglary	T	Title History
H	Death Certificate	U	Subrogation Financial/Assets
I	Incident	V	Vandalism - Auto
J	Locate Defendant/Witness	W	Weather Report
K	EMS/Rescue Squad	X	Fire - Auto
L	Supplemental/Addendum	Y	Photos
M	MV-140 (NY Only)	Z	Disposition of Charges

## ClaimCenter Metropolitan Integration Architecture

ClaimCenter integrates with Metropolitan with the following types of code, only some of which are directly modifiable or configurable:

- **Messaging plugins** – Built-in messaging plugins are able to send outgoing messages to Metropolitan. These messaging plugins repeatedly try to send the request to the Metropolitan servers until it acknowledges the request. An acknowledgement of the request indicates receipt of the request, not that the report is ready yet. A different part of ClaimCenter queries Metropolitan regularly to determine if the report is ready. These plugins are minimally configurable through configuration files.
- **Preupdate rules** – Claim preupdate rules verify that all the required data is available for a claim. You can modify these rules.
- **Report templates** – ClaimCenter includes Gosu templates generate XML data to send to Metropolitan, based on ClaimCenter claim data. You can modify these templates.
- **Event-handling rules** – ClaimCenter includes event handling rules for Metropolitan internal messaging events. The messaging events send messages to the Metropolitan messaging plugins to handle outgoing transport and asynchronous replies. These event-handling rules are part of the built-in implementation. You cannot modify these event-handling rules.
- **Document support** – Once Metropolitan’s APIs indicate that a report is available, Metropolitan provides a URL to download the report. ClaimCenter support for documents allows police reports and other reports to be stored as documents within ClaimCenter in the claim file. Built-in Metropolitan messaging plugins use document management APIs to add the document to ClaimCenter.

**IMPORTANT** ClaimCenter document content storage reference implementation is not intended to replace a full-featured document management system. For maximum data integrity and feature set, you must use a complete commercial document management system (DMS). Implement a new `IDocumentMetadataSource` and `IDocumentContentSource` plugin to communicate with your DMS. For details, see “Document Management” on page 199.

- **Internal state machine that manages Metropolitan report status flow** – The built-in Metropolitan integration includes an internal state machine that manages flow of Metropolitan report status. This state machine is not

modifiable. Later in this topic there is an overview of this state machine. This state machine queries Metropolitan regularly using Metropolitan's APIs to determine if a report is ready.

Because these requests happen over the Internet, you must provide appropriate firewall protection for requests, and appropriate firewall holes to permit communication to Metropolitan.

Unlike the ClaimCenter integration to Insurance Service Organization (ISO), all communications with Metropolitan are initiated by ClaimCenter. After the original report request, ClaimCenter polls Metropolitan regularly. The consequence is that it is easier to configure network proxies since all requests are HTTP requests outgoing from ClaimCenter to Metropolitan, and a reverse proxy is not necessary. However, insulating ClaimCenter outgoing requests through a proxy is still a good network practice.

For more information about designing proxy servers and configuring an Apache web server to act as a proxy, see “Proxy Servers” on page 619.

## Metropolitan Configuration

There are three main places where you must configure Metropolitan:

- The main application configuration file, `config.xml`
- The Metropolitan properties file, `Metro.properties`
- Display strings in the `Metro` section of the `Localization` keys

### Enabling Metropolitan Configuration

Metropolitan integration can be enabled or disabled by changing one parameter in the main application configuration file, `config.xml`. The corresponding Boolean parameter is defined as follows:

```
<param name="EnableMetroIntegration" value="true"/>
```

Related Metropolitan-related messaging plugins are defined in `config.xml`. You can make minor configuration changes, such as retry times, to these destination settings. See “Metropolitan Error Handling” on page 498.

### Metropolitan Properties File

The Metropolitan properties file `Metro.properties` contains most of the configuration information for the Metropolitan integration points.

Because the report requests are not pushed to ClaimCenter, ClaimCenter must poll the Metropolitan server regularly. One of the most important configuration parameters that you can make is the time interval between inquiries. This interval is effectively the delay for a polling request to the server to see if a report is available. The property `Metro.InquiryInterval` specifies the number of minutes to wait between requests, with a minimum of 60 minutes. The default setting in the base configuration is 150:

```
Metro.InquiryInterval = 150
```

Additionally, update the following properties specifically for each client with the basic ClaimCenter customer information. Properties include `CustAccount` and `CustBillingAccount`, which are account and billing numbers for each Metropolitan account so the client is charged appropriately. Metropolitan provides these numbers for each client.

The `CustNAIC` property is the NAIC number for the ordering company. If you do not know this number, put the value `NONE`.

The following is an example of the default base configuration properties in the `Metro.properties` file:

<code>Metro.CustNAIC</code>	= 11882
<code>Metro.CustCompanyName</code>	= Allstate Insurance
<code>Metro.CustAddress1</code>	= 123 West Ave
<code>Metro.CustAddress2</code>	=
<code>Metro.CustCity</code>	= Erie
<code>Metro.CustState</code>	= PA

```
Metro.CustZip      = 41222
Metro.CustAccount = PING
Metro.CustBillingAccount= MRDEMO
```

There are other properties in that file that typically do not require modification. Change the other properties only if Metropolitan changes the XML formats for date, time, and social security numbers. These property formats are in standard regular expression format.

## ClaimCenter Display Keys for Metropolitan Reports

In ClaimCenter Studio, you can set display keys to define your own error messages, property names, and other text to display for Metropolitan Reports.

To modify display keys, navigate in the Project window to **configuration** → **config** → **Localizations** and open the **display.properties** file for your locale.

For example, for US English, navigate in the Project window to **configuration** → **config** → **Localizations** → **en\_US**. Then double-click **display.properties** to open this file in the editor. You can modify any of the Metropolitan Report display keys by searching for **metro**, selecting a display key, and editing it. For example:

- **Metro.Activity.InsufficientData.Message** – Default setting is **Missing field(s)\:**.
- **Metro.Fields.AgentCity** – Default setting is **City of the investigating agency**.
- **Metro.Fields.ClaimNumber** – Default setting is **{Term.Claim.Proper} Number**.

The Metropolitan Reports error messages and property names are used in preupdate rules if there are required properties that are not defined. ClaimCenter creates an activity that indicates the missing properties so you know where to fix the report request, after which you can resubmit the request.

There are many other display keys that include **metro** that you can modify.

### See also

- “ClaimCenter Display Keys for Metropolitan Reports” on page 491

## Configuring Activity Patterns

The Metropolitan integration includes the following *activity patterns*. An activity pattern is a type of template for a user activity notification. If you log in to ClaimCenter as an administrator, you can modify these activity patterns by clicking the **Administration** tab and clicking **Activity Patterns** in the sidebar menu.

The Metropolitan activity patterns are:

- **Metropolitan Report Available** – If a report request succeeds and the report is attached it to the claim, ClaimCenter creates a **metropolitan\_report\_available** activity. ClaimCenter assigns it to the user that created the **MetroReport** entity associated with the request. The Metropolitan workflow uses this activity pattern.
- **Metropolitan Report Deferred** – For report requests that takes additional time to process, Metropolitan replies with the response **deferred**. ClaimCenter creates the activity **metropolitan\_report\_deferred**. The Metropolitan workflow uses this activity pattern.
- **Metropolitan Report Held** – If Metropolitan needs additional information from ClaimCenter before Metropolitan can process it, Metropolitan replies with the response **hold**. In response, ClaimCenter creates the activity **metropolitan\_report\_held**. The Metropolitan workflow uses this activity pattern.
- **Metropolitan Report Inquiry Failed** – Used if a request to Metropolitan other than the initial order request fails. A **metropolitan\_report\_inquiry\_failed** activity is generated and then the workflow retries. The Metropolitan workflow uses this activity pattern.
- **Metropolitan Report Request Failed** – For any report request that fails due to incomplete data or if initial order message fails. This activity pattern creates a **metropolitan\_request\_failed** activity and assigns it to the user that created the **MetroReport** entity associated with the request. The activity description text area

includes the type of report requested and the data that must be supplied to successfully submit the request. The Metropolitan workflow and the `Metro.gs` library uses this activity pattern.

- **No Metropolitan Report Available** –For any report request that fails after sending the request because no report is available. ClaimCenter creates a `metropolitan_report_unavailable` activity and assigns it to the user that created the `MetroReport` entity associated with the request. The Metropolitan workflow uses this activity pattern.

#### See also

- For more information about configuring activity patterns, see the “Understanding Activity Patterns” on page 225 in the *Application Guide*.

## Configuring the Messaging Plugin Retries

If the messaging plugins fail to send to Metropolitan, they retry after a time delay between tries, up to a maximum number of times. If Metropolitan does not acknowledge receipt successfully, the messaging plugins retry up to the maximum number of times at the specified interval. The default retry maximum is 3 times at an interval of 100 milliseconds.

To change the default settings, start Guidewire Studio and navigate in the left pane to **configuration** → **Messaging**. Click **Messaging** to open the Messaging editor and click the metro message, number 67, to edit its values. For example, you could set the following values to increase the amount of time between retries:

- **Initial Retry Interval** – Default value is 100 milliseconds. Increase it to 1000 milliseconds.
- **Max Retries** – Default value is 3. Increase it to 5.

This messaging destination is built-in and cannot be modified other than through the configuration settings documented in this topic.

#### See also

- For information on Studio Messaging editor settings, see “Messaging Editor” on page 137 in the *Configuration Guide*.
- For information on administration of messages, see “Monitoring and Managing Event Messages” on page 64 in the *System Administration Guide*.
- For a full description of the messaging system, see “Messaging and Events” on page 299.

## Metropolitan Report Templates and Report Types

ClaimCenter report requests to Metropolitan must be formatted in the correct XML format for that report type. To generate this request, ClaimCenter runs a specific a Gosu template, which is a text file with embedded Gosu code. This template generates the necessary XML-formatted text.

## Metro Report Types and Loss Types

ClaimCenter includes templates for all report types, and these report types are auto-configured to correspond to the standard loss types in the ClaimCenter reference configuration. For example, Auto Accident reports map to the Auto loss type, and Coroner Report maps to all base loss types. The typelist `MetroReportType.ttx` specifies the mapping between a Metropolitan report type and a ClaimCenter loss type. You can change this mapping, but be aware that Metropolitan specifies different data requirements and constraints for each report.

Different report types have different data requirements. Gosu templates and preupdate rules determine if the required data for the selected report type was correctly specified by the adjuster.

## Editing Gosu Template Files

You can customize report templates to support your own changes to the ClaimCenter data model. Also, you might need to customize reports if, at a later date, Metropolitan changes the XML format, such as changing tag names or adding required properties for certain report types. Review the latest version of the Metropolitan specification at the URL:

<https://metroweb.metroreporting.com/schema/index.php>

The Metropolitan report Gosu templates are stored, one template per report, in the following directory:

ClaimCenter/modules/configuration/config/web/templates/metroreport/...

For example, the `BurglaryReport.gst` template generates the Metropolitan burglary report request.

---

**IMPORTANT** You can modify Metropolitan report Gosu templates, but do not change template headers or footers.

---

A sample report template looks like the following:

```
<mrb:Insured>
  <mrb:CompanyName><%=claim.Insured.Company.Name%></mrb:CompanyName>
<% } else { %>
  <mrb:Last><%=claim.Insured.Person.LastName%></mrb:Last>
  <mrb:First><%=claim.Insured.Person.FirstName%></mrb:First>
  <mrb:Middle><=%=translator.formatNullToEmptyString(claim.Insured.Person.MiddleName)%></mrb:Middle>
<% } %>
  <mrb:Address1><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.AddressLine1)%>
  </mrb:Address1>
  <mrb:Address2><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.AddressLine2)%>
  </mrb:Address2>
  <mrb:City><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.City)%></mrb:City>
  <mrb:State><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.State)%></mrb:State>
  <mrb:Zip><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.PostalCode)%></mrb:Zip>
  <mrb:Phone><=%=translator.formatPhoneNumber(claim.Insured.PrimaryPhoneValue)%></mrb:Phone>
</mrb:Insured>
```

As you can see in this example, the templates use `translator` classes to format the data by using an object with the symbol `translator`. This utility class enables report templates to use a Metropolitan utility class that translates strings in report templates with the following methods:

- `formatCustDate` – Format date to MMDDYYYY format, which is the required Metropolitan date format.
- `formatCustTime` – Format time to HHMM 24 hours clock format, which is the required time format.
- `formatPhoneNumber` – Format the phone number to 10 digits without spaces or other characters.
- `formatNullToEmptyString` – Translate a null object reference to an empty string. This method is particularly useful for optional properties in XML templates.

## Mapping Report Types to Gosu Template Files

The mapping between template files and the report types is specified by the XML configuration file `MetroReportTemplateMapping.xml`. Your Gosu template file names must all end in `.gst`. The following example includes one `<report>` element that maps the Metropolitan report A (auto accident report) to a specific Gosu template file:

```
<Report>
  <Type>A</Type>
  <Template>MetroAutoAccidentReport.gst</Template>
</Report>
```

## Specific Report Template Notes

### Claims Insured By Companies

For a claim insured by a company, ClaimCenter puts the company's information in `<insured>`.

### Auto Accident Reports and Null Drivers

For the Auto Accident report type, if the specified driver is null, ClaimCenter puts Parked in the driver's first name and in the last name within the report request. The assumption is that, at the time the auto accident happened, the car was parked and there was no driver inside.

### Worker's Compensation

In the ClaimCenter base configuration, worker's compensation claims do not include all the requisite data that Metropolitan requires for reports. For worker's compensation reports, ClaimCenter puts the claimant's information in the <driver> property because there is no direct mapping from the ClaimCenter data model to the Metropolitan XML data document.

Consequently, worker's compensation claims cannot generate some reports. The user interface automatically removes unsupported report options if the claim is a worker's compensation claim.

## Adding New Report Types

You can add new Metropolitan report types if Metropolitan makes new report types available.

### To add new report types to the ClaimCenter Metropolitan report integration

1. Create a report template for the new report type.
2. Save the template in the directory:  
`ClaimCenter/modules/configuration/config/web/templates/metroreport/...`
3. Modify the file `MetroReportTemplateMapping.xml` to add mapping information for the new report type.
4. Add the new type to the report type mapping file `MetroReportType.ttx`.
5. Update the ClaimCenter preupdate rules to check for all required properties for the new report type.
6. Regenerate Java API and SOAP API files.
7. Rebuild and redeploy the application WAR file to make the new mappings and files available at runtime.

### See also

- For information on regenerating Java API and SOAP API files, see “Regenerating Integration Libraries” on page 22.

## Metropolitan Entities, Typelists, Properties, and Statuses

The following ClaimCenter entities describe and define the ClaimCenter-Metropolitan integration.

### MetroReport Entity

The entity `MetroReport` describes one report from Metropolitan, both before a response is returned and afterward. There is an array of `MetroReport` objects in a `MetroReports` property on `Claim`.

### MetroReportType Typelist

Each `MetroReport` entity has an associated `MetroReportType`. The `MetroReportType` typelist defines the possible types of Metropolitan reports adjusters can request, such as Auto Accident, Auto Theft, Coroner Reports, Title History, and others. The `MetroReportType` typelist also maps Metropolitan report types to loss types. For example, an adjuster can request an Auto Accident type of report only if the loss type on the claim is Auto.

The following example shows a MetroReportType typecode for a burglary report, which adjusters can request only if the lost type is Auto or PR.

```
<typecode code="G" name="Burglary" desc="Burglary" priority="7">
  <category typelist="LossType" code="AUTO"/>
  <category typelist="LossType" code="PR"/>
</typecode>
```

## MetroAgencyType Typelist

Each MetroReport entity has an associated MetroAgencyType. The MetroAgencyType typelist defines the possible investigating agency types for a report. Users who request reports from the user interface can specify the investigating agency type. If a requester does not have the information, the MetroAgencyType property can be null.

The following examples show MetroAgencyType typecodes for investigating agency types, with codes, names and descriptions.

```
<typecode code="PD" name="Police Department" desc="Police Department" priority="1"/>
<typecode code="CO_PD" name="County Police" desc="County Police" priority="2"/>
```

## Document Entity

Each MetroReport entity has a foreign key reference to a Document entity, which is the document generated by Metropolitan if a report request results in an actual report.

If Metropolitan successfully returns a report, the built-in code requests that the document content source plugin implementation registered in IDocumentContentSource.gwp add the document to the claim.

## Metropolitan Report Status and How It Changes

Each MetroReport entity has a MetroReportStatus property that indicates the current status of the report request. The following list of report statuses provide a sense of the flow of the report request process. This list of statuses also helps you understand the meaning of user-visible text in the application user interface.

The possible values of the MetroReportStatus property are:

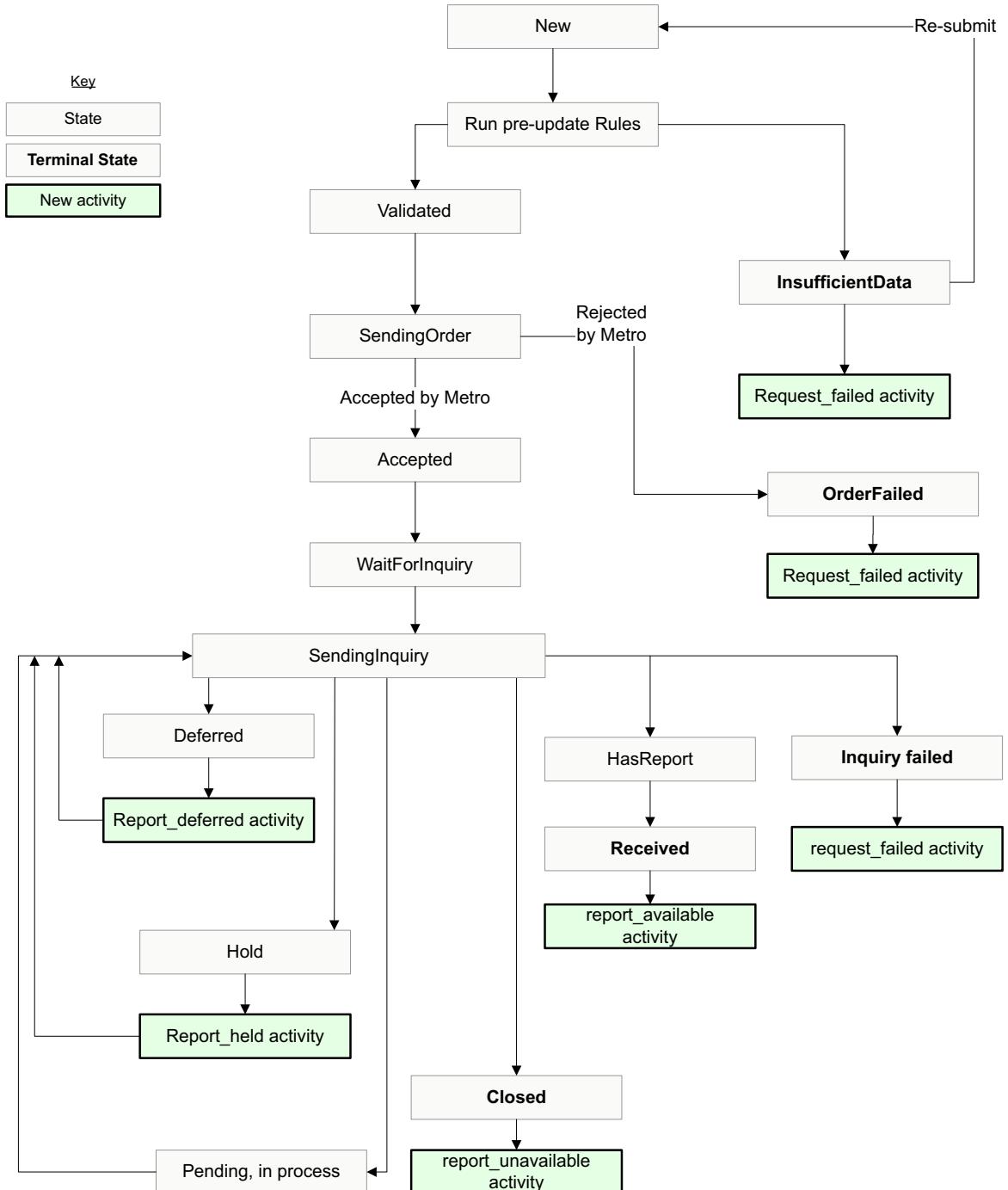
Metropolitan report status	Meaning
none	No order was located that matched the information supplied.
accepted	The order was accepted by the Metropolitan Reporting Bureau.
closed	The order was canceled, or it cannot be completed for some reason.
deferred	An image was returned with a notice that the data source takes some additional time to provide the requested information. The order is still open.
downloadingreport	The system is in the process of downloading the report.
duplicate	The report request failed due to a duplicate request.
error	The inquiry sent could not be matched to any existing order in the Metropolitan Reporting Bureau system.
hasreport	The report is ready on the external server.
hold	The order is awaiting additional information from you.
inquiryfailed	The inquiry request failed based on the result sent back from the Metropolitan Reporting Bureau.
insufficientdata	Some of the required fields are missing.
new	The initial status of the report request.
oderfailed	The order request failed based on the result sent back from the Metropolitan Reporting Bureau.
pending	The order is in the system and is currently awaiting a response from the data source.
received	The Report has been received—downloaded successfully—to your server.

Metropolitan report status	Meaning
sendinginquiry	The report inquiry file has been sent and is waiting for the response from the Metropolitan Reporting Bureau.
sendingorder	The order has been sent and is waiting for the response from the Metropolitan Reporting Bureau.
validated	The request has been validated and ready to be sent out.

The process for requesting Metropolitan reports and the statuses through which the process flows are fixed in ClaimCenter. You cannot modify the process flow, nor can you modify or extend the set of process statuses. The only ways to configure the Metropolitan report request process are by setting the `InquiryInterval` parameter and other settings documented in this topic.

The following diagram illustrates the Metropolitan report request process and the changes in status that can occur for a specific report request. The diagram also illustrates the points in the process at which ClaimCenter generates user activities related to a specific report request.

### Metropolitan Workflow States and Activities



**See also**

- “Configuring Activity Patterns” on page 491

## Customizing Metropolitan Timeouts

The built-in integration of Metropolitan reports has two time cycles in the workflow:

- A delay after sending the request before requesting the report.
- A delay after asking for the report, if the Metropolitan server does not have the report before checking again if the report is ready.

Both time cycles have associated time values in the workflow. The defaults are:

- `Metro.InquiryInterval` – After submitting the original request, ClaimCenter waits 2.5 hours before inquiring initially. You can modify this value in the `metro.properties` file.
- `Metro.OrderTimeout` – After requesting the report, ClaimCenter waits for the response 8 hours. You can modify this timeout in the `metro.properties`. This is exposed in Gosu as the property `metroReport.PastOrderTimeout`. It returns `true` only if the request started and has been waiting longer than the corresponding timeout value.

Additionally, if the overall workflow does not finish completely within one week, then ClaimCenter stops it and creates an error activity. You can modify this timeout in the `metro.properties` file in the property `Metro.WorkflowTimeout`. This is exposed in Gosu as the `MetroReport` property `metroReport.PastWorkflowTimeout`. It returns `true` only if the workflow started and has been running longer than the corresponding workflow timeout value.

In the `metro.properties` file, specify the time values for the two delays and the timeout using a number followed by a suffix for the unit. Use `d` for days, `h` for hours, and `m` for minutes. For example, use `2d` for “two days.” Use a combination of values, such as `3d2h`, for “three days and two hours.”

## Metropolitan Error Handling

Keep in mind the following facts about error-handling with Metropolitan:

- Preupdate rules must ensure that all required data for each report type is specified before a report request is sent to Metropolitan.
- If Metropolitan denies the report request, ClaimCenter updates the report status and creates a new activity for the adjuster to examine and resend the request if appropriate.
- The messaging plugins are configurable to retry sending to Metropolitan until a maximum number of times is reached. If no acknowledgement returns from Metropolitan, the messaging plugins retries up to the maximum at the specified time interval.
- Metropolitan sometimes changes the format of the XML files that Metropolitan expects for requests, and sometimes adds required properties. Review the latest version of the XML specifications at:  
<https://metroweb.metroreporting.com/schema/index.php>

**See also**

- “Configuring the Messaging Plugin Retries” on page 492

---

part VIII

# Policy and Contact Integrations



# Contact Integration

This topic discusses how to integrate ClaimCenter with an external contact management system other than ContactManager.

The base ClaimCenter configuration has built-in support for integration with Guidewire ContactManager. For more information, see “Integrating ContactManager with Guidewire Core Applications” on page 45 in the *Contact Management Guide*.

This topic includes:

- “Integrating with a Contact Management System” on page 501
- “Configuring Contact Links” on page 506
- “Contact Web Service APIs” on page 508

**See also**

- “Plugin Overview” on page 163
- “Web Services Introduction” on page 31

## Integrating with a Contact Management System

ClaimCenter integrates with an external contact management system through the `ContactSystemPlugin` plugin interface. This plugin interface integrates with an external system that supports create, retrieve, update, search, and finding duplicates. To integrate with an external contact management system, write your own implementation of the `ContactSystemPlugin` plugin and register it in Studio.

---

**IMPORTANT** If you integrate with ContactManager, do not write your own implementation. For integration with ContactManager, ClaimCenter provides an implementation of the `ContactSystemPlugin` plugin interface, `gw.plugin.contact.ab800.ABContactSystemPlugin`. See “Integrating ContactManager with Guidewire Core Applications” on page 45 in the *Contact Management Guide*.

---

Your implementation of the contact system plugin defines the contract between ClaimCenter and the code that initiates requests to an external contact management system. ClaimCenter relies on this plugin to:

- Retrieve a contact from the external contact management system.
- Search for contacts in the external contact management system.
- Add a contact to an external contact management system.
- Update a contact in an external contact management system.

ClaimCenter uniquely identifies contacts by using unique IDs in the external system, known as the Address Book Unique ID (AddressBookUID). This unique ID is analogous to a public ID, but is not necessarily the same as the public ID, which is the separate property PublicID.

When ClaimCenter needs to retrieve a contact from the contact management system, it calls one of the `retrieveContact` methods of this plugin. The method called depends on whether ClaimCenter needs to retrieve any related contacts along with the contact being retrieved. In both cases, ClaimCenter passes the address book unique ID of the contact. If related contacts are required, a list of relationships to retrieve is passed in an alternate method signature. Your plugin implementation needs to get the contact from the external system by using whatever network protocol is appropriate. Your plugin implementation must then populate either a `Contact` object or a graph that possibly includes related contacts.

For this plugin, the contact record object that you must populate is a `Contact` entity instance. An additional method, `retrieveRelatedContacts`, retrieves the related contacts as specified in the collection of relationships passed in. This method must return those related contacts merged into the contact graph that was passed in.

### Inbound Contact Integrations

For inbound integration, ClaimCenter publishes a WS-I web service called `ContactAPI`. This web service enables an external contact management system to notify ClaimCenter of updates, deletes, and merges of contacts in that external system. When ClaimCenter receives a notification from the external contact management system, ClaimCenter can update its local copies of those contacts to match. See “Contact Web Service APIs” on page 508.

## Asynchronous Messaging with the Contact Management System

When a user in ClaimCenter creates or updates a local `Contact` instance, ClaimCenter runs the Event Fired rule set. These rules determine whether to create a message that creates or updates the `Contact` in the contact management system. In the default configuration, the process of sending the message to the contact system involves several messaging objects:

- A built-in `ContactSystemPlugin` plugin implementation.
- A built-in messaging destination that is responsible for sending messages to the contact system. The messaging destination uses two plugins:
  - A message transport (`MessageTransport`) plugin implementation called `ContactMessageTransport`. ClaimCenter calls the message transport plugin implementation to send a message.
  - A message request (`MessageRequest`) plugin implementation called `ContactMessageRequest`. ClaimCenter calls the message request plugin to update the message payload immediately before sending the message if necessary.

For more information about messaging and these plugin interfaces, see “Messaging and Events” on page 299

In the default configuration, ClaimCenter calls the following two methods of `ContactSystemPlugin`:

- `createAsyncUpdate` – At message creation time, Event Fired rules call this method to create a message.
- `sendAsyncUpdate` – At message send time, the message transport calls this method to send the message.

### Detailed Contact Messaging Flow

1. At message creation time, the Event Fired rules call the `ContactSystemPlugin` method called `createAsyncUpdate` to actually create the message. The Event Fired rules pass a `MessageContext` object to the `ContactSystemPlugin` method `createAsyncUpdate`. The `createAsyncUpdate` method uses the

MessageContext to determine whether to create a message for the change to the contact, and, if so, what the payload is for that message.

2. At message send time on the batch server, ClaimCenter sends the message to the messaging destination. There are two phases of this process:

- a. ClaimCenter calls the message request plugin to handle late-bound AddressBookUID values. For example, suppose the AddressBookUID for a contact is unknown at message creation time but is known when at message send time. ClaimCenter calls the beforeSend method of the ContactMessageRequest plugin to update the payload before the message is sent. For related information, see “Late Binding Data in Your Payload” on page 339.
- b. ClaimCenter calls the message transport plugin to send the message. The message transport implementation finds the ContactSystemPlugin class and calls its sendAsyncUpdate method to actually send the message. An additional method argument includes the modified late-bound payload that the message request plugin returned.

## ClaimCenter Synchronizing and Mapping

The ContactSystemPlugin provides methods enabling ClaimCenter to access the properties that have been defined as synchronizable and persistent in the ClaimCenter mapping files. These methods are:

- `getSyncablePropertiesForType` — Returns the properties that have been defined as affecting the sync status in ClaimCenter.
- `getPersistPropertiesForType` — Returns the properties that have been defined as persistent in ClaimCenter.
- `getAllMappedPropertiesForType` — Returns the properties mapped between ClaimCenter and ContactManager.
- `getSyncableRelationshipsForContactType` — Returns the set of relationships that have been defined as synchronizable for a given Contact type.

### See also

- “ContactMapper Class” on page 285 in the *Contact Management Guide*
- “Linking in ClaimCenter” on page 194 in the *Contact Management Guide*
- “Synchronizing ClaimCenter and ContactManager Contacts” on page 200 in the *Contact Management Guide*

## Contact Retrieval

To support contact retrieval, do the following in your `retrieveContact` method:

1. Determine what object type to populate locally for your contact to connect to the external system.

For example, suppose you define the web service in Studio with the name `MyContactSystem` and that web service has a `MyContact` XML type from a WSDL or XSD file. For this example, suppose your contact retrieval web service returns a `MyContact` object.

**Note:** For more information about web services and how ClaimCenter imports types from external web services, see “Calling WS-I Web Services from Gosu” on page 71.

2. Write your web service code that connects to the external contact management system. Whatever type it returns must contain equivalents of the following `Contact` properties:

`PrimaryPhone`, `TaxID`, `WorkPhone`, `EmailAddress2`, `FaxPhone`,  
`HomePhone`, `CellPhone`, `DateOfBirth`, and `Version`.

If the contact is a person, it must have the properties `FirstName` and `LastName`.

If the contact is a company, it must have the property `Name`, containing the company name.

3. Your plugin code synchronously waits for a response.

4. To return the data to ClaimCenter, create a new instance of a contact in the current transaction's bundle.
5. Populate the new contact entity instance with information from your external contact.
6. Return that object as the result from your `retrieveContact` method.

If you ever need the ClaimCenter implementation to retrieve more fields from the external contact management system, it is best to extend the data model for the `Contact` entity.

**Note:** If you need to view or edit these additional properties in ClaimCenter, remember to modify the contact-related PCF files to extract and display the new field.

## Contact Searching

To support contact searching, your plugin must respond to a search request from ClaimCenter. ClaimCenter calls the plugin method `searchContacts` to perform the search. The details of the search are defined in a contact search criteria object, `ContactSearchCriteria`, which is a method argument. This object defines the fields that the user searched on.

The important properties in this object are:

- `ContactIntrinsicType` – The type of contact. To determine if the contact search is for a person or a company, use code such as the following:

```
var isPerson = Person.Type.isAssignableFrom(searchCriteria.ContactIntrinsicType)  
If the result is true, the contact is a person rather than a company.
```

- `FirstName`
- `Keyword` – The general name for a company name (for companies) or a last name (for people)
- `TaxID`
- `OrganizationName`
- `Address.AddressLine1`
- `Address.City`
- `Address.State.Code`
- `Address.PostalCode`
- `Address.Country.Code`
- `Address.County`

Refer to the Data Dictionary for the complete set of fields on the search criteria object that you could use to perform the search. However, the built-in implementation of the user interface might not necessarily support populating those fields with non-null values. You can modify the user interface code to add any existing fields or extend the data model of the search criteria object to add new properties.

The second parameter to this method is the `ContactSearchFilter`, which defines some metadata about the search, including:

- The start row of the results to be returned.
- The maximum number of results, if you are just querying for the number of results, as opposed to the actual results.
- The sort columns.
- Any subtypes to exclude from the search due to user permissions.

For the return results, populate a `ContactResult` object, which includes the number of results from the query and, if required, the actual results of the search as `Contact` entities. It is not expected that these `Contact` entities will contain all the contact information from the external contact management system. These entities are expected to contain just enough information to display in a list view to enable the user to select a result.

For example, the default configuration of the plugin for ContactManager includes the following properties on the `Contact` entities returned as search results:

- `AddressBookUID` – The unique ID for the contact as `String`

- `FirstName` – First name as `String`
- `LastName` – Last name as `String`
- `Name` – Company or place name as `String`
- `DisplayAddress` – Display version of the address, as a `String`
- `PrimaryAddress` – An address entity instance
- `CellPhone` – Mobile phone number as `String`
- `HomePhone` – Home phone number as `String`
- `WorkPhone` – Work phone number as `String`
- `FaxPhone` – Fax phone number as `String`
- `EmailAddress1` – Email address as `String`
- `EmailAddress2` – Email address as `String`

For the full list of properties, see the `ABContactAPISearchResult` Gosu class in `ContactManager`. This class is in the package `gw.webservice.ab.ab800.abcontactapi`.

## Finding Duplicates

ClaimCenter calls the `findDuplicates` method in the plugin to find duplicate contacts for a specified contact.

The method takes two arguments:

- A `Contact` entity instance for which you are checking for duplicates
- A `ContactSearchFilter` that can specify subtypes to exclude from the results if contact subtype permissions are being used

The `findDuplicates` method must return an instance of `DuplicateSearchResult`, a class that contains a collection of the potential duplicates as `ContactMatch` objects and the number of results.

The `ContactMatch` object contains the `Contact` that was found to be a duplicate and a `boolean` property, `ExactMatch`, which indicates if the contact is an exact or a potential match. As with searching, the `Contact` returned in the `ContactMatch` object does not have to contain all the contact information from the external contact management system. The object contains just enough contact information to enable the ClaimCenter user to determine if the duplicate is valid.

The list of properties returned by `ContactManager` in its default configuration is in the `ABContactAPIFindDuplicatesResult` class in the package `gw.webservice.ab.ab800.abcontactapi`.

### To find duplicates

1. Query the external system for a list of matching or potentially matching contacts.
2. Optionally, if the results you receive are only summaries, if it is necessary for detecting duplicates, retrieve all the data for each contact from the external system.
3. Review the duplicates and determine which contacts are duplicates according to your plugin implementation.
4. Create a list of `ContactMatch` items, each of which contains a `Contact` and a `boolean` property that indicates if the contact is an exact match. Each of these `Contact` objects is a summary that contains many, but perhaps not all, `Contact` properties.
5. Create an instance of the concrete class `DuplicateSearchResult` with the collection of `ContactMatch` items. Pass the list of duplicates to the constructor.
6. Return that instance of `DuplicateSearchResult` from this method.

## Adding Contacts to the External System

To support ClaimCenter sending new contacts to the external contact management system, implement the `createContact` methods in your contact system plugin. This method must add the contact to the external system. Send as many fields on `Contact` as make sense for your external system. If you added data model extensions to `Contact`, you must decide which ones apply to the external contact management system data model. There may be side effects on the local contact in ClaimCenter, typically to store external identifiers.

### Alternative Method Signatures for Adding a Contact to an External System

The `createContact` method has two signatures:

```
createContact(Contact contact)
createContact(Contact contact, String transactionId)
```

The simpler method signature takes only a `Contact` entity.

The transaction ID parameter uniquely identifies the request. Your external system can use this transaction ID to track duplicate requests from ClaimCenter.

### Capturing the Contact ID from the External System

The `createContact` method returns a `ContactCreateResult` object to ClaimCenter. This object contains the `Contact` and a flag to indicate if the contact is in a pending create status in the external contact management system. Your implementation of the method must capture the ID for the new contact and for other entities in the contact graph from the external system. Use the captured value to set the address book UID on the local ClaimCenter contact and other entities in the contact graph. You must set the address book UID before your `createContact` method returns control to the caller. Typically, the external system is the system of record for issuing address book UIDs.

## Updating Contacts in the External System

If a ClaimCenter user updates a contact's information, ClaimCenter sends the update to the contact management system by calling the `updateContact` method of the contact system plugin. Just like the `createContact` method, one parameter to `updateContact` is a `Contact` entity. The second version of the method adds a transaction ID that can be used by the external contact management system to track if an update has already been applied.

## Configuring Contact Links

You can change the behavior when linking contacts to an external contact management system using the `ContactSystemLinkPlugin` plugin interface. ClaimCenter includes a default implementation called `ABContactSystemLinkPlugin`. The default implementation uses the existing `ContactSystemPlugin` implementation to actually initiate requests to the external contact management system.

There are two plugin methods to implement: `linkContact` and `link`.

### Link Contact

ClaimCenter calls the `linkContact` plugin method when ClaimCenter needs to link a contact to an external contact management system. The method takes as arguments:

- A ClaimCenter `Contact` entity instance
- A `boolean` flag that indicates whether to commit the change immediately. If `true`, explicitly commit the entity change in the `Contact` object's current bundle. If `false`, do not commit the change because the caller code is responsible for the bundle commit.

The method returns a `LinkResult` object that encapsulates the link status and relevant information. Status types include the following, which are static fields on the `LinkResult` class.

The following table lists the status options, and the name of the static method you can use to create a `LinkResult` object of that type.

Status	Description	Static method name	Static method arguments, which also become data inside the <code>LinkResult</code>
CREATED	The contact was created in the external system	<code>created</code>	• the address book UID
PENDING_CREATE	The contact is in <i>pending create</i> status.	<code>pendingCreate</code>	• the address book UID
EXACT	The contact is an exact match with a contact in the external system.	<code>exact</code>	• the address book UID
POTENTIAL_MATCH	The contact is a potential match with a contact in the external system.	<code>potential</code>	• potential matches, as a <code>DuplicateContactMatch</code>
ERROR	Some error	<code>error</code>	n/a

The default implementation tries several tactics, choosing the first attempt that succeeds:

- Search for exact matches
- If none found, search for potential matches
- If none found, create a new contact in the external system

In the default implementation, whether a contact create is *created* or marked as *pending create* will be determined by the utility class `ContactSystemApprovalUtil`. The contact system plugin built-in plugin implementation calls this utility class when attempting to create a contact in `ContactManager`.

The `linkContact` method must first determine whether to create the contact in remote contact management system, use an definitively matched contact, or pick from a list of potentially matched contacts. In the default implementation, the `linkContact` method calls its own `link` method to perform the actual linking of the contact to the remote contact management system.

### Link

ClaimCenter calls the `link` plugin method when it tries to link two contacts together.

In the default implementation, it sets the `AddressBookUID` on the ClaimCenter `Contact` to the value in the external contact management system, `ContactManager`.

The method takes as arguments:

- a contact, as a ClaimCenter `Contact` entity instance
- The address book UID of the contact in the external contact management system

The method returns nothing.

### Unlinking

There is no method in the plugin interface specifically for unlinking, but you may need to unlink the contact if the link is broken.

In the default implementation:

- the `linkContact` method initially unlinks the contact if the address book UID is non-null by calling the `link` method with a `null` address book UID.
- the `link` method unlinks the contact if the passed-in address book UID is `null`.

If you need to unlink a contact, you must remove all the address book UID values throughout the contact graph with the exception of related contacts. If you need to implement similar logic, review the default implementation code for guidance.

## Contact Web Service APIs

The web service `ContactAPI` provides external systems a way to interact with contacts in ClaimCenter. This web service is WS-I compliant. Refer to the implementation class in Studio for details of all the methods in this API. Each of the methods in this API that changes contact data in ClaimCenter requires a transaction ID, which must be set in the SOAP request header for the call.

### See also

- “Web Services Introduction” on page 31

### Delete a Contact

Before a contact management system deletes a contact that is in ClaimCenter, it must first query to see if the contact is in use. The `isContactDeletable` method checks to see if a contact with the address book UID passed in is currently in use in ClaimCenter. ClaimCenter uses a batch process to retire contacts that are not in use. Therefore, this method can return `false` and then sometime later return `true`, after the batch process has run.

If the call to `isContactDeletable` determines that the contact can be deleted, the external contact management system calls the `removeContact` method. This method indicates to ClaimCenter that the contact has been deleted from the external contact management system. The `removeContact` method then attempts to retire all the contacts with that addressbook UID in ClaimCenter.

### Update Contacts

You can update a contact from an external system by using the `updateContact` method. It returns nothing.

Because ClaimCenter can have many local instances of a `Contact`, `updateContact` uses the `ContactAutoSync` process to cause ClaimCenter to copy over the latest version from the contact management system. Depending on the configuration of the `ContactAutoSync` process, this update might happen immediately or might happen when the batch process is run.

### Merge Contacts

You can merge two contacts from an external system. You need to know the IDs of both contacts, and you must decide which one will survive after the merge.

To merge contacts, use the `mergeContacts` method. This method returns nothing.

Pass the following parameters in this order:

1. The ID of the contact to keep. This parameter is known as the *kept* contact.
2. The ID of the contact to delete. This parameter is known as the *deleted* contact.

Merging contacts causes the address book UID on the contacts in ClaimCenter that match the deleted contact to be changed to that of the kept contact. They are then copied using the `ContactAutoSync` process. As with `updateContact`, the timing of this update depends on the configuration of the `ContactAutoSync` process.

## Handling Rejection and Approval of Pending Changes

An external contact management system might, like ContactManager, have the concept of *pending changes*. Pending changes are changes that are applied in ClaimCenter to contacts that require approval in the contact management system before being applied there.

ContactAPI has methods that the external contact management system can call to indicate if pending create or pending update operations have been approved or rejected. In each case, the methods pass in a parameter that contains the context of the original change, usually the user, claim, and contact information. This information enables ClaimCenter to notify the user of the results of the operation. If the change is rejected, ClaimCenter creates an activity for the user giving the details of the change that was rejected. If the rejection was for a pending update, ClaimCenter also copies the contact data from the contact management system and attaches it as a note to the activity.

These pending change methods are:

- pendingCreateRejected
- pendingCreateApproved
- pendingUpdateRejected
- pendingUpdateApproved



# Claim and Policy Integration

This topic describes web services, plugin interfaces, and tools for communicating with policy administration systems such as Guidewire PolicyCenter.

This topic includes:

- “Policy System Notifications” on page 512
- “Policy Search Plugin” on page 517
- “Claim Search Web Service For Policy System Integration” on page 522
- “ClaimCenter Exit Points to the Policy and Contact Management Applications” on page 524
- “PolicyCenter Product Model Import into ClaimCenter” on page 524
- “Catastrophe Policy Location Download” on page 535
- “Policy Location Search Plugin” on page 539
- “Policy Refresh Overview” on page 539
- “Policy Refresh Plugins and Configuration Classes” on page 542
- “Policy Refresh Steps and Associated Implementation” on page 544
- “Determining the Extent of the Policy Graph” on page 546
- “Policy Refresh Entity Matcher Details” on page 548
- “Policy Refresh Relinking Details” on page 552
- “Policy Refresh Policy Comparison Display” on page 556
- “Policy Refresh Configuration Examples” on page 565

**See also**

- “Web Services Introduction” on page 31
- “Plugin Overview” on page 163

# Policy System Notifications

ClaimCenter includes a general architecture for notifications from ClaimCenter to policy administration systems. The only built-in notification type in the default implementation is to detect large losses.

## General Purpose Notification System

The ClaimCenter architecture for policy notifications is flexible enough to support multiple types of policy system notifications.

The main aspects of the ClaimCenter notification architecture:

- *Preupdate rules* that detect specific types of issues and raise messaging events
- *Notification handler classes* for each type of policy system notification. Each one of these notifications correspond to one messaging event name. There is exactly one built-in notification handler class, which detects large losses.
- *Event messaging rules* that delegate work to the notification handlers to create messages to submit to the messaging queue. To do this, the rules call the handler's `createMessage` method.
- *A messaging transport* that gets each message (asynchronously in a separate thread) and asks the appropriate handler to send the message.
- *A policy system plugin interface* that defines the contract between ClaimCenter and an actual policy administration system. ClaimCenter includes a built-in version of this plugin interface that connects to PolicyCenter web services.
- *Event message rules to handle resync.*

Each notification handler for each type of policy system notification corresponds to one messaging event name.

ClaimCenter calls the handler at the following times:

- In Event Fired rules, to create a message for the messaging queue
- At message send time to perform any last-minute actions then call the plugin to send the message
- If a claim is resynced

Each of these times correspond to different methods in the notification handler class. Each notification handler can define its own behavior at each of these times. If you make new notification types, Guidewire recommends following the general pattern of the built-in large loss notification class.

In the default configuration, ClaimCenter includes one notification handler:

`LargeLossPolicySystemNotification`. This handler delivers large loss notifications to the policy administration system.

It is important to note that the handler mostly represents the type of notification itself, not necessarily all the implementation details. For instance, the built-in large loss notification handler delegates the actual work of sending the large loss message to the currently-registered version of the plugin interface `IPolicySystemNotificationPlugin`. If you make new notification types, Guidewire recommends following this general pattern and put your code that actually contacts the external system in your `IPolicySystemNotificationPlugin` implementation.

## Large Loss Notification Implementation Details

The following table summarizes the rule sets that ClaimCenter uses for large loss notifications.

Purpose	Rule Set	Description
Preupdate rules to detect large losses	Large Loss Notification	<p>The Preupdate → TransactionSetPreupdate → Large Loss Notification rule is disabled by default in the base configuration. When enabled, the rule fires whenever transaction sets are created or changed. The rule's conditions determine if the claim exceeds the large loss threshold for the claim's policy type. They also determine if a message is in the queue or if PolicyCenter has been notified. If the conditions pass, the rule adds a <code>ClaimExceedsLargeLoss</code> event to the claim. You define the thresholds in the administration user interface of ClaimCenter.</p>
Event rules to support policy system notifications in general	Policy System Notification	<p>The EventMessage → EventFired → Policy System Notification rule is a general rule for all policy system notification events. In the base configuration, this rule is disabled. If enabled, the rules can do the following:</p> <ol style="list-style-type: none"> <li>Determine the event name. For a large loss, the event name is <code>ClaimExceedsLargeLoss</code>. If you register additional notification handlers, the event name is different.</li> <li>Find the policy system notification handler that supports the current event name. For large loss, the handler class is <code>gw.policy.notification.LargeLossPolicySystemNotification</code>.</li> <li>The rules call the handler's <code>createMessage</code> method to create an outgoing message to persist to the database in the same transaction as the change that triggers the large loss notification. This call does not actually send the message. Sending the message happens asynchronously in another thread.</li> </ol> <p>The handler's <code>createMessage</code> method delegates the actual sending behavior to the class that implements the <code>IPolicySystemNotificationPlugin</code> plugin interface.</p> <p>The <code>message.EventName</code> property encodes which notification occurred. The body of the message contains information that the notification handler might need. For large loss, for example, it would contain the size of the loss.</p> <p>At message send time, the messaging transport calls the notification handler's <code>send</code> method and does not use this rule set.</p> <p>If a claim resynchronization happens, a <code>ClaimResync</code> event, ClaimCenter drops all queued (pending/errant) messages for the destination. However, ClaimCenter first calls the <code>EventFired</code> rule set to preserve and queue messages. These event rules trigger code that checks the notification handler's <code>MessageResyncBehavior</code> property for how to handle it. If the value is <code>COPY_LAST</code>, then only one pending message corresponding to that notification will be copied, the last one by send order.</p>

ClaimCenter can notify a policy administration system if a claim reaches a critical threshold, defined by policy type. The policy administration system can take appropriate actions to notify the policy's underwriter. Guidewire ClaimCenter is pre-configured to support this type of notification. When the claim exceeds the threshold, ClaimCenter creates a message using the messaging system. A special message transport plugin sends this message to your policy administration system and sends the claim's policy number, loss date, and the total gross incurred. This feature is called *large loss notification*.

Guidewire PolicyCenter includes a built-in integration to support this notification from ClaimCenter. PolicyCenter exposes a web service interface called `ClaimToPolicySystemNotificationAPI`. ClaimCenter calls this PolicyCenter web service across the network to add a referral reason and an activity in PolicyCenter. For more information about this from an application perspective, see “Large Loss Notifications” on page 511 in the *Application Guide*.

**Note:** If you use the built-in integration to PolicyCenter, you do not need to know the API details of the web service. For API details of this PolicyCenter web service, refer to the PolicyCenter documentation (the *PolicyCenter Integration Guide*).

If you want to use a policy administration system other than PolicyCenter, you can do so simply by writing your own implementation of the plugin interface `IPolicySystemNotificationPlugin`. That plugin is the code that actually notifies the external system.

---

**IMPORTANT** The definitions of the thresholds for what counts as a large loss are defined from within ClaimCenter, not in the policy administration system.

---

There is a `Claim` entity property called `LargeLossNotification` that tracks whether ClaimCenter has sent a notification for this claim. That property contains a typecode from the typelist `LargeLossNotification`. For the typecode values, see “Large Loss Notifications” on page 511 in the *Application Guide*.

The PolicyCenter implementation of the notification web service does the following:

1. **Finds the policy** – Finds the policy from its policy number, and throws an exception if it cannot find it
2. **Adds a referral reason** – Creates a referral reason on the policy for the large loss
3. **Adds an activity** – Adds a new activity to examine the large loss.

On the PolicyCenter side, you can modify the built in implementation of this API to do additional things. You could also have different code paths depending on the size of the gross total of the loss.

## Enabling Large Loss Notification

Follow the detailed instructions in “Enabling Large Loss Notification Integration” on page 99 in the *Installation Guide*.

To use the built-in plugin that connects to PolicyCenter, remember to set up your web services to connect to PolicyCenter. In Studio, in its Web Services editor, you can set the server, port, and authentication settings in that dialog.

If you want to define your authentication in Studio, set the web service to use HTTP authentication and specify your credentials there.

Alternatively, you can choose to leave the authentication setting on None and modify the code in the class `PCPolicySystemNotificationPlugin` to add authentication credentials. Look for the line:

```
_policySystemAPI = new ClaimToPolicySystemNotificationAPI()
```

Immediately afterward, add a line like the following and pass your user name and password:

```
_policySystemAPI.addHandler(new GWAAuthenticationHandler( "su", "gw" ))
```

For the full procedure, see “Enabling Large Loss Notification Integration” on page 99 in the *Installation Guide*.

## Using a Policy Administration System Other than PolicyCenter

If you use a policy administration system other than PolicyCenter, you must write your own implementation of the `IPolicySystemNotificationPlugin` plugin interface. Instead of contacting PolicyCenter, you can send the notification to your policy administration system through whatever API is appropriate. This might be web services but it could be some other remote procedure call. However, in the default implementation, the request must be synchronous.

---

**IMPORTANT** The remote procedure call to the external system must be synchronous. This is because a messaging transport calls the plugin implementation as part of its send process. Assuming there are no exceptions that your code throws, when your plugin methods complete, the message transport acknowledges the message immediately afterward. The message transport must be certain the external system got the request before acknowledging the message.

---

The `IPolicySystemNotificationPlugin` plugin has only a single method that you must implement, a method called `claimExceedsLargeLossThreshold`. It takes a loss date (as a `Calendar` object), a policy number (a `String` value), the gross total incurred (as a `String`), and a transaction ID (a `String`). Its method signature is:

```
claimExceedsLargeLossThreshold(java.util.Calendar lossDate, String policyNumber,  
String grossTotalIncurred, String transactionId) : void
```

The transaction ID is a identifier that ClaimCenter creates. Use this ID to avoid processing the same notification twice. For example, suppose ClaimCenter sends a notification and the remote system processes it but the reply back to ClaimCenter never arrives due to network failure. ClaimCenter does not know that the notification successfully delivered and retries using the same transaction ID. The remote system must detect that it has already processed a notification with this transaction ID and throw the exception `PolicySystemAlreadyExecutedException`. ClaimCenter catches this exception and marks the message as successfully delivered.

Plugin methods in this interface can throw `PolicySystemRetryableException` or `PolicySystemAlreadyExecutedException`. Typically the plugin talks to a remote system with web services over the SOAP protocol. The exception class `PolicySystemRetryableException` indicates a temporary problem contacting the remote system. The exception class `PolicySystemAlreadyExecutedException` if the remote system already processed a notification with that ID.

If you implement this plugin using a web service you must wrap any exceptions thrown by the web service. The web service throws exceptions specific to its WSDL so you will have to do something like:

```
try {  
    // call web service ...  
} catch (e : ...web service exception...) {  
    throw new gw.plugin.policy.PolicySystemRetryableException(e.message)  
}
```

### The Message Transport Acknowledges the Message

If no exceptions occur during the plugin notification method call, after the method completes, ClaimCenter marks the message to indicate that the policy administration system correctly received the message. This is called message acknowledgement. For more information about messaging acknowledgements, refer to “Messaging and Events” on page 299.

Additionally, each notification handler can set additional messaging-specific properties on the other objects as appropriate. For example, the built-in large loss notification handler sets the claim property `LargeLossNotificationStatus`.

## Adding Other Notification Types

The only built-in notification type is the large loss notification, but you can add custom types.

### To add new notification types

1. Create a new notification class that extends the class `gw.policy.notification.PolicySystemNotificationBase`. That base class contains some default behavior.
2. In your class definition, set up the ClaimCenter event name that corresponds to your notification handler. Also set up a static method that can instantiate an instance of your class. Follow the approach in the large loss handler:

```
/** The event name for this notification */  
public static final var EVENT_NAME : String = "ClaimExceedsLargeLoss"  
  
/** Singleton instance of this notification strategy */  
public static final var INSTANCE : LargeLossPolicySystemNotification =  
    new LargeLossPolicySystemNotification()  
  
private construct() {  
    super(EVENT_NAME)  
}
```

Replace `LargeLossPolicySystemNotification` with your class name and replace the `String` value `"ClaimExceedsLargeLoss"` with your event name.

3. Implement the `createMessage` method to generate a message. This method takes a message context object, which is the object that Event Fired rules get to assist in message creation. The large loss notification uses code like the following:

```
override function createMessage(context : MessageContext) {
    var claim = context.Root as Claim
    var amt = FinancialsCalculationUtil.getTotalIncurredGross().getAmount(claim)
    var renderedAmt = CurrencyUtil.renderAsCurrency(amt)
    var msg = context.createMessage(renderedAmt)
    msg.MessageRoot = claim
    claim.LargeLossNotificationStatus = "InQueue"
}
```

Note that it sets a field on the claim itself to indicate the claim notification is in queue. Also note that the method does not need to indicate what type of notification it is. Even if you add additional notifications, you do not need to add a special property to the message entity to identify the type of notification. Instead, your message code that runs later can detect which notification type by checking the `message.EventName` property.

4. Implement the `send` method, which ClaimCenter calls from the messaging transport that gets messages from the send queue. The message transport gets messages one by one from the send queue in another thread. Any changes on the message happen in different database transactions than the original change that triggered creation of the message. The `send` method takes a reference to the instance of the policy system notification plugin, the `Message` entity, and the transformed payload `String`. The transformed payload `String` is a variant of the `Message.payload`, and might be different if the messaging destination used a `MessageRequest` plugin in addition to the `MessageTransport` plugin. Generally speaking it is best to use the transformed payload rather than `Message.payload`. Your version of this method might just do what large loss does, which is call the plugin method that matches your notification type. If you added additional methods to your plugin implementation, you will need to cast the plugin reference to your specific implementation class type before calling custom methods on it. The large loss version looks like:

```
override function send(plugin : IPolicySystemNotificationPlugin, message : Message,
                      transformedPayload : String) {
    var claim = message.Claim
    plugin.claimExceedsLargeLossThreshold(claim.LossDate.toCalendar(),
                                          claim.Policy.PolicyNumber, transformedPayload, message.PublicID)
}
```

5. Implement the `afterSend` method. ClaimCenter calls this immediately after sending the message. You can use this as a hook to set messaging-specific fields on the `Claim`. For example:

```
override function afterSend(message : Message, status : MessageStatus) {
    if (status == GOOD) {
        message.Claim.LargeLossNotificationStatus = "Sent"
    } else if (status == NON_RETRYABLE_ERROR) {
        message.Claim.LargeLossNotificationStatus = "None"
    }
}
```

6. Implement the `MessageResyncBehavior` property getter. ClaimCenter responds differently during resync based on the value:

- If the value is `DROP`, ClaimCenter does not resend the notification during resync.
- If the value is `COPY_LAST`, then ClaimCenter copies only one pending notification message corresponding to that notification type for that messaging destination for that claim. ClaimCenter uses the last one, by send order.
- If the value is `COPY_ALL`, then ClaimCenter copies all pending notification messages for that notification type for that messaging destination for that claim.

For example, a simple implementation of this looks like:

```
override property get MessageResyncBehavior() : MessageResyncBehavior {
    return COPY_LAST
}
```

7. Register your notification type handler with the system by modifying the class `gw.policy.notificationPolicySystemNotificationList`. This is a configurable list of all policy system

notifications. Adding a notification to this list makes it known to the policy system notification event messaging rules and to the messaging transport `PolicySystemNotificationMessageTransport`. Although you cannot add new methods to the actual interface for the `IPolicySystemNotificationPlugin`, you can add new notifications to this list. If you do this, override the notification `send` method to directly call whatever mechanism is used for calling the policy administration system.

The built-in version of this list looks like the following:

```
class PolicySystemNotificationList {  
  
    /** List of all available notifications */  
    public static var ALL : List<PolicySystemNotificationBase>  
        = { LargeLossPolicySystemNotification.INSTANCE }.freeze()  
  
    /** Never instantiated */  
    construct() {}  
  
}
```

To add another notification class called `abc.integration.MyNotificationHandler`, change the list to look like this:

```
class PolicySystemNotificationList {  
  
    /** List of all available notifications */  
    public static var ALL : List<PolicySystemNotificationBase>  
        = { LargeLossPolicySystemNotification.INSTANCE ,  
            abc.integration.MyNotificationHandler.INSTANCE }.freeze()  
  
    /** Never instantiated */  
    construct() {}  
  
}
```

8. Add new preupdate rules that detect whatever situation you are interested in. If you detect the condition, raise the event with code such as:

```
claim.addEvent("MyEventName")
```

9. If you have not already enabled the notification system rules and plugins, see “Enabling Large Loss Notification Integration” on page 99 in the *Installation Guide*.

10. Test your notification system with a development version of ClaimCenter and your policy administration system.

## Policy Search Plugin

The policy search plugin enables ClaimCenter users to search for a policy, review the list of possible choices, select a specific policy, and retrieve the full details of that policy. The policy details are stored with the claim as a local copy. The `IPolicySearchAdapter` plugin interface defines a search method and a retrieve method to support standard policy search and retrieval. This plugin defines an additional retrieval method to handle the special case of policy refresh, which is discussed later in this topic.

If you use Guidewire PolicyCenter, you can use a built-in implementation of this plugin that queries PolicyCenter for policies. To use this plugin, in Studio, navigate in the Project window to `configuration` → `config` → `Plugins` and then double-click `IPolicySearchAdapter.gwp`. Then register the Gosu class `gw.plugin.pcintegration.pc800.PolicySearchPCPlugin`. For detailed instructions on this integration, see “Enabling Integration between ClaimCenter and PolicyCenter” on page 97 in the *Installation Guide*.

If you use a policy administration system other than PolicyCenter, you must write your own implementation of this plugin interface.

The base configuration of ClaimCenter also includes a demonstration version of this plugin implementation, which generates example policies.

To understand the flow of ClaimCenter policy search, it is important to distinguish the two concepts of search versus retrieve from the context of ClaimCenter:

- **Search means getting policy summaries, possibly many of them** – *Search* involves a call to an external system to retrieve policy summaries only, returning zero, one, two, or a very large number of results. The `searchPolicies` method receives a set of criteria and returns a subset of the full policy information, which in the typical case determines the policy that the user wants. The return result is a `PolicySearchResultSet`. This object has a `Summaries` property that contains an array of `PolicySummary` entities—an array of policy summaries.
- **Retrieval means getting a single, unique, full policy** – The two `retrieve` methods take policy summaries that contain a specific policy number, an effective date, such as a loss date, and other search information. They retrieve a single, unique version of the policy that is effective as of a specific effective date.

#### Typical Chronological Flow of Policy Search and Retrieval

The chronological flow for policy search and retrieval is as follows. In particular, note the distinction between the words search and retrieve, and between summaries and entire policy.

1. The user requests a policy search in the ClaimCenter user interface.
2. The policy search user interface pages use a PCF page to display the search fields.
3. The PCF pages encode user-entered data in properties in a `PolicySearchCriteria` object. This object is non-persistent—it is described in the ClaimCenter Data Dictionary, but has no corresponding table in the database for persisting it.
4. To begin the search, ClaimCenter calls `IPolicySearchAdapter.searchPolicies(PolicySearchCriteria)`, which returns a `PolicySearchResultSet` entity. This entity contains an array of `PolicySummary` entities to display as summaries for the user to select.

At some future time, ClaimCenter must retrieve the entire policy, not merely the summary. At that time, ClaimCenter uses the `PolicySummary` entity to retrieve the policy. Because of this usage, the `PolicySummary` must contain all the information required for policy retrieval.

**IMPORTANT** If any policy-related properties are required to retrieve a single unique policy, store that information in the `PolicySummary` entity. Extend the data model with new properties as necessary to store that information. For example, suppose at policy retrieval time you want to return only a single vehicle on the policy based on special search information passed to the search. Copy that information from the `PolicySearchCriteria` to each `PolicySummary` entity. For more information about returning a subset of vehicles or real estate properties, see “Policy Retrieval Additional Information” on page 520.

5. The selected summary is stored in the `NewClaimPolicyDescription` entity on the `NewClaimPolicy` object, which is part of the wizard user interface.
6. The user selects a summary to be retrieved by using the user interface.
7. To retrieve a single unique policy from the external system, ClaimCenter calls `IPolicySearchAdapter.retrievePolicyFromPolicySummary(PolicySummary)`. This method must return a `PolicyRetrievalResultSet`, which contains a `Policy` entity. For more details about the properties of `PolicyRetrievalResultSet`, see “Policy Retrieval Additional Information” on page 520.
8. If you refresh—reload—the `Policy` in the user interface, ClaimCenter calls `IPolicySearchAdapter.retrievePolicyFromPolicy(Policy)` with the current local version of the policy. If you need any properties from the original policy search or retrieval at the time of policy refresh, you must set them on the `Policy` entity during original policy retrieval. The `retrievePolicyFromPolicy` method can use these properties. This method also returns a `PolicyRetrievalResultSet` entity.

**IMPORTANT** Implement the plugin method `retrievePolicyFromPolicy` to support ClaimCenter refreshing—reloading—the policy.

In other words, if you are searching for a policy, in the typical case the following methods are called:

- Clicking the **Search** button calls the `IPolicySearchAdapter` method `searchPolicies`.
- Clicking the **Next** button to leave the first page calls the `IPolicySearchAdapter` method `retrievePolicyFromPolicySummary`.

If you never return to the first page, there are no more calls to the plugin's `searchPolicies` method.

However, the path is slightly different for commercial auto and commercial property policies, for which there is an extra page in which you can select risk units. That page is titled either **Select Involved Policy Vehicles** or **Select Involved Policy Properties**. For these types of policies, the `retrievePolicyFromPolicySummary` method call happens only after you click the **Next** button on that extra risk unit selection page.

It does not matter what kind of search criteria you use, such as policy number or policy type. ClaimCenter calls `searchPolicies` once and then calls `retrievePolicyFromPolicySummary`.

Every time you hit the **Search** button, ClaimCenter calls the plugin's `searchPolicies` method. In other words, if you do 10 searches you get 10 calls to that method.

If you do not search for a policy, then you are creating a new, unverified policy. This approach does not trigger a call to the `IPolicySearchAdapter` at all. However, you can go back to the first page and click the **Search** button to switch your approach and search for verified policies.

If you searched for a policy and you return to the first page, ClaimCenter again calls `searchPolicies` as soon as you enter the page. If you leave the page without changing anything, there are no additional calls to the plugin. However you can deselect the policy, which triggers a call to `searchPolicies`, and then do more searches, and then change the policy. In that case, ClaimCenter calls `searchPolicies` once per search, and calls `retrievePolicyFromPolicySummary` again when you click **Next**.

### Multicurrency and Policies

If you use multicurrency mode, you must set the currency on the policy in the `Policy.Currency` property.

### Policy Search Example and Additional Information

The following example shows a simple policy search plugin's main function in Java:

```
public PolicySearchResultSet searchPolicies(PolicySearchCriteria policySearchCriteria)
    throws RemoteException {
    int maxPolicySearchResults = 25; // must agree with MaxPolicySearchResults in config.xml
    // get whatever properties you need from the PolicySearchCriteria...
    LossType lossType = policySearchCriteria.getLossType();
    String vin = policySearchCriteria.getVin();
    PolicyType policyType = policySearchCriteria.getPolicyType();
    String propertyCity = policySearchCriteria.getPropertyAddress().getCity();

    // [at this point, send these properties across to a remote system and get results back]
    // Populate the result object with an array of policy summaries
    // In this case just return one result
    PolicySearchResultSet policySet = (PolicySearchResultSet)EntityFactory.getInstance().newEntity(
        PolicySearchResultSet.class);
    PolicySummary policySummary =
        (PolicySummary) EntityFactory.getInstance().newEntity(PolicySummary.class);
    policySummary.setPolicyNumber("00-00001");

    // [set any other desired properties...]
    PolicySummary[] policySummaryArray = new PolicySummary[1];
    policySummaryArray[0] = policySummary;
    policySet.setSummaries(policySummaryArray);
    policySet.setUncappedResultCount(1);

    if (policySet.getUncappedResultCount() > maxPolicySearchResults) {
        policySet.setResultsCapped(true);
    } else {
        policySet.setResultsCapped(false);
    }

    return policySet;
}
```

If your code path does not support a very large number of results, you can set a maximum number of results to show. You set this number in the `MaxPolicySearchResults` configuration parameter in the `config.xml` file. In the default configuration, this parameter is set to 25. This setting affects only the number of rows that ClaimCenter shows. The plugin must observe the same limitation during search. Never return more than that maximum number of results.

Set the `PolicySearchResultSet.UncappedResultCount` property to the number of results found according to the search criteria, even if this number exceeds the number of results actually returned in the result set. Also set the `PolicySearchResultSet.ResultsCapped` property to `true`. If you do so, the ClaimCenter user interface can show a message that informs the user when there are “too many results found to display”.

You can also add as many additional search criteria for policy searches as you want, as follows:

1. Extend the `PolicySearchCriteria` object’s data model with new extension properties just like you would extend the data model of any other ClaimCenter entity.
2. If appropriate, modify the PCF pages to prompt for new user data that relate to those properties. Alternatively, the new search criteria could be set programmatically based on some other factors other than new user data. For instance, the context within the ClaimCenter user interface could be such a factor.
3. Alter the PCF files appropriately to set these extension properties from form field data.

Sometimes policies have a very large number of insured risks, such as a large number of insured vehicles or buildings. In some cases you might need details only for a specific risk involved in the claim. The search criteria can identify the specific risk, for example, by passing a vehicle’s Vehicle Identification Number (VIN). By adding this search criterion, the policy search plugin can return only the coverages for that risk with the policy summary. If the plugin does not use some search criteria, ignore that search criteria data. It might be appropriate to remove those search fields from the user interface if they are always ignored.

### Policy Retrieval Additional Information

As discussed earlier, *policy retrieval* refers to returning a single policy, rather than a list of matches. Refer to the chronological flow for how this happens in a typical search and retrieval.

The return result of the `retrievePolicyFromPolicySummary` plugin method is an entity called `PolicyRetrievalResultSet`. It has properties of its own but is mostly a shell for a `Policy` entity within the `PolicyRetrievalResultSet.Result` property.

Once this method retrieves the full details for a policy, it creates a new `Policy` object and populates its properties from data from the external system. The method adds that to a new `PolicyRetrievalResultSet` entity and returns it.

The `Policy` object is a fairly complicated object with a lot of subobjects that must be populated. One aspect that is particularly difficult is associating contacts—people and companies—as the insured, as the agent, as interested parties, and so on. These associations are set by linking the contact to the policy and describing the ways in which the contact is related as a set of roles. Refer to the ClaimCenter Data Dictionary for the complete set of properties on the `Policy` entity.

There are three possible outcomes of a policy retrieval, and the cases can be distinguished by values in the `PolicyRetrievalResultSet`:

- **Successful** – The retrieval parameters map to a single, unique policy. The result entity has a `Result` property that contains a `Policy` entity, rather than `null`. Also, the result entity has a `NotUnique` property set to `false`.
- **Unsuccessful, multiple matches** – Unsuccessful because retrieval parameters map to multiple policies. The result entity has a `Result` property that contains a `Policy` entity, rather than `null`. The result entity has a `NotUnique` property set to `true`.
- **Unsuccessful, no match** – Unsuccessful, because retrieval parameters do not map to any policies. The result entity has a `Result` property that contains `null`.

One way to implement policy retrieval is to return the entire policy, including all subobjects. However, you can choose to save bandwidth and return only a subset of the vehicles or real estate properties. Perhaps return only objects included in the original search criteria that triggered the search, the summary of which a user clicked on, and thus triggered a retrieval. For example, if a search was triggered based on a specific vehicle, that information can be used only to retrieve that vehicle and not other vehicles on the policy. If you want to do this, be careful to note the following special properties on the `Policy` entity:

- If a policy insures vehicles, your plugin can return a subset of vehicles, perhaps only one vehicle. If you return a subset of vehicles, set the `Policy.TotalVehicles` property to the actual number of insured vehicles. Do not simply set it to the number of vehicles in `Policy.Vehicles`. Setting the total correctly allows the user interface to notice this difference between the two numbers. The application shows a message saying that not all vehicles on the policy are available in ClaimCenter.
- If a policy insures real estate properties, your plugin can return a subset of real estate properties, perhaps only one real estate property. If you return a subset of properties, set the `Policy.TotalProperties` property to the actual number of insured real estate properties. Do not set it to the number of real estate properties in `Policy.Properties`. Setting the total correctly causes the user interface to show the difference between the two numbers. The application shows a message saying that not all real estate properties on the policy are available in ClaimCenter.

The following example shows what to do in the policy retrieval plugin implemented in Gosu:

```
override function retrievePolicyFromPolicySummary( policySummary : PolicySummary ) :  
    PolicyRetrievalResultSet  
  
    var p = new Policy()  
    p.policyNumber = "123"  
    p.policyType = PolicyType.TC_AUTO_PER  
    p.policyStatus = PolicyStatus.TC_EXPIRED  
    p.isVerified = true  
  
    var insured = new Person();  
    insured.EmailAddress1 = "insured@testmail.com"  
    insured.EmailAddress2 = "jdoe@central.org"  
    insured.FirstName = "insured_firstname"  
    insured.HomePhone = "532-453-0989"  
    insured.LastName = "insured_lastname"  
    insured.LicenseNumber = "CL50976800"  
    insured.LicenseState = State.TC_CA  
    insured.Occupation = "Technical Writer"  
    insured.Preferred = Boolean.TRUE  
    insured.Prefix = NamePrefix.TC_MR  
    insured.PrimaryPhone = PrimaryPhoneType.TC_WORK;  
  
    // Set all your other properties on the Policy entity.  
    ...  
  
    var policySet = new PolicyRetrievalResultSet();  
    policySet.NotUnique = Boolean.FALSE // if successful, and there is a single match  
    policySet.Result = p  
  
    // Return the result set.  
    return policySet  
}
```

## Modifying New Claim Wizard to Reload Policy

The New Claim Wizard retrieves policy information from an external system. On the first page of the wizard the user typically does a policy search. ClaimCenter searches an external system for a policy for the new claim or creates a new one, based on the current policy description. If the policy description matches the claim's current policy, the wizard's `setPolicy` method does nothing.

The search uses the registered policy search plugin and returns a list of policy summaries, which are extremely simplified versions of policies. ClaimCenter shows these summaries in a list view so you can choose a policy. When you click **Next**, the wizard asks the policy search plugin for the policy for the picked summary. ClaimCenter sets this policy as the claim's policy.

However, suppose you return to the first step in the wizard and pick a new policy, and click **Next** again. ClaimCenter must decide whether to request the policy search plugin to retrieve another policy or instead keep—reuse—the current policy. ClaimCenter makes this decision by using a method called `isSamePolicy` on a policy summary. The wizard compares the currently picked summary with the policy that is on the claim. If `isSamePolicy` returns `true`, the wizard assumes that the claim's current policy is still the one wanted. If `isSamePolicy` returns `false`, the wizard replaces the claim's current policy with a new one, fetched with the policy search plugin. If the user explicitly picks a different external policy, `isSamePolicy` returns `false`, which triggers ClaimCenter to refresh the policy.

To decide whether two policies are the same, ClaimCenter compares a small number of fields to see if they match. By default, the set of policy summary fields ClaimCenter checks are the following: `PolicyType`, `PolicyNumber`, `EffectiveDate`, and `ExpirationDate`. To compare additional fields, modify the enhancement file `PolicySummary.gsx`, which implements the policy summary `isSamePolicy` method. You can make additions or removals to that array of property names to customize how ClaimCenter compares policy summaries to determine if two policies match.

For example, suppose you want to assume that two policy summaries with different loss dates must always trigger refreshing the policy. Modify `isSamePolicy` and add "`LossDate`" to the array of `String` values that represent property names.

## Claim Search Web Service For Policy System Integration

If you use a supported version of PolicyCenter, PolicyCenter uses the ClaimCenter web service `PCClaimSearchIntegrationAPI` web service to retrieve claim summaries based on search criteria. Typically you do not need to call this web service directly. PolicyCenter calls this web service automatically if you configure PolicyCenter to connect to ClaimCenter for claim search.

If you use a policy administration system other than PolicyCenter, your policy administration system can call this web service. However, this web service is written specifically to support a data model very similar to what PolicyCenter needs. If you use a policy administration system other than PolicyCenter, consider writing your own custom web services that directly address integration points between ClaimCenter and your specific policy administration system.

**Note:** The `ClaimAPI` contains some similar methods. For example, the `ClaimAPI` web service provides the `findPublicIdByClaimNumber` method and the `PCPolicySearchIntegrationAPI` web service provides the `getClaimByClaimNumber` method. However, neither web service returns a complex claim graph. Instead, add custom web services to search for claims or export claim summaries in the data model format your policy administration system needs. Only export the claim fields and subobjects that are necessary.

### Search for Claims

If PolicyCenter or another policy administration system wants to identify a claim by a claim number and an effective date, it can call the `searchForClaims` method in this interface. It takes a `PCClaimSearchCriteria` object, which encapsulates the search criteria from the policy administration system. It includes the properties described in the following table.

Property	Type	Description
<code>BeginDate</code>	<code>Calendar</code>	The begin date.
<code>EndDate</code>	<code>Calendar</code>	The end date.
<code>Lob</code>	<code>String</code>	Line of business, specified by its name
<code>PolicyNumbers</code>	<code>String[]</code>	An array of policy numbers, as an array of <code>String</code> values

Set some or all of these properties and pass the `PCClaimSearchCriteria` to the `searchForClaims` method.

The result is an array of `PCClaim` entities, which are effectively like claim summary objects. They do not contain the full claim detail or object graph. The each object contains the following fields.

Property	Type	Description
<code>ClaimNumber</code>	<code>String</code>	The claim number
<code>LossDate</code>	<code>Calendar</code>	The loss date
<code>PolicyNumber</code>	<code>String</code>	The policy number of the best matched policy
<code>PolicyTypeName</code>	<code>String</code>	The policy type name
<code>Status</code>	<code>String</code>	The claim status, as a string
<code>TotalIncurred</code>	<code>BigDecimal</code>	The total incurred amount on the claim.

The following example uses the `PCClaimSearchCriteria` constructor to assemble the object, and then calls the web service method:

```
String[] polNums = { "abc:234234", "abc:298734" };
PCClaimSearchCriteria crit = new PCClaimSearchCriteria(beginDate, endDate, theLOBName, polNums);

// call the API
PCClaim results = pcclaimSearchCriteria.searchForClaims(crit);

// get results
print(results[0].ClaimNumber);
print(results[0].Status);
print(results[1].ClaimNumber);
print(results[1].Status);
```

### Get Number of Matched Claims

If you want to get only the number of results that match, PolicyCenter or another policy administration system calls the `getNumberOfClaims` method. It takes the same `PCClaimSearchCriteria` as the `searchForClaims` method. The `getNumberOfClaims` method returns an integer number of matching claims.

The following example uses the `PCClaimSearchCriteria` constructor to assemble the object, and then calls the web service method:

```
String[] polNums = { "abc:234234", "abc:298734" };
PCClaimSearchCriteria crit = new PCClaimSearchCriteria(beginDate, endDate, theLOBName, polNums);

// call the API
int totalResults = pcclaimSearchCriteria.searchForClaims(crit);
```

### Get Claim Detail

If PolicyCenter or another policy administration system wants more detail, it calls this web service interface's `getClaimByClaimNumber` method. It takes a claim number and returns a `PCClaimDetail` entity. This object contains much more information about a claim, such as the remaining reserves and the loss cause. Refer to the Data Dictionary or the plugin reference Javadoc for details of the full set of properties.

### User Claim View Permission

PolicyCenter can directly direct a PolicyCenter user to the ClaimCenter user interface. To make this process work smoothly, it might be necessary to give a PolicyCenter user permission to view a claim. PolicyCenter calls the `PCPolicySearchIntegrationAPI` web service interface method `giveUserClaimViewPermission` to give the user permission to view the claim. This method takes a claim public ID and a user name, both as `String` values.

## ClaimCenter Exit Points to the Policy and Contact Management Applications

There are screens in the ClaimCenter user interface in which a user can directly open another application in a separate browser window. This feature is implemented by using the PCF widget `ExitPoint`, which in the base configuration sends a URL to a corresponding PCF widget called `EntryPoint` in the target Guidewire application.

In the base configuration, ClaimCenter provides `ExitPoint` widgets that can:

- Open PolicyCenter to directly view and edit policy information.
  - The file `ClaimPolicyGeneral.pcf` provides a **View Policy in Policy System** button that uses the `ViewPolicy` exit point.

To set the URL used in this PCF file for the exit point to PolicyCenter, edit the configuration parameter `PolicySystemURL` in `config.xml`. Set this parameter to the URL for the server, such as:

`http://localhost:8180/pc`

See “Enabling Integration between ClaimCenter and PolicyCenter” on page 97 in the *Installation Guide*.

- The file `PolicyLocationSearchResultsLV.pcf` provides a cell in the search results list view that uses the `ViewPolicy` exit point.

The URL for the exit point to PolicyCenter is defined in the ClaimCenter file `suite-config.xml`. For example:

```
<product name="pc" url="http://localhost:8180/pc"/>
```

You can change the `url` value if the URL to your policy system is different.

- Open ContactManager to directly view and edit contact information. The file `AddressBookSearchScreen.pcf` provides an **Open in ContactManager** button that uses the `GoToAB` exit point.

The URL for the exit point to ContactManager is defined in the ClaimCenter file `suite-config.xml`. For example:

`http://localhost:8280/ab`

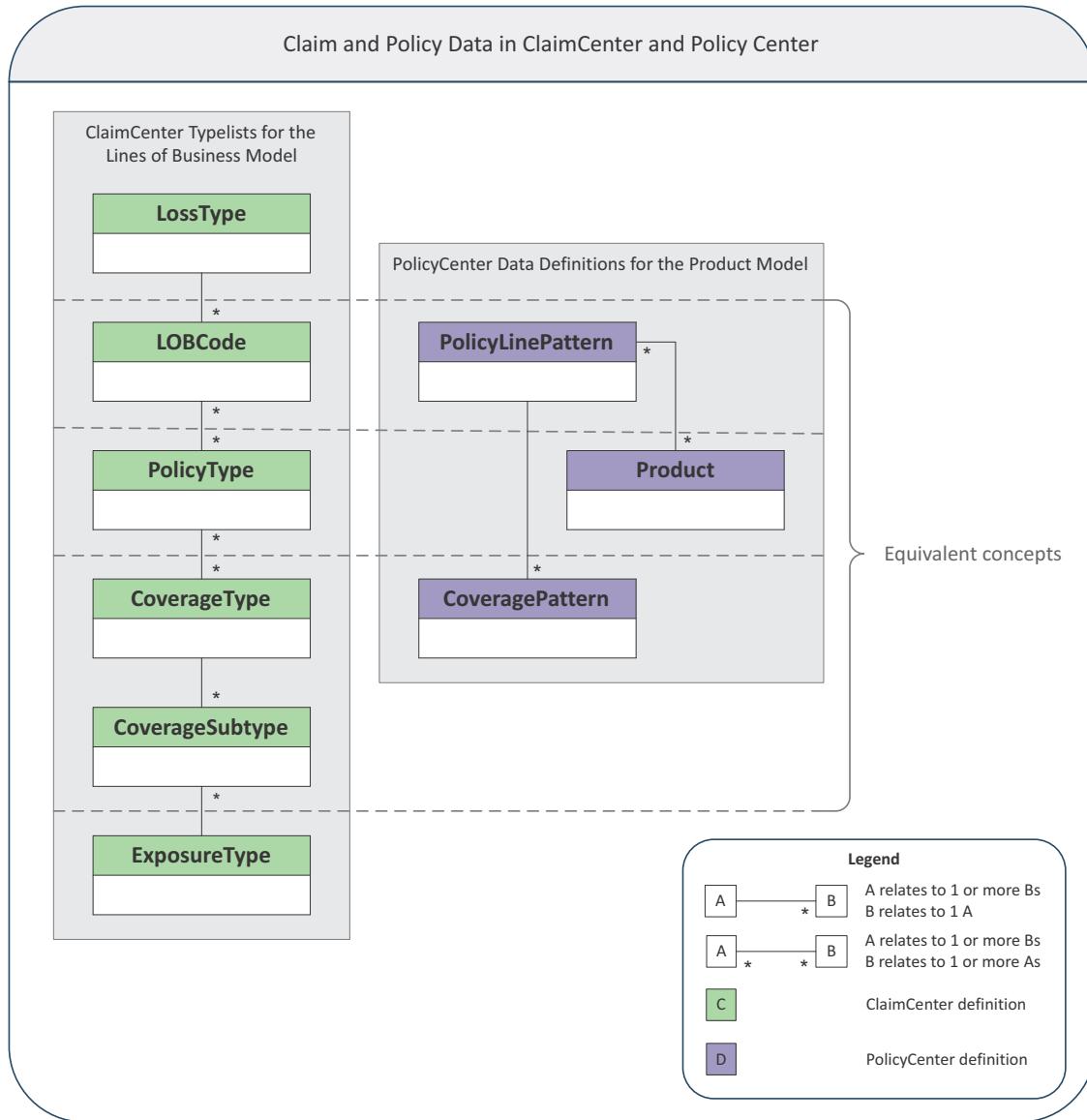
See “Integrating ContactManager with ClaimCenter in QuickStart” on page 46 in the *Contact Management Guide*.

You can integrate with a system other than a Guidewire core application, such as a policy system that is not Guidewire PolicyCenter. If you want to send the system additional parameters, you can add them to the URL configuration parameter for that system. If the configuration parameter is absent or commented out or set to the empty string, exit point buttons for that system in the user interface are hidden.

## PolicyCenter Product Model Import into ClaimCenter

If you run instances of ClaimCenter and PolicyCenter together, you must keep your lines of business model synchronized with your product model. When you change your PolicyCenter product model, you must merge the changes with your ClaimCenter lines of business model to keep them synchronized. PolicyCenter provides the ClaimCenter Typelist Generator to help you synchronize your ClaimCenter line of business model with your PolicyCenter product model. The ClaimCenter Typelist Generator is a command line tool.

The ClaimCenter lines of business model uses some data definitions from the PolicyCenter product model, as the following diagram shows.



For example, LOB codes in ClaimCenter are equivalent to policy lines in PolicyCenter. Policy types are equivalent to products, and coverage types are equivalent to coverages. The generator adds new LOB typecodes to the ClaimCenter LOB typelist that correspond to codes for new PolicyCenter policy lines. The generator adds new typecodes to the typelists for policy type and coverage type in a similar way.

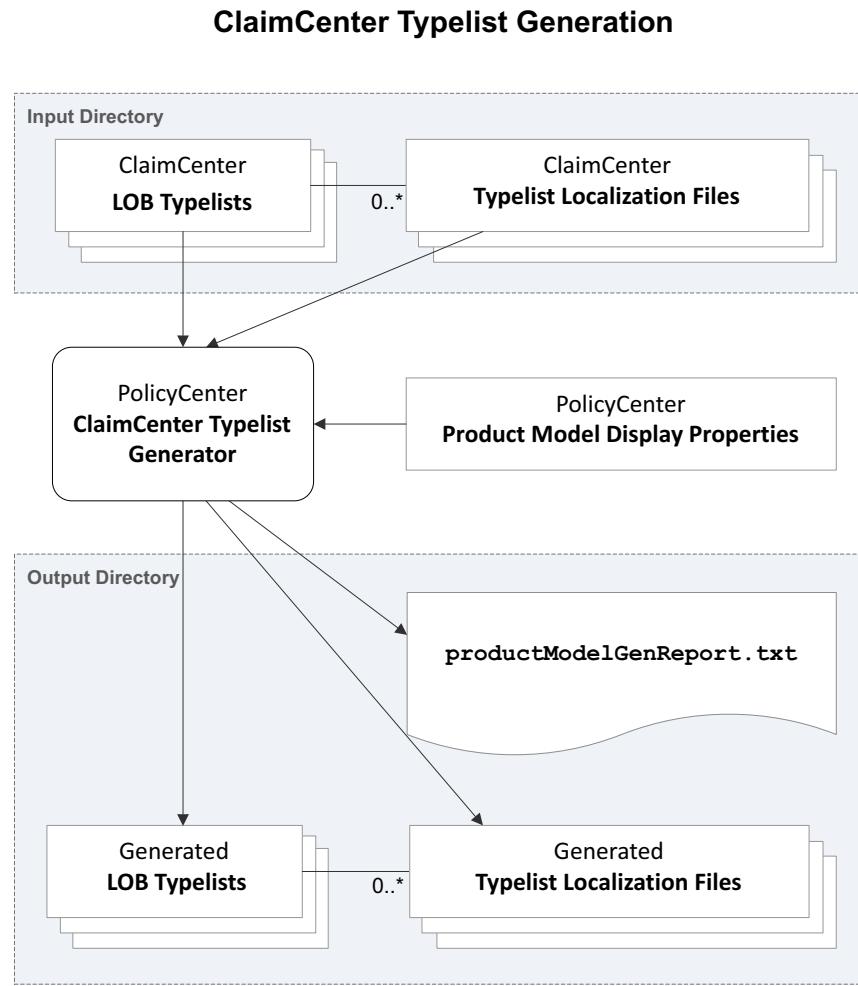
However, the ClaimCenter lines of business model has typelists in its hierarchy that fall above and below its PolicyCenter equivalents. For example, LOB codes in ClaimCenter link to loss types. The generator does not know about loss types. You must link new LOB codes to their parent loss types manually in ClaimCenter Studio. In a similar way at the bottom hierarchy, you must link new coverage types/coverage subtypes from PolicyCenter to exposure types in ClaimCenter.

## Configuring the ClaimCenter Typelist Generator

When you run the ClaimCenter Typelist Generator, you specify the following options:

Option	Description
Input directory	Location from where the generator reads ClaimCenter typelists and typelist localization files.
Output directory	Location from where the generator writes ClaimCenter typelists and typelist localization files.
Map new coverages to general damage exposure	Specify whether the generator associates new coverages from PolicyCenter with the generic General Damage exposure type in the base configuration of ClaimCenter. Select this option if you work in development mode or want to demonstrate the products together. For more information, see “Linking PolicyCenter Coverages to the ClaimCenter General Damage Exposure Type” on page 528.

The following diagram illustrates the basic operation of the ClaimCenter Typelist Generator.



### Input Files

The ClaimCenter Typelist Generator reads the following ClaimCenter typelist files as input:

- `LOBCode.ttx`
- `PolicyType.ttx`

- CoverageType.ttx
- CoverageSubtype.ttx
- ExposureType.ttx
- CovTermPattern.ttx
- LossPartyType.ttx

Always put the latest versions of your ClaimCenter lines of business typelist files in the input directory. With input files, the generator preserves links between LOB codes and loss types that you configured in ClaimCenter Studio. In a similar way, providing input files preserves links between coverage types and exposure types. The generator also preserves typecodes that exist in ClaimCenter from other, third-party policy administration systems.

**Note:** The ClaimCenter Typelist Generator reads in and writes out the `ExposureType.ttx` file only if you configure the generator to map new coverages to the General Damage exposure type.

### Input and Output Directories

The ClaimCenter Typelist Generator can use the same directory for the input and output directories. The generator reads input files when it starts and writes output files when it finishes. When the input and output directories are the same, your typelist and properties files are overwritten with generated changes. This arrangement works well for demonstration situations. However, using the same directory for input and output prevents you from using a diff tool to determine what typecodes the generator changed.

---

**WARNING** Do not configure the ClaimCenter Typelist Generator to use the ClaimCenter base configuration directory, `ClaimCenter/modules/cc/config/extensions` for input or output. Instead, use the custom configuration directory, `ClaimCenter/modules/configuration/config/extensions`. Otherwise, ClaimCenter does not start.

---

### Development or Demonstration Environment

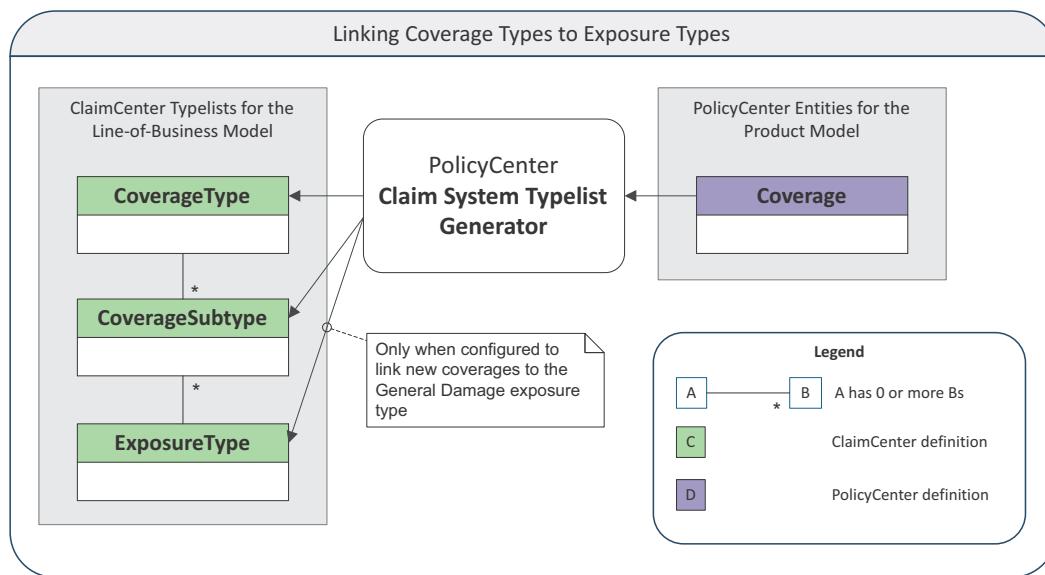
In a development or demonstration environment, you generally set up a PolicyCenter and a ClaimCenter instance on the same machine. If so, the generator can use the ClaimCenter directory that holds the lines of business typelist files as the input and output directories. This avoids steps to manually copy files to and from ClaimCenter. However, this approach does not work if your ClaimCenter and PolicyCenter instances are configured with multiple locales.

### Production Environment

In a production environment, do not configure the ClaimCenter Typelist Generator to use ClaimCenter directories for input or output.

## Generated Coverage Subtypes

In the ClaimCenter lines of business model, you link coverage types to exposure types through coverage subtypes.



Coverage subtypes essentially duplicate their coverage types. From a technical perspective, they help implement many-to-many relationships between coverages and exposures on claims.

For new coverages in PolicyCenter, the generator creates corresponding coverage types and subtypes in `CoverageType.ttx` and in `CoverageSubtype.ttx`. To link generated coverage types to generated subtypes, the generator adds them as categories of each other. Use the generated subtypes to link corresponding coverage types to exposure types in ClaimCenter Studio.

## Linking PolicyCenter Coverages to the ClaimCenter General Damage Exposure Type

You can configure the ClaimCenter Typelist Generator to associate new coverages from PolicyCenter with the General Damage exposure type in ClaimCenter. If you select to map new coverages to general damage exposure when you run the generator, then you must include the typelist file `ExposureType.ttx` in the input directory. The generator updates the `GeneralDamage` typecode by adding new coverage subtypes as categories. You select to map new coverages to general damage exposure by running the generator with `-Dmap_coverages=true`.

If you configure the generator to link new coverages to `GeneralDamage`, you can demonstrate intake of First Notice of Loss (FNOL) without further configuration in ClaimCenter Studio. ClaimCenter displays a generic exposure page, which lets users open new claims against policies with the new coverages.

In a production environment however, Guidewire recommends that you map coverages to more specific exposure types in ClaimCenter Studio. Do not configure the generator to link new coverages to the General Damage exposure type if you plan to use more specific exposure types. Otherwise, you must find and delete links to the General Damage before you create new links to correct exposure types.

## Preserving Third-Party Claim System Codes in Generated Typelists

PolicyCenter knows which codes in ClaimCenter typelists come from PolicyCenter by their source system categories. A value of PC indicates ClaimCenter codes that originate in PolicyCenter. The ClaimCenter Typelist Generator adds, changes, or deletes only codes that originate in PolicyCenter.

Any value for source system category other than PC, including the absence of a source system category, indicates ClaimCenter codes that originate somewhere other than PolicyCenter. Those typecodes pass through the generator unchanged from input to output.

## Merging PolicyCenter Localization with ClaimCenter Localization

If you configure your PolicyCenter and ClaimCenter instances with multiple locales, the ClaimCenter Typelist Generator helps you merge PolicyCenter localization with ClaimCenter localization. For more information, see “Typelist Localization” on page 534.

## Running the ClaimCenter Typelist Generator

Before you run the ClaimCenter Typelist Generator, you must:

- Know the location of the input and output directories.
- Place the ClaimCenter typelist and typelist localization files in the input directory.

The generator will preserve the lines of business codes that you configured in ClaimCenter Studio or imported from other third-party policy administration systems.

### To run the ClaimCenter Typelist Generator

1. At a command prompt, navigate to the *PolicyCenter\bin* directory.

2. Type the following command:

```
gwpc cc-typelist-gen -Dinput_dir=input_dir -Doutput_dir=output_dir -Dmap_coverages=true_or_false
```

Where the command line arguments are:

Argument	Value
-Dinput_dir	Specify the input directory. Required.
-Doutput_dir	Specify the output directory. Required.
-Dmap_coverages	Specify true to map new coverages to general damage exposure. Optional. Default is false.

For more information about the command line arguments, see “Configuring the ClaimCenter Typelist Generator” on page 526.

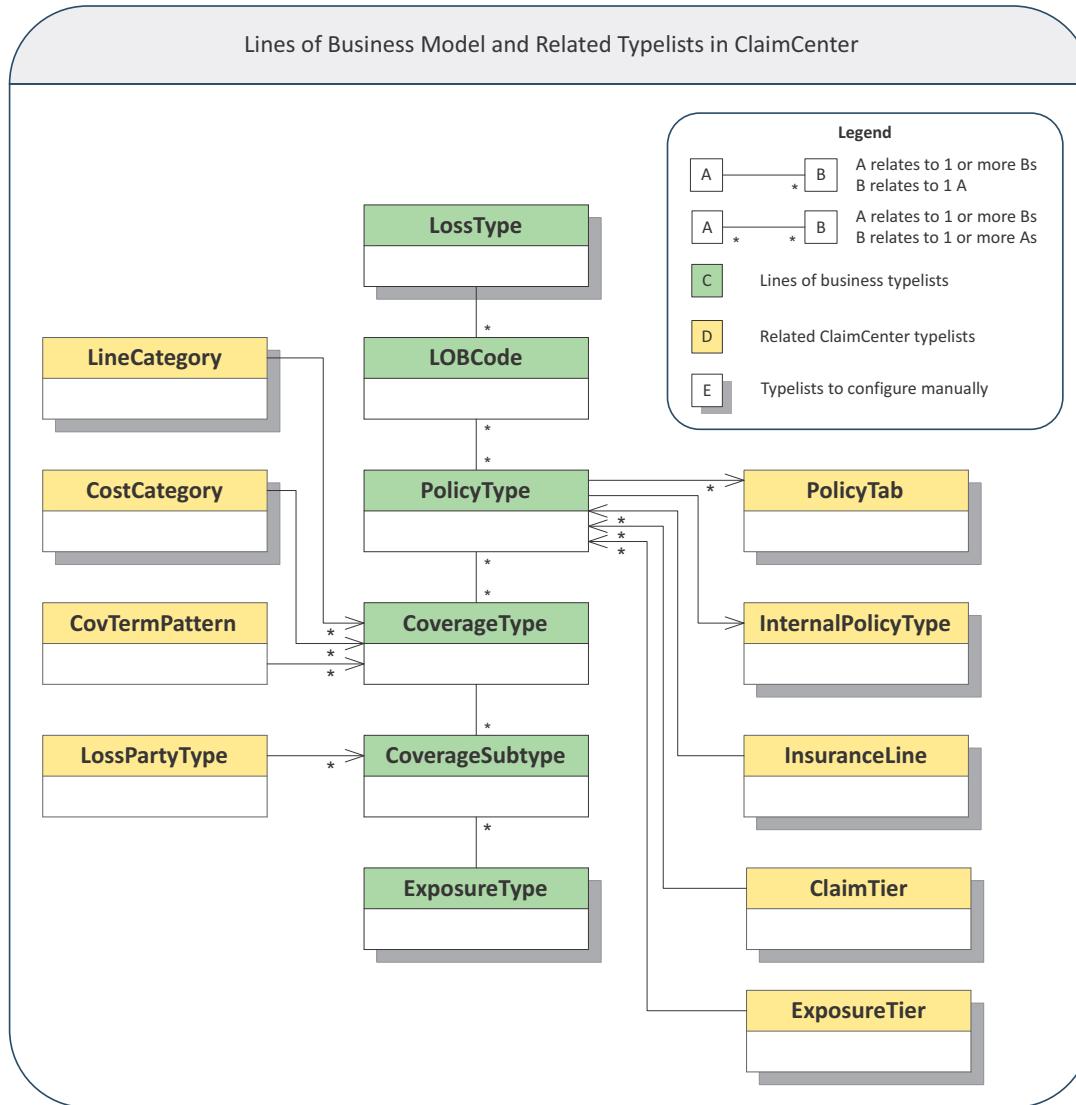
3. Check the output directory for generated files and the generation report *productModelGenReport.txt*.

Use the generation report to:

- Determine success or failure of the generation command
- If the command succeeded, identify new coverage types to map to exposure types in ClaimCenter Studio

## Using Generated Typelists in ClaimCenter

After you run the ClaimCenter Typelist Generator, you must perform additional steps to fully merge changes from PolicyCenter with your ClaimCenter lines of business model. The following diagram highlights the typelists that you may need to configure in ClaimCenter Studio.



### Suggested steps

1. Make sure to copy generated files to ClaimCenter directories.
2. Use the generation report `productModelGenReport.txt` to determine what coverage types and LOB codes that the generator added.
3. Use a difference-detection (`diff`) tool on your input and output typelist files to determine other typecodes that the generator changed, such as policy types.
4. In ClaimCenter Studio, link generated lines of business typecodes to the top and bottom of the lines of business model:
  - a. For new LOBCode typecodes, add appropriate `LossType` typecodes as parents.
  - b. For new `CoverageSubtype` typecodes, add appropriate `ExposureType` typecodes as children.

For more information, see “Linking New Coverage Types to Exposure Types” on page 532.

5. In ClaimCenter Studio, link generated lines of business codes to related ClaimCenter typelists:
  - a. For new PolicyType typecodes, add PolicyTab typecodes as categories.
  - b. For new PolicyType typecodes, add one InternalPolicyType typecode, commercial or personal, as a category.
  - c. For new PolicyType typecodes, add them as categories to appropriate InsuranceLine, ClaimTier, and ExposureTier typecodes.
  - d. For new CoverageType typecodes, add them as categories to appropriate LineCategory and CostCategory typecodes.
- Note that the generator writes an updated CovTermPattern.ttx file, so you do not need to change CovTermPattern in ClaimCenter Studio.
6. If you add or remove CoverageSubtype typecodes, run the generator again with your latest ClaimCenter typelist files. The generator adjusts LossPartyType for you.
7. Add references to new codes in ClaimCenter Gosu classes and other configuration files that relate to coverages types and policy types.
8. Search ClaimCenter for references to lines of business codes that the generator removed, such as references in rules. Fix obsolete references by deleting them or changing them to use other codes.

### [Copying Generated Files to ClaimCenter](#)

After you run the ClaimCenter Typelist Generator, copy the generated files to ClaimCenter. Skip this step if you configured the generator to use the ClaimCenter directory for lines of business typelist files as its input and output directory.

#### **To copy generated files to ClaimCenter**

1. Copy generated typelist files (.ttx) to ClaimCenter/modules/configuration/config/extensions.
2. Copy generated typelist localization files (.properties) to corresponding locale directories in ClaimCenter/modules/configuration/config/locale.

For more information, see “Typelist Localization” on page 534.

---

**WARNING** Do not copy generated typelist files (.ttx) or typelist localization files (.properties) to any directory within the ClaimCenter base configuration directory, ClaimCenter/modules/cc. Instead, use the custom configuration directory, ClaimCenter/modules/configuration/config/extensions. Otherwise, ClaimCenter does not start.

---

### [Using the Generation Report to Identify Added Coverages and LOB Codes](#)

The ClaimCenter Typelist Generator writes a generation report, productModelGenReport.txt, in the output directory. The report includes lines for:

- Coverage subtypes that the generator added to CoverageType that it did not link to exposure types in ExposureType.
- LOB codes that the generator added to LOBType that it did not link to loss types in LossType.

The following example shows lines from the generation report.

```
...
Warning: LOB Code [BOPLine] is not mapped to any loss types.
Warning: LOB Code [GLLine] is not mapped to any loss types.
Warning: LOB Code [BusinessAutoLine] is not mapped to any loss types.

...
Warning: Coverage subtype [BOPBuildingCov] is not mapped to any exposure types.
```

Warning: Coverage subtype [BOPOrdinanceCov] is not mapped to any exposure types.  
 Warning: Coverage subtype [BOPPersonalPropCov] is not mapped to any exposure types.  
 ...

Values in square brackets are typecodes that were not linked. In ClaimCenter Studio, you must link reported LOB codes to loss types, and you must link reported coverage subtypes to exposure types.

If you choose to link new coverages to the General Damage exposure type when running the generator, the generation report does not include lines for new coverage types. The generator linked them all to typecode GeneralDamage in ExposureType. In this case, use a diff tool and compare input and output versions of CoverageSubtype.ttx to identify new coverage types that you must link to exposure types.

## Linking New Coverage Types to Exposure Types

The ClaimCenter Typelist Generator adds generic typecodes in CoverageSubtype.ttx for new coverage types. The generic coverage subtypes have the same codes, names, and descriptions as the corresponding coverages.

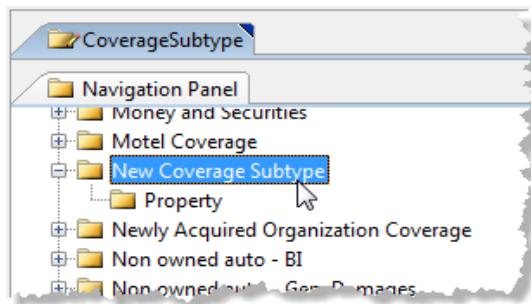
Generally, you want to link a coverage type to a single exposure type. Sometimes, you want to link a coverage to several exposure types. For example, you want liability coverage on a claim to allow exposures for injured people and damaged property.

### To link a new coverage to a single exposure type

1. In ClaimCenter Studio, the Resources pane, under the Lines of Business node, select **CoverageSubtype**.
2. In the Navigation Panel for **CoverageSubtype**, select the new coverage subtype.
3. If the generator linked new coverages to the General Damage exposure type, on the **Children (ExposureType)** tab, select the row for **GeneralDamage** and click **Remove**.
4. On the **Children (ExposureType)** tab, click **Add**.
5. In the **Code** column, press **CTRL+SPACE** and select an exposure type from the list.

### Result

The new coverage is linked to the exposure type through its generic coverage subtype. In the following example, **New Coverage Subtype** links to the **ClaimCenter Property** exposure type.



### To link a new coverage to several exposure types

1. In the Resources pane, under the Lines of Business node, select **CoverageType**.
2. In the Navigation Panel for **CoverageType**, find the new coverage type and click the plus sign (+) next to it. The panel displays the generic coverage subtype underneath the new coverage type.
3. Right-click the generic coverage subtype, and then click **Delete Typecode**.
4. For each ClaimCenter exposure type you want to link to the new coverage type:
  - a. In the Navigation Panel for **CoverageType**, select the new coverage type.
  - b. On the **Children (CoverageSubtype)** tab, click **Add New...**

c. In the New CoverageSubtype dialog, specify the following:

Field	Action
Code	Enter a lowercase code that combines the code for the new coverage type with the code for the exposure type you want to link. Abbreviate the two parts of the code if needed. For example, enter nwcvrge-pptydmg for a coverage subtype that links NewCoverageType to PropertyDamage.
Name	Enter a name that combines the name of the new coverage type with the name of the exposure type you want to link. For example, enter New Coverage Type - Property.
Description	Generally, enter the same value that you enter for Name.
Priority	Enter -1.

d. Click OK.

The Children (CoverageSubtype) tab shows the coverage subtype you added.

e. In the Navigation Panel for CoverageType, select your new coverage subtype.

You may need to click the plus sign (+) next to the new coverage type to see your new coverage subtype.

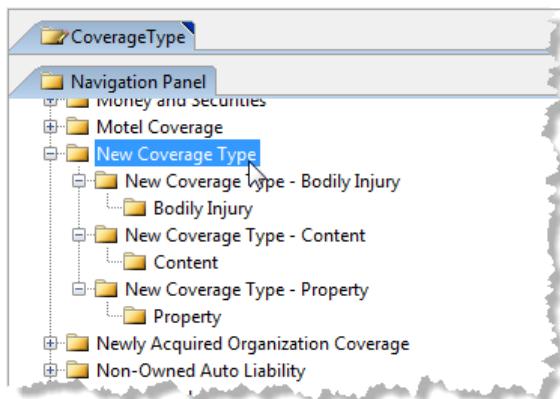
f. In the Children (Exposure) tab, click Add Existing

g. Press CTRL+SPACE, and then select an exposure type from the list.

h. If you want to link another exposure type to the new coverage type, repeat this procedure beginning at step a.

## Result

The new coverage is linked to several exposure types through new coverage subtypes that you added. In the following example, New Coverage Type links to the Bodily Injury, Content, and Property exposure types.



## Adding References to New Codes in Gosu Classes and Other Configuration Files

For ClaimCenter to take full advantage of new codes from PolicyCenter, you must add references to new codes in page configurations (.pcf), Gosu classes (.gs), and other configuration files.

### Notes on what to change for new coverages

- ClaimCenter accepts data in ACORD format and maps the data to new claims. Review the following files for two places to add ACORD mappings for new coverage types:
  - `Classes.gw.fnolmapper.acord.impl.AcordExposureMapper.gs`
  - `OtherResources.datatypes.acord.fnolmapper.typecodemapping.xml`
- To support ISO Claim Search for new coverage types, add entries to:
  - `OtherResources.iso.ISOCoverageCodeMap.csv`

- ClaimCenter has a number of methods that categorize PIP coverages. These methods govern the display of benefits tabs in ClaimCenter. For new or removed PIP coverage types, review the following Gosu class:
  - `Classes.libraries.PolicyUI.gsx`

#### Notes on what to change for new policy types

- Review the programming logic for exposure tier and claim tier mapping. You may need to change the logic in the following Gosu classes for new policy types:
  - `Classes.gw.entity.GWClaimTierEnhancement.gsx`
  - `Classes.gw.entity.GWExposureTierEnhancement.gsx`
- Review the programming logic for setting initial value on new claims. ClaimCenter generally sets the LOB code on new claims based on the loss type for the claim. You may need to change the logic in the following Gosu function:
  - `Classes.libraries.ClaimUI.setInitialValues()`
- To enforce aggregate limits and policy periods for new policy types, review the settings in the following files:
  - `OtherResources.xsds.aggregateLimitUsed-config.xml`
  - `OtherResources.xsds.policyPeriod-config.xml`

## TypeList Localization

If you configure your PolicyCenter and ClaimCenter instances with multiple locales, the ClaimCenter TypeList Generator helps you with typelist localization. The generator produces an updated typelist localization file for each locale. A *typelist localization file* contains translated typecode names and descriptions for a specific locale for all typelists in a Guidewire instance. To learn how to set up locales, see “Working with Regional Formats” on page 67 in the *Globalization Guide*.

When you run the generator, it produces updated ClaimCenter localization files with names and descriptions for new typecodes that come from PolicyCenter. The generator preserves names and descriptions for typecodes that do not originate in PolicyCenter.

The generator does not remove names and descriptions for typecodes that originated in PolicyCenter and now are deleted. This means that localized strings for typecodes that are removed from PolicyCenter remain in the ClaimCenter typelist localization files. The generator cannot distinguish localization properties that refer to deleted PolicyCenter typecode and properties for typecodes that originate from elsewhere.

#### To use the ClaimCenter TypeList Generator to produce updated typelist localization files

1. In the input directory, create a locale directory for each locale in your Guidewire instances.

For example, your instances have three locales, US English, Canadian English, and Canadian French. Create the following directories in the input directory:

- `INPUT_DIRECTORY/en_CA/`
- `INPUT_DIRECTORY/en_US/`
- `INPUT_DIRECTORY/fr_CA/`

2. Copy files named `typelist.properties` from each ClaimCenter locale directory to corresponding input local directories. ClaimCenter locale directories are located in:

`ClaimCenter/modules/cc/config/locale/`

3. Run the generator.

In the output directory, the generator creates localization folders for each locale and writes an updated `typelist.properties` files in each of them. For example:

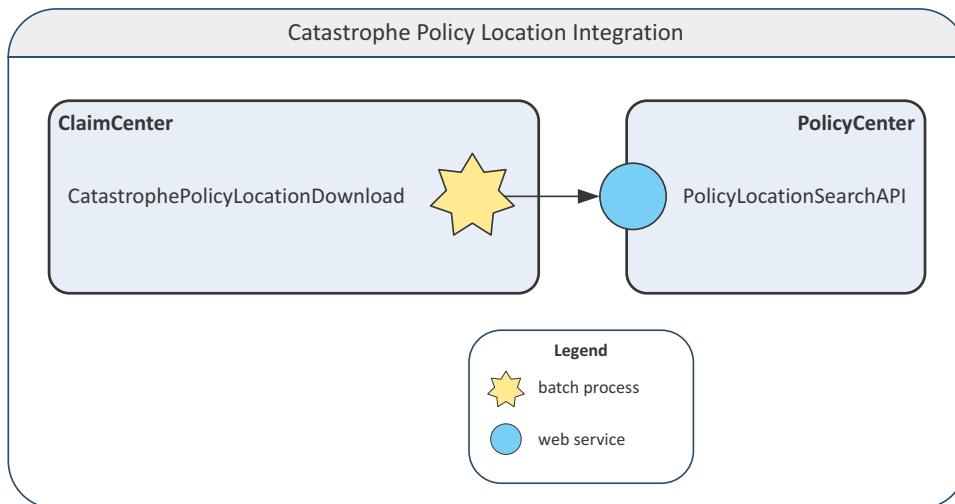
- `OUTPUT_DIRECTORY/en_CA/typelist.properties`
- `OUTPUT_DIRECTORY/en_US/typelist.properties`
- `OUTPUT_DIRECTORY/fr_CA/typelist.properties`

4. Copy the typelist localization files from their output directories to the corresponding ClaimCenter locale directories.

**IMPORTANT** The ClaimCenter Typelist Generator always produces a `typelist.properties` file for the default locale in your PolicyCenter instance, regardless whether you configure the instance for multiple locales. If your ClaimCenter instance has a single locale, then ignore the `typelist.properties` file that the generator produces.

## Catastrophe Policy Location Download

ClaimCenter provides the Catastrophe Policy Location Download batch process to retrieve policy locations from PolicyCenter or other, third-party policy administration systems. The batch process requests policy locations that lie within a geographic area of interest that ClaimCenter administrators set for a designated catastrophe.



If you integrate PolicyCenter with ClaimCenter, the default implementation of the `CatastrophePolicyLocationDownload` batch process calls the `PolicyLocationSearchAPI` web service that PolicyCenter publishes. The batch process is written entirely in Gosu, so you can modify the default implementation to download policy locations from third-party policy administration systems.

This topic includes:

- “Catastrophe Policy Location Overview” on page 535
- “Catastrophe Areas of Interest Data Model” on page 536
- “Catastrophe Policy Location Download Processing” on page 538

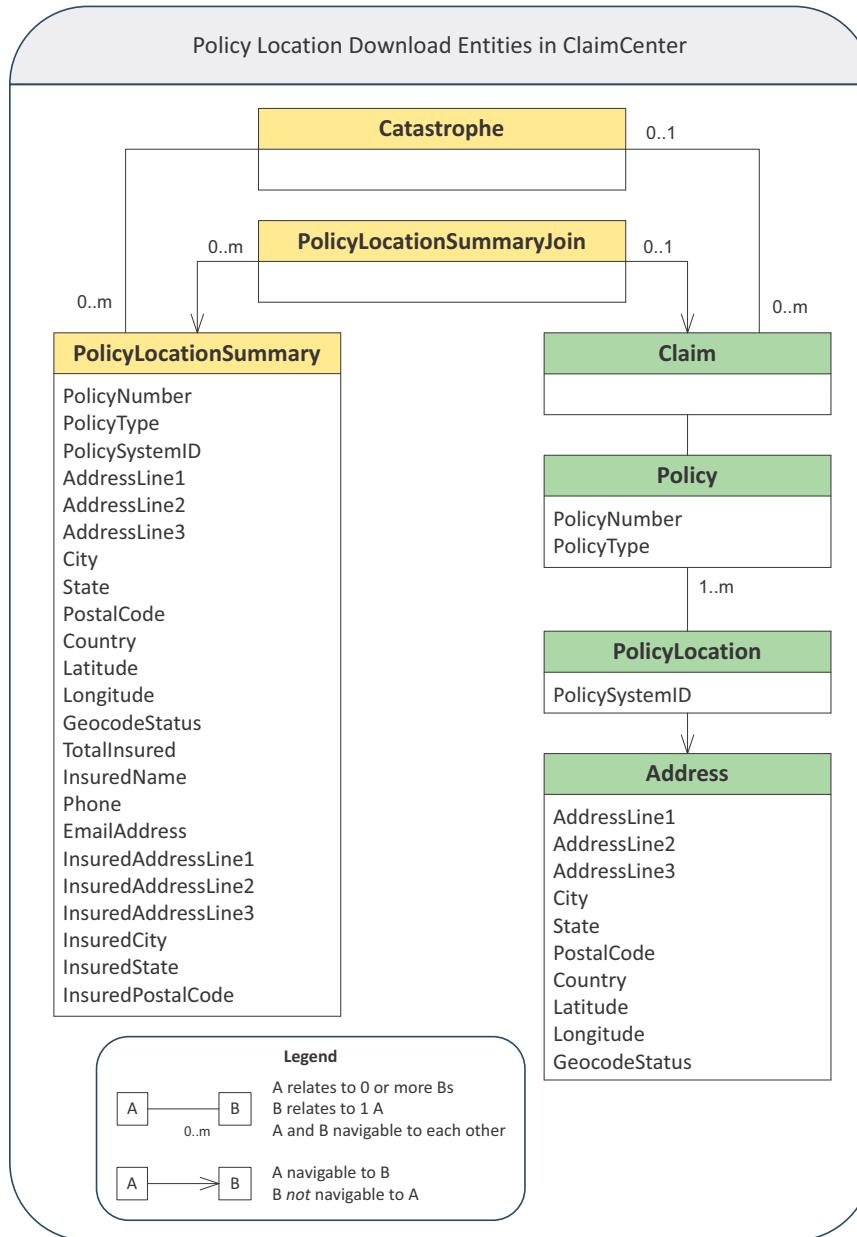
### Catastrophe Policy Location Overview

In ClaimCenter, administrators define catastrophes. To specify the area of interest for a catastrophe, an administrator clicks **Edit**, drags a bounding box on the catastrophe map, and then clicks **Set Catastrophe Area of Interest**. The next time that the batch process runs, ClaimCenter downloads policy locations from a policy administration system based on the area of interest and the effective date of the catastrophe. After the batch process downloads a fresh set of policy locations for the catastrophe, users who view the catastrophe map can see them plotted on the catastrophe map.

For more information, see “Catastrophes and Disasters” on page 155 in the *Application Guide*.

## Catastrophe Areas of Interest Data Model

The CatastrophePolicyLocationDownload batch process uses several entities in the ClaimCenter data model.



The batch process creates a **PolicyLocationSummaryJoin** to associate a **PolicyLocationSummary** and a **Claim** if all the following conditions are true:

- The **Claim.LossLocation.Address** exactly matches the address fields on a **PolicyLocation.Address** in the **Claim.Policy.PolicyLocations** array. The geocode fields **Latitude** and **Longitude** are not used for matching.
- The **PolicyLocationSummary.PolicyNumber** equals the **Claim.Policy.PolicyNumber**.
- The **PolicyLocationSummary.PolicySystemID** equals the **PolicyLocation.PolicySystemID** of the matching **PolicyLocation** in the **Claim.Policy.PolicyLocations** array.

The default Gosu implementation of the batch process does not verify that the address fields on the **PolicyLocationSummary** exactly match the address fields on the matching **PolicyLocation.Address** in the

`Claim.Policy.PolicyLocations` array. You can modify the Gosu implementation to add this additional consistency check.

## The Policy Location Summary Entity

The catastrophe heat map in ClaimCenter plots policy locations based on `PolicyLocationSummary` instances. A `PolicyLocationSummary` instance contains information about a policy location, regardless of any claim filed against it in ClaimCenter. The catastrophe heat map does not use the `Claim.Policy.PolicyLocations` array to plot policy locations.

### Creation of Policy Location Summaries

The batch process `CatastrophePolicyLocationDownload` calls the `PolicyLocationSearchAPI` web service to download policy location information from PolicyCenter. The batch process uses the downloaded information to create `PolicyLocationSummary` instances, as well as to create `PolicyLocationSummaryJoin` instances when it finds matching `Claim` instances. Besides the batch process, ClaimCenter creates and maintains `PolicyLocationSummaryJoin` instances in response to updated `Claim` instances that match a `PolicyLocationSummary` instance.

### Properties of Policy Location Summaries

The important properties of the `PolicyLocationSummary` are:

- `PolicyNumber` – Policy number from the external policy administration system, such as PolicyCenter
- `PolicySystemID` – Identifier of the external policy administration system, such as PolicyCenter
- Policy location address – Names of location address properties are the same as on the `Address` entity
- `Latitude`, `Longitude`, and `GeocodeStatus` – Geocode attributes for the policy location address
- Insured's contact information:
  - `InsuredName`
  - `Phone`
  - `EmailAddress`
  - `Insured*` – Names of the insured's address fields begin with `Insured`.
- `TotalInsured` – Total insured value for the policy location, as a `currencyamount`

## The Policy Location Summary Join Entity

A `PolicyLocationSummaryJoin` instance associates a `PolicyLocationSummary` with a matching catastrophe `Claim`. Usually, significantly more policy location summaries exist than matching catastrophe claims.

### Properties of Policy Location Summary Joins

The `PolicyLocationSummaryJoin` entity associates `Claim` and a `PolicyLocationsSummary`. The entity has two properties, both required:

- `PolicyLocationSummary` – a foreign key to a `PolicyLocationSummary`
- `Claim` – a foreign key to a matching catastrophe `Claim`

These two foreign keys serve the only purpose of a `PolicyLocationSummaryJoin` instance, associating a `PolicyLocationSummary` with a matching catastrophe `Claim`.

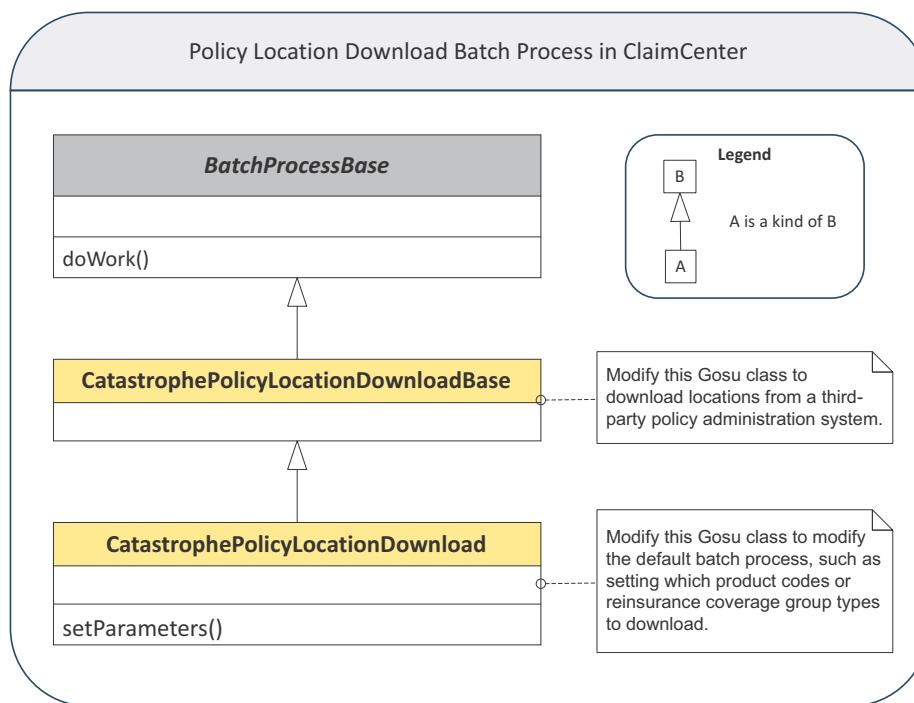
## Catastrophe Policy Location Download Processing

When it runs, the Catastrophe Policy Location Download batch process looks for catastrophes where the last modified date is later than the date of the last download of policy locations. For each catastrophe that qualifies, the batch process updates the policy locations in the modified area in two phases:

- 1. Location Query Phase** – Queries PolicyCenter or a third-party policy administration system for locations within the area of interest through the `PolicyLocationsSearchAPI` web service that PolicyCenter publishes
- 2. Claim Matching Phase** – Matches policy locations downloaded from the policy administration system with policies on claims that already are in ClaimCenter

### Catastrophe Policy Download Batch Process Implementation Classes

Like all batch processes, `CatastrophePolicyLocationDownload` extends the `BatchProcessBase` abstract class. It does so indirectly by extending `CatastrophePolicyLocationDownloadBase`, which in turn extends `BatchProcessBase` directly.



Generally, you modify the `CatastrophePolicyLocationDownload` class to modify the default batch process. For example, modify the `setParameters` method to set the arrays or product codes or reinsurance coverage group types to download. The default product codes are `Homeowners` and `CommercialProperty`. The default reinsurance coverage group type is `Property`. PolicyCenter defines its full set of reinsurance coverage group types with the `RICoverageGroupType` typelist.

To download locations from a third-party policy administration system, modify the `CatastrophePolicyLocationDownloadBase` class.

### Catastrophe Policy Download Batch Process Error Handling

The default implementation of the batch process attempts to connect with PolicyCenter through its `PolicyLocationsSearchAPI` web service. If the web service is unavailable when the batch process runs, it fails. However, the batch process retries to connect to the web service the next time the batch process runs.

## Policy Location Search Plugin

To perform a policy location search, ClaimCenter calls methods of the `PolicyLocationSearchPlugin` plugin interface. Each return result has properties to represent the policy number, the address, the latitude, the longitude, product code, and insured value.

ClaimCenter include a default implementation of this plugin interface that makes a web service call to PolicyCenter to search policy locations. The default implementation is:

```
gw.plugin.pcintegration.pcVERSION.location.PolicyLocationSearchPluginImpl
```

The default implementation uses this related class for encapsulating the return results:

```
gw.plugin.pcintegration.pcVERSION.location.CCPolicyLocationInfoImpl
```

If you use PolicyCenter, you may want to edit those files to make your own modifications, or base a new version of the plugin on this version.

If you use a different policy system, write your own implementation of this plugin. Next, update the plugin registry in Studio to use your implementation instead of the default plugin implementation.

There is only one method in this interface:

```
CCTPolicyLocationInfo[] findPolicyLocationsWithinBoundingBox(PolicyLocationSearchCriteria criteria)
```

This method finds policy locations within a bounding box, as well as other search criteria defined by `PolicyLocationSearchCriteria`. Properties include:

- `EffDate` – effective date, as `Date`
- `ProductCodes` – product codes, as an array of `String` objects
- `TopLeftLat`, `TopLeftLong`, `BottomRightLat`, `BottomRightLong` – bounding box in latitude and longitude
- `Handle` – The handle, which is used for returning multiple requests of results for the same query, as a `String`
- `StartingOffset` – the starting offset in the results, for large result sets, as an `int`
- `Count` – the number of items to return in this search

In your version of the `findPolicyLocationsWithinBoundingBox` method, use the properties in the `PolicyLocationSearchCriteria` to initiate a query in the external policy system.

To support large result sets and paged results, the plugin supports multiple queries to the same result set using a unique value called the *handle*. The handle is in the search criteria object as the `Handle` property. The handle identifies a single persistent query. The `StartingOffset` number in the search criteria is the index within the result set to start returning more results. ClaimCenter increases the value of `StartingOffset` on each request to get the next batch of results.

In the default implementation, PolicyCenter retrieves all results in memory on the first request with that handle value. PolicyCenter caches the results and reuses them for additional requests with the same handle value.

If you write your own implementation of this plugin, follow a similar pattern. Cache search results and store the handle value on external system so subsequent search queries with the same handle return results quickly.

Return the results in an array of `gw.plugin.policy.location.CCPolicyLocationInfo` objects. The `CCTPolicyLocationInfo` interface defines the set of properties on the object. In Studio, you can view the interface directly. Alternatively, in Studio, look at the `CCTPolicyLocationInfoImpl` implementation class mentioned earlier for how to implement that class in Gosu and to review the list of properties on the results.

## Policy Refresh Overview

Each claim includes a copy of the associated policy that was in force on the date of loss. This policy was retrieved from the external policy administration system. At a later time, you can update the policy from the external policy administration system. For example, if you change the date of loss, there may be different policy

coverages in force on that updated date. At the end of policy refresh, the older copy of the policy is deleted entirely. This entire process is called *policy refresh*. The copy of the policy in ClaimCenter is called a *policy snapshot* and often is a subset of the policy information in the policy administration system.

#### See also

For information on policy refresh from a feature and user interface point of view, see “Refreshing the Policy Snapshot on a Claim” on page 94 in the *Application Guide*.

## Preserving Relationships Between Claims and Policies with Policy Refresh

The technical challenge of policy refresh logic is to preserve the relationship between the claim and the policy. Think of the policy as a subgraph of the claim graph. Various objects within the claim graph might have links directly to objects within the policy graph. For example, an exposure object in the claim has a foreign key to a coverage in the policy. When you refresh a policy, ClaimCenter replaces policy-specific entities with new ones retrieved from the policy administration system. However, there may be important differences between the old copy of the policy and the newer copy of the policy. For example:

- There could be new objects, such as a new claim contact (`ClaimContact`).
- There could be changed objects, such as changed coverage details.
- There could be objects with no equivalent in the new policy, such as a coverage that no longer exists. If this happens, ClaimCenter deletes the object by the end of policy refresh, assuming there are no errors that would prevent it.

At a technical level, the refresh process includes the following high-level steps:

1. ClaimCenter determines the set of entities in the policy graph. For more details of how ClaimCenter determines which entities are in the policy graph, see “Determining the Extent of the Policy Graph” on page 546.
2. ClaimCenter compares the original copy of the policy (the current policy) with the new policy and determines which current policy objects correspond to which objects in the new policy. The comparison happens using a set of classes called entity matchers. An *entity matcher* knows how to compare two policy objects of the same entity type to determine whether they logically represent the same thing. For example, if there is only one contact in both the current policy and the new policy, are they the same person or an entirely different person? For more information about matchers, see “Matching Overview” on page 541 and “Policy Refresh Entity Matcher Details” on page 548.

**WARNING** Be sure you understand matching before modifying or writing any matching code. If you do not match and relink objects successfully, you could corrupt the claim/policy graph.

3. ClaimCenter displays important differences between the current policy and the new policy. If you confirm the policy refresh, you know which policy objects ClaimCenter will add, update, or delete. You can also view any warnings or errors for policy refresh. An example of a warning is that the effective dates on a coverage changed. An example of an error is that there exists a payment on a coverage that no longer exists. In contrast to warnings, any error blocks policy refresh from completing. If there are no errors, you can confirm the policy refresh. You can configure warnings and errors in Gosu as part of the difference display configuration. For more details, see “Policy Refresh Errors, Warnings, and Informational Messages” on page 559.
4. ClaimCenter proceeds to replace the policy. A critical part of this is relinking objects between the claim and the policy. *Relinking* requires fixing foreign keys from a claim object to a policy object, or from a policy object to a claim object. It is critical that ClaimCenter fixes any links that are potentially broken when replacing the current policy graph with a new policy graph. For example, an exposure with a foreign key to a policy coverage must link to the newer copy of the policy coverage. ClaimCenter encapsulates critical parts of relinking logic in objects called relink handlers. A *relink handler* defines the behavior when a foreign key between a policy object and a claim object might break during policy refresh. The default behavior depends on direction of the broken link, and whether the object exists on the new policy. For details, see “Policy

Refresh Relinking Details” on page 552.

**WARNING** Be sure you understand relinking before modifying or writing any relinking code. If you do not match and re-link objects successfully, you could corrupt the claim/policy graph.

5. ClaimCenter purges the old policy objects from the database. ClaimCenter deletes the database rows, rather than retiring them. Even if the object did not change between the old and new policy, the old object is deleted because the new object completely replaces the old object. Because relinking already finished, any foreign keys to the old object already link to the new copy of the object.

**Note:** You can configure preservation behavior for entities that normally policy refresh removes. The built-in configuration does this for contacts, and for addresses that are the claim’s loss location. For details, see “Policy Refresh Relinking Details” on page 552.

6. Finally, all cumulative changes during policy refresh commit to the database in one database transaction.

## Matching Overview

For some objects matching is easy if the object has another property that is both uniquely identifying and immutable (unchangeable). For example, a contact has a Address Book Unique ID (ABUID) property. If the ABUID values match, it is a definitive match.

For other entities, this is more complex and requires testing multiple properties. For example, a vehicle’s matcher can match the vehicle identification number (VIN) if it is present. Otherwise, the matcher compares other properties such as the license plate number, the state, or the vehicle serial number.

A small number of entity types match based on matching related objects, such as parent objects. For example, the default behavior is to match two CovTerm objects based on having matching parent Coverage objects.

For some but not all policy objects, there is a field containing a policy object Policy System ID (PSID) in the PolicySystemId property. The PSID is a unique ID that your external policy administration system specifies as uniquely representing the object in the policy administration system. If two objects have the same entity type and a matching PSID, it is a definitive match. The policy administration system must ensure that the PSID is both unique and immutable.

**Note:** Some entities do not have the PolicySystemId property. Usually those entities already have another uniquely identifying immutable property, for example contacts have the ABUID in its AddressBookUID property. However, some entities might have multiple types of unique immutable IDs.

If you add new entity types that add identifying fields to the subtype, write a new matching class for each new entity type. You must do this even if you use a PSID to match that entity type. However, if you add new entity types that do not add identifying fields to the subtype, you do not need to write a new matching class for the subtype.

### See also

For more information about entity matchers, see “Policy Refresh Entity Matcher Details” on page 548.

## Differences Between Policy Refresh and Policy Select

If you select an entirely different policy from the user interface (the process called *policy select*), a similar policy refresh happens.

There are differences between a policy refresh and a policy select:

- With policy refresh, the policy represents the same policy though it may have changed since last downloaded.
- With policy select, the entire policy including the policy number and the entire graph might be completely different. Although both policy refresh and policy both set a flag that indicates the policy is verified, policy

select is the recommended approach for verifying a policy. For example, the old policy may have been entered manually with incomplete or incorrect information, perhaps because the external policy administration system was offline when originally entered.

However, from a technical standpoint for implementing policy refresh integration, the implementation is the same. In both cases, ClaimCenter calls the policy refresh plugin and the behavior of the plugin typically would be basically the same.

## How Policy Refresh Handles Special Situations

The following list describes the policy refresh behavior in the ClaimCenter base configuration:

Retrieved policy	During policy refresh...
Contact no longer present on the policy as insured, and is added as an excluded party	The insured becomes the former insured. ClaimCenter adds the contact as an excluded party
Coverage added	ClaimCenter shows the refreshed policy with the coverage.
Coverage incident or exposure limits changed (or the limit currency changed)	ClaimCenter updates the coverage limits on the policy. ClaimCenter provides a warning if the limit decreases.
Effective date or the expiration date changed on the coverage	ClaimCenter provides a warning.
PIP aggregate limits lowered	ClaimCenter provides a warning.
Policy currency changed	<ul style="list-style-type: none"> <li>• If the currency is editable then ClaimCenter provides a warning.</li> <li>• If the currency is read-only then ClaimCenter blocks the refresh with an error.</li> </ul>
Policy period changed	ClaimCenter provides a warning.
Risk unit added to the policy (a vehicle, for example)	ClaimCenter adds the risk unit to the claim. It is possible to modify an incident to use the new risk unit (the vehicle).
Risk unit coverage removed that is used by an exposure	<ul style="list-style-type: none"> <li>• If the exposure is still open, ClaimCenter blocks the refresh. The user must close the exposure first to continue.</li> <li>• If there are non-reserving transactions on the exposure, or the net incurred is greater than 0, then ClaimCenter blocks the refresh.</li> <li>• If the exposure is closed, meaning that there are no transactions except reserves, and the net incurred is zero, ClaimCenter allows the refresh.</li> </ul>
Spelling of an insured name changed	<p>The exact action is dependent on the matching logic:</p> <ul style="list-style-type: none"> <li>• If the contact is uniquely identified, ClaimCenter treats the change as a name change.</li> <li>• If no unique identification is present, however, then the default behavior in the base configuration is to use the name as an identifier (using fall-back matching criteria). ClaimCenter considers this change to be the addition of a new contact.</li> </ul>
Workers' comp class code removed	ClaimCenter blocks the refresh with an error.

## Policy Refresh Plugins and Configuration Classes

ClaimCenter defines a policy refresh plugin interface called `IPolicyRefreshPlugin`. This plugin defines the interactions between ClaimCenter and the policy refresh logic.

If you use PolicyCenter, use the built-in plugin implementation called `PCPolicyRefreshPlugin`. This class implements default behavior for the PolicyCenter data model.

If you use a policy administration system other than PolicyCenter, write a new plugin implementation. Base your plugin implementation on the included plugin implementation class `BasicPolicyRefreshPlugin`, which is implemented in Gosu. This class extends the internal base class `PolicyRefreshPluginBase`, which implements core behaviors of policy refresh. This internal base class is implemented in Java and is not visible in Studio.

In the default configuration, each built-in plugin implementation class relies on a policy refresh configuration class to handle nearly all of the work. This approach encapsulates all code related to policy refresh in one place rather than implementing it directly in a plugin implementation class. All the built-in configuration are written in Gosu and are visible in Studio.

The default configuration defines a base interface called `PolicyRefreshConfiguration`. The `PolicyRefreshConfiguration` interface defines the basic contract between a policy refresh plugin and its policy refresh configuration object. There are several included implementations of this interface, as well as a subinterface, that correspond to the different plugin implementations. The following table summarizes the plugin implementation classes and their configuration classes.

Class	Purpose	Configuration implementation class that it uses
<code>BasicPolicyRefreshPlugin</code>	<p>Integrate policy refresh with a policy administration system other than PolicyCenter. This is also called the <i>basic example plugin</i>.</p> <p><b>IMPORTANT:</b> If you extend the PolicyCenter data model, you must modify the policy refresh plugin to identify any new entities.</p>	<p>Uses the configuration class <code>BasicPolicyRefreshConfiguration</code> in the same package. It extracts the set of policy-only entities from an existing claim-and-policy graph. The code walks foreign key/array references and determines which entities that are created in ClaimCenter through the <code>IPolicySearchAdapter</code> plugin. This defines the set of policy-specific entity instances. See “Policy Refresh Entity Matcher Details” on page 548 for related discussion.</p> <p>If you want to write your own configuration class to work with this plugin, base your class on a new implementation of <code>ExtendablePolicyRefreshConfiguration</code>, which extends its superinterface <code>PolicyRefreshConfiguration</code>. Override the <code>getPolicyOnly</code> method to specify the entities that are part of the policy graph.</p>
<code>PCPolicyRefreshPlugin</code>	<p>Integrate policy refresh with Guidewire PolicyCenter.</p> <p><b>IMPORTANT:</b> If you extend the PolicyCenter data model, you do not need to modify the policy refresh plugin for any new entities. Extension entities are automatically detected when ClaimCenter imports the web service that PolicyCenter publishes.</p>	<p>Uses the configuration class <code>PCPolicyRefreshConfiguration</code> in the same package. This configuration object finds entities to replace during policy refresh by examining XSD types in the web service definition to connect to PolicyCenter. See “Policy Refresh Entity Matcher Details” on page 548 for related discussion. This plugin implementation relies on the built-in policy search plugin implementation for PolicyCenter: <code>gw.plugin.pcintegration.pcVERSION_NUMBER.PolicySearchPCPlugin</code>.</p>
<code>PolicyRefreshPluginBase</code>	The internal base class for the other two policy refresh plugin implementations that this table mentions. Do not directly extend this class.	The plugin base class uses the configuration base class <code>PolicyRefreshConfigurationBase</code> . This base class is important because it implements the core functionality. Refer to it in Studio.

You could use one of several different strategies to write a custom policy refresh plugin:

- For a policy administration system other than PolicyCenter, base your implementation on the built-in implementation `BasicPolicyRefreshPlugin`. Write an entirely new implementation of the `ExtendablePolicyRefreshConfiguration` interface, which is defined in Gosu and visible in Studio. Override the `getPolicyOnly` method to specify the entities that are part of the policy graph. In the plugin, simply override the constructor to return your configuration object implementation.
- For expert integration programmers only, you could choose to re-implement the interface entirely with your a completely new design. This approach is not recommended. If you completely reworked the plugin implementation, you could consider not using any of the built-in configuration object interfaces or classes.

You might not even need to change the plugin and configuration objects in a significant way. Instead you might need only to modify the more easily-customized parts of the policy refresh system:

- **Entity matchers** – For more information, see “Policy Refresh Entity Matcher Details” on page 548.
- **Difference display tree user interface** – For more information, see “Policy Refresh Policy Comparison Display” on page 556. This is also the place to configure warnings and errors that appear during difference display.

### Policy Refresh Configuration Base Class

The `PolicyRefreshConfigurationBase` abstract class controls almost all ClaimCenter interface configuration for policy refresh, either directly or indirectly. This class consists entirely of property getters, most of which define mappings between various entities and the classes used to manage those entities. To override these values, override property getters in your own subclass of the `PolicyRefreshConfigurationBase` class.

The following list describes the most important property getters in the `PolicyRefreshConfigurationBase` class:

Property getter	Used to...	See...
<code>MatcherTypes</code>	Determine which entities in the existing and new policies are logically equivalent	“Policy Refresh Entity Matcher Details” on page 548
<code>DisplayTree</code>	Construct the policy comparison tree within ClaimCenter	“Policy Refresh Policy Comparison Display” on page 556
<code>DiffDisplayTypes</code>	Configure how ClaimCenter displays a specific entity type	“Policy Refresh Policy Comparison Display” on page 556
<code>CustomRelinkerTypes</code>	Determine how ClaimCenter relinks or fixes broken foreign key links that became broken as you refreshed a policy	“Policy Refresh Relinking Details” on page 552
<code>RelinkFilters</code>	Construct special logic that is outside the standard Policy Refresh behavior	“Policy Refresh Relinking Details” on page 552

## Policy Refresh Steps and Associated Implementation

The following table describes the steps in a standard policy refresh. The rightmost column describes the location of the implementation for this step as well notes about the behavior.

#	Step Summary	Description	Implementation
1	In the ClaimCenter policy summary screen, click Policy Refresh or Policy Select	To refresh the policy in ClaimCenter, click Policy Refresh. To verify the policy, click Policy Select.	See the ClaimCenter policy summary screen for PCF details.
2	Get new policy from external system	After the user selects to refresh the policy or select a new policy, ClaimCenter retrieves the new policy from the policy administration system.	ClaimCenter calls the policy search plugin ( <code>IPolicySearchAdapter</code> ) to find and retrieve the latest version policy using its <code>retrievePolicyFromPolicy</code> method.

#	Step Summary	Description	Implementation
3	Get policy-specific entities	ClaimCenter determines which entity instances references through the policy are policy-specific.	<p>This step happens in the policy refresh plugin method called compare. That method returns results encapsulated as a PolicyComparison object (that is an interface name).</p> <p>In the base configuration, the policy refresh plugin instantiates the built-in class PolicyComparer. The PolicyComparer class in its extractPolicyGraph method returns a set of policy objects. In the default implementation, this object calls the configuration object's getPolicyOnly method. The configuration object getPolicyOnly method is the method you would typically override to modify the built-in behavior.</p> <p>There are two approaches to generating the list of policy objects. See "Policy Refresh Entity Matcher Details" on page 548.</p>
4	Find which objects to remove, add, or update	ClaimCenter compares the local existing policy and the new policy from the policy search looking for entities that were removed, added, or updated.	<p>Sorts the entity instances from both policies into three sets:</p> <ul style="list-style-type: none"> <li>• Objects removed from the existing policy</li> <li>• Objects added on the new policy</li> <li>• Objects that are logically the same on the two policies. In ClaimCenter terminology, these objects matched.</li> </ul> <p>This step happens in the policy refresh plugin method called compare.</p> <p>In the base configuration, the matching process is coordinated by an implementation of the PolicyRefreshConfiguration interface. Such a class extends the PolicyRefreshConfigurationBase Gosu class. PolicyRefreshConfigurationBase contains a mapping in its MatcherTypes property. That property specifies the mapping between:</p> <ul style="list-style-type: none"> <li>• An entity type</li> <li>• Entity matcher class (EntityMatcher subclass) that knows how to compare two entity instances of that type.</li> </ul> <p>This mapping behavior is critical to how the comparison between the existing and new policies is performed. The entity matchers determine whether an entity is classified as added, removed, or matched in the policy comparison code.</p> <p>When you add new entities to the policy data model, typically you need to add a corresponding entity matcher class for that entity. If no EntityMatcher is defined for a given type, then ClaimCenter uses an EntityMatcher defined for the nearest supertype. If you add a new entity subtype to the data model that defines no new identifying fields, you do not need to create a new entity matcher for the subtype.</p>
5	Relinking	ClaimCenter finds foreign key links between the claim and policy that would be broken if the new policy replaced the existing policy.	<p>Foreign keys may link from non-policy entities to the policy entities or from policy entities to non-policy entities.</p> <p>This step happens in the policy refresh plugin method called compare.</p> <p>In the base configuration, the matching process is coordinated by the PolicyRefreshConfiguration implementation class. The policy configuration class knows how to create the relink handler objects that perform the relinking. See "Policy Refresh Relinking Details" on page 552 for more details.</p>

#	Step Summary	Description	Implementation
6	Select risk units	<b>Only for commercial auto and commercial property lines,</b> ClaimCenter displays a user interface for selecting risk units in the new policy. For example, you can select which insured vehicles or properties are relevant to this particular claim.	The user can check one or more risk items in a list for this commercial line of business. For example, for vehicles ClaimCenter displays one row for each vehicle with a checkbox for each item to select it. Each vehicle row includes the vehicle identification number (VIN), vehicle make, and vehicle model.
7	Display differences between old and new policy	ClaimCenter displays differences between the two policies to the user, with appropriate errors and warnings.	The DiffDisplay interface is the mechanism for configuring: <ul style="list-style-type: none"> <li>• How the application displays differences between objects in the user interface</li> <li>• How the application displays WARNING, ERROR, or INFO messages as a result of any differences.</li> </ul> Each entity can have a corresponding DiffDisplay object to define unique display, warnings, or errors.  If no difference item is present for some entity type, ClaimCenter uses the default behavior defined in EntityDiffDisplayBase (and no messages are generated).  See "Policy Refresh Policy Comparison Display" on page 556.
9	Relink the claim and the policy graphs	If the user approves the policy refresh, ClaimCenter attempts to relink broken links between Claim-related entities and Policy-related entities. In other words, determine the links between the claim and the old policy and reproduce those links between the claim and the new policy.	This step happens in the policy refresh plugin method called <code>relink</code> .  In the base configuration, the policy refresh plugin base plugin uses a class called <code>PolicyRelinker</code> . See "Policy Refresh Relinking Details" on page 552.
9	Remove the old policy graph	ClaimCenter deletes all objects on the old policy. Note that ClaimCenter deletes, not retires, the objects.	This step happens in the policy refresh plugin method called <code>removeOldPolicy</code> .
10	Commit all changes	ClaimCenter commits the bundle. This persists all related database changes in a single transaction.	This step happens in internal code.

## Determining the Extent of the Policy Graph

The policy refresh process knows what entities to replace because the objects in the current claim must either belong to the policy graph or not belong to the policy graph. The definition of the extent of the policy graph affects how ClaimCenter compares, relinks, and replaces the policy.

### The Policy Graph within a Claim

The formal definition of the policy graph is the set of all entities in the claim graph that the `IPolicySearchAdapter` retrieves from the external policy administration system. Assure the design of your data model graph cleanly separates the objects that belong to the policy graph from those that do not.

If an object in the claim graph is not part of the policy graph, ClaimCenter treats the object as part of the claim-specific part of the claim graph. Claim objects remain after a policy refresh, although some of their foreign keys that reference objects in the policy graph may change.

If an object in the claim graph is part of the policy graph, policy refresh replaces that object with a newer version, generally speaking. If an object used to exist but is absent on the new policy, for unusual cases you can optionally configure it to remain rather than deleting it.

#### See also

To learn how to retain policy-related objects that the policy administration system removed, see “Policy Refresh Relinking Details” on page 552.

## The Policy Refresh Plugin Determines the Objects in the Policy Graph

The policy refresh plugin determines the exact set of policy graph objects using one of the following strategies:

- **XSD/WSDL approach** – Programmatically examining the remote service that implements the policy retrieval plugin (`IPolicySearchAdapter`) and interpreting the types returned. For example, examine the XSD types in the WSDL that defines the web service connection to the external policy administration system.
- **Manual approach** – Writing code that traverse the claim and finds all known policy-related entity types in the graph and return the set of objects of those types.

In the base configuration, ClaimCenter provides examples of both approaches:

- The XSD/WSDL approach exists in the policy refresh plugin that is specific to PolicyCenter, `PCPolicyRefreshPlugin`. The policy graph can be automatically determined from the implementation of the `IPolicySearchAdapter`. The `PCPolicyRefreshPlugin` class examines XSD types from the WSDL file called `pcintegrationVERSION.wsdl`. This code is implemented in Java and is not visible in Studio.
- The manual approach exists in the `BasicPolicyRefreshPlugin` class, which calls out to the `BasicPolicyRefreshConfiguration` that uses code like the following:

```
/*
 * This method extracts all Policy-only entities from the existing Policy (which is linked
 * to a Claim and other non-Policy entities). Note that this implementation uses the getEntireArray()
 * method, which returns all members of the array including retired beans. Including retired beans is
 * necessary here because of issues purging policy entities if retired beans have a foreign key to
 * one of those entities.
 */
override function getPolicyOnly(existingPolicy : Policy) : Set<KeyableBean> {
    var policyOnly = new NonNullSet<KeyableBean>()
    policyOnly.add(existingPolicy)
    getEntireArray(existingPolicy, "Roles", ClaimContactRole).each(\ r -> {
        policyOnly.add(r)
        policyOnly.add(r.ClaimContact)
        includeContact(policyOnly, r.ClaimContact.Contact)
    })
    getEntireArray(existingPolicy, "ClassCodes", ClassCode).each(\ c -> policyOnly.add(c))
    getEntireArray(existingPolicy, "StatCodes", StatCode).each(\ s -> policyOnly.add(s))
    getEntireArray(existingPolicy, "Endorsements", Endorsement).each(\ e -> policyOnly.add(e))
    getEntireArray(existingPolicy, "RiskUnits", RiskUnit).each(\ ru -> includeRiskUnit(policyOnly, ru))
    getEntireArray(existingPolicy, "Coverages", Coverage).each(\ cov -> includeCoverage(policyOnly, cov))
    getEntireArray(existingPolicy, "PolicyLocations", PolicyLocation).each(\ p ->
        includeLocation(policyOnly, p))
    return policyOnly
}
```

Some of the utility functions that this code calls are necessary to query the database for results that include retired objects. It is necessary to find the retired instances in order to support proper purging of old policy objects.

#### See also

“Including Retired Entities in Query Results” on page 174 in the *Gosu Reference Guide*.

## Extending the Data Model for the Policy Graph

If your policy refresh plugin uses the manual approach you extend the policy graph in the policy administration system and in ClaimCenter, you must perform additional configuration steps. ClaimCenter calls the policy

refresh plugin for to determine the extent of the policy graph within the claim graph. You must assure your plugin gives the correct answer.

For example, if you base your implementation on `BasicPolicyRefreshPlugin`, you must modify the `getPolicyOnly` method. Modify the method so it includes any instances of your extended entity types that belong to the policy graph in the returned set. If you do not modify the method, ClaimCenter considers your new entity types part of the claim-specific part of the claim graph.

## Policy Refresh Entity Matcher Details

To replace the existing policy with the new policy and display any differences to the end user, ClaimCenter compares the two policies. This comparison determines:

- Which entities were removed from the current policy
- Which were added in the new policy
- Which are present in both and may be unmodified or updated. The ClaimCenter terminology for an object existing in both the old policy and the new policy is that the objects *match*.

In the default implementation of policy refresh, the entity matching happens in classes called *entity matcher* classes. Entity matcher classes determine which entities in the existing and new policies are logically equivalent. Entity matchers all implement the interface `gw.api.bean.compare.EntityMatcher`. This interface defines a single method: `doEntitiesMatch`, which takes two instances of the type. It returns `true` if and only if the two entities being compared are logically equivalent.

Use of entity matcher classes is the only way to determine whether two entities are equal for policy refresh. In the context of policy refresh, entity matching is not the same as the `Object.equals()` contract or the Gosu operators `==` nor `==>`. Generally speaking, two entities are logically equivalent if they are identical on the set of identifying properties, which is usually a subset of all the properties on an object.

Typically, identifying properties are properties on an object itself. Sometimes, an identifying property is a relation to another. For example, two coverage terms match if and only if they relate to a matching coverage.

Some policy objects contain a policy object Policy System ID (PSID) in the `PolicySystemId` property. The PSID is a unique ID that your external policy administration system specifies as uniquely representing the object in the policy administration system. For objects that have a PSID, populate the property so you can match object instances. Two policy objects of the same entity type with matching PSID values definitively match. The policy administration system is responsible for ensuring that the PSID is both unique and immutable.

---

**IMPORTANT** If you add new entity types, you must write a new matching class for each new entity type, even if you use a PSID to match that entity type.

---

If you write a new matcher implementation, you must provide a default (no-argument) constructor.

## Matching Related Objects

If an entity matcher must reference other related entities, it also implements the initializable matcher interface `gw.api.bean.compare.InitializableMatcher`. The entity matcher retrieves those entity instances through a matcher context object, which is an instance of `MatcherContext`.

If an entity matcher needs to reference other related entities:

- The entity matcher must also implement interface `gw.api.bean.compare.InitializableMatcher`.
- The entity matcher must retrieve the related entity instances through `MatcherContext`.

The `InitializableMatcher` interface defines the method:

```
public void init(MatcherContext context)
```

If a given matcher implements this interface, the matcher class can implement this method to get (and save in a instance variable) an instance of `gw.api.bean.compare.MatcherContext` to use during matching. Use this object to find an `EntityMatcher` for any given entity type when comparing entities related to the two entity instances passed into `gw.api.bean.compare.EntityMatcher.doEntitiesMatch(a, b)`. To make this easy to use, there is an abstract base class called `gw.api.bean.compare.InitializableMatcherBase` that defines the `doRelatedEntitiesMatch` method. This method compares two related objects and returns true if and only if they match. In other words, that method:

1. Finds matcher for the types passed in as arguments
2. Next, uses that matcher to match the entity by calling the matcher's `doEntitiesMatch` method.

## Registering Matcher Classes

Entity matchers are registered in a `PolicyRefreshConfiguration` implementation in its getter function for the `MatcherTypes` property. The `MatcherTypes` property returns a `java.util.Map` that maps entity types to their entity matcher classes. In the base configuration, the `PolicyRefreshConfigurationBase` class defines the mapping for all built-in existing matchers. For example, this defines the mapping from `Address` to `AddressMatcher`. ClaimCenter uses this map to match entities between the current policy and the new policy.

The `MatcherTypes` property getter in the configuration object looks like the following:

```
/*
 * Returns a map of type-->EntityMatcher type that is used to match entities between
 * the old and new policies.
 */
@Override
public Map<IEntityType, Class<EntityMatcher<KeyableBean>> getMatcherTypes() {
    return {
        Address -> AddressMatcher.class,
        Building -> BuildingMatcher.class,
        Contact -> ContactMatcher.class,
        ...
    };
}
```

In general, if you add a new entity to the policy data model—for any entity that `IPolicySearchAdapter` can retrieve—then you must also add a corresponding `entityMatcher` class for that entity. You must then map that entity in the `MatcherTypes` getter mapping logic.

If ClaimCenter can not find an `entityMatcher` class for a given type, then it uses the `entityMatcher` defined for the nearest supertype. Therefore, you do not need to define an `entityMatcher` class for an entity that you add as new entity subtype if all of the following are true:

- The new entity subtype does not define any identifying fields.
- An appropriate matcher class exists for a supertype of the entity.

---

**IMPORTANT** If you write new entity matcher classes, remember to register them in your `PolicyRefreshConfiguration` implementation in the getter function for the `MatcherTypes` property.

---

## Best Practices for Entity Matcher Classes

Guidewire recommends the following for writing matcher classes:

1. Search for a definitive match if you are writing a custom entity matcher.

Definitive matches are not always possible, however. If a definitive match is not possible through the `AddressBookUID`, `PolicySystemID`, or other identifier, then have your entity matcher class attempt to match on a fallback criteria. It is for this reason that Guidewire recommends that you always populate `PolicySystemID` for objects that have them to make definitive matching easier. As mentioned earlier, not all objects have that property, typically because there is another unique immutable field, such as the contact `AddressBookUID` property.

## 2. Match only on fields in an entity that uniquely identify instances of that entity.

You can also test associated (linked) entities in the case in which those identify the given entity. For example, the `PolicyLocation` associated with a `LocationBasedRU`. If this is not possible, you can write fallback logic that finds probable matches and prioritize the closest match. This is the task of a prioritizer class, which an entity matcher returns from its `getMatchPrioritizer` method. You can also use this class to sort by other criteria in the case in which multiple matches are found. Refer to the API Reference documentation for details of a prioritizer.

In the base configuration, ClaimCenter policy refresh uses its own internal algorithm to determine the closest match if it finds multiple matches. However, ClaimCenter uses the prioritizer if it can determine no other way to narrow down the closest match.

## Entity Matcher Classes in the Base Configuration

In the base configuration, the `ContactMatcher` class provides default matching for all `Contact` entities. It is overridden for both the `Person` and `Company` entities by the `PersonMatcher` and `CompanyMatcher` matcher implementations respectively. In the base configuration, the `ContactMatcher` class has logic similar to the following:

```
/*
 * Matches on Address Book UID, then TaxID, then display name.
 */
override function doEntitiesMatch(c1:Contact, c2:Contact) : boolean {

    if(areBothNotNull(c1.AddressBookUID, c2.AddressBookUID)) {
        return c1.AddressBookUID.equals(c2.AddressBookUID)
    }

    if (areBothNotNull(c1.PolicySystemId, c2.PolicySystemId)) {
        return c1.PolicySystemId == c2.PolicySystemId
    }

    if(not c1.Subtype.equals(c2.Subtype)) {
        return false
    }

    if(areBothNotNull(c1.TaxID, c2.TaxID)) {
        return c1.TaxID.equals(c2.TaxID)
    }

    // Attempt to match on display name if none of the above worked.
    return c1.DisplayName.equals(c2.DisplayName)
}
```

In this code:

- As `ContactManager` is the final authority for contact information, ClaimCenter considers property `AddressBookUID` a definitive match if the same value is present on both entities.
- Failing that, the `PolicySystemId` is a definitive match if present and both values match.
- In the same manner, ClaimCenter uses the same process for other identifiers such as the Tax ID and the DUNS ID.
- If none of the previous IDs match, the last test is a non-definitive one. ClaimCenter checks to see if the company name matches. This is the fallback logic. It is highly unlikely that two or more companies with the same name actually appear on the same policy. However, if this does happen, ClaimCenter will determine the closest match based on other criteria. For example, if the two different companies had two different primary addresses, the Policy Refresh logic would see this and match appropriately.

## Other Custom Matcher Logic in the Base Configuration

The following list describes other examples of matcher logic in the ClaimCenter base configuration.

---

Address	This matcher logic is similar to the CompanyMatcher described previously. The AddressMatcher logic attempts to definitively match the AddressBookUID if it is present on both entities. Failing this, it uses fall-back logic to find a probable match based on the AddressLines, City, County, State, Country, and PostalCode.
RUCoverage	The RUCoverageMatcher attempts to definitively match the PolicySystemId, then matches on the associated RiskUnit.
PolicyLocation	The PolicyLocationMatcher attempts to definitively match the PolicySystemId, then matches on the associated addresses (if both are non-null).

---

## Example of Changing Matching Criteria

You might need to change the matching criteria for some entities. This topic shows how you might change the matching criteria.

The default VehicleMatcher logic follows these steps:

- If the PolicySystemID values match, the Vehicle objects match.
- If the VIN numbers match, the Vehicle objects match.
- If the SerialNumber values match, the Vehicle objects match.
- If both the LicensePlate and the State (the registered state for the vehicle) match, the Vehicle objects match.
- If none of the previous fields match, the Vehicle objects do not match.

The code for this default logic is:

```
class VehicleMatcher extends MatcherBase<Vehicle> {

    /**
     * Match on identifying information, or a combination of unique
     * characteristics.
     */
    override function doEntitiesMatch(v1 : Vehicle, v2 : Vehicle) : boolean {

        // Do policy system IDs match?
        if (areBothNotNull(v1.PolicySystemId, v2.PolicySystemId)) {
            return v1.PolicySystemId == v2.PolicySystemId
        }

        // Do vehicle identification numbers match?
        if(areBothNotNull(v1.Vin, v2.Vin)) {
            return v1.Vin==v2.Vin
        }

        // Do serial numbers match?
        else if(areBothNotNull(v1.SerialNumber, v2.SerialNumber)) {
            return v1.SerialNumber==v2.SerialNumber
        }

        // Do license plates match?
        else if(areBothNotNull(v1.LicensePlate, v2.LicensePlate)
                and areBothNotNull(v1.State, v2.State)) {
            return v1.LicensePlate==v2.LicensePlate and v1.State==v2.State
        }

        return false // No identifying properties exist on both objects.
    }
}
```

To alter this behavior so that a match can also occur on the Make and Model properties of the Vehicle objects, add an additional branch to the if/else statement.

```
class VehicleMatcher extends MatcherBase<Vehicle> {
```

```

/*
 * Match on identifying information, or a combination of unique
 * characteristics.
 */
override function doEntitiesMatch(v1 : Vehicle, v2 : Vehicle) : boolean {

    // Do policy system IDs match?
    if (areBothNotNull(v1.PolicySystemId, v2.PolicySystemId)) {
        return v1.PolicySystemId == v2.PolicySystemId
    }

    // Do vehicle identification numbers match?
    if (areBothNotNull(v1.Vin, v2.Vin)) {
        return v1.Vin==v2.Vin
    }

    // Do serial numbers match?
    else if(areBothNotNull(v1.SerialNumber, v2.SerialNumber)) {
        return v1.SerialNumber==v2.SerialNumber
    }

    // Do license plates match?
    else if(areBothNotNull(v1.LicensePlate, v2.LicensePlate)
            and areBothNotNull(v1.State, v2.State)) {
        return v1.LicensePlate==v2.LicensePlate and v1.State==v2.State
    }
        // Added condition - Do vehicle make and model match?
    else if(areBothNotNull(v1.Make, v2.Make) and areBothNotNull(v1.Model, v2.Model)) {
        return v1.Make==v2.Make and v1.Model==v2.Model
    }

    return false // No identifying properties exist on both objects.
}

}

```

## Policy Refresh Relinking Details

Relinking is the process of repairing broken foreign keys from a claim entity to a policy entity or from a policy entity to a claim entity. It is important that ClaimCenter repairs any links that potentially are broken whenever replacing the old policy graph with a new policy graph.

For example, if an `Incident` on a `Claim` refers to a `RiskUnit` on the `Policy`, then the claim incident foreign key to the policy risk unit potentially must be repaired. The default behavior is as follows:

- If the old and new risk units differ, ClaimCenter sets the foreign key on the incident to the new risk unit on the refreshed policy
- If the old risk unit does not exist on the refreshed policy, ClaimCenter sets the foreign key on the incident to null.

ClaimCenter encapsulates critical parts of relinking through objects called *relink handlers*. Relink handlers define the behavior when foreign keys between claim objects and policy objects break during policy refresh. The default behavior depends on the direction of the broken link, as the following table describes.

Direction of link	Relink handler behavior
Claim to policy	If there is a match, move target of link to new policy object
	If no match exists and the foreign key property in the data model is set to <code>nullOk="true"</code> , set the foreign key to null. Otherwise, throw an exception.
Policy to claim	If there is a match, move source of link to new policy-specific object. Otherwise, do nothing because soon ClaimCenter will delete the old policy object.

You can customize this behavior by changing an existing custom relink handler or by adding a new one that implements the interface `gw.api.policy.refresh.relink.RelinkHandler`. The `RelinkHandler` interface defines relinking behavior for all foreign key fields on that entity type.

As a convenience, there is another interface called `gw.api.policy.refresh.relink.LinkHandler` that defines more granular per-link (for each foreign key field) behavior. To use this more granular behavior, define the `RelinkHandler` implementations that extend `PerLinkHandler`. Next, use the `register` method to define the relink behavior on each `LinkHandler` that you want to define.

The default behavior for relinking is typically satisfactory for most entity types and does not require new relinking handler classes. However, you might need to customize relink handlers if you want to handle a broken link in a special way. For example, suppose a link is to an object that no longer exists in the new policy. In the default case, that would cause an error. In special cases, you might want to define some more complex compensation behavior. Contact Customer Support if you think you need to modify behavior of links broken during policy refresh.

Configure any new relink handlers in the `PolicyRefreshConfiguration` property getter called `CustomRelinkerTypes`. See the `PolicyRefreshConfigurationBase` class for the default mapping.

Some examples of built-in custom relink handling include:

- Relinking `ContactTag` objects to a new policy contact.
- Unlinking the `Deductible` entity from financial transactions when removing the corresponding `Coverage` object.

Re-link handlers must be defined for the specific entity type that owns the link. This means that you cannot specify relink handlers as shared code in the supertype for all its subtypes. This is notable because in contrast, entity matchers for new subtypes can share implementation in their supertype.

### Example Relink Handler Class for Contact Tag Objects

In the base configuration, ClaimCenter provides the following example of a custom relink handler class. The class `ContactTagRelinkHandler` class relinks `ContactTag` objects to the new policy `Contact`.

```
class ContactTagRelinkHandler extends PerLinkHandler<ContactTag> {  
    construct() { register(ContactTag, "Contact", new TagLinkHandler()) }  
  
    /*  
     * LinkHandler for ContactTag--(Contact)-->Contact  
     */  
    private class TagLinkHandler extends BaseLinkHandler<ContactTag> {  
  
        /*  
         * Handle the case where the Contact is in the removed set.  
         */  
        override function handleRemovedLinkTarget(relinkItem:RelinkItem<ContactTag>,  
            relinkCtx:RelinkContext, target:KeyableBean) {  
            // Ignore if not matched.  
        }  
  
        /*  
         * Handle the case where the new Contact already has the ContactTag being re-linked.  
         */  
        override function handleClaimToPolicyLink(relinkItem:RelinkItem<ContactTag>,  
            relinkContext:RelinkContext) {  
            var target = relinkItem.Value as Contact  
  
            if(relinkContext.PolicyComparison.hasMatch(target)) {  
                var newBean = relinkContext.PolicyComparison.findMatch(target).Compare  
  
                if(not hasContactTag(relinkItem.Owner, newBean)) {  
                    relinkItem.setValue(newBean)  
                }  
            }  
            else {  
                // Do nothing if no match.  
            }  
        }  
    }  
}
```

```

        handleRemovedLinkTarget(relinkItem, relinkContext, target)
    }

    private function hasContactTag(tag:ContactTag, contact:Contact) : boolean {
        return contact.Tags.firstWhere(\ t -> t.Type==tag.Type) != null
    }

}

```

## Relink Filters

ClaimCenter uses *relink filters* to define special logic that is outside the default policy refresh behavior. Relink filters are only for logic that needs to execute before or after relinking. ClaimCenter executes relink filters before the relinking step (in a `preProcess` method) and after the relinking step (in a `postProcess` method). Relink filters implement the interface `gw.api.policy.refresh.relink.RelinFilter`.

Examples of default relink filters are described in the following table.

Relink filter class	Purpose
ContactPreservingFilter	Moves any removed Contacts and moves them to roles with the added prefix <code>former_</code>
ClaimLinkPreservingFilter	Preserves any policy objects if they are removed and referenced from claim entities
CopyInternalFieldsFilter	Copies internal fields from policy entities that are matched to their corresponding replacements in the new policy

Refer to Studio for the implementation of these classes and for more information about:

- Using preserving filters to prevent ClaimCenter from deleting objects
- The relink context object

### Best Practices for Writing Relink Filters

Guidewire recommends the following for writing relink filters:

1. Relink filters are only for logic that needs to execute before or after relinking.

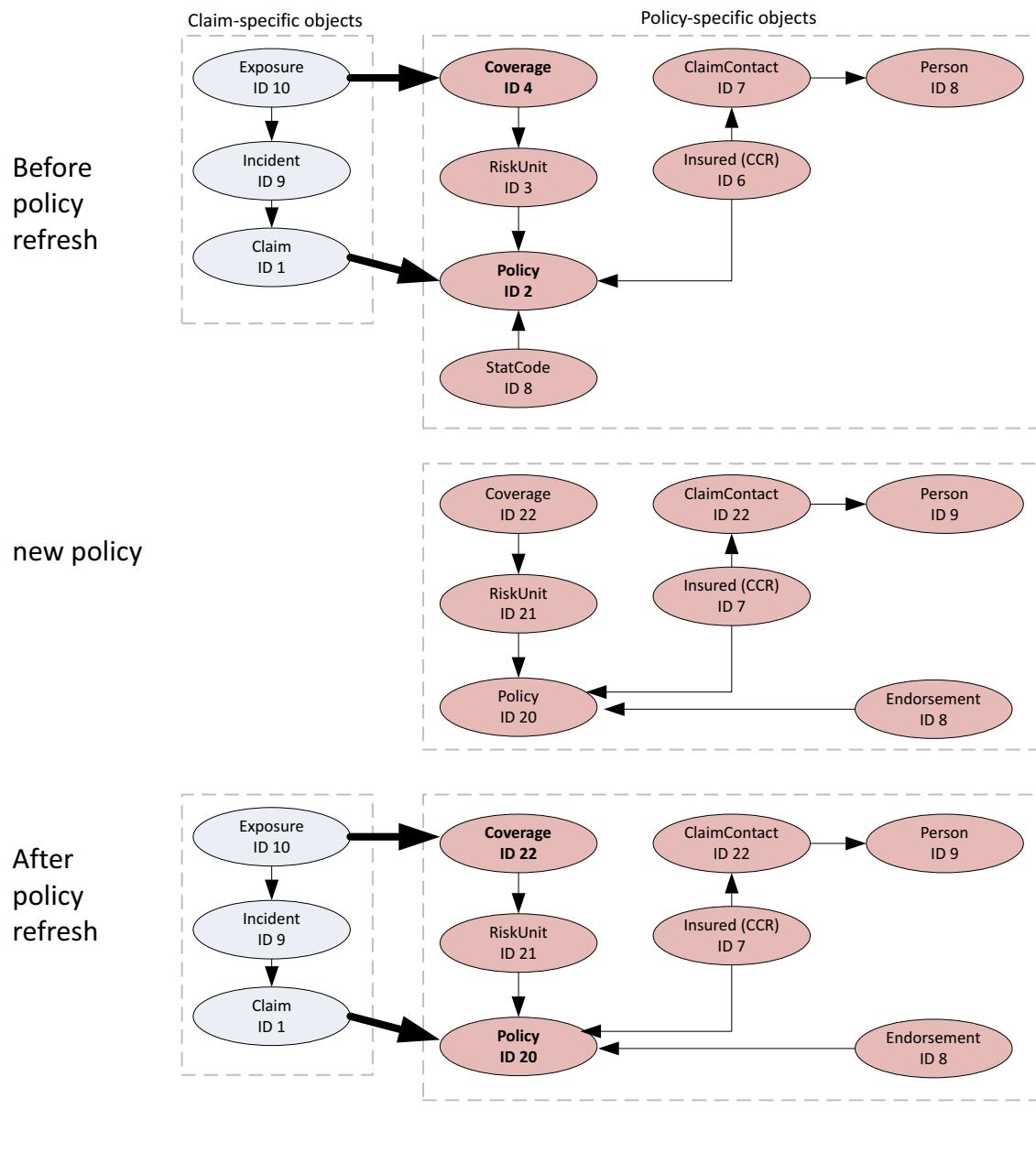
**IMPORTANT** Relink filters must be written carefully to preserve links correctly. If you think you need a relink filter, contact Guidewire Customer Support for guidance.

2. Use the `ClaimLinkPreservingFilter` to preserve entities instead of trying to preserve entities directly from the `gw.api.policy.refresh.relink.RelinContext.addExcludedFromRemoval` in a `RelinkHandler` class. Attempting to preserve entities from the handler can lead to subtle order dependencies in how ClaimCenter performs relinking.
3. Limit preservation behavior to entities that have no outgoing foreign keys (such as `Address`) and use preservation sparingly.
4. If you need to handle only a few foreign keys differently, refer to earlier in the topic about the link handler called `PerLinkHandler`. In that case, do not use a relink filter.
5. As with all relinking code, be mindful of how the relinking can potentially interact with other ClaimCenter processes such as Preupdate rules or bundle callbacks.

## Example Policy Refresh Matching and Relinking - With Relink Set Information

Let us consider the earlier example of relinking and removal. The following diagram shows the state of the claim-policy graphs both before and after policy refresh.

### Policy Refresh and Relinking



→ A regular foreign key link that does not cross the claim-policy boundary

→ Foreign Key link between a claim-specific object and a policy-specific objects. These are the links that will break when replacing the policy. It is these links that must be relinked.

Note: this is a simplified diagram. In a real-world system, there would also be links that point from the policy-specific objects to one of the claim-specific objects. ClaimCenter also relinks those links during policy refresh.

Notice the following differences between the old and new policy:

- The new policy no longer has the `StatCode`
- The new policy has a new entity, an `Endorsement`
- The claim graph's `Exposure` with ID=10 now points to the new `Coverage`
- The claim graph's `Claim` with ID=1 now points to the new `Policy`

After getting the new policy graph, ClaimCenter traverses the new policy. Next, ClaimCenter invokes matcher classes on entities of the same type and in the same graph position in the two graphs.

For this example:

- The following entities are logically the same and the matcher classes add these to the matched set: `Policy`, `RiskUnit`, `Coverage`, `ClaimContactRole`, `ClaimContact`, `Person`.
- The `StatCode` object is absent in the new policy and thus becomes part of the set of removed objects.
- The `Endorsement` from the new policy object is absent in the old policy and thus becomes part of the set of added objects.

After the policy comparison, ClaimCenter must find all links between the group of policy objects and the group of non-policy claim objects. In this simplified example, the links are `Claim` → `Policy` and `Exposure` → `Coverage`. These are the foreign key references that ClaimCenter must relink as part of completing policy refresh.

**Note:** In a real-world example, there might also be links from policy objects to claim objects.

## Policy Refresh Policy Comparison Display

ClaimCenter displays a side-by-side comparison of the current and new policy versions in the **Policy Comparison** screen at policy refresh. Within this screen, it is possible to configure the following:

- The objects and properties that ClaimCenter shows in the policy comparison tree.
- The order, labels, and values for objects and properties that ClaimCenter shows in the **Policy Comparison** screen.
- The warning and error messages that ClaimCenter generates if it detects an issue during policy refresh.

The **Policy Comparison** screen uses two main types of objects to configure differences and display the differences:

- **Difference objects** – These represent a match between an entity from the current policy graph and new policy graph, or any detected property differences.
- **Difference display objects** – These display a diff object, and can generate errors and warnings.

### Difference Objects

A *difference object* (a `Diff` object, or a `diff`) represents a match between an entity from both current and new policy graphs, or a difference in a property. There are two types of difference objects: entity difference objects and property difference objects.

#### Entity Difference Objects

An entity difference object has a name like `entityDiff`, for example `PolicyDiff`. ClaimCenter creates an entity difference object in the policy comparison tree in the following situations:

- The object has been added or removed from the policy.
- The object has child objects that are changed, added, or removed.
- The object has properties that changed.

A logical pair of entities consists of either:

- One entity from each policy graph that the matcher classes matched.

- One entity from only one policy graph, which means that an entity was not matched and the new policy either added or removed the entity from the policy.

The following foreign key properties can have the following values:

Difference object property	Meaning of value
SourceValue	If the current entity exists, contains a foreign key to the entity. If an element of a policy is being added, then Diff.SourceValue is null. This means that the entity appears only in the new policy graph.
CompareValue	If the new entity exists, contains a foreign key to the entity. If an element of a policy is being removed, then Diff.CompareValue is null. This means that the entity appears only in the current policy graph.

### Property Difference Objects

ClaimCenter creates a property difference (PropertyDiff) if an important policy property changed, such as one of the following:

- CancellationDate
- PolicySystemPeriodID
- ProducerCode

### Difference Display Objects

At a programming level, the main mechanism for configuring both how ClaimCenter displays differences on the policy refresh Policy Comparison screen is the DiffDisplay interface. Each entity type has a corresponding object that implements DiffDisplay to display differences for objects of that type. Objects that implement this interface are called *difference display objects* or simply DiffDisplay objects.

Difference display objects are also the mechanism for generating warnings, errors, or informational messages as a result of any differences. See “Policy Refresh Errors, Warnings, and Informational Messages” on page 559. Any errors actually block policy refresh from proceeding.

There are different types of `entityDiffDisplay` classes that display entity-level information for each type of entity in the policy graph, and a single `PropertyDiffDisplay` class. The following summarizes the differences:

Class	Purpose
<code>entityDiffDisplay</code>	<p>Each <i>entity difference display class</i> (an object with name <code>entityDiffDisplay</code>) defines how to display difference information for that entity type as well as what refresh messages to display, if any. Every <code>entityDiffDisplay</code> class extends <code>EntityDiffDisplayBase&lt;T&gt;</code>.</p> <p>For any entity type that appears in the policy graph but for which there is no specific <code>entityDiffDisplay</code> class, ClaimCenter uses the entity difference display base class called <code>EntityDiffDisplayBase</code>. In practice, for the defaults ClaimCenter relies on <code>KeyableBeanDiffDisplay</code>, which extends <code>EntityDiffDisplayBase</code>.</p> <p>In the default configuration, ClaimCenter defines custom difference display objects for the following entity types: <code>Policy</code>, <code>RUCoverage</code>, <code>PolicyCoverage</code>, <code>ClassCode</code>, and <code>PropertyItem</code>.</p>
<code>PropertyDiffDisplay</code>	<p>The property difference display class (<code>PropertyDiffDisplay</code>) displays property-level information for all properties. This class identifies how to display difference information for properties, regardless of the entity to which the property belongs. There is only one class with this name. Unlike the naming pattern of entity difference display classes, property difference display objects are not multiple variants of the name for each entity type or property name</p> <p><b>Note:</b> The class <code>gw.plugin.policy.refresh.ui.PropertyDiffDisplay</code> class is an internal Java class that you cannot modify.</p>

## Adding Difference Display Objects (or Modifying Mappings)

### To configure how ClaimCenter displays an entity

1. If an `entityDiffDisplay` class exists for that entity, then modify that existing `entityDiffDisplay` class.
2. If no `entityDiffDisplay` class exists for that entity, then do both of the following:
  - a. Create a new `entityDiffDisplay` class that extends `EntityDiffDisplayBase<T>`.
  - b. Add an appropriate map element to the `DiffDisplayTypes` getter to the `PolicyRefreshConfiguration` object that you are using, or modify the built-in `PolicyRefreshConfigurationBase` class.

To see the mapping in the base configuration, see the `PolicyRefreshConfigurationBase` Gosu class in the `DiffDisplayTypes` property getter.

In the base configuration, ClaimCenter defines `entityDiffDisplay` classes for the following:

- Policy
- Coverage
- PolicyCoverage
- RUCoverage
- ClassCode
- PolicyLocation
- PropertyItem

In the base configuration, the code for this getter looks similar to the following:

```
override property get DiffDisplayTypes() : Map<IEntityType, Class<DiffDisplay>> {
    return { ClassCode -> ClassCodeDiffDisplay,
             KeyableBean ->KeyableBeanDiffDisplay,
             Policy -> PolicyDiffDisplay,
             PolicyCoverage -> PolicyCoverageDiffDisplay,
             PropertyItem -> PropertyItemDiffDisplay,
             RUCoverage -> RUCoverageDiffDisplay

             /* Custom DiffDisplay for Policy to illustrate one way to customize the hierarchy
              * that gets displayed. For demo purposes, currently unregistered in this
              * configuration file.
              * If you use this, comment out the entry for PolicyDiffDisplay above.
              */
             // Policy -> CustomPolicyDiffDisplay
    }
}
```

If the map does not list a specific policy graph entity type, then ClaimCenter uses the `KeyableBeanDiffDisplay` class as the default, which extends `EntityDiffDisplayBase` as the default.

### See also

- “Policy Refresh Policy Comparison Display” on page 556 for details on how to configure the way in which ClaimCenter renders the **Policy Comparison** screen.

## Getting the Difference Object from an Entity Difference Display Object

Every `entityDiffDisplay` class inherits a `Diff` property from the `EntityDiffDisplayBase` class that it extends. Use the `Diff` property to access the related `entityDiff` instance:

- Use `Diff.SourceValue` to access fields on the current policy.
- Use `Diff.CompareValue` to access fields on the new policy.

For example, the following code determines if the effective date on the new policy differs from the effective date on the current policy by more than seven days. If so, the code generates an error message to that effect.

```
if( Diff.CompareValue.EffectiveDate != Diff.SourceValue.EffectiveDate ) {
    if (Diff.SourceValue.EffectiveDate.daysBetween(Diff.CompareValue.EffectiveDate) > 7 ) {
        messages.add(UIMessage.error("The effective date has changed by more than 7 days."))
    }
}
```

## Getting the Type of Difference from an Entity Difference Display Object

Every `entityDiffDisplay` class inherits a `Type` property from the `EntityDiffDisplayBase` class that it extends. This property specifies the type of difference involved between the current and new entity instances. The different values of `Type` have the following meanings:

Type property value	Meaning
ADDED	The entity instance appears only in the new policy.
REMOVED	The entity instance appears only in the existing or current policy.
CHANGED	The entity instance appears in both policies, but its properties and/or child entities are not the same.
UNCHANGED	The entity instance appears in both policies and its properties and/or child entities are the same.
MOVED_TO MOVED_FROM	<p>The entity instance appears in both the new and the existing policies. However, it does not have the same parent in each policy.</p> <p>This situation can occur if you move a single child from one parent to another. For example, suppose a business has several garage locations and several vehicles. Between two versions of the policy, it is possible for a single vehicle to appear garaged in one location on the current policy. It can then appear in a different location in the new policy. The entity thus appears twice in the comparison tree:</p> <ul style="list-style-type: none"> <li>The entity will have the <code>MOVED_FROM</code> type status on the old parent and will appear in the comparison tree using the <code>REMOVED</code> icon.</li> <li>The entity will also have the <code>MOVED_TO</code> type status on the new parent and will appear in the comparison tree using the <code>ADDED</code> icon.</li> </ul>

For example, the following code from `PolicyDiffDisplay` looks at the current and new versions of the policy and determines if the currency changed on the policy. If so, it generates either an error or a warning, depending on whether the currency on the policy is editable. In general, a policy is editable if the claim associated with this policy does not have any transactions.

```
if(Type==CHANGED) {
    if(Diff.SourceValue.Currency!=Diff.CompareValue.Currency) {
        if(not Diff.SourceValue.CurrencyEditable) {
            messages.add(UIMessage.error(displaykey.PolicyRefresh.DiffDisplay.Policy.CurrencyChange))
        }
        else {
            messages.add(UIMessage.warning(displaykey.PolicyRefresh.DiffDisplay.Policy.CurrencyChange))
        }
    }
}
```

## Policy Refresh Errors, Warnings, and Informational Messages

Each `DiffDisplay` object is responsible for generating any messages for that entity type by returning a list of messages in its `getMessages` method. Policy refresh messages include error, warning, or information messages. ClaimCenter encapsulates each message in an object of type `gw.api.web.UIMessage`.

Generally speaking, a difference display object makes messages that are related only to the entity for which the given `DiffDisplay` object is responsible.

- An `ERROR` message is the only message type that actually blocks the policy refresh from completing. Define these for incompatible changes that cannot be resolved automatically using existing mechanisms. Avoid

throwing exceptions from within the policy refresh process unless you cannot represent them at the user interface level by an appropriate ERROR message.

**IMPORTANT** If ClaimCenter identifies an error condition, it removes the **Finish** button from the **Policy Comparison** screen. This prevents the policy refresh from completing.

- **WARNING** messages define issues that might pose a problem, but are not automatically conditions that block policy refresh.
- **INFO** messages define any other informative message or suggested action as a result of the policy refresh. In the base configuration, there are no information messages generated for built-in entity types.

ClaimCenter lists all messages in order of severity. In other words, ClaimCenter lists all errors first, followed by warnings, and then by info messages.

You can configure all message types. In configuring messages:

- Construct each policy refresh message as an instance of `gw.api.web.UIMessage`.
- Group a set of `UIMessage` objects through the use of `gw.api.web.UIMessageList`.
- Use the method `entityDiffDisplay.getMessages` to construct the `UIMessageList` object, which ClaimCenter renders in the user interface as the set of error, warning, and informational messages.

## Creating a Policy Refresh Message

To create a policy refresh message, a difference display object creates a `UIMessage` object.

Use the following syntax to call static methods on the `UIMessage` class:

```
UIMessage.error(msgText)
UIMessage.warning(msgText)
UIMessage.info(msgText)
```

Always use a display key for the message text. For example:

```
UIMessage.warning(displaykey.PolicyRefresh.DiffDisplay.Policy.PolicySystemPeriodIDChange))
```

In addition to being shown in the policy comparison screen, error messages also block the refresh process. There is no additional functionality associated with warning or info messages.

In the base configuration, there are a few `entityDiffDisplay` classes that do not create messages. One, for example, is the `PolicyCoverageDiffDisplay` class. However, of the classes that do create messages, most of these classes create messages directly in the `getMessages` method. The one exception is the `CoverageDiffDisplay` class, which merely calls a `performValidation` method from within the `getMessages` method. ClaimCenter actually creates the messages within this `performValidation` method, which exists in `gw.plugin.policy.refresh.CoverageValidator`.

## Returning a Policy Refresh Message List to Encapsulate Messages

ClaimCenter encapsulates a set of `UIMessage` objects to show to the user in a `UIMessageList` object.

### To construct a UI message list object

1. Create a new `UIMessageList` object using the `new` operator, for example:  
`messages = new UIMessageList()`
2. Define the condition to use for comparing the two policy versions, for example:  
`Type==CHANGED`
3. Compare the new and current entities or properties that are of interest, for example:  
`Diff.SourceValue.Currency!=Diff.CompareValue.Currency`
4. Construct a `UIMessage` object and add it to the `UIMessageList` object using the list's `add` method.

To illustrate:

```
var messages = new UIMessageList()
if(Type==CHANGED) {
    if( Diff.SourceValue.Currency!=Diff.CompareValue.Currency) {
        messages.add(UIMessage.error(displaykey.PolicyRefresh.DiffDisplay.Policy.CurrencyChange))
    }
}
```

For each entity type that can require a message, you must override the corresponding `entityDiffDisplay.getMessages` method. The `getMessages` method receives a `PolicyRefreshMessageContext` object, discussed later in this topic. The `getMessages` method returns a `UIMessageList` that is either empty or contains one or more messages to show within ClaimCenter.

The `PolicyRefreshMessageContext` object provides context for the difference display object code to construct errors and warning related to a policy refresh. A `PolicyRefreshMessageContext` object has two methods that are useful in policy refresh message logic that each returns the set of non-policy claim entities linked to a given entity:

- The `findClaimEntitiesLinkedTo` method returns all non-policy claim entities
- The `findClaimEntitiesOfTypeLinkedTo` method filters for entities of a specific type.

For example, the following code appears in the base configuration `PropertyItemDiffDisplay` class:

```
if (this.Type==REMOVED) {
    if (ctx.findClaimEntitiesOfTypeLinkedTo(Diff.SourceValue, PropertyContentsScheduledItem).
        HasElements) {
        result.add(UIMessage.warning(displaykey.PolicyRefresh.DiffDisplay.PropertyItem.
            MissingPropertyItem(Diff.SourceValue)))
    }
}
```

The logic for this code is the following:

- If this property item is on the current policy, but not on the new policy...
- And, if there are any entities of type `PropertyContentsScheduledItem` on the claim that are linked to the property item on the current version of the policy...
- Then, add the following warning message:  
`The PropertyItem {0} has been removed from the policy and is referenced by the claim.`  
In the display key string, `{0}` is the display name for the property item.

## Configuring Visibility in the Comparison Tree

ClaimCenter uses a RowTree PCF widget to generate the policy comparison tree in the **Policy Comparison** screen. The policy comparison tree displays differences between a current and new policy as a collapsible hierarchy of objects. You can expand or collapse the hierarchy through the use of the +/- controls to expand or collapse a portion of the tree. ClaimCenter shows the +/- controls for entities with child entities only.

It is possible for the leaves of the policy comparison tree (the entries that do not have +/- controls) to be either properties or entities. In the base ClaimCenter application, you can identify which is an entity or which is a property in the following ways:

- The label format for entities is `entityType:DisplayName`, with a colon.
- The label format for properties is simply `DisplayName`. The ClaimCenter base configuration does not use colons for properties.
- The visual indicator for changed entities is to use an icon to indicate the change. The ClaimCenter base configuration shows visual indicators for added or removed entities only.
- The visual indicator for changed properties is to simply show the actual difference, for example, the two different cancellation dates.

### Setting Visibility

ClaimCenter uses the `entityDiffDisplay.isChildVisible` method to determine what to show in the policy comparison tree. In the base ClaimCenter configuration, Guidewire provides an overridden definition of this method in class `EntityDiffDisplayBase`:

- If you want to globally modify the logic for displaying children, modify the `isChildVisible` method in `EntityDiffDisplayBase`.
- If you want to modify the logic for displaying children for a specific type of entity, then override the `isChildVisible` method in the `entityDiffDisplay` class for that entity. For example, if you want to override the logic for displaying coverages in the policy comparison tree, then you need to modify the `isChildVisible` method in `CoverageDiffDisplay`.

ClaimCenter automatically excludes several sets of properties from the policy comparison tree:

1. ClaimCenter automatically excludes properties used to track an object internally, for example, ID, `CreateTime`, and `CreateUser`. This behavior occurs in internal code and you cannot configure it.
2. ClaimCenter programmatically excludes additional properties that you specify through the `PolicyRefreshConfigurationBase.EntityPropertiesToIgnoreForComparison` getter. You configure this getter to identify additional properties to always omit from the tree. In the base application, ClaimCenter excludes the following properties:
  - `Contact.AutoSync`
  - `ClaimContact.ClaimantFlag`
  - `ClaimContactRole.PartyNumber`

## Construction of the Policy Comparison Tree

To construct the policy comparison tree, ClaimCenter does the following:

1. ClaimCenter instantiates an `entityDiff` object for the following:
  - One for every matched pairs of entities
  - One for every unmatched entityClaimCenter uses logic from the matcher classes to determine if one entity matches another. See “Policy Refresh Entity Matcher Details” on page 548 for details.
2. ClaimCenter organizes the `entityDiff` instances into a tree structure, using the `PolicyRefreshConfigurationBase.DisplayTree` getter.  
ClaimCenter uses the `DisplayTree` getter to convert the policy graph into a one-to-many hierarchy for viewing in the ClaimCenter interface. In actuality, the policy data model, and, therefore, the graph of `entityDiff` objects, is not a tree. However, ClaimCenter uses a `RowTree` widget to display the information in the ClaimCenter interface. The `DisplayTree` property transforms the graph of `entityDiff` objects into a tree for viewing purposes. See “Difference Objects” on page 556 for more information.
3. ClaimCenter instantiates an `entityDiffDisplay` object for each `entityDiff` object. The `entityDiffDisplay` object specifies how to display the entity differences:
  - Each `entityDiffDisplay` determines whether to display its data, and if so, how to display the data.
  - Each `entityDiffDisplay` has a type, which is typically set to ADDED, CHANGED, or REMOVED.
  - Each `entityDiffDisplay` is responsible for generating warning or error messages at the top of the `Policy Comparison` screen. There are no messages for `PropertyDiffDisplay`. ClaimCenter only generates messages at the `entityDiffDisplay` level.

**Note:** Class `PropertyDiffDisplay` exists in the `gw.plugin.policy.refresh.ui` package. It is an internal Java class that you cannot modify.
4. ClaimCenter renders the policy comparison tree based on the logic defined by the `entityDiffDisplay` instances.

### Defining the Tree Using the Display Tree Property Getter

The `PolicyRefreshConfigurationBase` class contains the `DisplayTree` property getter. The `DisplayTree` getter builds the policy comparison tree using multiple `addOneTo...` methods to build the tree by connecting child `entityDiff` objects to parent `entityDiff` objects.

The only thing you can configure in the `DisplayTree` getter is the mapping from a given parent to a given child. You cannot create relationships that do not already exist in the policy graph. For example, you cannot display vehicle coverages as the child of a contact.

The `DisplayTree` getter logic uses the following important methods:

<code>addOneToMany</code>	Adds one-to-many child elements to a given parent element
<code>addOneToOne</code>	Adds a single child element to a given parent element

These methods have three parameters, which you use to specify a parent/child relationship in the policy comparison tree:

<code>parentType</code>	An <code>IEntityType</code> for the parent entity type to which the <code>entityDiff</code> object refers.
<code>propertyName</code>	A String for the property name of the parent property that points to the child.
<code>childType</code>	An <code>IEntityType</code> for the child entity type to which the <code>entityDiff</code> object refers.

The `entityDiff` graph consists of a number of *nodes* (the objects in the graph) and *edges* (the connections between pairs of related nodes). To construct the tree, ClaimCenter traverses the entire `entityDiff` graph. In doing so, ClaimCenter compares every connection between the tree nodes to the list of relationships specified in the `addOneToMany` and `addOneToOne` methods.

In performing the comparison:

- If ClaimCenter finds a relationship in which all parameters match, then it creates an `entityDiffDisplay` instance from the child `entityDiff` and attaches it to the tree at the appropriate point.
- If ClaimCenter does not find a relationship in which all parameters match, then it does not create an `entityDiffDisplay` for that edge or connection.

Guidewire recommends that you pay close attention to how you modify the `DisplayTree` getter, as minor errors can result in ClaimCenter not displaying nodes in the tree. For example, ClaimCenter does not create an `entityDiffDisplay` instance and does not display the information in the policy comparison tree if you do the following:

- If you identify the wrong `IEntityType` for the parent or for the child
- If you misspell the name of the relationship

**WARNING** Pay close attention to how you modify the `DisplayTree` getter, as minor errors can result in ClaimCenter not displaying nodes in the tree.

## Configuring Labels and Display Order

It is possible to configure the following items in the policy comparison tree:

- Entity and property labels
- Entity and property display order

### Configuring Entity Labels

In the base configuration, ClaimCenter uses the following format for entity labels:

`entityType: DisplayName`

Use `entity.EntityName` display keys to format the entity type.

Alternatively, you can configure how ClaimCenter displays an entity label by overriding the `Label` getter in the appropriate `entityDiffDisplay` class. For example, to change how ClaimCenter displays the label for the policy root in the policy comparison tree, add code similar to the following to `PolicyDisplayDiff`:

```
override property get Label() : String {
    return "Policy (ID: " + Diff.SourceValue.PolicyNumber " ")"
}
```

## Configuring Property Labels

ClaimCenter generates property labels through an internal lookup search for a display key with the following format:

```
entity.entityType.propertyName
```

Thus, you configure property labels by changing the value of the display key. For example, suppose that you want to change the label `Policy System Period ID` to `Policy Period ID`. You need merely find the correct display key, in this case, `entity.Policy.PolicySystemPeriodID`, and change it to the desired value.

Display key	Value
<code>entity.Policy.PolicySystemPeriodID</code>	Policy System Period ID
<code>entity.Policy.PolicyPeriodID</code>	Policy Period ID

## Configuring Display Order

The policy comparison tree `RowTree` widget controls property and entity display order. In the base application:

- ClaimCenter lists properties first, in alphabetical order by sort expression.
- ClaimCenter lists entities next, in alphabetical order by sort expression.

For example:

```
Policy: 54-847654
Cancellation Date
Policy System Period ID
Producer Code
Covered Vehicle: Honda GX
Person: Allen Robertson
Person: Karen Egertson
```

Notice that properties labels do not contain colons.

Within the **Policy Comparison** screen, ClaimCenter labels the left-most column as **Entity or Property** and provides a small triangular sort icon. This column maps to the `RowTree` widget on the `PolicyComparisonScreen PCF`.

Within the `RowTree` widget, this column maps to the left-most widget cell, with an ID of `LabelField`. This cell contains the following sorting-related properties:

- `sortBy`
- `sortDirection`
- `sortOrder`

You use the `sortBy` property on the `PolicyComparisonScreen PCF` to define logic for dynamically generating a sort value for each child element. ClaimCenter then sorts the child elements based on their sort values.

You set the `sortBy` property by adding a `getSortBy` function and defining that function on the `Code` tab. In the base configuration, the `getSortBy` function looks similar to the following:

```
uses gw.plugin.policy.refresh.ui.DiffDisplay
uses gw.plugin.policy.refresh.ui.EntityDiffDisplay

function getSortBy (diff : DiffDisplay) : String {
```

```
    return ((diff.typeis EntityDiffDisplay) ? "B:" : "A:") + diff.Label
}
```

In the base configuration, this function checks to see if the child element is of type `EntityDiffDisplay` or not:

- If the child element is an `EntityDiffDisplay`, then its sort expression is “B:” plus the label.
- If the child element is not an `EntityDiffDisplay`, then it is a `PropertyDiffDisplay` and its sort expression is “A:” plus the label.

As a consequence:

- ClaimCenter sorts all the properties first as they all start with “A:” and then orders the properties by their label value.
- ClaimCenter sorts all the entities afterwards as they all start with “B:” and then orders the entities by their label values.

Be aware that the strings “A:” and “B:” are somewhat arbitrary. It is possible to replace these strings with other strings that sort the properties and entities in the desired way (such as “1”, “2”, “3”...).

To modify the sort order for properties or entities, you need to modify the `getSortBy` function defined on the **Code** tab of the `PolicyComparisonScreen` PCF. To access the **Code** tab, you need to select the entire PCF.

For example, suppose that you want to configure the sort order so that ClaimCenter displays the `Producer Code` property before any other property. To do so, modify the `getSortBy` function in the following manner:

```
function getSortBy (diff : DiffDisplay) : String {
    if (diff.Label == "Producer Code") {
        return "A:Producer Code"
    }
    else {
        return ((diff.typeis EntityDiffDisplay) ? "C:" : "B:") + diff.Label
    }
}
```

## Changing Icons for Added, Removed, Unchanged, and Changed

You can change the icons that ClaimCenter displays for added, removed, unchanged, or changed (updated) differences. Change the appropriate display keys that reference file names in the images folder:

- `Web.PolicyRefresh.PolicyComparisonScreen.IconFieldFileName.Added`
- `Web.PolicyRefresh.PolicyComparisonScreen.IconFieldFileName.Removed`
- `Web.PolicyRefresh.PolicyComparisonScreen.IconFieldFileName.Unchanged`
- `Web.PolicyRefresh.PolicyComparisonScreen.IconFieldFileName.Changed`

## Policy Refresh Configuration Examples

This topic includes:

- “Adding a New Claim-specific Entity to Policy Refresh” on page 565
- “Adding a New Policy-specific Entity to Policy Refresh” on page 566
- “Modifying Your Policy System for Changes to Entities in the Policy Graph” on page 568
- “Additional Tasks to Configure Policy Refresh for New Policy-Specific Entities” on page 570

### Adding a New Claim-specific Entity to Policy Refresh

Suppose you add a new entity type, `ClaimEntity_Ext`, that is claim-specific and not part of the policy graph. In `ClaimEntity_Ext` you include a foreign key to `Claim`. In turn, you extend `Claim` with a one-to-one element to provide a symmetrical link back to `ClaimEntity_Ext`.

To assure `ClaimEntity_Ext` is part of the claim graph, do not modify the policy refresh plugin to include it in the set of entity types that comprise the policy graph. Thus, ClaimCenter knows that your new `ClaimEntity_Ext` entity is part of the claim graph.

Now, suppose your new `ClaimEntity_Ext` has an array property that contains `Vehicle` instances. The `Vehicle` entity is a built-in type that belongs to the policy graph. For array properties, database-level foreign keys must reside on the child objects (`Vehicle`) to relate the children to their parent objects (`ClaimEntity_Ext`). So, you extend `Vehicle` with a foreign key to `ClaimEntity_Ext`.

### Example of Extending the Metadata Definitions for a New Claim-specific Entity Type

You define your new entity `ClaimEntity_Ext` in this metadata definition file:

```
ClaimCenter/config/extensions/ClaimEntity_Ext.eti
```

The contents of the file might look like the following example.

```
<?xml version="1.0"?>
<entity
    xmlns="http://guidewire.com/datamodel"
    entity="ClaimEntity_Ext"
    table="claimentity"
    type="retireable">
    <implementsEntity name="Extractable"/>
    <foreignkey name="Claim" fkentity="Claim"/>
    <array name="Vehicles" arrayentity="Vehicle"/>
</entity>
```

In `ClaimCenter/config/extensions/Vehicle.etc`, extend the `Claim` entity with a one-to-one element.

```
<onetoone fkentity="ClaimEntity_Ext" name="ClaimEntity_Ext" nullok="true"/>
```

In `ClaimCenter/config/extensions/Vehicle.etc`, extend the `Vehicle` entity with a foreign key.

```
<foreignkey name="ClaimEntity_Ext" fkentity="ClaimEntity_Ext"/>
```

The changes described in this example are common types of data model changes. Because your new entity belongs to the claim graph and not the policy graph, the policy refresh system places no additional demands on the configuration.

### How Policy Refresh Automatically Updates Objects in the Claim and Policy Graphs

When policy refresh happens, `ClaimEntity_Ext` entity instances remain unchanged within the claim graph. For links to `ClaimEntity_Ext` instances from refreshed `Vehicle` instances, the policy refresh system relinks them automatically. The directionality of the foreign key runs from a policy object to a claim object. Directionality of foreign keys is an important distinction when reviewing the default relinking behavior of the policy refresh system.

### Adding a New Policy-specific Entity to Policy Refresh

Suppose you add a policy entity type, `PolicyEntity_Ext`, that is policy-specific and not part of the claim graph. In `PolicyEntity_Ext` you include a foreign key to `Policy`. In turn, you extend `Policy` with an array property to provide a collection of symmetrical links back to its `PolicyEntity_Ext` instances. A policy can have many policy extensions, but each policy extension belongs to only one policy.

To assure `PolicyEntity_Ext` is part of the policy graph, you must modify the policy refresh plugin to include it in the set of entity types that comprise the policy graph. Thus, ClaimCenter knows that your new `PolicyEntity_Ext` entity is part of the policy graph.

Now, suppose your new `PolicyEntity_Ext` has a foreign key to `Exposure`. The `Exposure` entity is a built-in type that belongs to the claim graph. In turn, you extend `Exposure` with an array property that contains `PolicyEntity_Ext` instances. An exposure can have many policy extensions, but each policy extension belongs to only one exposure.

**See also**

For information on how to change the policy refresh plugin, see “Determining the Extent of the Policy Graph” on page 546.

### Example of Extending the Metadata Definitions for a New Policy-specific Entity Type

You define your new entity `PolicyEntity_Ext` in this metadata definition file:

```
ClaimCenter/config/extensions/PolicyEntity_Ext.eti
```

The contents of the file might look like the following example.

```
<?xml version="1.0"?>
<entity
    xmlns="http://guidewire.com/datamodel"
    entity="PolicyEntity_Ext"
    table="policyentity"
    exportable = "true"
    type="retireable">
    <implementsEntity name="Extractable"/>
    <foreignkey name="Policy" fkentity="Policy"/>
    <foreignkey name="Exposure" fkentity="Exposure"/>
    <column
        name="Name"
        type="shorttext"
        desc="Policy Entity Name"/>
</entity>
```

In `ClaimCenter/config/extensions/Policy.etc`, extend the `Policy` entity with an array of policy extensions.

```
<array name="PolicyEntity_Exts" arrayentity="PolicyEntity_Ext"/>
```

In `ClaimCenter/config/extensions/Exposure.etc`, extend the `Exposure` entity with an array of policy extensions.

```
<array name="PolicyEntity_Exts" arrayentity="PolicyEntity_Ext"/>
```

The changes described in this example are common types of data model changes. Because your new entity belongs to the policy graph, the policy refresh system requires you to perform additional configuration tasks beyond those associated with standard data model extensions. You must:

- Modify the policy refresh plugin to include `PolicyEntity_Ext` in the set of entity types that comprise the policy graph.
- Modify your policy administration system to correspond to the changes you make in the ClaimCenter policy graph.
- Perform configuration tasks in other parts of the policy refresh system to complete your addition of new policy-related entities to the ClaimCenter policy graph. These tasks include:
  - Adding new policy-related entities to the policy refresh display tree
  - Creating custom matcher classes
  - Creating customer relinker classes

**See also**

- For information on how to modify the policy refresh plugin, see “Determining the Extent of the Policy Graph” on page 546.
- For information on how to modify your policy administration system, see “Modifying Your Policy System for Changes to Entities in the Policy Graph” on page 568.
- For information on additional configuration tasks in the policy refresh system, see “Additional Tasks to Configure Policy Refresh for New Policy-Specific Entities” on page 570

### How Policy Refresh Automatically Updates Objects in the Claim and Policy Graphs

When policy refresh happens, ClaimCenter replaces any `PolicyEntity_Ext` entity instances, which belong to the policy graph. For foreign key links from `PolicyEntity_Ext` instances to exposures, the policy refresh system relinks them automatically. The directionality of the foreign key runs from a policy object to a claim object.

Foreign key links from updated `PolicyEntity_Ext` instances to other updated policy-specific instances in the refreshed policy graph do not require relinking. The policy refresh system retrieves valid links between policy-specific instances from the policy administration system whenever policy refresh refreshes a policy snapshot in ClaimCenter.

## Modifying Your Policy System for Changes to Entities in the Policy Graph

Whenever you add a new entity type to the policy graph in ClaimCenter, policy refresh requires you to perform additional configuration tasks beyond those associated with standard data model extensions. Tasks depend on whether your policy administration system is PolicyCenter or a third-party system.

This topic includes:

- “Tasks in PolicyCenter for New Policy-Specific Entities” on page 568
- “Tasks in Third-Party Policy Administration Systems for New Policy-Specific Entities” on page 570

### Tasks in PolicyCenter for New Policy-Specific Entities

Whenever you add a new entity type to the policy graph in Claim Center, you must create a corresponding new entity type in PolicyCenter. This topic continues the example of adding the `PolicyEntity_Ext` entity type to the policy graph in ClaimCenter.

#### 1. Add New Policy-Specific Entity Types to the PolicyCenter Data Model

In PolicyCenter Studio, define a new `PolicyEntity_Ext` entity in this metadata definition file:

`PolicyCenter/config/extensions/PolicyEntity_Ext.eti`

The contents of the PolicyCenter metadata definition file differ slightly from the ClaimCenter version. In PolicyCenter, the `PolicyPeriod` entity type corresponds to the ClaimCenter `Policy` entity type. So, the foreign key elements in the two versions differ.

```
<?xml version="1.0"?>
<entity
    xmlns="http://guidewire.com/datamodel"
    entity="PolicyEntity_Ext"
    table="policyentity"
    exportable = "true"
    type="retireable">
    <implementsEntity name="Extractable"/>
    <foreignkey name="PolicyPeriod" fkentity="PolicyPeriod"/>
    <column
        name="Name"
        type="shorttext"
        desc="Policy Entity Name"/>
</entity>
```

In the file `PolicyCenter/config/extensions/PolicyPeriod.etx`, extend the `PolicyPeriod` entity with array of policy extensions.

```
<array arrayentity="PolicyEntity_Ext" cascadeDelete="true" name="PolicyEntities_Ext"/>
```

#### 2. Add a Gosu Class for each New Policy-Specific Entity Type

In PolicyCenter Studio, create or modify Gosu classes to match your changes to the data model. Prefix the class names with CC. Continuing our example from above, create a Gosu class called `CCPolicyEntity_Ext`.

```
package gw.webservice.pc.pcVERSION.ccintegration.ccentities

class CCPolicyEntity_Ext {
    var _name : String as Name
    construct() {
    }
}
```

The purpose of the Gosu class `CCPolicyEntity_Ext` is to mirror the `PolicyEntity_Ext` entity in ClaimCenter. Follow this pattern for all custom policy graph entities.

Note the following qualities of these special CC classes:

- They do not need public property getters and setters.
- They have an empty constructor.

### 3. Modify the Policy Generator and ClaimCenter Policy Classes

In PolicyCenter Studio, modify `CCPolicyGenerator`, specifically the `generatePolicy` method. Change the method to instantiate the new Gosu classes and copy data from the persistent entities as needed. Continuing the example of the entity `PolicyEntity_Ext` and the Gosu class `CCPolicyEntity_Ext`, the modified `generatePolicy` method looks like the following sample Gosu code.

```
uses gw.webservice.pc.pcVERSION.ccintegration.ccentities.CCPolicy

/*
 * Generate a ClaimCenter Policy object and related objects from a PolicyCenter PolicyPeriod.
 */
class CCPolicyGenerator {
    ...

    /*
     * Create the ClaimCenter Policy and all related objects for the given PC PolicyPeriod.
     */
    public function generatePolicy( pcPolicyPeriod : PolicyPeriod ) : CCPolicy {
        ...

        // Generate ClaimCenter versions of custom PolicyCenter entities.
        for (pcEntity in pcPolicyPeriod.PolicyEntities_Ext) {

            // Instantiate a ClaimCenter instance.
            var newCCPolicyEntity_Ext = new CCPolicyEntity_Ext()

            // Copy properties from the PolicyCenter instance to the ClaimCenter instance.
            newCCPolicyEntity_Ext.Name = pcEntity.Name

            // Add the populated ClaimCenter instance to the ClaimCenter policy instance.
            _policy.addToPolicyEntities_Ext(newCCPolicyEntity_Ext)
        }

        return _policy
    }
}
```

Add the adder function in the Gosu class `CCPolicy.gs`:

```
class CCPolicy
{
    var _policyEntities = new ArrayList<CCPolicyEntity_Ext>()
    ...

    function addToPolicyEntities_Ext(policyEntity : CCPolicyEntity_Ext) : void {
        _policyEntities.add(policyEntity)
    }
}
```

Start the PolicyCenter application to verify your data model extensions and Gosu modifications.

### 4. Refresh the WSDL in ClaimCenter

In ClaimCenter Studio, refresh the WSDL from PolicyCenter. PolicyCenter recursively reflects on the `CCPolicy` Gosu class and its related types, including the new ClaimCenter-related Gosu classes you added. There now will be new XSD types in the WSDL files that allow ClaimCenter to retrieve policies from PolicyCenter.

ClaimCenter now knows from the WSDL about the new Gosu classes that you added to PolicyCenter, such as `CCPolicyEntity_Ext`.

By default, ClaimCenter automatically copies data from a ClaimCenter Gosu object (in our example, `CCPolicyEntity_Ext`) to the corresponding entity (in our example, `PolicyEntity_Ext`). The copying occurs only if the following conditions are met:

- **The type names match** – Other than the prefix CC on the Gosu type within ClaimCenter
- **The type properties match** – Specifically, matching names and types of fields

The rules listed above are only the default rules. You can modify the rules to do a custom mapping by modifying the file `pc-to-cc-data-mapping.VERSION.xml`. For example, you could not require the CC prefix for some entity types, or you could do some other custom matching.

## Tasks in Third-Party Policy Administration Systems for New Policy-Specific Entities

If you use a policy administration system other than PolicyCenter, use the `BasicPolicyRefreshConfiguration` configuration class in ClaimCenter to include new policy-specific entity types in the policy refresh system.

In the class `BasicPolicyRefreshConfiguration`, modify the `getPolicyOnly` method. In our continuing example, the changes look like the following.

```
class BasicPolicyRefreshConfiguration extends ExtendablePolicyRefreshConfiguration {
    ...
    /*
     * This method extracts all Policy-only entities from the existing Policy (which is linked
     * to a Claim and other non-Policy entities).
     */
    override function getPolicyOnly(existingPolicy : Policy) : Set<KeyableBean> {
        // Use helper method getEntireArray to include retired entities, if entity is retireable
        getEntireArray(existingPolicy, "PolicyEntities_Ext", PolicyEntity_Ext).each(
            \ g ->includePolicyEntity_Ext(policyOnly, g))
        return policyOnly
    }
}
```

### See also

For information about the policy refresh plugin and the `BasicPolicyRefreshConfiguration` class, see “Policy Refresh Plugins and Configuration Classes” on page 542.

## Additional Tasks to Configure Policy Refresh for New Policy-Specific Entities

Whenever you add a new entity type to the policy graph, you must perform additional configuration tasks related to policy refresh, regardless what policy administration system you use. If you performed the preceding tasks properly, custom entities in the policy administration system become part of the new policy graph in ClaimCenter and are replaced when appropriate.

You must perform the following additional configuration tasks in other parts of the policy refresh system to complete your addition of new policy-related entities to the ClaimCenter policy graph:

- “Customizing the Policy Refresh Display Tree for New Policy-specific Entities” on page 571
- “Customizing Policy Refresh Matchers for New Policy-specific Entities” on page 571
- “Customizing Policy Refresh Relinkers for New Policy-specific Entities” on page 572

### Customizing the Policy Refresh Display Tree for New Policy-specific Entities

Whenever you add custom entities to the policy graph in ClaimCenter, you must add the custom entities to the policy refresh display tree in the **Policy Refresh Wizard**. The wizard compares the differences between the current snapshot of a policy in ClaimCenter and a refresh of the policy snapshot from the policy administration system.

Optionally create custom `DiffDisplay` objects to customize the appearance of node entries for custom entities in the policy refresh comparison screen. Create a custom `DiffDisplay` and register it with

`PolicyRefreshConfigurationBase`. Using our continuing example, create the following Gosu class and note the use of Gosu generics in the first line.

```
class PolicyEntity_ExtDiffDisplay extends EntityDiffDisplayBase<PolicyEntity_Ext> {  
    construct(theDiff : EntityDiff<ClassCode>, theType : DiffDisplay.Type) {  
        super(theDiff, theType)  
    }  
  
    override function getMessages(ctx : PolicyRefreshMessageContext) : UIMessageList {  
        var result = new UIMessageList()  
        if (Type==REMOVED) {  
            result.add(UIMessage.error("hello I am an error"))  
        }  
  
        return result;  
    }  
}
```

Next, add your new `DiffDisplay` class to the `DiffDisplayTypes` property map in `PolicyRefreshConfigurationBase`, as the following sample Gosu code shows.

```
override property get DiffDisplayTypes() : Map<IEntityType, Class<DiffDisplay>> {  
    return {  
        ...  
        PolicyEntity_Ext -> PolicyEntity_ExtDiffDisplay,  
        ...  
    }  
}
```

Next, in `PolicyRefreshConfigurationBase`, modify the getter for the `DisplayTree` property so it includes a reference to new custom entities in their appropriate places in the policy graph. For instance, if the customization introduces a new `PolicyEntity_Ext[]` array property on `Policy` called `AllPolicyEntities` that refers to `PolicyEntity_Ext` entities, then you would add the following association.

```
tree.addOneToMany(policyRoot, "AllPolicyEntities", PolicyEntity_Ext)
```

Note that the association must satisfy the following conditions:

- **The cardinality of the property must match** – If it is an array property, use the method `addOneToMany`. If it is a one-to-one property, use the method `addOneToOne`.
- **The property name must match** – If the data model property is called `AllPolicyEntities`, then you must pass the string `"AllPolicyEntities"` as the second parameter to `addOneToMany` (or to `addOneToOne`).
- **The property type must match or be a supertype** – If the property is `PolicyEntity_Ext`, then the third parameter must either be `PolicyEntity_Ext`, or a supertype of `PolicyEntity_Ext`.

#### See also

- “Policy Refresh Policy Comparison Display” on page 556
- “Policy Refresh Wizard” on page 95 in the *Application Guide*

#### Customizing Policy Refresh Matchers for New Policy-specific Entities

While technically not necessary, it is a practical requirement that you write your custom matchers in Gosu for custom policy graph entities. Next, register your matchers in the policy refresh configuration file. Without a custom matcher, the default behavior relies on object identity to determine if a given current policy-related entity instance matches a given refreshed policy-related entity instance.

Object identity is almost always an inappropriate way to match policy-related entity instances. Instead, create a Gosu class for your own matcher. The class you extend to create your class depends on your matching requirements for your new policy-related entity:

- If your matcher matches instances of your new entity only on its own properties and not the properties of related entity types, extend the class `gw.api.bean.compare.MatcherBase`.

- If your matcher matches instances of your new entity on properties of related entity types, extend the class `gw.api.bean.compare.InitializableMatcherBase`.

The following example Gosu matcher class extends the `InitializableMatcherBase` class.

```
package gw.plugin.policy.refresh.matcher

uses gw.api.bean.compare.InitializableMatcherBase

/*
 * Entity matcher for a PolicyEntity_Ext.
 */
class PolicyEntity_ExtMatcher extends InitializableMatcherBase<PolicyEntity_Ext> {

    override function doEntitiesMatch(p0 : PolicyEntity_Ext, p1 : PolicyEntity_Ext) : boolean {
        return p0.Name.equals(p1.Name)
    }
}
```

Next, register your new matcher class with policy refresh by adding it to the `MatcherTypes` property map in `PolicyRefreshConfigurationBase`, as in this example.

```
override property get MatcherTypes() : Map<IEntityType, Class<EntityMatcher<KeyableBean>>> {
    return {
        ...
        PolicyEntity_Ext -> PolicyEntity_ExtMatcher,
        ...
    }
}
```

#### See also

[“Policy Refresh Entity Matcher Details” on page 548](#)

#### Customizing Policy Refresh Relinkers for New Policy-specific Entities

Whenever you add new policy-specific entities to the policy graph in ClaimCenter, create custom `Relinker` classes for them. The default relinking behavior suffices for most cases without requiring a custom `Relinker`. If a custom entity does require special relinking behavior, create a custom `Relinker` and register it with `PolicyRefreshConfigurationBase`.

For example:

```
package gw.plugin.policy.refresh.relink.handler

uses gw.api.policy.refresh.relink.PerLinkHandler
uses gw.api.policy.refresh.relink.IgnoreIfNotMatchedLinkHandler

class PolicyEntity_ExtRelinkHandler extends PerLinkHandler<PolicyEntity_Ext> {

    construct() {

        register(PolicyEntity_Ext, "PolicyEntity_Ext",
            new IgnoreIfNotMatchedLinkHandler<PolicyEntity_Ext>())
    }

}
```

The `CustomRelinkerTypes` property getter returns a map. In the file `PolicyRefreshConfigurationBase`, add a row like the following:

```
override property get CustomRelinkerTypes() : Map<IEntityType, Class<RelinkHandler>> {

    return {
        ...
        PolicyEntity_Ext -> PolicyEntity_ExtRelinkHandler,
        ...
    }
}
```

#### See also

[“Policy Refresh Relinking Details” on page 552](#)

# Other Integration Topics



# Archiving Integration

ClaimCenter supports archiving a claim as a serialized stream of data. You can store the serialized data in files, in a document storage system, or in a database as a binary large object. Each serialized stream is a XML document.

This topic includes:

- “Overview of Archiving Integration” on page 575
- “Archiving Storage Integration Detailed Flow” on page 579
- “Archive Retrieval Integration Detailed Flow” on page 582
- “Archive Source Plugin Utility Methods” on page 583
- “Upgrading the Data Model of Retrieved Data” on page 585

**See also**

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

---

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

---

## Overview of Archiving Integration

ClaimCenter supports archiving a claim as a serialized XML document. You can store the data in a file, in a document storage system, or in a database as a single binary large object.

There is one archiving-related plugin in ClaimCenter. For more information about the archive source plugin, see “Archive Source Plugin” on page 577.

## Archiving Integration and Archiving Eligibility Flow

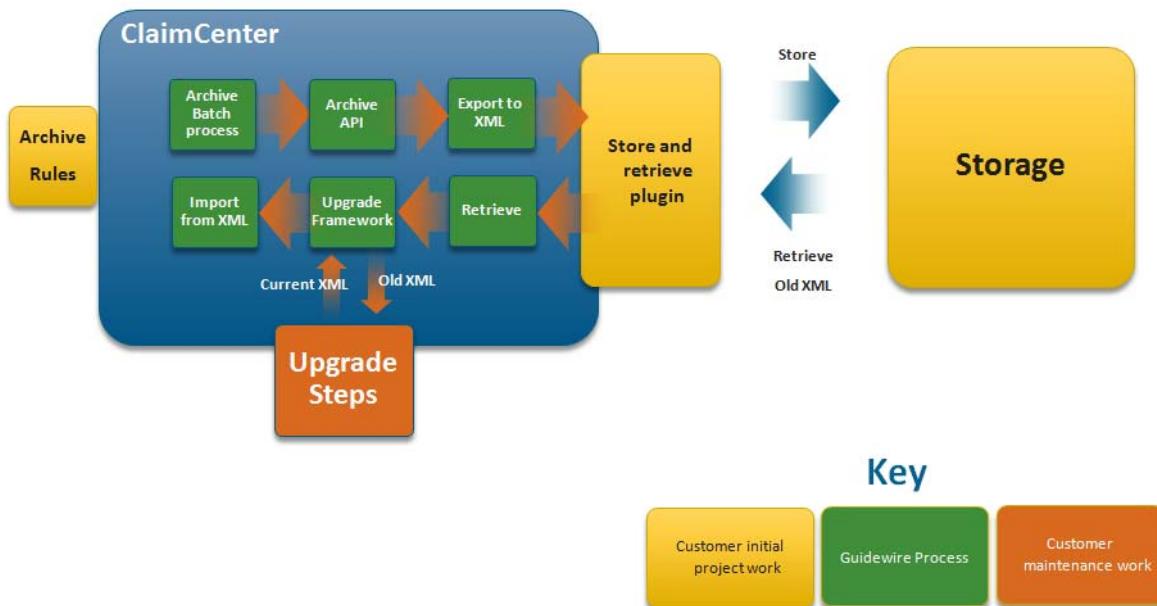
The high-level integration flow for archival storage:

1. The ClaimCenter archiving batch process runs.
2. ClaimCenter determines whether to skip or exclude claims from archiving. To determine what claims to skip or exclude, ClaimCenter runs the archiving rule set. ClaimCenter uses the following objects and properties to determine archive eligibility:
  - `ClaimInfo.ExcludedFromArchive` - a Boolean flag that if set to `true` prevents ClaimCenter from even attempting to archive the claim. If the value is `true`, ClaimCenter skips the claim and does not run the archiving rule set for this claim again until this flag is reset to `false`. For an excluded claim, the `ClaimInfo.ExcludedReason` property contains a `String` that describes the reason for the exclusion.
  - `Claim.DateEligibleForArchive` - a date that determines the earliest archive eligibility. If the current date is before this date, ClaimCenter does not attempt to archive the claim nor does the archiving rule set run for this claim.
3. ClaimCenter encodes the claim into an in-memory XML representation for one XML document.
4. ClaimCenter serializes each XML document.
5. ClaimCenter calls a customer-written archive source plugin to store the data.
6. The archive source plugin sends archive data to an external system. You implement this plugin to connect to your own archive source.
7. ClaimCenter deletes the claim and all its subobjects but preserves the associated info object (`ClaimInfo`). The info object is a small entity that represents the original claim but contains only high-level information for the claim.

The high-level integration flow for archiving retrieval:

1. The user identifies a claim to retrieve.
2. Synchronously, ClaimCenter finds the associated info object (`ClaimInfo`) for the claim.
3. ClaimCenter tells the archiving storage plugin to retrieve the XML format archive data from the external archive.
4. ClaimCenter de-serializes the XML data into an in-memory representation of the stored XML.
5. If the archive data is from an earlier data model of ClaimCenter, ClaimCenter next runs upgrade steps on the object. This includes built-in upgrade triggers as well as customer-written upgrade steps that extend built-in upgrade steps. For example, the customer upgrade steps might upgrade data in data model extension properties. For more information, see “Upgrading the Data Model of Retrieved Data” on page 585
6. ClaimCenter converts the temporary object into a real entity, upgrades the data model if necessary, and finally persists it to the database.

The following diagram summarizes the elements of this system.



## Archive Source Plugin

Customers using archiving must implement the `IArchiveSource` plugin interface. The responsibility of this plugin is storage and retrieval of archive data from the backing store. The backing store typically is on a different physical computer. The backing store could be anything.

Common choices for a backing store include:

- Files on a remote file system
- A document management system document
- A database row containing a large binary object

ClaimCenter includes a demonstration implementation of this plugin interface. The included implementation is `gw.plugin.archiving.ClaimInfoArchiveSource`. This demonstration implementation writes the archive data as files to the server's local file system.

The superclass of the built-in plugin implementation is the base class `gw.plugin.archiving.ArchiveSource`.

---

**IMPORTANT** Guidewire provides an archive source plugin implementation for demonstration purposes only. Do not use this in a production system. Implement your own archiving plugin that stores data on an external system.

---

To store archive data, ClaimCenter calls the plugin's `store` method synchronously, waiting for it to complete or fail.

To retrieve archive data, ClaimCenter calls the plugin's `retrieve` method. The only argument the `retrieve` method gets is a root info object, which is an instance of an entity type that implements the `RootInfo` interface. A root info object encapsulates a summary of one archived object. The root info object remains in the database after archiving most of the data.

In ClaimCenter, the root info object always is the `ClaimInfo` object. The `ClaimInfo` object describes a claim, including its archiving status. In the APIs or the documentation where you see the interface type `RootInfo`, the

this refers to `ClaimInfo`. Your plugin implementation must store enough reference information into the `ClaimInfo` object to determine how to retrieve any archived document from its backing store.

### Archive Source Plugin Methods and Archive Transactions

It is important to understand that ClaimCenter calls some archive source plugin methods inside an archive transaction and other methods outside a transaction. The behavior varies by plugin method. Be careful to understand any changes to database data outside the transaction, especially with respect to any error conditions.

For example, ClaimCenter always calls the plugin method `storeFinally` outside the main database transaction for the storage request. If the storage request fails, all ClaimCenter data changes never commit to the database. However, any changes you make during the call to `storeFinally` do persist to the database. For details, see “Archive Source Plugin Storage Methods” on page 580.

---

**WARNING** Be extremely careful that you understand potential failure conditions and the relationship between each archive method and associated database transactions.

---

For details for each plugin method, refer to reference information grouped by purpose of the methods:

- “Archive Source Plugin Storage Methods” on page 580
- “Archive Source Plugin Retrieve Methods” on page 583
- “Archive Source Plugin Utility Methods” on page 583

### Contact Info Objects and Other Archiving Objects That Persist

Most archiving documentation focuses on the main graph of objects and the one `RootInfo` object at the top of the graph. For ClaimCenter, the root info object is the `ClaimInfo` object.

However, there are other info objects, most notably the `ContactInfo` object. The `ContactInfo` object represents an archived contact. The application persists the `ContactInfo` object so that users can search for a claim by important contacts on the claim, even for archived claims.

For ClaimCenter, the application creates `ContactInfo` objects only on archive. ClaimCenter creates `ContactInfo` objects for the following:

- the one contact representing the insured
- one or more contacts represent the claimants

Internal code implements the creation of these items. You can add more `ContactInfo` objects or objects of custom entity types. To add additional info objects, create the objects in the `updateInfoOnRestore` and `updateInfoOnRestore` methods in the archive source plugin.

### Set Encryption Algorithm For Archived Objects with Encrypted Properties

The `config.xml` configuration parameter `DefaultXmlExportEncryptionId` specifies the unique encryption ID of an encryption plugin.

If archiving is enabled, the application uses that encryption plugin to encrypt any encrypted fields during XML serialization. For more about encryption unique IDs, see “Writing Your Encryption Plugin” on page 251.

---

**WARNING** To avoid accidentally archiving encrypted properties as unencrypted data, be sure to set the parameter `DefaultXmlExportEncryptionId`.

---

# Archiving Storage Integration Detailed Flow

ClaimCenter implements archive integration as a work queue.

## Archive Writers and Workers

ClaimCenter performs the following archiving tasks to create archive writers and workers:

1. First, the application confirms that the network archive source system is available. It is possible that the archive source is unavailable due to network problems or configuration issues. The archive source plugin returns its status in its `getStatus` method. If the archive source is unavailable for any reason, then the writer work process logs an information message but does not create archive work items.
2. If the archive source is available, the batch process called Archiving Item Writer finds objects that are potentially eligible for archiving. The batch process creates a work item as a row in the database for every object that is eligible for archiving.

In ClaimCenter, the only eligible object for archiving is a claim.

The archive worker process performs all the following steps to process the archive items.

3. Each worker process performs eligibility checks on each work item. Some eligibility criteria is defined in internal code.

For example, a claim is ineligible if it has aggregate limits, or is a payee on a bulk invoice.

Additionally, ClaimCenter calls the rule set Default Group Claim Archiving Rules. That rule set contains additional eligibility checks. You can enable, disable, delete, or modify eligibility checks to meet your business requirements.

4. For each work item, ClaimCenter performs the following steps all within a single database transaction:

- a. The worker copies some internal properties from the `Claim` to the `ClaimInfo` object.
- b. The worker compares all data model extension properties on `Claim` and `ClaimInfo` entities. If any extension properties have the same name and compatible types, then the worker copies those property values from the `Claim` to the `ClaimInfo`.
- c. The worker calls the `IArchiveSource` plugin method `updateInfoOnStore`. It takes a single argument, which is a `ClaimInfo` object. In the plugin definition, this parameter is declared as an object that implements the `RootInfo` interface. A `ClaimInfo` object encapsulates information about a single claim including its archive-related status information. If you need to add or modify properties on `ClaimInfo` in addition to properties mentioned in *step b*, set these properties in your `updateInfoOnStore` method.

- d. The worker *tags* the archived root object (the claim) specified in the work item. Next, the worker recursively tags all entities in the domain graph whose parent object was tagged.

The process of tagging includes setting the `ArchivePartition` property on the root object to a non-null value. The tagging process helps the application determine what data to delete at the end of the archiving process.

- e. The worker internally generates a structure that represents an XML document for the archived `Claim` and its subobjects.

- f. The worker serializes the XML structure into a stream of XML data as bytes. The worker outputs these bytes as an `java.io.InputStream` object.

- g. The worker deletes most of the archived data. However, ClaimCenter does **not** delete the `ClaimInfo` object. This info object remains in the database. It summarizes the archived claim including its archiving status.

- h. The worker calls the archive source plugin `store` method. The first argument to the method is the input stream that contains the serialized XML document. The second argument is the `ClaimInfo` object that

encapsulates archive-related properties of the claim. In a real production system, the archive source plugin would send the data to an external system along with any related metadata from the `ClaimInfo` object. For example, the archive source might contain details that link the archive data to that `ClaimInfo` object.

5. The worker commits the database transaction. This ends the database transaction for the preceding database changes. If any changes before this step threw an exception, all changes in this entire transaction are rolled back.
6. As the last step in the archive process for each work item, the worker calls the archive source plugin method `storeFinally`. ClaimCenter calls this method independent of whether the archive transaction succeeded. For example, if some code threw an exception and the archive transaction never committed, ClaimCenter still calls this method. Use this method to do any final clean up. To make any changes to entity data, you must run your Gosu code within a call to `Transaction.runWithNewBundle(block)`. That API sets up a bundle for your changes, and then commits the changes to the database in one transaction. For details, see “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide*.

#### See also

- “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide*
- “Configuring Distributable Work Queues” on page 128 in the *System Administration Guide*
- “Archiving and Restoring Claims from External Systems” on page 156

### Error Handling During Archive

If the archive process fails in any way, consult both of the following:

- application logs
- the `Archive Info` page within the `Server Tools` page.

To view the `Archive Info` page, you must set the `config.xml` configuration parameter `ArchiveEnabled` to `true`.

### Archive Source Plugin Storage Methods

ClaimCenter calls various `IArchiveSource` methods during the archive store process. The following list of methods defines the main storage methods:

- `prepareForArchive(info : RootInfo)`
- `updateInfoOnStore(info : RootInfo)`
- `store(graph : InputStream, info : RootInfo)`
- `storeFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)`

For ClaimCenter, when you see the interface type `RootInfo` in documentation or code in Studio, this always refers to a `ClaimInfo` entity instance.

#### `prepareForArchive(info : RootInfo)`

The method call for `prepareForArchive` occurs outside the archive transaction. Thus:

- ClaimCenter does not roll back any changes if the archiving operation fails.
- ClaimCenter does not commit any changes automatically if the archiving operation succeeds.

You use this method for the (somewhat unusual) case in which you want to prepare some data regardless of whether the domain graph actually archives successfully. The method has no transaction of its own. If you want to update data, then you must create a bundle and commit that bundle yourself.

In the demonstration plugin implementation, this method does nothing.

**updateInfoOnStore(info : RootInfo)**

ClaimCenter calls the `updateInfoOnStore` method after the application preps the data for archive but before calling the `store` method. Despite the name of the `updateInfoOnStore` method, this method is not the only place to modify the info object. See the following topic on the `store` method below.

ClaimCenter calls this method inside the archiving transaction. This enables you to make additional updates on `RootInfo` objects.

For example, use this method to write logic to update calculated fields on the `ClaimInfo` object that ClaimCenter uses for aggregate reports or searches.

As the method call is inside the archiving transaction, if that transaction fails, then ClaimCenter does not commit any updates made during the call to the database.

Depending on what kind of action you need to do in `updateInfoOnStore`, you might need similar or complementary changes in the plugin methods `updateInfoOnRetrieve` and `updateInfoonDelete`.

In the demonstration plugin implementation, this method does nothing.

---

**WARNING** You can modify the root info object (`ClaimInfo`) object in the `updateInfoOnStore` method. However, it is dangerous and unsupported to modify any other objects in the claim graph in this method.

---

**store(graph : InputStream, info : RootInfo)**

The `store` method is the main configuration point for taking XML data (in the form of an `InputStream`) and transferring it to an external archive source system.

ClaimCenter calls the `store` method inside the archiving transaction, after deleting rows from the database, but before performing the database commit. Your implementation of this method must store the archive XML document.

During the method call, the archive process passes in the `java.io.InputStream` object that contains the generated XML document. This is the data that your archive source plugin must send to the external archive backing store.

The archive process passes in the `RootInfo` object to the plugin so that it can insert or update additional reference information to help with restore. Generally speaking, this is not the best place to make changes to the root info object. Normally, it is best to make those changes in the `updateInfoOnStore` method.

Generally speaking, it is best to change no entity data at all in the `store` method. Within the `store` method (or any other part of the data base transaction), the only properties that are safe to change are the non-array properties on the `RootInfo` object. The only safe reason to change entity data in the `store` method is to add a unique retrieval ID to an extension property on the `RootInfo` object. Only make this change from the `store` method if your external archive source requires this data for retrieval and that ID is only available after sending data.

If your plugin is unable to store the XML document, then ClaimCenter expects the plugin to throw an error. ClaimCenter treats this error as a storage failure and rolls back the transaction. The transaction rollback also rolls back any changes to objects that you set up in your `updateInfoOnStore` method call.

**storeFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)**

ClaimCenter calls this method outside the archiving transaction after completing that transaction. The `finalStatus` parameter value indicates if the archiving delete operation was successful. Check this value. This allows the archive storage system to reverse any changes that were not part of the transaction in the rare case in which the delete transaction fails.

To change any data, you must perform any entity instance modification within a call to `Transaction.RunWithNewBundle(block)`. See “Running Code in an Entirely New Bundle” on page 342 in the

### Gosu Reference Guide.

If this method is called with errors, `finalStatus` is something other than `success`. The `cause` parameter contains a list of `String` objects that describe the cause of any failures. The `String` objects are the text for the exception cause hierarchy.

This permits two different design strategies:

- The recommended technique is to send and commit your data in the external source in the `store` method. On failure conditions, in `storeFinally` you can **back out** of any changes in the external data store. For example, delete any temporarily-created files.
- Alternatively, you can perform a two stage commit. In other words, send the data to the external system in the `store` method but finalize it in `storeFinally` if and only if `finalStatus` indicates success.

It is important to be careful about what kinds of work you do in the `storeFinally` method to properly handle error conditions. If the `storeFinally` method throws an exception, the application logs the exception but the main database transaction already completed. Be sure to carefully catch any exception and handle all error conditions. There is no rollback or recovery that the application can perform if `storeFinally` does not complete its actions due to errors or exceptions within the `storeFinally` implementation.

## Archive Retrieval Integration Detailed Flow

As one part of the retrieve process, *retrieval* is the step of getting data from an archive source back into the main database. It is up to you provide the necessary hooks into the retrieve process.

For example, in ClaimCenter, with archiving enabled, the **Advanced Search** screen displays a **Retrieve from Archive** button if the search query returns an archived item.

Within ClaimCenter, archive document retrieval has the following overall flow:

1. The user identifies an object to retrieve.
2. Synchronously, the retrieve request work queue finds items to retrieve
3. ClaimCenter calls the `IArchiveSource` plugin method called `retrieve`. ClaimCenter passes this method a reference to a `RootInfo` object. Your plugin implementation is responsible for returning the object binary data that describes the XML data that was originally stored. Your plugin implementation must provide this data as an `InputStream` object.
4. ClaimCenter clears the data on `ClaimInfo` that it copied from the `Claim` object during the initial archive process.
5. ClaimCenter creates any data that relates to the retrieve. For example, notes and history events.
6. ClaimCenter performs any upgrade steps defined for the data. See “Upgrading the Data Model of Retrieved Data” on page 585.
7. ClaimCenter calls `IArchiveSource.updateInfoOnRetrieve`. This method gives you a chance to perform additional operations in the same bundle as the retrieval operation, after ClaimCenter recreates all the entities but before the commit.  
*In the demonstration plugin*, this method sets the `Claim.DateEligibleForArchive` property to a date based on the `DaysRetrievedBeforeArchive` configuration parameter. This ensures that the application does not immediately archive the object again but instead waits until it is eligible again.
8. ClaimCenter commits the bundle.
9. ClaimCenter calls `IArchiveSource.retrieveFinally`. It is up to you to decide what additional handling the retrieved data requires, if any. For example, you can use this method to perform final clean up on files in the archive source. Guidewire does **not** recommend, nor does it support, the use of this method to make changes to data after you retrieve it. Any attempt to commit these changes in the `retrieveFinally` method invokes

the Preupdate and Validation rules. This is undesirable and unsupported. As ClaimCenter commits the main transaction for the retrieve process, it does not run these rules. Therefore, it is possible, for example, to retrieve objects that do not pass validation rules. Committing additional changes in `retrieveFinally` could fail validation, causing only those changes to be rolled back, not the entire retrieve request.

---

**WARNING** Do not changes to the retrieved data in `retrieveFinally`.

---

Before you delete the returned XML document, first consider whether it is important to compare what the archived item looked like initially with a subsequent archive of the same item.

If the retrieval process does not complete successfully, then consult the application logs as well as the [Server Tools → Archive Info](#) page. If you cannot view the [Archive Info](#) page, you must set the configuration parameter `ArchiveEnabled` to `true` in `config.xml`.

#### See also

- “Configuring Claim Archiving” on page 573 in the *Configuration Guide*
- “Archiving and Restoring Claims from External Systems” on page 156

## Archive Source Plugin Retrieve Methods

ClaimCenter calls the following `IArchiveSource` methods during the archive retrieval lifecycle:

- `retrieve(info : RootInfo) : InputStream`
- `updateInfoOnRetrieve(info : RootInfo)`
- `retrieveFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)`

For ClaimCenter, when you see the interface type `RootInfo` in documentation or code in Studio, this always refers to a `ClaimInfo` entity instance.

### `retrieve(info : RootInfo) : InputStream`

The retrieve method is the main configuration point for retrieving XML data from the external archive source. You must return the data as an `InputStream`. The archive process passes the `RootInfo` object to the plugin method to assist your plugin implementation to identify the correct data in the external system.

For archive retrieval to work correctly, the `RootInfo` object must store enough information to determine how to retrieve the XML document from the backing store.

### `updateInfoOnRetrieve(info : RootInfo)`

The retrieval process calls this method within the retrieval transaction. This occurs after the archive process populates the bundle of the `RootInfo` object with the objects produced by parsing the XML returned by the earlier call to `retrieve(RootInfo)`.

From within the `updateInfoOnRetrieve` method, you can modify the new objects in the object graph as well as the `RootInfo` object.

### `retrieveFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)`

The retrieval process calls this as the last retrieval step, outside of the retrieve transaction. The `finalStatus` parameter tells if the retrieve was successful. Use this plugin method to perform other actions on the storage. For example, deleting the file, or marking its status.

## Archive Source Plugin Utility Methods

ClaimCenter calls these utility methods as needed during the archive process:

- `updateInfoonDelete(info : RootInfo) : List<Pair<IEntityType, List<Key>>`

- `delete(info : RootInfo)`
- `retrieveSchema(platformMajor : int, platformMinor : int, appMajor : int, appMinor : int, extension : int) : InputStream`
- `storeSchema(platformMajor : int, platformMinor : int, appMajor : int, appMinor : int, extension : int, schema : InputStream)`

For ClaimCenter, when you see the interface type `RootInfo` in documentation or code in Studio, this always refers to a `ClaimInfo` entity instance.

Your plugin must implement all of these methods. Refer to the demonstration plugin for guidance.

#### `updateInfoonDelete(info : RootInfo) : List<Pair<IEntityType, List<Key>>>`

At some later time after archiving, a user or batch process may decide to delete the archived object entirely, including the data that remained in the local database. This is called *purgung*. During purging, you may want to delete the archived data on the external store, but you are not required to do so.

During purging, ClaimCenter calls the `updateInfoonDelete` method to determine additional objects that ClaimCenter must also delete if it deletes the object represented by `RootInfo`. ClaimCenter never calls this method during the main archiving step. The delete process calls this method within the transaction in which it deletes the `RootInfo` object and related entities from the active database.

If your archive source plugin creates extension entity objects in the `updateInfoOnArchive` method that link to the `RootInfo` object, you must return these objects from `updateInfoonDelete`. This causes the application to delete those extension entity instances.

---

**WARNING** Declaring these relationships in `updateInfoonDelete` is important because you may have created other objects that ClaimCenter cannot discover by looking at the foreign key references on the `RootInfo` object.

---

The return type is a list of pair (`Pair`) objects. Each pair object is parameterized on:

- an entity type
- A list of Key objects. The Key contain the relevant `object.ID` value for that object.

Typically, it is inappropriate to make entity changes in this method. If this method does make changes, the application commits those changes before calling the plugin method called `delete`. If you create objects in the `updateInfoOnArchive` that link to the `RootInfo` entity, return those objects in the return results of this method.

Generally speaking, if you mark an object to delete by returning it from this method, you also return all objects that link to those objects.

The order of the types and IDs in the list is important. Deletion happens in the order in the list and deletion is permanent. If object A links to object B, which links to the `RootInfo` object, you must return A before B.

Typically you would determine the correct order of types and then delete all objects of that object type all at once.

However, although usually unnecessary, it is supported to return one entity type more than once in the return list. Theoretically in complex configuration edge cases, this might be necessary to enforce proper ordering of the deletion process. For example, this allows you to enforce the following deletion order:

1. delete object ID 100 of type Type1.
2. delete object ID 900 of type Type2.
3. delete object ID 101 of type Type1, a type already mentioned.

**delete(info : RootInfo)**

In the `delete` method, your plugin implementation must delete from the backing store the data identified by the specified `RootInfo` object. ClaimCenter calls the `delete` method during the purge process.

**IMPORTANT** In your implementation of the `delete` method, you must delete any documents and associated document metadata linked to the archived object.

The delete process calls this method *after* ClaimCenter deletes associated objects return from `updateInfoonDelete`. See earlier in this topic for details of `updateInfoonDelete`.

**retrieveSchema(platformMajor : int, platformMinor : int, appMajor : int, appMinor : int, extension : int) : InputStream**

This method retrieves the XSD associated with a *data model + version number* combination as a `FileInputStream`. The XSD describes the format of the XML files for that version.

**storeSchema(platformMajor : int, platformMinor : int, appMajor : int, appMinor : int, extension : int, schema : InputStream)**

This method stores the XSD associated with the specified *data model + version number* combination. The XSD describes the format of the XML files for that version.

## Upgrading the Data Model of Retrieved Data

After the archiving plugin retrieves archived data, ClaimCenter deserializes the data into an in-memory object that represents the entity instance. This in-memory object is **not** yet a real entity instance. ClaimCenter loads the data into an object called an archived entity, represented by the interface `IArchivedEntity`. The application runs upgrade trigger steps on the object. Upgrade steps such as adding a column happen on this archived entity object.

Only after all upgrade triggers run, ClaimCenter attempts to convert the data to a real entity that matches the current data model of the application. If this process fails, the entire restore from archive fails. If it succeeds, the application commits the new data to the main application database.

ClaimCenter also links the retrieved data with its info object (`ClaimInfo`).

For more about writing upgrade triggers, see “Using the `IDatamodelUpgrade` Plugin” on page 141 in the *Upgrade Guide*.



# Custom Batch Processes

You can add custom batch processes to perform background tasks. You can view batch processes (and work queues) in the Internal Tools page in ClaimCenter.

---

**IMPORTANT** For another type of customer-defined background process, see “What are Startable Plugins?” on page 271.

---

This topic includes:

- “Creating a Custom Batch Process” on page 587
- “Custom Batch Processes and MaintenanceToolsAPI” on page 594

## Creating a Custom Batch Process

You can create batch processes to perform background tasks. ClaimCenter users can start, stop, or track progress of the batch process in the user interface.

Each batch process must provide a class that implements the `BatchProcess` interface. Although `BatchProcess` is an interface, it is **not** technically defined as a ClaimCenter *plugin*. You do not register it in the Studio Plugins editor. Also, other characteristics of plugins do not apply to this class, such plugin parameters and persistence rules. You need to ensure that ClaimCenter creates instances of your new class. To do this, there is a plugin interface called `IProcessesPlugin` that creates all the batch process objects. The `IProcessesPlugin` implementation simply instantiates all batch processes.

### To create a custom batch process

1. In Studio, edit the `BatchProcessType` typecode.
  - a. Add an element to represent your own batch process.
  - b. In the typecode editor, for the new typecode, add one or more categories as they apply for your batch process. Use the `BatchProcessTypeUsage` typelist with following values:  
`UIRunnable` - the process is runnable both from the user interface and from the web service APIs.

**APIRunnable** - the process is only runnable from web service APIs.

**Schedulable** - the process can be scheduled.

**MaintenanceOnly** - only run the process if the system is at maintenance run level.

For more information about adding categories to typecodes, see “Working with TypeLists in Studio” on page 275 in the *Configuration Guide*. Note that you must add at least one category or else your batch process cannot run.

**Note:** Creating this new typecode adds support for the new batch process within `BatchProcessInfo.pcf` and `IMaintenanceToolAPI`.

2. Create a class that extends the `BatchProcessBase` class (`gw.processes.BatchProcessBase`). This base class implements the `BatchProcess` interface. The only required method you must override is the `dowork` method, which takes no arguments. Do your batch processes work in this method. For more information about this class, including some optional methods to override, see “Batch Process Implementation Using the Batch Process Base Class” on page 589.
3. Modify the application code to create your new batch process. In the built-in implementation, there is no code that instantiates your new batch process. When ClaimCenter needs to create a new batch process, it calls the registered plugin for the `IProcessesPlugin` interface. There is a built-in `IProcessesPlugin` implementation that adds all the built-in batch processes. To add your batch process type, either reimplement the existing `IProcessesPlugin` implementation and base yours on the code in the built-in implementation. The existing implementation looks like:

```
package gw.plugin.processes
uses gw.plugin.processing.IProcessesPlugin
uses gw.processes.BatchProcess
uses gw.util.ClaimHealthCalculatorBatch
uses gw.util.PurgeMessageHistory
use gw.util.CatastropheClaimFinderBatch

@Export
class ProcessesPlugin implements IProcessesPlugin {

    construct() {
    }

    override function createBatchProcess(type : BatchProcessType, arguments : Object[]) : BatchProcess {
        switch(type) {
            case BatchProcessType.TC_CLAIMHEALTHCALC:
                return new ClaimHealthCalculatorBatch();
            case BatchProcessType.TC_PURGEMESSAGEHISTORY:
                return new PurgeMessageHistory(arguments);
            case BatchProcessType.TC_CATASTROPHECLAIMFINDER:
                return new CatastropheClaimFinderBatch(arguments);
            default:
                return null
        }
    }
}
```

Add a new case statement to create your batch process, based on the typecode code that you created in the `BatchProcessType` typelist. For example, suppose you added the ABC typecode and your batch process class is `MyProcess`. Add the following lines to the `switch` statement (optionally passing the `arguments` array passed to `createBatchProcess`):

```
case "ABC":
    return new MyProcess(arguments);
```

If you already implemented this plugin because you already added a batch process, if you add more batch processes just add more case statements as needed.

4. In Studio, in the Plugins editor, register your new plugin for the `IProcessesPlugin` plugin interface. If you already implemented this plugin because you already added a batch process, if you add more batch processes just add more case statements as needed and skip this step.
5. If you want to schedule your new batch process, add the entry to the XML file `scheduler-config.xml`

## Batch Process Implementation Using the Batch Process Base Class

Each batch process must have an implementation of the `BatchProcess` interface, which defines the contract with ClaimCenter. The easiest approach is to write a new Gosu class that extends the batch process base class: `gw.processes.BatchProcessBase`. This base class defines the basic implementation of the `BatchProcess` interface.

**Note:** Although `BatchProcess` is an interface, it is not technically defined as a ClaimCenter *plugin*. You do not register it in the Studio Plugins editor. Also, other characteristics of plugins do not apply to this class, such as plugin parameters and persistence rules. However, you do need to specially create instances of your new class. There is a plugin interface called `IProcessesPlugin` that you must implement to use batch processes. The `IProcessesPlugin` implementation simply instantiates **all** batch processes. For more information, see “Creating a Custom Batch Process” on page 587.

Strictly speaking, the only required method for you to override is the `dowork` method, which takes no arguments. You can override other methods if you need to, but the base class defines meaningful defaults.

It is critical to note that the `dowork` method does not have a bundle to track the database transaction. If you want to modify any data, you must use the `Transaction.RunWithNewBundle(...)` API to create a bundle.

---

**IMPORTANT** To modify entity data in your batch process, you must use the `RunWithNewBundle` API. For more information about creating bundles, see “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide*.

---

Ensure that your main `dowork` method frequently checks the `TerminateRequested` flag. If it is `true`, exit from your code. For example, if you are looping across a database query, exit from the loop. For more information, see “Request Termination” on page 589.

---

**IMPORTANT** ClaimCenter calls the `requestTermination` method in a different thread from the thread that runs your batch process `dowork` method.

---

The following subtopics list other property getters or methods that you can call or override.

### Check Initial Conditions

ClaimCenter calls the batch process `checkInitialConditions` method to determine whether to create and start the batch process. If it returns `true`, ClaimCenter starts the batch process. If the initial conditions are not met, this method must return `false`. If you return `false`, ClaimCenter skips this batch process right now. The batch process schedule defines the next time that ClaimCenter calls this method. The batch process base class always returns `true`. Override this method if you need to customize this behavior. For example, you could check if system conditions are necessary for this batch process to run, such as the server run level.

### Request Termination

If a user clicks the `Stop` button on the Batch Process Info page, this request a batch process to terminate. ClaimCenter calls the batch process `requestTermination` method to terminate a batch process if possible.

Your batch process must shut down any necessary systems and stop your batch process if you receive this message. If you cannot terminate your batch process, return `false` from this method. The batch process base class always returns `false`, which means that the request did not succeed. The base class also sets an internal `TerminateRequested` flag that you can check to see if a terminate request was received.

---

**IMPORTANT** ClaimCenter calls the `requestTermination` method in a different thread from the thread that runs your batch process `dowork` method.

---

For typical implementations, use the following pattern:

- Override the `requestTermination` method and have it return `true`. When the user requests termination of the batch process, the application calls your overridden version of the method.
- Ensure that your main `doWork` method **frequently** checks the `TerminateRequested` flag. In your `doWork` code, exit from your code if that flag is set. For example, if you are looping across a database query, exit from the loop and return.

Override the `requestTermination` method and return `false` if you genuinely **cannot** terminate the process soon. Be warned that if you do this, you risk the server shutting down before your process completes.

**WARNING** Although you can return `false` from `requestTermination`, Guidewire strongly recommends you design your batch process so that you actually can terminate the action. It is critical to understand that returning either value does **not** prevent the application from shutting down or reducing run level. ClaimCenter delays shutdown or change in run level for a period of time. However, eventually the application shuts down or reduces run level independent of this batch process setting. For maximum reliability and data integrity, design your code to frequently check and respect the `TerminateRequested` property.

ClaimCenter writes a line to the system log to indicate whether the batch process says it could terminate. In other words, the log line includes the result of your `requestTermination` method.

If your batch process can only run one instance at a time, returning `true` does **not** remove the batch process from the internal table of running batch processes. This means that another instance cannot run until the previous one completes. For more information about exclusive-running batch processes, see “Exclusive” on page 590.

### Exclusive

The property getter for the `Exclusive` property for a batch process determines whether another instance of this batch process can start while this process is running. The base class implementation always returns `true`. Override this method if you need to customize this behavior. This value does not affect whether other batch process classes can run. It only affects the current batch process class.

For maximum performance, be sure to set this `false` if possible. For example, if your batch process takes arguments in its constructor, it might be specific to one entity such as only a single `Claim` entity. If you want to permit multiple instances of your batch process to run in parallel, you must ensure your batch process class implementation returns `false`. For example,

```
override property get Exclusive() : boolean {  
    return false  
}
```

**Note:** For Java implementations, you must implement this property getter as the method `isExclusive`, which takes no arguments.

### Description

The `getDescription` method gets the batch type’s description. The base class gets this string from the batch type typecode description. Override this method if you need to customize this behavior.

### Detail Status

The property getter for the `DetailStatus` property gets the batch type’s detailed status. The base class defines a default simple implementation. Override the default property getter to provide more useful detail information about the status of your batch process for the Administration user interface. The details might be important if your class experiences any error conditions.

**Note:** For Java implementations, you must implement this property getter as the method `getDetailStatus`, which takes no arguments.

## Progress Handling

The Progress property in the batch process dynamically returns the progress of the batch process. The base class returns text in the form "x of y" where x is the amount of work completed and y is the total amount of work. If y is unknown, returns just "x". These values are determined from the OperationsExpected and OperationsCompleted properties. From Java, this is the `getProgress` method.

The following list includes related properties and methods you can use that the base class implements:

- OperationsExpected property - a counter for how many operations are expected, as an `int` value. From Java this counter is the `getOperationsExpected` method.
- OperationsCompleted property - a counter for how many operations are complete, as an `int` value. From Java this counter is the `getOperationsCompleted` method.
- incrementOperationsCompleted method - this no-argument method increments an internal counter for how many operations completed. For example, suppose your batch process iterates across entities of some type with special conditions and specially handle each found item. For each entity you modify, call this method once to track the progress of this batch process. The user interface can display this number or debugging code can track this number to see the progress. This method returns the current number of operations completed. For each entity for which you have an error condition, call this method once to track the progress of this batch process.
- OperationsFailed property - an internal counter for the number of operations that failed. From Java this counter is the `getOperationsFailed` method.
- incrementOperationsFailed method - this method increments an internal counter for how many operations failed. For example, suppose your batch process iterates across entities of some type with special conditions and specially handle each found item. For each entity for which you have an error condition, call this method once to track the progress of this batch process. The user interface can display this number or debugging code can track this number to see the progress. This method returns the current number of operations failed. You must also call the `incrementOperationsCompleted` method.

---

**IMPORTANT** For any operations that fail, call both the `incrementOperationsFailed` method and the `incrementOperationsCompleted` method.

- 
- Finished property - return a Boolean value to indicate whether the process completed. Completion says nothing about the errors if any. From Java, this is the `isFinished` method.

## Type

The base class maintains a Type property, which contains the batch process type, as a `BatchProcessType` type-code. From Java this counter is the `getType` method.

## Example Batch Process to Purge Workflows

The following Gosu batch process purges old workflows. It takes one parameter that indicates the number of days for successful processes. Pass this parameter in the constructor to this batch process class. In other words, your implementation of `IProcessesPlugin` must pass this parameter such as the new `MyClass(myParameter)` when it instantiates the batch process. If no parameter is missing or null, it uses a default system setting.

```
package gw.processes

uses gw.processes.BatchProcessBase
uses java.lang.Integer
uses gw.api.system.PLConfigParameters
uses gw.api.admin.WorkflowUtil

class PurgeWorkflows extends BatchProcessBase
{
    var _succDays = PLConfigParameters.WorkflowPurgeDaysOld.Value

    construct() {
        this(null)
    }
}
```

```

construct(arguments : Object[]) {
    super("PurgeWorkflows")
    if (arguments != null) {
        _succDays = arguments[0] != null ? (arguments[0] as Integer) : _succDays
    }
}

override function doWork() : void {
    WorkflowUtil.deleteOldWorkflowsFromDatabase( _succDays )
}
}

```

**IMPORTANT** Remember that if you want to modify entity data in your batch process, you must use the `runWithNewBundle` API. For more information about creating bundles, see “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide*.

### Example Batch Process to Enforce Urgent Activities

The following is a sample customer batch process. It implements a notification scheme such that any urgent request must be handled within 30 minutes. If it is not handled within that time range, the batch process notifies a supervisor. If still not resolved in 60 minutes, it sends a message further up the supervisor chain.

If there are no qualified activities, it returns false so that it will not create a process history. If there are items to handle, it increments the count. The application uses this count to display batch process status “n of t” or a progress bar. If there are no contact email addresses, the task fails and the application flags it as a failure.

This example checks `TerminateRequested` to terminate the loop if the user or the application requested to terminate the process.

In this Gosu example, it does not actually send the email. Instead it prints to the console. You can change this to use the real email APIs if desired.

```

package sample.processes
uses gw.processes.BatchProcessBase
uses java.util.Map
uses gw.api.util.DateUtil
uses java.lang.StringBuilder
uses java.util.HashMap
uses gw.api.profiler.Profiler

class TestBatch extends BatchProcessBase
{
    static var tag = new gw.api.profiler.ProfilerTag("TestBatchTag1","A sample tag")
    var work : ActivityQuery

    construct() {
        super( "TestBatch" )
    }

    override function requestTermination() :Boolean {
        super.requestTermination() // set the TerminationRequested flag
        return true // return true to signal that we will attempt to terminate in our doWork method
    }

    override function doWork() : void { // no bundle
        var frame = Profiler.push(tag);
        try {
            work = find(a in Activity where a.Priority == "urgent" and a.Status == "open"
                and a.CreateTime < DateUtil.currentDate().addMinutes( -30 ) )
            OperationsExpected = work.getCount()
            var map = new HashMap<Contact, StringBuilder>()
            for (activity in work) {
                if (TerminateRequested) {
                    return;
                }
                incrementOperationsCompleted()
                var haveContact = false
                var msgFragment = constructFragment(activity)
                haveContact = addFragmentToUser(map, activity.AssignedUser, msgFragment) or haveContact
                var group = activity.AssignedGroup
            }
        }
    }
}

```

```
haveContact = addFragmentToUser(map, group.Supervisor, msgFragment) or haveContact
if (activity.CreateTime < DateUtil.currentDate().addMinutes( -60 )) {
    while (group != null) {
        group = group.Parent
        haveContact = addFragmentToUser(map, group.Supervisor, msgFragment) or haveContact
    }
}
if (!haveContact) {
    incrementOperationsFailed()
    addFragmentToUser(map, User.util.UnrestrictedUser, msgFragment)
}
if (not TerminateRequested) {
    for (addressee in map.Keys) {
        sendMail(addressee, "Urgent activities still open", map.get(addressee).toString())
    }
}
finally {
    Profiler.pop(frame)
}
}

private function constructFragment(activity : Activity) : String {
return formatAsURL(activity) + "\n\t"
+ " Subject: " + activity.Subject
+ " AssignedTo: " + activity.AssignedUser
+ " Group: " + activity.AssignedGroup
+ " Supervisor: " + activity.AssignedGroup.Supervisor
+ "\n\t" + activity.Description
}

private function formatAsURL(activity : Activity) : String {
return "http://localhost:8080/cc/Activity.go(${activity.id})"

// TODO: you must ADD A PCF ENTRYPPOINT THAT CORRESPONDS TO THIS URL TO DISPLAY THE ACTIVITY.
}

private function addFragmentToUser(map : Map<String, StringBuilder>, user : User,
msgFragment : String) : boolean {
if (user != null) {
    var email = user.Contact.EmailAddress1
    if (email != null and email.trim().length > 0) {
        var sb = map.get(email)
        if (sb == null) {
            sb = new StringBuilder()
            map.put(email, sb)
        }
        sb.append(msgFragment)
        return true
    }
}
return false;
}

private function sendMail(contact : Contact, subject : String, body : String) {
var email = new Email()
email.Subject = subject
email.Body = "<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">\n" +
    "<html>\n" +
    "  <head>\n" +
    "    <meta http-equiv="content-type"\n" +
    "      content="text/html; charset=UTF-8"\n" +
    "    <title>${subject}</title>\n" +
    "  </head>\n" +
    "  <body>\n" +
    "    <table>\n" +
    "      <tr><th>Subject</th><th>User</th><th>Group</td><th>Supervisor</th></tr>" +
    body +
    "    </table>" +
    "  </body>" +
    "</html>"
email.addToRecipient(new EmailContact(contact))
EmailUtil.sendEmailWithBody(null, email);
}
}
```

To use this entry point, use the following PCF entry point in the file `ClaimCenter/modules/configuration/pcf/EntryPoints/Activity.pcf`:

```
<PCF>
<EntryPoint authenticationRequired="true" id="Activity" location="ActivityForward(actvtIdNum)">
<EntryPointParameter locationParam="actvtIdNum" type="int"/>
</EntryPoint>
</PCF>
```

Also include the following `ActivityForward` PCF file as the `ActivityForward.pcf` in each place in the PCF hierarchy that you would like it:

```
<PCF>
<Forward id="ActivityForward">
<LocationEntryPoint signature="ActivityForward(actvtIdNum : int)" />
<Variable name="actvtIdNum" type="int"/>
<Variable
 initialValue="find(a in Activity where a.ID == new Key(Activity, actvtIdNum))"
 name="actvt"
 type="Activity"/>
<ForwardCondition action="ClaimForward.go(actvt.Claim);
ActivityDetailWorksheet.goInWorkspace(actvt)"/>
</Forward>
</PCF>
```

**IMPORTANT** Remember that if you want to modify entity data in your batch process, you must use the `runWithNewBundle` API. For more information about creating bundles, see “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide*.

## Batch Process History

A built-in startable plugin creates a process history entry for every run of each batch process. Refer to the `ProcessHistory` entity in the Data Dictionary for details of its properties.

To get the current batch process history entity from a batch process instance, get its `ProcessHistory` property. From Java, this is the `getProcessHistory` method.

If there is no work or if the batch process constructor throws an exception, the application capture and logs those errors also.

In ClaimCenter, there is a batch process that purges process history entities when they are old.

## Custom Batch Processes and MaintenanceToolsAPI

If you use the `MaintenanceToolsAPI` web service to start a batch process, you can identify a batch process with the predefined strings as the command names. For custom batch processes, pass the code value of your batch process `BatchProcessType` typecode.

This feature requires your `BatchProcessType` typecode to have the category `UIRunnable` or `APIRunnable`. When you create your typecode in Studio, refer to the `Categories` tab in Studio to add new categories.

For more information about adding categories to typecodes, see “Working with Typelists in Studio” on page 275 in the *Configuration Guide*.

# Free-text Search Integration

ClaimCenter free-text search is an alternative to database search that can return results faster for certain search requests than database search. Free-text search depends on an external full-text search engine, the Guidewire Solr Extension. Free-text search provides two plugins that connect free-text search in ClaimCenter with the Guidewire Solr Extension.

This topic includes:

- “Free-text Search Plugins Overview” on page 595
- “Free-text Load and Index Plugin and Message Transport” on page 596
- “Free-text Search Plugin” on page 598

**See also**

- “Search Overview” on page 345 in the *Configuration Guide*
- “Free-text Search Setup” on page 85 in the *Installation Guide*

## Free-text Search Plugins Overview

ClaimCenter free-text search depends on two Guidewire plugins that connect ClaimCenter to a full-text search engine. The search engine is a modified form of Apache Solr in a special distribution known as the *Guidewire Solr Extension*. The Guidewire Solr Extension runs in a different instance of the application server than the instance that runs your ClaimCenter application.

The free-text search plugin interfaces are:

- `ISolrMessageTransportPlugin` – Called by the Indexing System rules in the Event Fired ruleset whenever contacts on claims change. The plugin extracts the changed data and sends it in an indexing document to the Guidewire Solr Extension for loading and incremental indexing.
- `ISolrSearchPlugin` – Called by the **Search → Contacts** search screen to send users’ criteria to the Guidewire Solr Extension and receive the search results.

ClaimCenter provides the plugin implementations and the Guidewire Solr Extension software. Do not try connecting the free-text plugins to your own installation of Apache Solr.

---

**IMPORTANT** Guidewire does not support replacing the plugin implementations that ClaimCenter provides with custom implementations to other full-text search engines. Guidewire supports connecting the free-text plugins only to a running instance of the Guidewire Solr Extension.

---

## Connecting the Free-text Plugins with the Guidewire Solr Extension

If you configure free-text search for external operation, the free-text plugins connect to the Guidewire Solr Extension through the HTTP protocol. The plugin implementations obtain the host name and port number for the Guidewire Solr Extension application from parameters in the `solrserver-config.xml` file. In the base configuration, the `port` parameter specifies the standard Solr port number, 8983. If you set up the Guidewire Solr Extension with a different port number, modify the `port` parameter to match your configuration.

In the base configuration, the `host` parameter specifies `localhost`, which generally is correct for development environments. For production environments however, Guidewire requires that you set up the application server instance for the Guidewire Solr Extension on a host separate from the one that hosts ClaimCenter. For production environments, you must modify the `host` parameter to specify the remote host where the Guidewire Solr Extension runs.

If you configure free-text search for embedded operation, the plugins connect to the Guidewire Solr Extension without using the HTTP protocol. With embedded operation, the Guidewire Solr Extension runs as part of the ClaimCenter application, not as an external application. With embedded operation, the plugins ignore any host name and port number parameters specified in `solrserver-config.xml`. Free-text search does not support embedded operation in production environments.

## Enabling and Disabling the Free-text Plugins

The free-text plugins are disabled in the base configuration of ClaimCenter. Even if you enable the free-text plugins, the implementations do not fully operate unless the `FreeTextSearchEnabled` parameter in `config.xml` is set to `true`. In addition, the `ISolrMessageTransportPlugin` requires you to enable the `CCSolrMessageTransport` message destination. After you enable the free-text plugins, use the `FreeTextSearchEnabled` parameter to toggle them on and off, along with other free-text resources.

## Running the Free-text Plugins in Debug Mode

You can run the free-text plugins in debug mode. With debug mode enabled, the plugins generate messages on the server console to help you debug changes to free-text search fields. The free-text plugin implementations have a debug plugin parameter that lets you enable and disable debug mode. You can enable debug separately for each plugin.

In the base configuration, the `debug` parameters are set to `true`. Whenever you use free-text search in a production environment, set the `debug` parameters for each plugin to `false`.

## Free-text Load and Index Plugin and Message Transport

ClaimCenter provides the free-text load and index message transport to send changed claim contact data to the Guidewire Solr Extension for loading and incremental indexing. In the base configuration of ClaimCenter, the plugin is disabled.

The primary components of this message transport are:

- A `MessageTransport` plugin implementation with the following qualities:

- The implementation class is `gw.solr.CCSolrMessageTransportPlugin`.
- The plugin name in the plugins registry in Studio is `SolrMessageTransportPlugin`
- This class implements the interface called `ISolrMessageTransportPlugin`, which is a subinterface of `MessageTransport` with additional methods. In the default configuration, in the plugins registry, this plugin implementation specifies its interface name as `ISolrMessageTransportPlugin` instead of `MessageTransport`.
- A messaging destination

Generally, you do not need to modify the `CCPSolrMessageTransportPlugin` implementation if you add or remove free-text search fields.

To edit the registry for the `CCSolrMessageTransportPlugin` interface, in the **Resources** pane of ClaimCenter Studio, navigate to **resources** → **gw** → **plugin** → **solr** → **ISolrMessageTransportPlugin**.

#### See also

- “[Messaging and Events](#)” on page 299

## Message Destination for Free-text Search

The implementation of `CCSolrMessageTransportPlugin` depends on the `CCSolrMessageTransport` message destination. If the message destination is not enabled or is enabled but not started, the plugin cannot send index documents to the Guidewire Solr Extension. Use the **Messaging** editor in Studio to enable the `PCSolrMessageTransport` destination. Use the **Event Messages** page on the **Administration** tab in the application to start and stop the destination.

If you enabled the free-text search feature, enable the message transport implementation. The implementation class is `gw.solr.CCSolrMessageTransportPlugin`, which is an implementation of the `MessageTransport` plugin interface.

The plugin itself is enabled in the plugins registry in the default configuration. However, you must enable the relevant messaging destination in the Messaging editor in Studio.

In the **Messaging** editor in Studio, ensure the following are set:

- **Enabled** – *checked*
- **Destination ID** – 69
- **Name** – `Java.MessageDestination.SolrMessageTransport.Policy.Name`
- **Transport plugin** – `ISolrMessageTransportPlugin`

**Note:** This is not the fully-qualified name. This is the name for the plugin in the Plugins registry.

- **Poll interval** – 1000
- **Events**
  - ??????

## Plugin Parameters

The plugin registry for the item with name `ISolrMessageTransportPlugin` has the following plugin parameters.

Plugin parameter	Description	Default value
<code>debug</code>	For development servers only, specifies whether to generate messages on the server console and in the server log to help debug changes to free-text search fields. For production, always set to true.	true
<code>commitImmediately</code>	Whether the Guidewire Solr Extension indexes and commits each new or changed index document before receiving and indexing the next one.  If you set this parameter to <code>false</code> , the Guidewire Solr Extension receives a batch of index documents from ClaimCenter before it indexes and commits them. You configure the batch size in the <code>autocommit</code> section of the <code>solrconfig.xml</code> file.	false

## Free-text Search Plugin

ClaimCenter provides the free-text search plugin `ISolrSearchPlugin` to send free-text search requests to the Guidewire Solr Extension and receive the search results. In the base configuration of ClaimCenter, the plugin is disabled. The plugin registry specifies the following Gosu implementation:

```
gw.solr.CCSolrSearchPlugin
```

If you add or remove free-text search fields from your configuration, you must modify the implementation of `CCSolrSearchPlugin`.

To edit the registry for the `ISolrSearchPlugin` interface, in the **Resources** pane of ClaimCenter Studio, navigate to **configuration** → **Plugins** → **gw** → **plugin** → **solr** → **ISolrSearchPlugin**.

### Plugin Parameters

The plugin registry for `ISolrSearchPlugin` has the following plugin parameters.

Plugin parameter	Description	Default value
<code>debug</code>	Whether to generate messages on the server console and in the server log to help debug changes to free-text search fields.	true
<code>fetchSize</code>	Determines the maximum number of query results that the Guidewire Solr Extension returns for each search request.	100

### See also

- “Modifying Free-text Search for Additional Fields” on page 369 in the *Configuration Guide*

# Servlets

You can define simple web servlets for your ClaimCenter application using the `@Servlet` annotation.

## Implementing Servlets

You can define simple web servlets inside your ClaimCenter application. You can define extremely simple HTTP entry points to custom code using this approach. These are separate from web services that use the SOAP protocol. These are separate also from the Guidewire PCF entrypoints feature.

Use this technique to define arbitrary Gosu code that a user or tool can call from a configurable URL. You can define a Gosu block that can determine from the URL whether your servlet owns the HTTP request.

Servlets provide no built-in object serialization or deserialization, such as is done in the SOAP protocol.

The ClaimCenter servlet implementation uses standard Java classes in the package `javax.servlet.http` to define the servlet request and response. The base class for your servlet must extend `javax.servlet.http.HttpServlet` directly, or extend a subclass of it.

Extending the base class `javax.servlet.http.HttpServlet` provides no inherent security for the servlet. By default users can trigger servlet code without authenticating. This is dangerous in a production system. ClaimCenter includes a utility class called `gw.servlet.ServletUtils` that you can use in your servlet to require authentication.

**WARNING** You must add your own authentication system for servlets to protect information and data integrity. Carefully read “[Implementing Servlet Authentication](#)” on page 601. If you have any questions about server security, contact Guidewire Customer Support.

### Creating a Basic Servlet

#### To create a basic servlet

1. Write a Gosu class that extends the class `javax.servlet.http.HttpServlet`.

## 2. Register the servlet class in the file:

ClaimCenter/configuration/config/servlets.xml

Add one <servlet> element that references the fully qualified name of your class, for example:

```
<servlets xmlns="http://guidewire.com/servlet">
  <servlet class="com.example.servlets.MyServletClass"/>
</servlets>
```

## 3. Add the @Servlet annotation on the line before your class definition.

You must pass arguments to the constructors to specify which URLs your servlet handles. There are multiple versions of the constructors for different use cases.

You provide a search pattern to test against the *servlet query path*. The servlet query path is every character after the computer name, the port, the web application name, and the word "/service". In other words, your servlet gets the servlet URL substring identified as *YOUR\_SERVLET\_STRING* in the following URL syntax:

- `@Servlet(pathPattern : String)`

Use this annotation constructor syntax for high-performance pattern matching, though with less flexibility than full regular expressions.

This annotation constructor declares the servlet URL pattern matching to use Apache

`org.apache.commons.io.FilenameUtils` wildcard syntax. Apache `FilenameUtils` syntax supports Unix and Windows filenames with limited wildcard support for ? (single character match) and \* (multiple character match) similar to DOS filename syntax. The syntax also supports the Unix meaning of the ~ symbol. Matches are always case sensitive.

- `@Servlet(pathMatcher : block( path : String ) : boolean)`

Use this annotation constructor syntax for highly flexible pattern matching.

You pass a Gosu block that can do arbitrary matching on the servlet query path. Performance depends greatly on what you do in your block. As a parameter in parentheses for the annotation, pass a Gosu block that takes a URL String. Write the block such that the block returns `true` if and only if the user URL matches what your servlet handles.

If you only use the `startsWith` method of the `String` argument, for example

`path.startsWith("myprefix")`, the servlet URL checking code is high performance.

You could do more flexible pattern matching with other APIs, such as full regular expressions. Using regular expressions lowers performance for high volumes of servlet calls. The following example uses regular expressions with the `matches` method on the `String` type:

```
@Servlet(\ path : String ->path.matches("/test(/.*)?"))
```

This example servlet responds to URLs that start with the text "/test" in the servlet query path, and optionally a slash followed by other text.

## 4. Override the `doGet` method to do your actual work. Your `doGet` method takes a servlet request object (`HttpServletRequest`) and a servlet response object (`HttpServletResponse`).

## 5. In your `doGet` method, determine what work to do using the servlet request object.

**WARNING** You must add your own authentication system for servlets to protect information and data integrity. Carefully read "Implementing Servlet Authentication" on page 601. If you have questions about server security, contact Guidewire Customer Support.

Important properties on the request object include:

- `RequestURI` – Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
- `RequestURL` – Reconstructs the URL the client used to make the request.
- `QueryString` – Returns the query string that is contained in the request URL after the path.
- `PathInfo` – Returns any extra path information associated with the URL the client sent when it made this request.

- **Headers** – Returns all the values of the specified request header as an Enumeration of String objects.

For full documentation on this class, refer to these Sun Javadoc pages:

<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html>  
<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html>

6. In your doGet method, write an HTTP response using the servlet response object. For example, the following simple response sets the content MIME type and the status of the response (OK):

```
resp.ContentType = "text/plain"
resp.setStatus(HttpServletRequest.SC_OK)
```

To write output text or binary data, use the `Writer` property of the response object. For example:

```
resp.getWriter.append("hello world output")
```

For example, the following simple Gosu servlet works although it provides no authentication:

```
package mycompany.test

uses gw.servlet.Servlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet

@Servlet( \ path : String ->path.matches("/test(/.*)?"))
class TestingServlet extends HttpServlet {

    override function doGet(req: HttpServletRequest, response: HttpServletResponse) {

        // ** SECURITY WARNING - FOR REAL PRODUCTION CODE, ADD AUTHENTICATION CHECKS HERE!

        // Trivial servlet response to test that the servlet responds
        response.setContentType = "text/plain"
        response.setStatus(HttpServletRequest.SC_OK)
        response.getWriter.append("I am the page " + req.PathInfo)
    }
}
```

Add an element in the `servlets.xml` file for your new servlet class. Run the QuickStart server at the command prompt: `gwcc dev-start` and test the servlet at the URL:

<http://localhost:PORT/cc/service/test>

Change the port number as appropriate for your application.

For a production servlet, you must add authentication. See “[Implementing Servlet Authentication](#)” on page 601.

## Implementing Servlet Authentication

Extending the base class `javax.servlet.http.HttpServlet` provides no inherent security for your servlet. By default, anyone can trigger servlet code without authenticating. This is dangerous in a production system, generally speaking. ClaimCenter includes a utility class called `gw.servlet.ServletUtils` that you can use in your servlet to require authentication.

**WARNING** You must add your own authentication for your servlet to protect information and data integrity. If you have any questions about server security, contact Guidewire Customer Support.

There are three methods in the `ServletUtils` class, each of which corresponds to a different source of authentication credentials. The following table summarizes each `ServletUtils` method. In all cases, the first argument is a standard Java `HttpServletRequest` object, which is an argument to your main servlet method `doWork`.

Source of credentials	ServletUtils method name	Description	Method arguments
Existing ClaimCenter session	<code>getAuthenticatedUser</code>	<p>If this servlet shares an application context with a running Guidewire application, there may be an active session token. If a user is currently logged into ClaimCenter, this method returns the associated <code>User</code> object.</p> <p>Always check the return value. The method returns <code>null</code> if authentication failed. For example:</p> <ul style="list-style-type: none"> <li>• there is no active authenticated session with correct credentials</li> <li>• the user exited the application</li> <li>• the session ID is not stored on the client</li> <li>• the session ServiceToken timeout has expired</li> </ul>	<ul style="list-style-type: none"> <li>• <code>HttpServletRequest</code> object</li> <li>• A Boolean value that specifies whether to update the date and time of the session</li> </ul>
HTTP Basic authentication headers	<code>getBasicAuthenticatedUser</code>	<p>Even if there is no active session, you can use HTTP Basic authentication. This method gets the appropriate HTTP headers for name and password and attempts to authenticate. You can use this type of authentication style even if there is an active session. This method forces creation of a new session. The method gets the headers to find the user name and password and returns the associated <code>User</code> object.</p> <p>Always check the return value. The method returns <code>null</code> if authentication failed.</p> <p>For login problems, this method might throw the exception  <code>gw.api.webservice.exception.LoginException</code>.</p>	<ul style="list-style-type: none"> <li>• <code>HttpServletRequest</code> object</li> </ul>
Arbitrary user/ login name pairs	<code>login</code>	<p>Use the <code>login</code> method to pass an arbitrary user and password as <code>String</code> values and authenticate with ClaimCenter. For example, you might use a corporate implementation of single-sign-on (SSO) authentication that stores information in HTTP headers other than the HTTP Basic headers. You can get the username and password and call this method. This method forces creation of a new session.</p> <p>Always check the return value. The method returns <code>null</code> if authentication failed.</p> <p>For login problems, this method might throw the exception  <code>gw.api.webservice.exception.LoginException</code>.</p>	<ul style="list-style-type: none"> <li>• <code>HttpServletRequest</code> object</li> <li>• Username as a <code>String</code></li> <li>• Password as a <code>String</code></li> </ul>

You can combine the use of multiple methods of `ServletUtils` in your code.

For example, a typical design pattern is to first call the `getAuthenticatedUser` method to test whether there is an existing session token that represents valid credentials. If the `getAuthenticatedUser` method returns `null`, attempt to use HTTP Basic authentication by calling the method `getBasicAuthenticatedUser`.

The following code demonstrates this technique in the doGet method and calls to a separate method to do the main work of the servlet.

```
package mycompany.test

uses gw.servlet.Servlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet

@Servlet( \ path : String ->path.matches("/test(/.*)?""))
class TestingServlet extends HttpServlet {

    override function doGet(req: HttpServletRequest, response: HttpServletResponse) {
        //print("Beginning call to doGet()...")

        // SESSION AUTH : Get user from session if the client is already signed in.
        var user = gw.servlet.ServletUtils.getAuthenticatedUser(req, true)
        //print("Session user result = " + user?.DisplayName)

        // HTTP BASIC AUTH : If the session user cannot be authenticated, try HTTP Basic
        if (user == null)
            try {
                user = gw.servlet.ServletUtils.getBasicAuthenticatedUser(req)
                //print("HTTP Basic user result = " + user?.DisplayName)
            }
            catch (e) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                    "Unauthorized. HTTP Basic authentication error.")
                return // Be sure to RETURN early because authentication failed!
            }

        if (user == null) {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                "Unauthorized. No valid user with session context or HTTP Basic.")
            return // Be sure to RETURN early because authentication failed!
        }

        // IMPORTANT: Execution reaches here only if a user succeeds with authentication.
        // Insert main servlet code here before end of function, which ends servlet request
        doMain(req, response, user )
    }

    // this method is called by our servlet AFTER successful authentication
    private function doMain(req: HttpServletRequest, response: HttpServletResponse, user : User) {
        assert(user != null)

        var responseText = "REQUEST SUCCEEDED\n"+
            "req.RequestURI: '${req.RequestURI}'\n" +
            "req.PathInfo: '${req.PathInfo}'\n" +
            "req.RequestURI: '${req.RequestURI}'\n" +
            "authenticated user name: '${user.DisplayName}'\n"

        // debugging in the console
        //print(responseText)

        // for output response
        response.ContentType = "text/plain"
        response.setStatus(HttpServletResponse.SC_OK)
        response.getWriter.append(responseText)
    }
}
```

#### To test session authentication servlet code

1. In Studio, create the `mycompany.test.TestingServlet` as shown earlier in this topic.
2. Register this servlet in `servlets.xml`.
3. Run the QuickStart server at the command prompt: `gwcc dev-start`  
However, do not yet log into the ClaimCenter application.
4. Test the servlet with no authentication by going to the URL:  
`http://localhost:PORT/cc/service/test`

You see a message:

```
HTTP ERROR 401  
Problem accessing /cc/service/test. Reason:  
Unauthorized. No valid user with session context or HTTP Basic.
```

5. Log into the ClaimCenter application with valid credentials.

6. Test the servlet with no authentication by going to the URL:

```
http://localhost:PORT/cc/service/test
```

You see a message:

```
REQUEST SUCCEEDED  
req.RequestURI: '/cc/service/test'  
req.PathInfo: '/test'  
req.RequestURI: '/cc/service/test'  
authenticated user name: 'Super User'
```

For testing of the HTTP Basic authentication, sign out of the ClaimCenter application to remove the session context. Next, re-test your servlet and use a tool that supports adding HTTP headers for HTTP Basic authentication.

## Alternative APIs for Authentication

Guidewire recommends using the `ServletUtils` APIs for managing authentication. See “[Implementing Servlet Authentication](#)” on page 601. If you use `ServletUtils`, you can use the session key if available and if not you can use HTTP Basic authentication headers or custom headers.

An alternative approach for authentication is to use the legacy class `AbstractGWAAuthServlet` to translate the security headers in the request and authenticate with the Guidewire server. There is a subclass called `AbstractBasicAuthenticationServlet`, which authenticates using HTTP Basic authentication. You can view the source code to both classes in Studio. These classes are provided primarily for legacy use because they do not support using more than one type of authentication at run time.

### Abstract Guidewire Authentication Servlet Class

To use the session key created from a Guidewire application that shares the same application context, you can write your servlet to extend the class `AbstractGWAAuthServlet`. You must override the following methods:

- `doGet` – Your main task is to override the `doGet` method to do your main work. ClaimCenter already authenticates the session key if it exists before calling your `doWork` method. For more information about the parameters in this method, see “[Creating a Basic Servlet](#)” on page 599.
- `authenticate` – Create and return a session ID
- `storeToken` – You can store the session token in this method, but you can also leave your method implementation empty.
- `invalidAuthentication` – Return a response for invalid authentication. For example:

```
override function invalidAuthentication( req: HttpServletRequest,  
                                      resp: HttpServletResponse ) : void {  
    resp.setHeader( "WWW-Authenticate", "Basic realm=\"Secure Area\""  
    resp.setStatus( HttpServletResponse.SC_UNAUTHORIZED )  
}
```

### Abstract HTTP Basic Authentication Servlet Class

The `AbstractBasicAuthenticationServlet` class extends `AbstractGWAAuthServlet` to support HTTP Basic authentication.

Your main task is to override the `doGet` method to do your main work. ClaimCenter already authenticates the using HTTP Basic authentication headers before calling your `doWork` method. For more information about the parameters in this method, see “[Creating a Basic Servlet](#)” on page 599.

Also, override the `isAuthenticationRequired` method and return `true` if authentication is required for this request.

The following example responds to servlet URL substrings that start with the string `/test/`. If an incoming URL matches that pattern, the servlet simply echoes back the `PathInfo` property of the response object, which contains the path.

```
package mycompany.test

uses gw.servlet.Servlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet
uses gw.api.util.Logger

@Servlet( \ path : String ->path.matches("/test(/.*)?"))
class TestingServlet extends gw.servlet.AbstractBasicAuthenticationServlet {

    override function doGet(req: HttpServletRequest, resp : HttpServletResponse) {

        print("servlet test url: " + req.RequestURI)
        print("query string: " + req.QueryString)

        resp.ContentType = "text/plain"
        resp.setStatus(HttpServletResponse.SC_OK)
        resp.getWriter.append("I am the page " + req.PathInfo)
    }

    override function isAuthenticationRequired( req: HttpServletRequest ) : boolean {

        // -- TODO -----
        // Read the headers and return true/false if user authentication is required
        // -----
        return true;
    }
}
```

This servlet responds to URLs with the word `test` in the service query path, such as the URL:

`http://localhost:8080/cc/service/test/is/this/working`

To test this, you must use a tool that supports adding HTTP Basic authentication headers for the username and password.

Note that the text `"/test"` in the URL is the important part that matches the servlet string. Change the port number and the server name to match your application.

Your web page displays the following:

`I am the page /test/is/this/working`

Use this basic design pattern to intercept any of the following:

- A single page URL
- An entire virtual file hierarchy, as shown in the previous example
- Multiple page URLs that are not described in traditional file hierarchies as a single root directory with subdirectories. For example, you could intercept URLs with the regular expression:

`"(/.*)?/my_magic_subfolder_one_level_down"`

That would match all of the following URLs:

`http://localhost:8080/cc/service/test1/my_magic_subfolder_one_level_down`  
`http://localhost:8080/cc/service/test2/my_magic_subfolder_one_level_down`  
`http://localhost:8080/cc/service/test3/my_magic_subfolder_one_level_down`



# Data Extraction Integration

ClaimCenter provides several mechanisms to generate messages, forms, and letters in custom text-based formats from ClaimCenter data. If an external system needs information from ClaimCenter about a claim, it can send requests to the ClaimCenter server by using the HTTP protocol.

This topic includes:

- “Why Gosu Templates are Useful for Data Extraction” on page 607
- “Data Extraction Using Web Services” on page 608

## Why Gosu Templates are Useful for Data Extraction

Incoming data extraction requests include what Gosu template to use and what information the request passes to the template. With this information, ClaimCenter searches for the requested data such as claim data, which is typically called the *root object* for the request. If you design your own templates, you can pass any number of parameters to the template. Next, ClaimCenter uses a requested template to extract and format the response. You can define your own text-based output format. See “Gosu Templates” on page 347 in the *Gosu Reference Guide*.

Possible output formats include:

- A plain text document with *name=value* pairs
- An XML document
- An HTML or XHTML document

You can fully customize the set of properties in the response and how to organize and export the output in the response. In most cases, you know your required export format in advance and it must contain dynamic data from the database. Templates can dynamically generate output as needed.

Gosu templates provide several advantages over a fixed, pre-defined format:

- With templates, you can specify the required output properties, so you can send as few properties as you want. With a fixed format, all properties on all subobjects might be required to support unknown use cases, so you must send all properties on all subobjects.

- Responses can match the native format of the calling system by customizing the template. This avoids additional code to parse and convert fixed-format data.
- Templates can generate HTML directly for custom web page views into the ClaimCenter database. Generate HTML within ClaimCenter or from linked external systems, such as intranet websites that query ClaimCenter and display the results in its own user interface.

The major techniques to extract data from ClaimCenter using templates are as follows:

- For user interaction, write a custom servlet that uses templates** – Create a custom servlet. A servlet generates HTML (or other format) pages for predefined URLs and you do not implement with PCF configuration. See “Servlets” on page 599. Your servlet implementation can use Gosu templates to extra data from the ClaimCenter database.
- For programmatic access, write a custom web service that uses templates** – Write a custom web service. Custom web services let you address each integration point in your network. Follow these design principles:
  - Write a different web service API for each integration point, rather than writing a single, general purpose web service API.
  - Name the methods in your web service APIs appropriately for each integration point. For example, if a method generates notification emails, name your method `getNotificationEmailText`.
  - Design your web service APIs to use method arguments with types that are specific to each integration point.
  - Use Gosu templates to implement data extraction. However, do not pass template data or anything with Gosu code directly to ClaimCenter for execution. Instead, store template data only on the server and pass only typesafe parameters to your web service APIs.

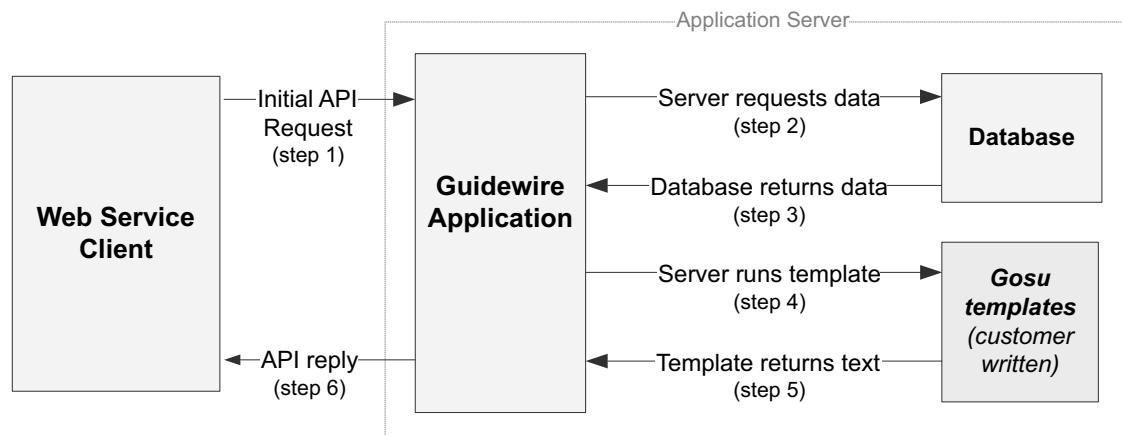
For more information, see “Data Extraction Using Web Services” on page 608.

## Data Extraction Using Web Services

You can write your own web services that use Gosu templates to generate results. For more information about writing Gosu templates, see “Gosu Templates” on page 347 in the *Gosu Reference Guide*.

The following diagram illustrates the data extraction process for a web service API that uses Gosu templates..

**Web Service Data Extraction Flow**



Every data extraction request includes a parameter indicating which Gosu template to use to format the response. You can add an unlimited number of templates to provide different responses for different types of root objects. For example, given a claim as a root object for a template, you could design different templates to export claim data such as:

- A list of all notes on the claim
- A list of all open activities on the claim
- A summarized view of a claim

To provide programmatic access to ClaimCenter data, ClaimCenter uses Gosu, the basis of Guidewire Studio business rules. Gosu templates allow you to embed Gosu code that generates dynamic text from evaluating the code. Wrap your code with <% and %> characters for Gosu blocks and <%= and %> for Gosu expressions.

Once you know the root object, refer to properties on the root object or related objects just as you do as you write rules in Studio.

Before writing a template, decide which data you want to pass to the template.

For example, from a claim you could refer to many properties, including:

- `claim.LossDate` – Loss date
- `claim.LossType.Code` – Code for a typelist value
- `claim.LossType.Name` – Text description of the typelist value
- `claim.Policy.PolicyNumber` – Policy number (different from its public ID)

The simplest Gosu expressions only extract data object properties, such as `claim.ClaimNumber`. For instance, suppose you want to export the claim number. You might use the following template, which despite its short length is a valid Gosu template in its entirety:

```
The number is <%= claim.claimNumber %>.
```

At run time, Gosu runs the code in the template block “<%= ... %>” and evaluates it dynamically:

```
The number is H0-1234556789.
```

## Error Handling in Templates

By default, if Gosu cannot evaluate an expression because of an error or `null` value, it generates a detailed error page. It is best to check for blank or `null` values as necessary from Gosu so that you do not accidentally generate errors during template evaluation.

## Getting Parameters from URLs

If you want to get the value of URL parameters other than the root objects and/or check to see if they have a value use the syntax `parameters.get("paramnamehere")`. For instance, to check for the `xyz` parameter and export it:

```
<%= (parameters.get("xyz") == null)? "NO VALUE!" :parameters.get("xyz") %>
```

## Built in Templates

There are example Gosu templates included with ClaimCenter. Refer to the files within the ClaimCenter deployment directory:

```
ClaimCenter/modules/configuration/config/templates/dataextraction/*.gst
```

## Using Loops in Templates

Gosu templates can include loops that iterate over multiple objects such as every exposure or contact person associated with a claim. For example, suppose you want to display all claims associated with a policy. The corresponding template code with the root object `claims` might look something like this:

```
<% for (var thisClaim in claims) { %>
    Claim number: <%= thisClaim.ClaimNumber %>
<% } %>
```

This template might generate something like:

```
Claim number: HO-123456:C0001  
Claim number: HO-123456:C0002  
Claim number: HO-123456:C0003
```

## Structured Export Formats

HTML and XML are text-based formats, so there is no fundamental difference between designing a template for HTML or XML export compared to other plain text files. The only difference is that the text file must conform to HTML and XML specifications.

HTML results must be a well-formed HTML, ensuring that all properties contain no characters that might invalidate the HTML specification, such as unescaped “<” or “&” characters. This is particularly relevant for especially user-entered text such as descriptions and notes.

Systems that process XML often are very strict about syntax and well-formedness. Be careful not to generate text that might invalidate the XML or confuse the recipient. For instance, beware of unescaped “<” or “&” characters in a notes field. If possible, you could export data within an XML <CDATA> tag, which allows more types of characters and character strings without problems of unescaped characters.

## Handling Data Model Extensions in Gosu

If you added data model extensions, such as new properties within the `Claim` object, they are available work within Gosu templates just as in business rules within Guidewire Studio. For instance, just use expressions like `myClaim.myCustomField`.

## Gosu Template APIs Common for Integration

### Gosu Libraries

You can access Gosu libraries from within templates similar to how you would access them from within Guidewire Studio, using the `Libraries` object:

```
<%= Libraries.Financials.getTotalIncurredForExposure(exposure) %>
```

### Java Classes Called From Gosu

Just like any other Gosu code, your Gosu code in your template use Java classes.

```
var myWidget = new mycompany.utils.Widget();
```

### See also

- For information about calling Java from Gosu, see “Calling Java from Gosu” on page 119 in the *Gosu Reference Guide*.

### Logging

You can send messages to the ClaimCenter log file using the ClaimCenter logger object. Access the logger from Gosu using the following code:

```
libraries.Logger.logInfo("Your message here...")
```

### Typecode Alias Conversion

As you write integration code in Gosu templates, you may also want to use ClaimCenter typecast mapping tools. ClaimCenter provides access to these tools by using the `$typecode` object:

```
typecode.getAliasByInternalCode("LossType", "ns1", claim.LossType)
```

This `$typecode` API works only within Gosu templates, not Gosu in general.

In addition, you can use the `TypecodeMapperUtil` Gosu utility class.

**See also**

- For more information on the Gosu utility class, see “Using Gosu or Java to Translate Typecodes” on page 144.



# Logging

ClaimCenter provides a robust logging system based on the open source Apache log4j project. The log4j system flexibly outputs log statements with arbitrary granularity at a specific logging level that indicate what to write to files. You can configure the logging system fully at runtime by using external configuration files. You can use the ClaimCenter logging system throughout your application code. However, special features of the logging system are particularly useful for integration developers, and this topic discusses them.

This topic includes:

- “Logging Overview For Integration Developers” on page 613
- “Logging Properties File” on page 614
- “Logging APIs for Java Integration Developers” on page 615

**See also**

“Configuring Logging” on page 25 in the *System Administration Guide*

## Logging Overview For Integration Developers

### Logging Elements

The logging system in ClaimCenter comprises these elements:

- **Logging properties files** – A `logging.properties` file specifies what information to log. For instance, you can choose to log absolutely everything relevant to integration, log plugin information, or log nothing. You can choose a large set of logging properties that log certain errors for certain cases or warnings for other cases. The logging properties file uses the format for the Java tool log4j.
- **Log files** – Create log files anywhere accessible from the server. You can create different types of log messages for different contexts in different files or even multiple server directories.
- **Code that triggers logging messages** – Many built-in parts of ClaimCenter can log information, warnings, errors, or arbitrary debugging information. As an integration developer, your code can log anything you want. However, code that triggers logging does not force log messages to be written to files. The logging properties file specifies what ClaimCenter actually does with any specific logging information.

## Logging Types: Category-based and Class-based

ClaimCenter provides two ways to configure logging:

- **Category-based logging** – ClaimCenter provides a hierarchical, abstract category system for logging that avoids dependencies on specific Java classes. Category-based logging operates independently of Java packages and supports quick and easy configuration of log levels, log file locations, and other logging settings. Guidewire strongly recommends that you use category-based logging.
- **Class-based logging** – If you have legacy Java code that uses `log4j` class-based logging, it might be easier to continue your use class-based logging. However, Guidewire strongly recommends that you migrate your `log4j` code to category-based logging so your code integrates with the pre-defined logging categories of ClaimCenter.

## Logging Properties File

The logging properties file (`logging.properties`) configures what to log and where to log it. The logging properties file is in the `log4j` format, which the Apache project defined for the `log4j` system. This topic only briefly mentions the details of how to configure `log4j` files such as these.

Each logging properties file is separated into blocks such as the following example:

```
# set logging levels for the category: "log4j.category.Integration.plugin"
log4j.category.Integration.plugin=DEBUG, PluginsLog
# To remove logging for that category, comment out the previous line with an initial "#"

# The following lines specify how to write the log, where to write it, and how to format it
log4j.additivity.PluginsLog=false
log4j.appenders.PluginsLog=org.apache.log4j.DailyRollingFileAppender
log4j.appenders.PluginsLog.File=C:/Guidewire/ClaimCenter/logs/plugins.log
log4j.appenders.PluginsLog.DatePattern = .yyyy-MM-dd
log4j.appenders.PluginsLog.layout=org.apache.log4j.PatternLayout
log4j.appenders.PluginsLog.layout.ConversionPattern=%-10.10X{server} %-4.4X{userID} %d{ISO8601} %p %m%n
```

In this example, the following line defines the class, log level, and a log name (`AdaptersLog`):

```
log4j.category.Integration.plugin=DEBUG, AdaptersLog
```

This line specifies to use a file appender (a standard local log file):

```
log4j.appenders.PluginsLog=org.apache.log4j.DailyRollingFileAppender
```

The line after that specifies the location of the log:

```
log4j.appenders.PluginsLog.File=C:/Guidewire/ClaimCenter/logs/plugins.log
```

### See also

<http://logging.apache.org/log4j/1.2/index.html>

## Logging Categories for Integration

Logging categories are hierarchical. For example, enabling a log file appender for the category `log4j.category.plugin` enables `log4j.category.plugin.IValidationAdapter`. If there are subcategories defined for that category, those are enabled also. If you use logging categories from Java code, use the `LoggerCategory` class, which includes several static instances of the `Logger` interface that are pre-defined for common use.

### See also

- “Category-based Logging” on page 615.
- For a current list of logging categories, see “Listing Logger Categories” on page 30 in the *System Administration Guide*

# Logging APIs for Java Integration Developers

## Category-based Logging

ClaimCenter provides an API to the log4j-based logging system by using the `LoggerCategory` class. The `LoggerCategory` class contains predefined static instances of the `Logger` interface. The `LoggerCategory` class is available to your Java code and your web services API client code.

For your Java plugins, logger configuration is automatic because the server already instantiated and configured a *logger factory*. A logger factory is the object that configures what to log and where to log it. A Java plugin automatically inherits the logging properties of the application server.

For web services API client code that runs on an external system, your code must explicitly set up the logger factory on the host system. Your web services API client code can set up a logger factory using the `LoggerFactory` class included in the SOAP API libraries.

### See also

If you want to use logging within a Java plugin, see “[Logger Classes](#)” on page 616.

## Setting Up a Logger Factory (SOAP Client Code)

For web services API client code that does not operate in the same Java virtual machine as ClaimCenter, you must configure `LoggerFactory` before writing a logging message. Otherwise, the underlying logging class does not know where to save log messages or the current *log level*. The good news is that you can use a standard logger properties file to programmatically configure a logger factory with the same format as the standard `logging.properties` file.

Configure the logger factory using the `LoggerFactory` class method `configure`. Pass the `configure` method a Java `Properties` object to initialize it with the same properties that are stored in the ClaimCenter `logging.properties` text file. If you want to load a properties text file similar to `logging.properties` into a `Properties` object, you can use the standard Java `Properties` class method called `load`. Once the `Properties` object is prepared, pass it to the `configure` method of the `LoggerFactory`.

The following is an example of this approach for web services API code:

```
// Set up the logger you are going to write to, which is a static
// instance of the LoggerCategory class that uses the category API.*
_logger = LoggerCategory.API;

if (!LoggerFactory.isConfigured()) {
    Properties loggingProps = new Properties();

    // Load the logging properties from the right directory on your server.
    String loggingPropsFile = "/Guidewire/log-config/cc/logging.properties";

    // grab the file and populate the Properties object
    try {
        loggingProps.load(new FileInputStream(loggingPropsFile));
    } catch (IOException io) {
        System.err.println("Cannot locate:" + loggingPropsFile);
        loggingProps = null;
    }

    // pass the Properties object to the logger factory to configure it
    LoggerFactory.configure(loggingProps);
}

_logger.info("Initializing logger for MyClassName...");
```

Alternatively, you can use another method signature for the `configure` method, `configure(java.io.File)`, which configures a logger factory directly from a properties file.

**See also**

The API Reference Javadoc for `gw.api.system.logging.LoggerFactory`

## Logger Classes

To use the `LoggerCategory` class, you first need an instance of the `Logger` class. The easiest way to get an instance is to use the static instances of this class predefined for common top level loggers, such as `PLUGIN` and `API`. You can access these loggers as properties of the `LoggerCategory` class. For instance, `LoggerCategory.PLUGIN` refers to the static instance of the `Logger` interface for plugins.

For example, you can use code like the following to log an error.

```
LoggerCategory.PLUGIN.error(Document + ", missing template: " + stringWithoutDescriptor);
```

To use this tool within a Java class, declare a private class variable of interface `Logger` like this.

```
private Logger _logger = null;
```

Then at runtime, get an instance of the `Logger` object. For example, use the built-in static instances of the `LoggerCategory` class.

```
_logger = LoggerCategory.PLUGIN;
```

You can now write to this logger object with `Logger` methods. The methods you typically use most are methods that log a message at a specific `log4j` logging level: `info`, `warn`, `trace`, `error`, and `debug`.

The following example logs a message at the `INFO` logging level.

```
_logger.info("Setting up logger for MySpecialCode...");
```

The logger does not append this message to a file unless the logger is configured to do so. You must set logging properties to enable that logging level and define a filename path for the log file output.

ClaimCenter uses the initial setup of the logging factory to determine the logging level for this logger based on its category. For plugins, it inherits the server settings. However, you can override the logging level by changing `logging.properties`. You can also temporarily adjust the logging level for a logger by using the **Set Log Level** screen in ClaimCenter. Changes made with the **Set Log Level** screen only persist until the ClaimCenter server is restarted.

If you want to use a logger for which there is no static instance, use one of two alternate calls to the `LoggerFactory` class. The most common approach is to create a logger as a sublogger of an existing logger.

```
Logger logger = LoggerFactory.getLogger(LoggerCategory.PLUGIN, "IApprovalAdapter");
logger.info("My info message here")
```

Alternatively, create a new root category.

```
LoggerCategory logger = LoggerFactory.getLogger("MyRootCategoryName");
logger.info("My info message here")
```

To configure your new category in `logging.properties`, define the new logger and give it its own appender, as the following example shows.

```
log4j.category.IApprovalAdapter=DEBUG, MyLog
log4j.additivity.MyLog=false
log4j.appender.MyLog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.MyLog.File=c:/gwlogs/messaging.log
log4j.appender.MyLog.DatePattern = .yyyy-MM-dd
log4j.appender.MyLog.layout=org.apache.log4j.PatternLayout
log4j.appender.MyLog.layout.ConversionPattern=%-10.10X{server} %-4.4X{userID} %d{ISO8601} %p %m%n
```

If you created a new root category, replace `IApprovalAdapter` in the example above with your new category.

**See also**

- “Logging Levels” on page 27 in the *System Administration Guide*
- “Set Log Level” on page 168 in the *System Administration Guide*

## Class-based Logging (Not Generally Recommended)

Instead of using abstract logging categories to identify related code, you can use the fully-qualified name of a class. By using a fully qualified class name, The class name and package define the hierarchy you use to define logging configuration settings.

**IMPORTANT** Guidewire strongly recommends you use the category-based approach, instead of the class-based approach described in this topic.

For instance, instead of your plugin code writing log messages with `LoggerCategory` to the `log4j.category.Integration.plugin.IValidationAdapter` class, configure a logger based on the actual class of your plugin. For example, you might use `com.mycompany.myadapters.myValidationAdapter`. To use the class-based approach, use the `Logger` and `LoggerFactory` classes.

Just as for category-based logging, plugin logger configuration is automatic because the server already instantiated and configured a *logger factory*. The logger factory configures what to log and where to log it. However, for web services API client code, you must explicitly set up the logger factory using the `LoggerFactory` class. Plugin code and web services API client code can set up a logger factory using the `LoggerFactory` class.

### To use class-based logging in your code

1. If the code is non-plugin code, first configure a logger factory.

For an example, see “Setting Up a Logger Factory (SOAP Client Code)” on page 615.

2. Configure the logger in the `logging.properties` file by using the class name instead of the category name.

3. Use a `LoggerFactory` instance to create a `Logger` instance.

4. Send logging messages to that `Logger` instance.

For a typical example for a plugin, set a class private variable.

```
private Logger _logger = null;
```

Then in your set up code, initialize the logger.

```
_logger = LoggerFactory.getLogger(MyJavaClassName.class);
```

And then you can send logger messages with it, with similar methods as in `LoggerCategory`.

```
_logger.info("Setting up logger ...");
```

### See also

“Category-based Logging” on page 615

## Dynamically Changing Logging Levels

You can change logging levels without redeploying ClaimCenter. For more information, see “Making Dynamic Logging Changes without Redeploying” on page 33 in the *System Administration Guide*.



# Proxy Servers

Guidewire recommends deploying proxy servers to insulate ClaimCenter from the external Internet. Because incoming requests are the most dangerous, this is most important for integrating ClaimCenter with Insurance Service Office (ISO) because ISO must initiate network requests directly to ClaimCenter. However, Guidewire recommends proxy servers for outgoing requests to Metropolitan Reporting Bureau and outgoing geocoding requests.

This topic includes:

- “Proxy Server Overview” on page 619
- “Configuring a Proxy Server with Apache HTTP Server” on page 620
- “Certificates, Private Keys, and Passphrase Scripts” on page 621
- “Proxy Server Integration Types for ClaimCenter” on page 622
- “Proxy Building Blocks” on page 623

**See also**

- For ISO, see “Insurance Services Office (ISO) Integration” on page 447.
- For Metropolitan, see “Metropolitan Reporting Bureau Integration” on page 487.
- For geocoding, see “Geographic Data Integration” on page 235.
- For configuring web services URLs based on configuration environment settings to support proxy servers, see “Working with Web Services” on page 119 in the *Configuration Guide*. For additional information about web services, see “Web Services Introduction” on page 31.

## Proxy Server Overview

Several ClaimCenter integration options require outgoing messages, including ISO integration, geocoding integration, and Metropolitan integration. In addition, ISO integration requires incoming requests from the Internet to ClaimCenter. Placing a proxy server between the external Internet and ClaimCenter insulates ClaimCenter from some types of attacks and partitions all network access for maximum security.

Additionally, some of the integration points require encrypted communication. Because encryption in Java tends to be lower performance than in native code that is part of a web server, encryption can be off-loaded to the proxy server. For example, instead of the ClaimCenter server directly encrypting HTTPS/SSL connections to an outsider server, ClaimCenter can contact a proxy server with standard HTTP requests. Standard requests are less resource intensive than SSL encrypted requests. The proxy server running fast compiled code connects to the outside service using HTTPS/SSL.

If a proxy server handles incoming connections from an external Internet service to ClaimCenter and not just outgoing requests from ClaimCenter, some people would call it a *reverse proxy server*. For the sake of simplicity, this topic refers simply to a server that handles incoming requests as a *proxy server*. Your server might handle only outgoing requests if you do not need to intercept incoming requests.

## Resources for Understanding and Implementing SSL

Some proxy server configurations use SSL encryption. Encryption concepts and proxy configuration details are complex, and full documentation on this process is outside the scope of ClaimCenter documentation.

For more information about SSL encryption and Apache-specific documentation related to SSL, refer to all of the following resources:

Encryption-related documentation	For more information, see this location
High-level overview of public key encryption	<a href="http://en.wikipedia.org/wiki/Public_key">http://en.wikipedia.org/wiki/Public_key</a>
Detailed description of public key encryption	<a href="ftp://ftp.pgp.org/pub/pgp/7.0/docs/english/IntroToCrypto.pdf">ftp://ftp.pgp.org/pub/pgp/7.0/docs/english/IntroToCrypto.pdf</a>
Detailed description of SSL/TLS Encryption	<a href="http://httpd.apache.org/docs/2.0/ssl/ssl_intro.html">http://httpd.apache.org/docs/2.0/ssl/ssl_intro.html</a>
Overview of Apache's SSL module	<a href="http://httpd.apache.org/docs/2.0/mod/mod_ssl.html">http://httpd.apache.org/docs/2.0/mod/mod_ssl.html</a>
Overview of Apache's proxy server module	<a href="http://httpd.apache.org/docs/2.0/mod/mod_proxy.html">http://httpd.apache.org/docs/2.0/mod/mod_proxy.html</a>

## Web Services and Proxy Servers

If your ClaimCenter deployment must call out to web services hosted by other computers, for maximum security always connect to it through a proxy server.

Be aware you can vary the URL to remote-hosted web services based on configuration environment settings on your server, specifically the `env` and `serverid` settings. For example, if running in a development environment, then directly connect to the remote service or through a testing-only proxy server. In contrast, if running in the production environment, then always connect through the official proxy server.

### See also

- For more information, see “Working with Web Services” on page 119 in the *Configuration Guide*.
- For additional information about web services, see “Web Services Introduction” on page 31.

## Configuring a Proxy Server with Apache HTTP Server

The Apache HTTP server is a popular open source web server that can be configured as a proxy server. This section is intended only if you need to use the ISO Apache HTTP server examples included with ClaimCenter. Also use this section to integrate the relevant security elements into a current Apache configuration for an existing Apache proxy server.

This section presents the generic Apache HTTP server configuration, and then the next section describes the different proxy *building blocks*. You can add one or more building blocks to your own Apache configuration file as appropriate.

## Apache Basic Installation Checklist

This section describes the high-level Apache installation and security instructions. A full detailed set of Apache instructions is outside the scope of this Guidewire documentation.

### Follow these high-level steps for installing Apache

1. Download Apache HTTP server. Get it from <http://httpd.apache.org>.

The Apache configuration files blocks listed in this topic were designed for Apache 2.2.X. Guidewire has observed problems with Apache HTTP versions older than version 2.2.3. Do not use older versions. To use these examples, use the latest 2.2.X release and confirm X is a number 3 or greater.

2. Install Apache HTTP server.
3. Download and install the SSL security Apache module.
4. Install Apache HTTP server as a background UNIX daemon or Windows service.
5. Configure the Apache directive configuration file.
6. Make appropriate changes to your firewall.

---

**IMPORTANT** If you already have some sort of corporate firewall, you must make holes in your firewall for all integration points.

---

7. Install any necessary SSL certificates and SSL keys.
8. Enable Apache modules. Enable the following Apache modules `mod_proxy` (proxies in general), `mod_proxy_http` (HTTP proxies), `mod_proxy_connect` (SSL tunneling), `mod_ssl` (SSL encryption).

## Certificates, Private Keys, and Passphrase Scripts

Security file	Description
<code>\$DestinationTrustedCACertFile</code>	File containing the certificate used to sign the destination web site.
<code>\$ReverseProxyTrustedCertFile</code>	File containing the certificate for the reverse proxy site. To ensure that the certificate is recognized by source systems, ensure a Trusted Certification Authority signs it
<code>\$ReverseProxyTrustedProtectedPrivateKeyFile</code>	File containing the private key used to decrypt the messages in the source to reverse proxy communication. This file is generally signed by a passphrase script, <code>\$ReverseProxyTrustedPassPhraseScript</code>

The `$ReverseProxyTrustedProtectedPrivateKeyFile` is very sensitive. If it is exposed, it may allow an elaborate attacker to impersonate your web site by coupling this exploit with DNS corruptions. Therefore, this private key must be secured by all means.

Rather than displaying that private key in a file, it is a common practice to secure that private key through a passphrase. The DMZ proxy would then be provided with both the protected private key file and with a script that would return the pass-phrase under specific security conditions. The logic of the script and the conditions for returning the right pass-phrase are the secured DMZ proxy's administrator responsibility. The script's goal is to prevent the pass-phrase to be returned if not called from the right proxy instance and from a non-corrupted environment.

# Proxy Server Integration Types for ClaimCenter

## ISO Proxy Communication

ClaimCenter and ISO exchange different types of messages:

ClaimCenter sends messages 1 and 2 to ISO. For these messages, use this configuration:

- “Downstream Proxy With Encryption” on page 624

ISO sends Message 3 asynchronously to ClaimCenter. For these message, use this configuration:

- “Upstream (Reverse) Proxy with Encryption for Service Connections” on page 624

## Metropolitan Proxy Communication

For Metropolitan reports, use the following configuration to connect to the Metropolitan Request URL:

- “Downstream Proxy With Encryption” on page 624

Use the same URL later to poll for availability of results:

ClaimCenter retrieves the report (after determining availability) using the separate URL called the Metropolitan Report Retrieval URL. Use the same configuration block, but with the different URL.

## Bing Maps Geocoding Service Communication

The geocoding plugin only initiates communications with geocoding services. It never responds to communications initiated from the external Internet. Therefore, you do not need a reverse proxy server to insulate the geocoding plugin and ClaimCenter from incoming Internet requests. However, Guidewire recommends insulating outgoing geocoding requests through a proxy server as a best practice.

### To configure the Bing Maps geocoding plugin to use a proxy server

1. In ClaimCenter Studio, configure the web service collection for the Bing Maps web service.
  - a. Navigate to the web service collection at:  
Resource → Classes → wsi → remote → microsoft → bingmaps
  - b. In the Web Service Collection editor, click **Add Setting** and then **Override URL**.
  - c. In the **Override URL** field, enter your proxy server URL and port number for the Bing Maps web service.
2. In the configuration of your Apache proxy server, add a configuration building block for the Bing Maps URL. Follow the pattern for configuration building blocks described in “Downstream Proxy With No Encryption” on page 623

If you use Guidewire ContactManager and geocoding is enabled within ContactManager, typically you can use the same configuration building block for communication between ContactManager and Bing Maps. Simply allow connections from the IP addresses of ClaimCenter and ContactManager in each configuration building block. Also, configure the web service collection for Bing Maps in Contact Manager Studio with the overriding URL of your proxy server.

### See also

- For more information on how to configure WS-I web services for proxy servers, see “Working with Web Services” on page 119 in the *Configuration Guide*.
- For more information about the geocoding plugin, see “Geographic Data Integration” on page 235.

## SSL Encryption for Users

You may want to use the Apache server to handle SSL encryption from users to ClaimCenter and reduce the processing burden of SSL encryption in Java on the ClaimCenter server. Use the following Apache configuration building block:

- “Upstream (Reverse) Proxy with Encryption for User Connections” on page 625

### See also

- For more information about SSL encryption, see the “Securing ClaimCenter Communications” on page 89 in the *System Administration Guide*.

## Proxy Building Blocks

The following subsections list building blocks of configuration text for Apache configuration files. For each building block, you must substitute all values that are prepended by dollar signs (\$). Actual Apache configuration files must not have the dollar sign in the actual file.

For instance, instead of:

```
Listen $PROXY_PORT_NUMBER_HERE
```

Replace it with:

```
Listen 1234
```

...to listen on port 1234.

---

**IMPORTANT** The dollar sign (\$) appears in configuration building blocks to indicate values that must be substituted with hard-coded values. Replace all these values, and leave no “\$” characters in the final file.

---

## Downstream Proxy With No Encryption

In this configuration, a source system calls the proxy, which transmits the request unchanged to the destination URL. The reply follows the opposite path unencrypted.

Use this Apache configuration building block:

```
#Disable forward proxying for security purposes
ProxyRequests Off

#The reverse proxy listens to the source system on the reverse proxy port.
Listen $PROXY_PORT_NUMBER_HERE

<VirtualHost *:$PROXY_PORT_NUMBER_HERE>
<Proxy *>
    Order Deny,Allow
    Deny from all

    # The Virtual Host accepts requests only from the source system
    Allow from $SourceSystem
</Proxy>

# The Virtual Hosts associates the packet to the destination URL
ProxyPass $SOURCE_URL $DESTINATION_URL

#Logs redirected to appropriate location
ErrorLog $ApacheErrorLog

</VirtualHost>
```

Replace the \$SOURCE\_URL value with the source URL. To redirect all HTTP traffic on all URLs on the source IP address and port, use the string “/”, which is just the forward slash, with no quotes around it.

Replace the \$DESTINATION\_URL value with the destination domain name or URL.

## Downstream Proxy With Encryption

In this configuration, a source system calls the proxy, which transmits the request to the destination URL. The reply follows the opposite path. The proxy to destination system communication is encrypted for the request and also for the reply.

Use this Apache configuration building block:

```
#SSL sessions are cached to ensure possible reuse across sessions
SSLSessionCache shm:$SSL_CACHE(512000)
SSLSessionCacheTimeout 300

#Disable forward proxying for security purposes
ProxyRequests Off

#The reverse proxy listens to the source system on the reverse proxy port.
Listen $PROXY_PORT_NUMBER_HERE

<VirtualHost *:$PROXY_PORT_NUMBER_HERE>
    <Proxy *>
        Order Deny,Allow
        Deny from all

        # The Virtual Host accepts requests only from the source system
        Allow from $SourceSystem
    </Proxy>

    # The Virtual Hosts associates the packet to the destination URL
    ProxyPass / $DestinationURL

    #Communication is encrypted on the reverse proxy to destination system leg
    SSLProxyEngine on

    #The Reverse proxy checks the destination's certificate
    #using the appropriate Trusted CA's certificate
    SSLProxyCACertificateFile $DestinationTrustedCACertFile

    #Logs redirected to appropriate location
    ErrorLog $ApacheErrorLog

</VirtualHost>
```

## Upstream (Reverse) Proxy with Encryption for Service Connections

In this configuration, a source system calls the reverse proxy, which transmits the request to the destination URL. The reply follows the opposite path. The source system to reverse proxy communication is encrypted (for both request and reply)

Use this Apache configuration building block:

```
#SSL sessions are cached to ensure possible reuse across sessions
SSLSessionCache shm:$SSL_CACHE(512000)
SSLSessionCacheTimeout 300

#Private keys are secured through a pass-phrase
SSLPassPhraseDialog exec:$ReverseProxyTrustedPassPhraseScript

#Disable forward proxying for security purposes
ProxyRequests Off

#The reverse proxy listens to the source system on the reverse proxy port.
Listen $REVERSEPROXY_PORT_NUMBER_HERE

<VirtualHost *:$REVERSEPROXY_PORT_NUMBER_HERE>
    <Proxy *>
        Order Deny,Allow
        Deny from all

        # The Virtual Host accepts requests only from the source system
        Allow from $SourceSystem
    </Proxy>

    # The Virtual Hosts associates the packet to the destination URL
    ProxyPass / $DestinationURL

    #Communication is encrypted on the source system to reverse proxy leg
```

```
SSLEngine    on
#The Virtual Host authenticates to the source system providing its certificate
SSLCertificateFile $ReverseProxyTrustedCertFile
#The communication security is achieved using the PrivateKey, which is secured
#through a pass-phrase script.
SSLCertificateKeyFile $ReverseProxyProtectedPrivateKeyFile
#Logs redirected to appropriate location
ErrorLog    $ApacheErrorLog
</VirtualHost>
```

## Upstream (Reverse) Proxy with Encryption for User Connections

Use the Apache server to handle SSL encryption from users to ClaimCenter and thus reduce the processing burden of SSL encryption in Java on the ClaimCenter server. Use the following configuration building block:

```
#Encrypted Reverse Proxy
<VirtualHost *:>portnumber>

#Allow from the authorized remote sites only
<Proxy *>
    Order Deny,Allow
    Allow from all
</Proxy>

# Access to the root directory of the application server is not allowed
<Directory />
    Order Deny,Allow
    Deny from all
</Directory>

#Access is allowed to the cc directory and its subdirectories for the authorized sites only
<Directory /cc>
    Order Deny,Allow
    Allow from all

# Never allow communications to be not encrypted
SSLRequireSSL

#The Cipher strength must be 128 (maximal cipher size authorized
#all communication secured
SSLRequire %{SSL_CIPHER_USEKEYSIZE} >= 128 and %{HTTPS} eq "true"
</Directory>

#Classic command to take into account an Internet Explorer issue
SetEnvIf User-Agent ".*MSIE.*" \
nokeepalive ssl-unclean-shutdown \
 downgrade-1.0 force-response-1.0

#Encryption secures the Internet to Encrypted Reverse Proxy communication
#Listing of available encryption levels available to Apache
SSLEngine      on
SSLCipherSuite ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+eNULL

#The Virtual Host authenticates to the user providing its certificate
SSLCertificateFile    conf/<certificate_filename>.crt

#The communication security is achieved using the PrivateKey, which is secured through
#a pass-phrase script.
SSLCertificateKeyFile  conf/<certificate_filename>-secured.pem

#The Virtual Host associates the request to the internal Guidewire product instance
ProxyPass        /<product>400 <url of the product server>
ProxyPassReverse    /<product>400 <url of the product server>

#Logs redirected to appropriate location
ErrorLog    logs/encrypted_<product>.log

</VirtualHost>
```

## Modify the Server to Receive Incoming SSL Requests

To enable ClaimCenter to respond to a request over SSL from a particular inbound connection, your proxy handles encryption. The connection between ClaimCenter and the proxy server remains unencrypted. Configure the proxy to know the URL and port (location) of the server that originates the request.

Edit your proxy server configuration so it is aware of:

- The externally-visible domain name of the reverse proxy server
- The port number of the reverse proxy server
- The protocol the client used to access the proxy server (in this case HTTPS)

To ensure your ClaimCenter server is aware of the proxy, edit the web application container server configuration CATALINA\_HOME\conf\server.xml on your ClaimCenter server. Add an additional connector as shown in the following XML snippet:

```
<!-- Define a non-SSL HTTP/1.1 Connector on port <port number> to receive decrypted  
communication from Apache reverse proxy on port 11410 -->  
<Connector acceptCount="100" connectionTimeout="20000" disableUploadTimeout="true"  
enableLookups="false" maxHttpHeaderSize="8192" maxSpareThreads="75" maxThreads="150"  
minSpareThreads="25" port="portnumber" redirectPort="8443" scheme="https" proxyName="hostname"  
proxyPort="portnumber">  
</Connector>
```

You must substitute the following parameters shown in the snippet:

<i>port</i>	Specifies the port number for the additional connector for access through the proxy.
<i>proxyName</i>	Identifies the deployment server's name.
<i>proxyPort</i>	Specifies the port for encrypted access through Apache.
<i>scheme</i>	Identifies the protocol used by the client to access the server.

After configuring the server.xml file, restart your application server.

### See also

- For more information about SSL encryption, see “Securing ClaimCenter Communications” on page 89 in the *System Administration Guide*.

# Java and OSGi Support

This topic describes ways to write and deploy Java code in ClaimCenter, including accessing entity data from Java. For example, you could implement ClaimCenter plugin interfaces using a Java class instead of a Gosu class. If you implement a plugin interface in Java, optionally you can write your plugin implementation as an OSGi bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java libraries. Guidewire recommends OSGi for all new Java plugin development.

**See also**

- You can write Gosu code that uses Java types. See “Calling Java from Gosu” on page 119 in the *Gosu Reference Guide*.
- For an introduction to Gosu from a Java perspective, see “Gosu Introduction” on page 15 in the *Gosu Reference Guide*.

This topic includes:

- “Overview of Java and OSGi Support” on page 627
- “Accessing Entity and Typecode Data in Java” on page 631
- “Accessing Gosu Classes from Java Using Reflection” on page 642
- “Gosu Enhancement Properties and Methods in Java” on page 643
- “Class Loading and Delegation for non-OSGi Java” on page 643
- “Deploying Non-OSGi Java Classes and JARs” on page 644
- “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 645
- “Advanced OSGi Dependency and Settings Configuration” on page 652
- “Updating Your OSGi Plugin Project After Product Location Changes” on page 653

## Overview of Java and OSGi Support

You can deploy Java code within ClaimCenter. There are several different ways you can use Java.

Typical customers write Java code primarily to implement ClaimCenter plugin interfaces.

However, you can write Java code that you can call from any Gosu code in ClaimCenter, such as from rule sets or other Gosu classes.

In all cases for Java development, you must use an IDE for Java development separate from ClaimCenter Studio. It is unsupported to add or modify Java class files in the ClaimCenter Studio user interface. Although ClaimCenter Studio does not hide user interface tools that add Java classes to the file hierarchies, those features are unsupported.

If you are deploying Java plugins or OSGi plugins, carefully read about your IDE options in “[Implementing Plugin Interfaces in Java and Optionally OSGi](#)” on page 628.

## Learning More About Entity Java APIs

If you want to deploy Java code and you do not use Guidewire entity data, the main thing you need to know is where to put Java classes and libraries. See “[Deploying Non-OSGi Java Classes and JARs](#)” on page 644.

If your Java code needs to get, set, or query Guidewire entity data, you must also understand how ClaimCenter works with entity data in Java. See “[Accessing Entity and Typecode Data in Java](#)” on page 631.

## Accessing Gosu Types from Java

From Gosu, you can call Java types, including added third-party Java classes and libraries. However, from Java you cannot access Gosu types without using a language feature called *reflection*. Reflection means to ask the type system at run time about types. Using reflection is not typesafe, which means you cannot catch some types of errors at compile time. For example, you must use reflection to access the following from Java: Gosu classes, Gosu interfaces, and Gosu enhancements. See the following topics:

- “[Accessing Gosu Classes from Java Using Reflection](#)” on page 642
- “[Gosu Enhancement Properties and Methods in Java](#)” on page 643

Note that some of the supported ClaimCenter types that you might use in Gosu are actually implemented in Java and thus require no special access from Java.

## Implementing Plugin Interfaces in Java and Optionally OSGI

Many customers write Java code primarily to implement plugin interfaces. For general information about plugins, see “[Plugin Overview](#)” on page 163.

Conceptually, there are two main steps to implement a plugin:

1. **Write a class that implements the interface** – See “[Implementing Plugin Interfaces](#)” on page 164
2. **Register your plugin implementation** – See “[Registering a Plugin Implementation Class](#)” on page 166

If you implement a plugin interface in Java, there are two ways to deploy your code:

- **Java plugin** – A Java class. You must use an IDE other than Studio. If you write your plugin in Java, you must regularly regenerate the Java API libraries to compile against the latest libraries. You can use any Java IDE. You can choose to use the included IntelliJ IDEA with OSGi Editor for Java plugin development even if you do not choose to use OSGi.
- **OSGi plugin** – A Java class encapsulated in an *OSGi* bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java libraries. To simplify OSGi configuration, ClaimCenter includes an application called IntelliJ IDEA with OSGi Editor. For details of deploying the files, see “[Deploying Non-OSGi Java Classes and JARs](#)” on page 644.

**IMPORTANT** Guidewire recommends OSGi for all new Java plugin development. ClaimCenter supports OSGi bundles only to implement a ClaimCenter plugin interface and any of your related third-party libraries. It is unsupported to install OSGi bundles for any other purpose.

For more information about the OSGi standard, refer to:

<http://www.osgi.org/Technology/WhyOSGi>

The most important benefits of OSGi for ClaimCenter plugin development are:

- Safe encapsulation of third-party Java JAR files. OSGi loads types in a way that reduces compatibility problems between OSGi bundles, or between an OSGi component and libraries that ClaimCenter uses. For example, dependencies on specific third-party packages and classes are explicit in manifest files and validated at startup time.
- Dependency injection support using *declarative services*. Declarative services use Java annotations and interfaces to declare dependencies between your code and other APIs. For example, your code does not declare that it needs a specific class for a task. Instead, your code uses Java interfaces to define which services it needs. The OSGi framework ensures the appropriate API or objects are available. The OSGi framework tracks dependencies and handles instantiates objects as needed. For example, your OSGi plugin might depend on a third-party OSGi library that provides a service. Your plugin code can use declarative services to access the service.

Beware of a terminology issue in Guidewire documentation with the word *bundle*:

- In nearly all Guidewire documentation, *bundle* refers to a programmatic abstraction of a database transaction and the set of database rows to update. See “Bundles and Database Transactions” on page 331 in the *Gosu Reference Guide*.
- In the OSGi standard, *bundle* refers to a registered OSGi component. ClaimCenter documentation relating to Java and plugins sometimes refers to *OSGi bundles*.

## IDE Options for Plugin Development in the Java Language

You can use any Java IDE that is separate from ClaimCenter Studio. It is unsupported to edit Java directly in Studio.

If you are writing an OSGi plugin implementation, there are several IDE options:

- **IntelliJ IDEA with OSGi Editor (included with ClaimCenter)** – A specially-configured instance of IntelliJ IDEA and included with ClaimCenter. This is a different application than ClaimCenter Studio. This IntelliJ IDEA instance includes a special IntelliJ IDEA plugin for OSGi plugin configuration. For OSGi plugin development, Guidewire recommends using IntelliJ IDEA with OSGi Editor. The included plugin editor configures OSGi configuration files such as the bundle manifest.
- **Other Java IDEs such as Eclipse** – You can use other Java IDEs such as Eclipse or your own version of IntelliJ IDEA. However, you must manually configure OSGi files and bundle manifest manually according to the OSGi standard.

The following table summarizes options for development environments for plugin development in Java.

IDE	Gosu plugin	Java plugin (no OSGi)	OSGi plugin (Java with OSGi)
ClaimCenter Studio	Yes	--	--
IntelliJ IDEA with OSGi Editor (included with ClaimCenter)	--	Yes	Yes
Eclipse or other Java IDE of your own choice	--	Yes	Yes, though requires manual configuration of OSGi files and bundle manifest.

## Inspections to Flag Unsupported Internal Java APIs

The Java API allows you to use the same Java types that you can use in Gosu. However, Guidewire specifies some methods and fields on these types for internal use only. Do not use any of these *internal APIs*. Guidewire indicates internal API methods and properties with the annotation @gw.lang.InternalAPI.

In Gosu, methods and fields with that annotation are hidden. Gosu code that uses internal APIs triggers compilation errors.

In Java, when you are using your own IDE separate from Studio, internal APIs are visible although unsupported. Depending on what IDE you use, you may require additional configuration of your IDE. The following table summarizes your options for internal API code inspections.

IDE	Java IDE Includes Internal API Inspection
IntelliJ IDEA with OSGi Editor (included with ClaimCenter)	Yes
Separate IntelliJ IDEA instance of your own choice	Optional installation. For compatibility with specific IntelliJ IDEA versions, please contact Guidewire Customer Support. See “Installing the Internal API Inspection in Your Own Instance of IntelliJ IDEA” on page 630
Eclipse or other Java IDE of your own choice	No. There is no equivalent code inspection available. If you use another IDE and you are unsure of the status of a particular method or field, navigate to it and see if the declaration has the annotation @gw.lang.InternalAPI.
ClaimCenter Studio	<b>WARNING:</b> ClaimCenter Studio does not support Java coding directly in the IDE. Do not create Java classes directly in ClaimCenter Studio. If you want to code in Java, you must use a separate IDE for Java development. See “IDE Options for Plugin Development in the Java Language” on page 629

### Installing the Internal API Inspection in Your Own Instance of IntelliJ IDEA

The recommended approach for Java development is with the included IntelliJ IDEA with OSGi Editor. If you instead choose to use your own separate instance of IntelliJ IDEA, you may be able to use the Internal API inspection, depending on your version of IntelliJ IDEA. For compatibility with specific IntelliJ IDEA versions, please contact Guidewire Customer Support.

---

**IMPORTANT** The IntelliJ IDEA with OSGi Editor automatically includes the code inspection for the @InternalAPI annotation. If you use this included application, you do not need to install any code inspections to flag internal API usage.

---

#### To install the IntelliJ IDEA Internal API Code Inspection

1. Open your separate instance of IntelliJ IDEA.
2. Navigate to **Settings → Plugins**.
3. Click **Install from disk...**
4. Navigate to the directory:  
`ClaimCenter/studio/configstudio/internal-api-idea-plugin/lib/`
5. Select the JAR in that directory.
6. Let IntelliJ IDEA restart after installing the plugin.
7. Navigate to **Settings → Inspections**.
8. Expand the node **Portability Issues**.
9. Check the box **Use of internal API**.
10. Click **OK**.

## Accessing Entity and Typecode Data in Java

An *entity* type is an abstract representation of Guidewire business data of a certain type. Define entity types in data model configuration files. For entity types in the default configuration, entity types have built-in properties and you can optionally add extension properties. User and Address are examples of entity types.

You can write Java code that accesses entity data from:

- Java plugin implementations
- OSGi plugin implementations (written in Java)
- Other Java classes called from Gosu.

After you call a script that regenerates ClaimCenter Java API libraries, you can write Java code that uses them to access entity data. See “Regenerating Integration Libraries” on page 22.

Using a separate Java-capable IDE, add the entity library directories as a dependency in your project in the separate IDE. Compile your Java code against these libraries.

---

**IMPORTANT** Do not create Java classes directly in Studio. To code in Java, you must use a separate IDE for Java development. For example, use a separate instance of IntelliJ IDEA or Eclipse.

---

Your Java code can get or set entity data or call additional *domain methods* with similar functionality in Java as in Gosu. For example, a Message object as viewed through the entity library interface has data getter and setter methods to get and set data. The Message object has `getPayload` and `setPayload` methods that manipulate the `Payload` property. Additionally, the object has additional methods that trigger complex logic, such as the `reportAck` method.

Understanding and using the entity libraries is critical in the following situations:

- **Java plugin development.** Most ClaimCenter plugin implementations must access entity data or change entity data. Also, some plugin interface methods explicitly have entity data as method arguments or return values. See “Plugin Overview” on page 163.
- **Other Java classes that use entity data.** Even if your Java code does not implement a plugin interface, your Java code can access entity data.

In both cases, your code can:

- take entity data as method arguments or return values
- search for entity data
- change entity data

When you are done writing your Java code, deploy your class files and JAR files:

- For non-OSGi Java, see “Deploying Non-OSGi Java Classes and JARs” on page 644
- For OSGi plugins, see “Using Third-Party Libraries in Your OSGi Plugin” on page 649s

## Regenerating Java API Libraries

Anytime you change the data model due to extensions or customizations, you must regenerate the entity libraries and compile your Java code against the latest libraries.

### To regenerate the Java API libraries

1. In Windows, open a command prompt.
2. Change your working directory with the following command:

```
cd ClaimCenter/bin
```

- Use the `regen-java-api` command:

```
gwcc regen-java-api
```

This command generates Java entity interfaces and typelist classes in libraries in the location:

```
ClaimCenter/java-api/lib
```

This command generates Javadoc documentation in the location:

```
ClaimCenter/java-api/doc
```

## Entity Packages and Customer Extensions from Java

In Gosu, you can refer to an entity type using the syntax simply `entity.ENTITYNAME` or simply the entity name because the package `entity` is always in scope.

In the ClaimCenter Java API, you can reference a type directly by its fully-qualified name. However, for ClaimCenter entity types, from Java the fully-qualified name of an entity is not `entity.ENTITYNAME` nor simply the entity name. The syntax `entity.ENTITYNAME` or using the entity name with no package is a shortcut of the Gosu type system.

If you want only the base configuration properties, the type name is the same in Java as in Gosu but the package varies by entity type. Some aspects of the fully-qualified names of the interfaces are configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635. ClaimCenter exposes each entity type as several interfaces, which the following table summarizes.

Terminology	Description	When it exists	Entity name suffix	For more information, including the Java package for the interface
<b>Entity types that Guidewire originally creates</b>				
<i>base entity interface</i>	Contains only the base configuration properties.	All entity types have a base entity interface.	n/a	“Accessing Entity Properties and Methods With Base and Core Extension Interfaces” on page 633
<b>Entity types that you originally create</b>				
<i>customer extension entity interface</i>	Includes customer data model extension properties.  If a core extension interface exists for that entity type, the customer extension interface extends from the core extension entity interface. Otherwise, the customer extension interface extends from the base entity interface.	Exists only if there are customer data model extensions.	Ext	“Customer Extension Entity Interface” on page 634
<i>customer entity interface</i>	Contains all properties	All entities that you create have this interface	n/a	“Entity Interfaces for Completely Custom Entity Types” on page 635.

## Accessing Entity Properties and Methods With Base and Core Extension Interfaces

From Java, the base entity interface package includes a *prefix* and a *subpackage* that defines a general area of functionality in the application. The general pattern for the fully-qualified base entity type name is:

```
PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME
```

For entity types that multiple Guidewire applications share, the pattern is:

```
gw.p1.SUBPACKAGE.entity.ENTITYNAME
```

For entities specific to the ClaimCenter application, the pattern is:

```
gw.cc.SUBPACKAGE.entity.ENTITYNAME
```

For example, fully-qualified name of the `Address` entity is:

```
gw.p1.contact.entity.Address
```

Instead of memorizing fully-qualified names of the entities, use the auto-completion features of your IDE. For example, if you type `Address` then type `CTRL+Space` in IntelliJ IDEA, the IDE presents options including the entity with the pattern described above. After you choose the correct entity type, the IDE inserts an `import` statement at the top of the file. For example, the IDE might add the following line:

```
import gw.p1.contact.entity.Address;
```

If you write Java code that implements a ClaimCenter plugin interface, interface method arguments and return types that reference entity types always uses the Java name for the type. The Java fully-qualified name for the entity type is different than the type name in Gosu, and in fact in Java is represented by three interfaces. Remember this difference as you review documentation examples that may show code in Gosu.

Some aspects of the fully-qualified names of the interfaces are configurable. See “Configuring Entity and Type-list Fully-Qualified Names from Java” on page 635.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

### Core Extension Entity Interfaces

For some entity types, the Guidewire platform defines a base version and then ClaimCenter extends the base entity interface with properties and methods for application-specific business logic.

Only for such entities, ClaimCenter adds a *core extension entity interface* with the suffix `CoreExt`. For example, the `Message` entity contains many properties in its base entity interface. However, each Guidewire application adds additional properties to `Message` for business logic unique to each application to reference a primary object for that message.

The package for the core extension entity interface is slightly different from the base entity interface. For entities that have core extension entity interfaces, the base entity interface has the pattern:

```
gw.p1.SUBPACKAGE.entity.ENTITYNAME
```

By definition, core extension entity interfaces are specific to the ClaimCenter application. Therefore, the pattern matches the pattern for application-specific entities:

```
gw.cc.SUBPACKAGE.entity.ENTITYNAMECoreExt
```

For example, Java code that references the base entity interface and core extension entity interface for the `Message` entity has the following lines at the top of the Java file:

```
import gw.p1.messaging.entity.Message;
import gw.cc.messaging.entity.MessageCoreExt;
```

The core extension entity interface extends the base entity interface, so it also contains all the methods of the base entity interface. To use application-specific properties, downcast an object reference to the core extension entity interface.

For example, in a ClaimCenter message transport plugin, to use the `Message.Claim` property, you must downcast to the *core extension interface* to access that property:

```
import gw.cc.claim.entity.Claim;
import gw.p1.messaging.entity.Message;
```

```

import gw.cc.messaging.entity.MessageCoreExt;
class MyMessageTransport implements MessageTransportPlugin {
    // ...
    public void send(Message message, String transformedPayload) {
        // IMPORTANT: To access some properties, cast Message to subinterface MessageCoreExt
        MessageCoreExt m = (MessageCoreExt) message;

        // access a property on MessageCoreExt that is absent on Message
        Claim c = m.getClaim();

        // ... use this Claim information and send a message...
    }
}

```

Alternatively, to access the application-specific properties or methods, you can use the customer extension entity interface. Only if there are customer data model extensions, ClaimCenter creates a customer extension entity interface, which extends from the base entity library or the core extension interface if it exists. For details, see “Customer Extension Entity Interface” on page 634.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

## Customer Extension Entity Interface

For each entity, if there are any customer data model extensions, there is an additional interface called the *customer extension entity interface*. If there are no customer data model extensions, ClaimCenter does not create this interface.

Important qualities of the customer extension entity interface:

- The interface contains your data model extension properties.
- The interface name has the suffix `Ext`.
- The interface package is different from the package of the base entity interface. The package prefix changes to the *Java API package prefix* as defined in the file `extensions.properties`. See the table in “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635.
- The customer extension entity interface extends from the base entity library or the core extension interface if it exists. Therefore if there is a customer extension entity interface, it contains the complete set of all possible properties defined by data model configuration files.
- Like all the other Java entity interfaces, the customer extension entity interface does not include properties added by Gosu enhancements. See “Gosu Enhancement Properties and Methods in Java” on page 643.

The Java fully-qualified name of the customer extension entity interface is:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAMEExt`

By default, `PACKAGE_PREFIX` is `extensions.cc`, and the subpackage varies by entity type in the entity declaration. For configuration of the `PACKAGE_PREFIX`, see “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635.

For example, consider the `Address` entity type. In Gosu, use either of the following syntax styles:

```

Address
entity.Address

```

The Java fully-qualified name of the base entity interface is:

`gw.pl.contact.entity.Address`

By default, the Java fully-qualified name of the customer extension interface is:

`extensions.cc.contact.entity.AddressExt`

If the customer extension entity interface exists, you can downcast a variable declared to the base entity interface to the customer extension entity class. The customer extension class contains the customer extension properties as well as all the properties and methods from the base and core extension entity interfaces.

For example, the following Java code downcasts to access extension properties:

```
import gw.pl.contact.entity.Address;
import extensions.cc.contact.entity.AddressExt;

...
public void sendAddressProperties ( Address a ) {
    // downcast from Address to AddressExt. See the import statements above.
    AddressExt addressWithExtension = (AddressExt) a;

    // use extension properties from Java...
    System.out.println(addressWithExtension.getMyExtensionProperty());
}
```

After modifying the data model configuration in ways that affect extension properties, remember to regenerate the Java libraries. See “Regenerating Java API Libraries” on page 631.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

### Entity Interfaces for Completely Custom Entity Types

If you create custom entity types (rather than extend existing ones), ClaimCenter creates a customer entity interface with the following Java fully-qualified name:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME`

By default:

- `PACKAGE_PREFIX` is `extensions.cc`, though can be configured. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635.
- `SUBPACKAGE` is defined by the `subpackage` attribute in the entity declaration (or the default) in the data model configuration file. If it is omitted, a default package is used. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

### Configuring Entity and Typelist Fully-Qualified Names from Java

You can control the Java package names for entity types that you add in ClaimCenter data model configuration XML files. As mentioned earlier, the generated entity package includes a package prefix and a subpackage using the following pattern for fully-qualified entity type names:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME`

There are several ways to configure the package of entity types in Java, as described in the following table.

Package configuration for Java entity or type-list types	How to configure	Affects types in the base configuration and not in the extensions folder	Affects types that contain customer extensions	Affects types that you add, including typelists and subtypes of existing entities
custom subpackage	<p>In an entity type or typelist declaration, there is an attribute called subpackage. Set this attribute to a custom subpackage.</p> <p>If you omit the subpackage attribute, ClaimCenter uses a default subpackage. See later in this table regarding configuring the default subpackage.</p>	No	No	<b>Yes</b>
default subpackage	<p>If you omit the subpackage attribute on the entity or typelist definition, ClaimCenter uses a default subpackage. To change the default subpackage, edit the file <code>extensions.properties</code> in Studio. Change the property <code>gw.pl.metadata.codegen.package.default.subpackage</code> to your desired subpackage.</p> <p>In the base configuration, the default subpackage is <code>claim</code>.</p> <p>In the base configuration of <code>ContactManager</code>, the default subpackage is <code>contact</code>.</p>	No	No	<b>Yes</b>
package prefix	<p>You can change the package prefix from Java for entity types that you create by editing the file <code>extensions.properties</code> in Studio. Change the property <code>gw.pl.metadata.codegen.package.prefix</code> to the desired prefix. In the default configuration, the package prefix is <code>extensions.cc</code>.</p> <p>This package prefix is used for generating entity type interfaces for two cases:</p> <ul style="list-style-type: none"> <li>Entity types that you originally create</li> <li>Customer extension entity interface, which ClaimCenter generates when you extend an existing entity type with new properties. See “Customer Extension Entity Interface” on page 634.</li> </ul> <p>In contrast, ClaimCenter ignores this package prefix configuration when generating:</p> <ul style="list-style-type: none"> <li>The base entity interface</li> <li>The core entity interface</li> </ul> <p>For typelists, the same rules apply for the package prefix in typelist classes. ClaimCenter uses the package prefix in generating typelist types for:</p> <ul style="list-style-type: none"> <li>Typelists that you originally create</li> <li>Customer extension typelist classes, which ClaimCenter generates when you extend an existing typelist. See “Typecode Classes from Java” on page 636.</li> </ul> <p><b>Warning:</b> In the file <code>extensions.properties</code>, there is an additional package prefix <code>gw.pl.metadata.codegen.package.prefix.internal</code>. That is for internal use only. Do not change it.</p>	No	<b>Yes</b>	<b>Yes</b>

## Typecode Classes from Java

The Java API exposes typelists and typecodes as Java classes, in contrast to entity types which ClaimCenter exposes as interfaces.

To access a typecode, first get a reference to the appropriate typelist class. The package naming is similar to the pattern for entity data, with some differences. See “Entity Packages and Customer Extensions from Java” on page 632.

ClaimCenter exposes each typelist type as several classes, which the following table describes:

- For the rightmost column, *TL* represents the typelist name, and *SUBPACKAGE* represents the subpackage. The typelist type declaration `subpackage` attribute specifies the *SUBPACKAGE*. This attribute only exists on the original `.tti` file that declares the typelist.
- If the typelist type declaration omits the `subpackage` attribute, a default is used based on whether the typelist is a *platform typelist*. If the typelist declaration attribute `platform` has value `true`, the typelist is a Guidewire platform typelist.
- Whether a typelist is a platform typelist also affects the package prefix. Refer to the table for details.

Terminology	Description	Fully-qualified type name
<b>Typelists that Guidewire defines</b>		
<i>base typelist class</i>	This class contains only the base configuration typecodes. All typelists that Guidewire creates have a base typelist class. This class contains typecodes in the following data model files: <code>.tti</code>	<p>For platform typelists:</p> <ul style="list-style-type: none"> <li>The class is <code>gw.p1.SUBPACKAGE.typekey.TL</code></li> <li>The default subpackage is <code>platform</code> and is not configurable.</li> </ul> <p>For non-platform typelists:</p> <ul style="list-style-type: none"> <li>The class is <code>gw.cc.SUBPACKAGE.typekey.TL</code></li> <li>The default subpackage is <code>claim</code>.</li> <li>For ContactManager, the default subpackage is <code>contact</code></li> <li>The default subpackage is configurable in the <code>extensions.properties</code> file. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635</li> </ul> <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> <li>Class name is <code>BatchProcessType</code></li> <li>Package is <code>gw.p1.batchprocessing.typekey</code></li> </ul>
<i>internal extension typelist class</i>	This class contains base typecodes and application-specific typecodes. This class exists only if the Guidewire platform defines a base version but ClaimCenter provides additional typecodes for application-specific use. This class contains typecodes in the following data model files: <code>.tti</code> , <code>.tix</code>	<p>For all internal extension typelist classes:</p> <ul style="list-style-type: none"> <li>The class is <code>gw.cc.SUBPACKAGE.typekey.TLConstants</code></li> <li>The subpackage is defined by the subpackage of the typelist this extends. If it is undeclared, the default is <code>platform</code> and is not configurable.</li> </ul> <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> <li>Class name is <code>BatchProcessTypeConstants</code></li> <li>Package is <code>gw.cc.batchprocessing.typekey</code></li> </ul>

Terminology	Description	Fully-qualified type name
<b>Typelist extensions and new typelists</b>		
<i>customer extension typelist class</i>	<p>This class contains base typecodes, application-specific typecodes, and customer typecodes. This class exists only if there are customer data model extensions. If an internal extension typelist class exists for that typelist type, the customer extension typelist class includes everything from the internal extension typelist class. Otherwise, the customer extension typelist class includes typecodes from the base typelist class. This class contains typecodes in the following data model files: .tti, .tix, .tx.</p>	<p>For all customer extension typelist classes:</p> <ul style="list-style-type: none"> <li>The class is: <code>PACKAGE_PREFIX.SUBPACKAGE.typekey.TLExtConstants</code></li> <li><code>SUBPACKAGE</code> is defined by the subpackage of the typelist this extends. Refer to the row in this table for the <i>base typelist class</i>.</li> <li>By default, <code>PACKAGE_PREFIX</code> is <code>extensions.cc</code> but is configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635</li> </ul> <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> <li>Class name is <code>BatchProcessTypeExtConstants</code></li> <li>Package is <code>extensions.cc.batchprocessing.typekey</code></li> </ul>
<i>customer typelist class</i>	<p>This class contains all typecodes. All typelists that you create have this class. Contains typecodes in the following data model files: .tti.</p>	<p>For all customer typelist classes:</p> <ul style="list-style-type: none"> <li>The class is <code>PACKAGE_PREFIX.SUBPACKAGE.typekey.TL</code></li> <li>The default subpackage is <code>claim</code>.</li> <li>For <code>ContactManager</code>, the default subpackage is <code>contact</code></li> <li>The default subpackage is configurable in the <code>extensions.properties</code> file. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635</li> <li>By default, <code>PACKAGE_PREFIX</code> is <code>extensions.cc</code> but is configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 635.</li> </ul> <p>For example, for typelist <code>Example</code> with omitted subpackage:</p> <ul style="list-style-type: none"> <li>Class name is <code>Example</code></li> <li>Package is <code>extensions.cc.claim.typekey</code></li> </ul>

### Getting Typecodes Using the get method

The static properties on a typelist that represent a typecode have the `TC_` prefix, just like from Gosu. However, to actually work with the typecode, you must call the `get` method on the static property. The `get` method returns the appropriate typecode object.

For example, suppose you want the typecode `Approval` from a fictional example typelist called `ExampleType`. Assume that the typelist is declared with the example subpackage `testsub`.

- If the type is in the base configuration for multiple Guidewire applications, use the syntax:  
`tc = gw.pl.testsub.typekey.ExampleType.TC_APPROVAL.get()`
- If the type is defined in the base configuration and extended for ClaimCenter, use the syntax:  
`tc = gw.cc.testsub.typekey.ExampleTypeConstants.TC_APPROVAL.get()`
- If the type is defined in a customer data model extension, use the syntax:  
`tc = extensions.cc.testsub.typekey.ExampleTypeExtConstants.TC_APPROVAL.get()`

If you forget to call the `get` method, the resulting object is inappropriate for comparing typecodes, as well as most other contexts. For related information, see “Comparing Entity Instances and Typecodes” on page 639.

Wherever possible, use the static constants for typecode values. Using the typecode static constants ensures that your code is type safe. Mistakes in the typecode string are caught as compile errors rather than only at run time. If the code is known at compile time, always use the static constants to get the typekey object.

In the rare cases where you need to get a typecode reference from a `String` value known only at run time, use the `getTypeKey` method on the typelist class. Be warned that the `getTypeKey` method uses reflection, and typecode values cannot be validated at compile time. For example:

```
tc = gw.pl.testsub.typekey.ExampleType.getTypeKey(typeCodeString)
```

### Entity Subtype Typelists

In addition to standard typelists, there are special typelists that exist only to distinguish *subtypes* of entities. These *subtype typelists* help ClaimCenter specify the entity subtype in each database row.

The typelist name matches the name of the entity type. Based on this typelist name, ClaimCenter creates the typelist classes for the Java API using the same basic pattern as regular typelists in “Typecode Classes from Java” on page 636.

If the entity subtype is shared among multiple Guidewire applications, the Java class name is:

```
gw.pl.SUBPACKAGE.typekey.ENTITYNAME
```

If ClaimCenter overrides the entity subtype, there is a typelist class with the Java class name:

```
gw.cc.SUBPACKAGE.typekey.ENTITYNAMEConstants
```

If you extend an entity subtype of a pre-existing entity, there is a typelist class with the Java name:

```
PACKAGE_PREFIX.SUBPACKAGE.typekey.ENTITYNAMEExtConstants
```

If you created the entity subtype, there is a typelist class with the Java name:

```
PACKAGE_PREFIX.SUBPACKAGE.typekey.ENTITYNAMEExtConstants
```

## Comparing Entity Instances and Typecodes

To compare entity instances for equality, always use the `entity.equals(otherEntity)` method. Using the `==` (double equals) operator is unsafe to test for equality.

To compare typecode values for equality, similarly use the `equals` method. However, be sure that you have a reference to the actual typecode class using the `get` method as mentioned in “Typecode Classes from Java” on page 636.

For example, suppose that the code `claim.getState()` returns a typecode type called `ClaimState` and you want to compare the result to a specific value.

It is incorrect to do either of the following because the Java `==` (double equals) operator is unsupported for comparing typecodes:

```
claim1.getState() == claim2.getState()  
claim.getState() == ClaimState.TC_OPEN.get()
```

The following line is incorrect because it omits the `get` method after getting a static instance by name:

```
claim.getState().equals(ClaimState.TC_OPEN)
```

Instead, use the following expression syntax:

```
claim.getState().equals(ClaimState.TC_OPEN.get())  
claim1.getState().equals(claim2.getState())
```

Be very careful with code that compares two entities or two typecodes.

## Entity Bundles and Transactions from Java

Before writing Java code, regenerate the Java API libraries. This process ensures that any data model changes or extensions are in the most recently generated libraries. See “Regenerating Java API Libraries” on page 631.

If you implement a Java plugin, see “Plugin Overview” on page 163 and “Special Notes For Java Plugins” on page 170.

## Getting a Reference to an Existing Bundle in Java

To use entity instances, in many cases you need a reference a *bundle*. A bundle is a programmatic abstraction that represents one database transaction. See “Bundles and Database Transactions” on page 331 in the *Gosu Reference Guide*.

There are some programming contexts in ClaimCenter in which there is a current database transaction, which means there is a current *bundle*. For example, the following code contexts include a current bundle:

- Code called from business rules
- Plugin interface implementation code, or code called by a plugin implementation
- Most PCF user interface application code

---

**WARNING** There are many contexts in which it is unsafe to commit a bundle. For example, all plugin code and most PCF code. For more information, see “Bundles and Database Transactions” on page 331 in the *Gosu Reference Guide*.

---

To get the current bundle from Java, use the same API as in Gosu:

```
gw.pl.persistence.core.Bundle b = gw.transaction.Transaction.getCurrent();
```

If there is no current bundle, you must create a bundle before creating entity instances or updating entity instances that you get from a database query.

## New Bundles In Java

In general, Java code does not need to create a new bundle because there is already a bundle to use for that kind of code context. There are rare cases in which you need to create a new bundle, but only do this if necessary. If you have questions, please contact Guidewire Customer Support.

---

**WARNING** There are many contexts in which it is unsafe to commit a bundle. For example, all plugin code and most PCF code. For more information, see “Bundles and Database Transactions” on page 331 in the *Gosu Reference Guide*.

---

You can directly create a new bundle using the `newBundle` method on the `Transaction` class:

```
import gw.pl.persistence.core.Bundle;
import gw.transaction.Transaction;

...
Bundle bundle = Transaction.newBundle();
```

Additionally, you can use the `runWithNewBundle` API, which is the same as in Gosu for this task. However, the corresponding Gosu method takes a Gosu block as an argument. From Java, the syntax is slightly different, with an anonymous class instead of a Gosu block. For example:

```
gw.transaction.Transaction.runWithNewBundle(new Transaction.BlockRunnable() {
    @Override
    public void run(Bundle bundle) {
        // your code here...
    }
    // The bundle commits automatically if no exceptions happen before the end of the run method
});
```

Also see “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide*.

## Creating New Entity Instances from Java

The recommended API for creating an entity instance is to call the `newInstance` method on the entity type’s `TYPE` property. Pass a bundle reference as a method argument. Depending on whether you need customer extension properties, the syntax varies. For discussion of the two separate interfaces for each entity type, see “Entity Packages and Customer Extensions from Java” on page 632.

From Java plugin code, there is a current bundle. You can use the `Transaction.getCurrent()` method to get the current bundle if one exists.

If there is a current bundle, create a new `Address` entity instance using the following code:

```
import gw.pl.persistence.core.Bundle;
import gw.pl.contact.entity.Address;
import extensions.cc.contact.entity.AddressExt;
import gw.transaction.Transaction;

...
Bundle b = Transaction.getCurrent();

// if you need only the base entity interface properties and methods
Address a1 = Address.TYPE.newInstance(b);

// if you need customer extension properties, downcast to the customer extension interface...
AddressExt a2 = (AddressExt) Address.TYPE.newInstance(b);
```

#### Alternative API for New Instances

Although not recommended for typical use, you can also create a new entity instance with the `newBeanInstance` method on the `Bundle` class as shown in the following example:

```
Bundle b = Transaction.getCurrent();
Address a1 = (Address) b.newBeanInstance(Address.TYPE.get());
```

This approach requires explicit downcasting from the base class of all entity types and thus it is easier to make mistakes that cannot be caught at compile time.

## Getting and Setting Entity Properties from Java

Entity properties appear from Java as getter and setter methods. Getter and setter methods are methods to get or set properties with names that start with `get` or `set`. For example, a readable and writable property named `MyField` appears as the methods `getMyField` and `setMyField`. Read-only properties do not expose a `set` method on the object. If the property named `MyField` contains a value of type `Boolean` or `boolean`, it appears in the interface as `isMyField` instead of `getMyField`.

Examples:

```
address.setFirstName("John");
lastName = address.getLastName();
tested = someEntity.isFieldname();
```

If the property is an extension property, you must use the entity extension interface as discussed in “Entity Packages and Customer Extensions from Java” on page 632. Read that topic carefully for the fully-qualified names of the interfaces and a code example of downcasting.

## Calling Entity Object Methods from Java

Most entity methods on entity instances appear as regular methods, for example:

```
claim.addEvent("MyCustomEventName");
```

In some cases you need to know how the method is declared before you can write the necessary Java code:

- The base entity interface may declare the method, in which case no special downcast is necessary.
- The core extension entity interface may declare the method, in which case downcast to that interface or the customer extension interface. See “Accessing Entity Properties and Methods With Base and Core Extension Interfaces” on page 633.
- The customer extension entity interface may declare the method, in which case you may need to downcast to the customer extension interface. See “Customer Extension Entity Interface” on page 634.
- A Gosu enhancement may declare the method. See “Gosu Enhancement Properties and Methods in Java” on page 643.

## Querying for Entity Data in Java

In the Java API, if you need to find entity instances, use the query builder API. See “Overview of the Query Builder APIs” on page 125 in the *Gosu Reference Guide*.

The Java API syntax to those classes is nearly identical to the syntax in Gosu for APIs that do not use Gosu blocks.

For some of the query API that uses Gosu blocks, there is no direct Java equivalent.

For other APIs that use blocks as arguments or return values, there are special Java-specific variants of the methods. Contact Guidewire Customer Support for details.

## Accessing Gosu Classes from Java Using Reflection

From Java, you must use *reflection* to access Gosu types. Reflection means asking the type system about types at run time to get data, set data, or invoke methods. Reflection is a powerful way of accessing type information at run time, but is not type safe. For example, language elements such as class names and method names are manipulated as `String` values. Be careful to correctly type the names of classes and methods because the Java compiler cannot validate the names at compile time.

To use reflection from Java or Gosu, use the utility class `gw.lang.reflect.ReflectUtil`. The `ReflectUtil` class contains various APIs to get information about a class, get information about features (properties and methods), and invoke object methods or instance methods.

For example, the following Java code calls a static method on a Gosu class using the `ReflectUtil` method `invokeStaticMethod`.

The Gosu class definition:

```
package test1

class MyGosuClass {
    public static function myMethod(input : String) : String {
        return "MyGosuClass returns the value ${input} as the result!"
    }
}
```

To call the static method from Java with the argument "hello", use the following code:

```
package test1;

import gw.lang.reflect.ReflectUtil;
import gw.transaction.Bundle;

public class MyClass {
    public void doIt() {

        // call a static method on a Gosu class
        String r = (String) ReflectUtil.invokeStaticMethod("test1.MyGosuClass", "myMethod", "hello") ;

        // print the return result
        System.out.print(r);
    }
}
```

If you write your own Gosu class and you want to call methods on it from Java, there is pattern that increases the degree of type safety. See the following procedure.

### To make your Java code that calls Gosu classes more typesafe

1. Create a Java interface containing only the set of methods that you need to call from Java. For getting and setting properties, define the interface using getter and setter methods, such as `getMyField` and `setMyField`.
2. Create a Gosu class that implements that interface. It can contain additional methods if desired if they are not needed from Java.

3. In code that needs to get a reference to the object, use `ReflectUtil` to create an instance of the desired Gosu class. There are several approaches:

- Create an instance using `ReflectUtil`.
- Alternatively, define a method on an object that creates an instance. Next, call that method with `ReflectUtil`.

In both cases, at compile time any new object instances returned by `ReflectUtil` have the type `Object`. At run time, downcast to your new Java interface and assign to a new variable of that interface type.

4. You now have an instance of an object that conforms to your interface. Call methods on it as you normally would from Gosu. The getters, setters, and other method names are defined in your interface, so the method calls are typesafe. For example, the Java compiler can protect against misspelled method names.

## Gosu Enhancement Properties and Methods in Java

Gosu enhancements are a way of adding properties and methods to a type, even if you do not control the source code to the class. Gosu enhancements are a feature of the Gosu type system. See “Enhancements” on page 227 in the *Gosu Reference Guide*.

Gosu enhancements are not directly available on types from Java.

You can use the `gw.lang.reflect.ReflectUtil` class to call enhancement methods and access enhancement properties. The syntax is more complex than it would be from Gosu. Because it uses reflection, it is less typesafe. There is more chance of errors at run time that were not caught at compile time. For example, if you misspell a method name passed as a `String` value, the Java compiler cannot catch the misspelled method name at compile time.

For more information about reflection and the `gw.lang.reflect.ReflectUtil` class, see “Accessing Gosu Classes from Java Using Reflection” on page 642.

## Class Loading and Delegation for non-OSGi Java

### Java Class Loading Rules

To load custom Java code into Gosu or to access Java classes from Java code, the Java virtual machine must locate the class file with a *class loader*. Class loaders use the fully-qualified package name of the Java class to determine how to access the class.

ClaimCenter follows the rules in the following list to load Java classes, choosing the first rule that matches and then skipping the rules listed after it:

#### 1. General delegation classes

The following classes *delegate load*, which means to delegate class loading to a parent class loader:

- `javax.*` - Java extension classes
- `org.xml.sax.*` - SAX 1 & 2 classes
- `org.w3c.dom.*` - DOM 1 & 2 classes
- `org.apache.xerces.*` - Xerces 1 & 2 classes
- `org.apache.xalan.*` - Xalan classes
- `org.apache.commons.logging.*` - Logging classes used by WebSphere

#### 2. Internal classes

If the class name starts with `com.guidewire`, then ClaimCenter delegate loads in general, but there are some internal classes that locally load.

---

**WARNING** Java code you deploy must never access any internal classes other than supported classes and documented APIs. Using internal classes is dangerous and unsupported. If in doubt about whether a class is supported, immediately ask Customer Support. Never use classes in the package `com.guidewire`.

---

### 3. All your classes

Any remaining non-internal classes load locally.

---

**WARNING** Java classes that you deploy must **never** have a fully-qualified package name that starts with `com.guidewire`. Additionally, never rely on classes with that prefix because they are internal and unsupported.

---

## Java Class Delegate Loading

If the ClaimCenter class loader delegates Java class loading, ClaimCenter requests the parent class loader to load the class, which is the ClaimCenter application. If the ClaimCenter application cannot find the class, then the ClaimCenter class loader attempts to load the class locally.

## Deploying Non-OSGi Java Classes and JARs

The following deployment instructions work for non-OSGi Java class files or JAR files, independent of whether your code uses Guidewire entity data. Carefully deploy Java class files and JAR files in the locations defined in this topic. Putting files in other locations is dangerous and unsupported. If you are deploying an OSGi plugin, see the separate section “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 645.

**Note:** This section discusses the deployment options for the Java API introduced in version 8.0. If you are using the deprecated Java API from ClaimCenter 7, see “Important Changes for Java Code” on page 53 in the *New and Changed Guide*.

Place your non-OSGi Java classes and libraries in the following locations. Any subdirectories must match the package hierarchy. The `shared` directory works for any non-OSGI Java code.

If the class implements a ClaimCenter plugin interface, you can define a separate plugin directory in the ClaimCenter Studio in the Plugins registry for that interface. In the following paths, `PLUGIN_DIR` represents plugin directory that you define in the Plugins registry. If the `Plugin Directory` field in the Studio Plugins registry is empty, the default is the plugin directory name `shared`. For more about defining plugin directories in ClaimCenter Studio, see “Adding a New Plugin Interface Implementation” on page 114 in the *Configuration Guide*. Also see “Registering a Plugin Implementation Class” on page 166.

If any Gosu class calls your Java code, for the `PLUGIN_DIR` value you can also use the value `Gosu` instead of `shared`. Be careful to notice the capitalization of `Gosu`.

Place non-OSGi Java classes in the following locations as the root directory of directories organized by package:

```
ClaimCenter/modules/configuration/plugins/shared/basic/classes  
ClaimCenter/modules/configuration/plugins/PLUGIN_DIR/basic/classes // for Java plugin code only
```

For example, for a class file with fully qualified name `mycompany.MyClass`, create files at one of the following:

```
ClaimCenter/modules/configuration/plugins/shared/basic/classes/mycompany/MyClass.class  
ClaimCenter/modules/configuration/plugins/PLUGIN_DIR/basic/classes/mycompany/MyClass.class
```

Place your libraries (JAR files) and any third-party libraries in the following locations:

```
ClaimCenter/modules/configuration/plugins/shared/basic/lib  
ClaimCenter/modules/configuration/plugins/PLUGIN_DIR/basic/lib // for Java plugin code only
```

# OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor

For a summary of the IDE options for OSGi plugin development, see “IDE Options for Plugin Development in the Java Language” on page 629.

This topic describes specific steps for OSGi plugin development:

- “Generate Java API Libraries” on page 645
- “Launch IntelliJ IDEA with OSGi Editor” on page 645
- “Create an OSGi-compliant Class that Implements a Plugin Interface” on page 646
- “Compiling and Installing Your OSGi Plugin as an OSGi Bundle” on page 647
- “Using Third-Party Libraries in Your OSGi Plugin” on page 649

## Generate Java API Libraries

Before starting work with Java and OSGi, regenerate the ClaimCenter Java API libraries with the `regen-java-api` tool. See “Regenerating Java API Libraries” on page 631. This is a requirement that is independent of which Java IDE that you use.

## Launch IntelliJ IDEA with OSGi Editor

For OSGi plugin development, Guidewire recommends that you use the included application IntelliJ IDEA with OSGi Editor. To launch IntelliJ IDEA with OSGi Editor, open a command prompt in the `ClaimCenter/bin` directory and type the following:

```
gwcc plugin-studio
```

**IMPORTANT** If you use other Guidewire applications, such as Guidewire ContactManager, each Guidewire application includes its own version of IntelliJ IDEA with OSGi Editor. Be sure to run this command prompt from the correct application product directory.

## Creating a New Project With OSGi Plugin Module

1. If this command is executed for the first time for one Guidewire product, IntelliJ starts with an empty workspace and no current project
2. To create a new project, select **Create New Project**. In the module type list, click **OSGi Plugin Module**. In the **Project name** field, type the project name. In the **Project location** field, type the project location. Do not yet click **Finish**. Alternatively, if you want to add or import an OSGi module to an existing empty project, in the **Project Structure** dialog, select **Empty Project** and set the **Project name** field. Next, add a module in the **Project Structure** dialog by clicking the plus sign (+). Choose either **New Module** (for new module) or **Import Module** (to select another module to import). For a new module, select the module type **OSGi Plugin Module**. In the **Module name** field, type the module name. Continue to follow the rest of this procedure.
3. Open the **More Settings** pane, which may be initially closed. Set the name of the new module as appropriate. By default, the **Bundle Symbolic Name** field matches the name of the module. You can optionally change the symbolic name to a different value in this dialog. The bundle symbolic name defines the main part of the output JAR file name before the version number. You can also optionally change the version of the bundle in the **Bundle Version** field.
4. Click **Finish**.
5. If this is a new project, you must set the project JDK:
  - a. Click **File → Project Structure → Project Settings**. In the **Project** section, set the **Project JDK** picker to your Java 7 JDK, which might be labelled 1.7.

- b. If there is no Java 7 JDK listed, click the **New...** button. Click **JDK**, then select your Java 7 JDK on your disk. Next, set the **Project JDK** picker to your newly-created Java 7 JDK configuration.
6. For advanced OSGi settings, see “Advanced OSGi Dependency and Settings Configuration” on page 652.

## Create an OSGi-compliant Class that Implements a Plugin Interface

Follow the procedure below to implement an OSGi plugin with IntelliJ IDEA with OSGi Editor.

### To implement a new OSGi plugin

1. Remember to regenerate the Java libraries. See “Generate Java API Libraries” on page 645.
2. In IntelliJ IDEA with OSGi Editor, navigate under your OSGi module to the **src** directory.
3. Create new subpackages as necessary. Right-click on **src** and choose **New → Package**. To follow along with an example of a simple startable plugin, create a **mycompany** package to contain your new classes.
4. Right-click on the desired package for your class.
5. Choose **New → New OSGi plugin**. IntelliJ IDEA with OSGi Editor opens a dialog.
6. In the **Plugin class name** field, enter the name of the Java class that you want to create. For our example, type **DemoStartablePlugin**.
7. In the **Plugin interface** field, enter a fully qualified name of the plugin interface you want to implement. Alternatively, to choose from a list, click the ellipsis (...) button. Type some of the name or scroll to the desired plugin interface. Select the desired interface and then click **OK**.
8. IntelliJ IDEA with OSGi Editor displays a dialog **Select Methods to Implement**. By default, all methods are selected. Click **OK**.
9. IntelliJ IDEA with OSGi Editor displays your new class with stub versions of all required methods.
10. If the top of the Java class editor has a yellow banner that says **Project SDK is not defined**, you must set your project SDK to a JDK. See “Creating a New Project With OSGi Plugin Module” on page 645. If the SDK settings are uninitialized or incorrect, your project shows many compilation errors in your new Java class.
11. If you have several tightly-related OSGi plugin implementations, you can optionally deploy them in the same OSGi bundle. If you have additional plugins to implement in this same project, repeat this procedure for each plugin implementation class.

For example, one OSGi bundle can encapsulate messaging code for a messaging destination. A messaging destination may implement the **MessageTransport** interface. The same messaging destination may optionally also implement the **MessageReply** and **MessageRequest** plugin interfaces. If the multiple plugin implementations have shared code and third-party libraries, you could deploy them in the same OSGi bundle.

### Example Startable Plugin in Java Using OSGi

The following example defines a class that implements the **IStartablePlugin** interface. The following class simply prints a message to the console for each application call to each plugin method. Use the following example to confirm you can successfully deploy a basic OSGi plugin using IntelliJ IDEA with OSGi Editor.

Create a class with fully-qualified name **mycompany.DemoStartablePlugin** with the following contents:

```
package mycompany;  
  
import aQute.bnd.annotation.component.Activate;  
import aQute.bnd.annotation.component.Component;  
import aQute.bnd.annotation.component.ConfigurationPolicy;  
import aQute.bnd.annotation.component.Deactivate;  
import gw.api.startable.IStartablePlugin;  
import gw.api.startable.StartablePluginCallbackHandler;  
import gw.api.startable.StartablePluginState;  
  
import java.util.Map;
```

```
@Component(provide = IStartablePlugin.class, configurationPolicy = ConfigurationPolicy.require)
public class DemoStartablePlugin implements IStartablePlugin {

    private StartablePluginState _state = StartablePluginState.Stopped;

    private void printMessageToGuidewireConsole(String s) {
        System.out.println("*****");
        System.out.println("***** --> STARTABLE PLUGIN METHOD = " + s);
        System.out.println("*****");
    }

    @Activate
    public void activate(Map<String, Object> config) {
        printMessageToGuidewireConsole("activate -- OSGi plugin init");
        // The Map contains the plugin parameters defined in the Plugins registry in Studio
        // There are additional OSGi-specific parameters in this Map if you want them.
    }

    @Deactivate
    public void deactivate() {
        printMessageToGuidewireConsole("deactivate -- OSGi plugin shutdown");
    }

    // Other IStartablePlugin interface methods...

    @Override
    public void start(StartablePluginCallbackHandler startablePluginCallbackHandler,
                      boolean b) throws Exception {
        printMessageToGuidewireConsole("start");
    }

    @Override
    public void stop(boolean b) {
        printMessageToGuidewireConsole("stop");
    }

    @Override
    public StartablePluginState getState() {
        printMessageToGuidewireConsole("getState");
        return _state;
    }
}
```

## Compiling and Installing Your OSGi Plugin as an OSGi Bundle

The main script for OSGi compilation and deployment is an Ant build script. See “Advanced OSGi Dependency and Settings Configuration” on page 652.

The Ant build script does the following:

1. compiles Java code
2. generates OSGi metadata
3. packages code and library dependencies into an OSGi bundle
4. installs an OSGi bundle to the correct ClaimCenter bundles directory as defined in the OSGi plugin project’s `build.properties` file. For related information, see “Advanced OSGi Dependency and Settings Configuration” on page 652. The script copies your final JAR file to the following ClaimCenter directory:  
`ClaimCenter/modules/configuration/deploy/bundles`

### To compile and install your OSGi plugin implementation as an OSGi bundle implementation

1. Open a command prompt in the directory that contains your OSGi plugin module.
2. Type the following command:

```
ant install
```

The command generates messages about compiling source files, generating files, copying files, and building a JAR file. If it succeeds, it prints:

```
BUILD SUCCESSFUL
```

3. Switch to or open the regular ClaimCenter Studio (not IntelliJ IDEA with OSGi Editor) application. Navigate in the Project pane to the path **configuration** → **deploy** → **bundles**. Confirm that you see the newly-deployed file **YOUR\_JAR\_NAME.jar**. The JAR name is based on the module symbolic name followed by the version.

**IMPORTANT** If the JAR file is not present at that location, check the Ant console output for the directory path that the script copied the JAR file. If you recently installed a new version of ClaimCenter at a different path, or moved your Guidewire product directory, you must immediately update your OSGi settings in your OSGi project. See “[Updating Your OSGi Plugin Project After Product Location Changes](#)” on page 653.

4. In ClaimCenter Studio (not IntelliJ IDEA with OSGi Editor), register your OSGi plugin implementation. Registering a plugin implementation defines where to find a plugin implementation class and what interface the class implements.
- In the Project window, navigate to **configuration** → **config** → **Plugins** → **registry**. Right-click on the item **registry**, and choose **New** → **Plugin**.
  - In the plugin dialog, enter the plugin name in the **Name** field. For our example, use **DemoStartablePlugin**.  
For plugin interfaces that only support one implementation, just enter the interface name without the package. For example, **IStartablePlugin**. The text you enter becomes the basis for the file name that ends in **.gwp**. The **.gwp** file represents one item in the Plugins registry.  
If the plugin interface supports multiple implementations, like messaging plugins or startable plugins, this can be any arbitrary name. If you are registering a messaging plugin, the *plugin name* must match the plugin name in fields in the separate **Messaging** editor in Studio.  
For our demonstration implementation of the **IStartablePlugin** interface, enter the plugin name **DemoStartablePlugin**.
  - In the plugin dialog, next to the **Interface** field, click the ellipsis (...), find the interface class, and select it. If you want to type it, enter the interface name without the package.  
For our demonstration implementation of the **IStartablePlugin** interface, find or type the plugin name **IStartablePlugin**.
  - Click **OK**.
  - In the Plugins registry main pane for that **.gwp** file, click the plus sign (+) and select **Add OSGi Plugin**.
  - In the **Service PID** field, type the fully-qualified Java class name for your OSGi implementation class. Note that this is different from the bundle name or the bundle symbolic name. The ellipsis (...) button does not display a picker to find available OSGi classes, so you must type the class name in the field.  
For our demonstration implementation of the **IStartablePlugin** interface, type the fully-qualified class name **mycompany.DemoStartablePlugin**.
  - Set any other fields that your plugin implementation needs, such as plugin parameters.

**IMPORTANT** For more information about the Plugins registry, see “[Plugin Overview](#)” on page 163

- Make whatever other changes you need to make in the regular ClaimCenter Studio application. For example, if your plugin is a messaging plugin, configure a new messaging destination. See “[Using the Messaging Editor](#)” on page 137 in the *Configuration Guide* and “[Messaging and Events](#)” on page 299. You may need to make other changes such as changes to your rule sets or other related Gosu code.
- Start the server.  
To run the server from Studio, if ClaimCenter Studio was already running when you installed or re-installed the bundle, there is an extra required step before running the server. Open a command prompt in the **ClaimCenter/bin** directory and type:  
`gwcc dev-deploy`

Next, start the server from Studio, such as clicking the **Run** or **Debug** menu items or buttons.

Alternatively, run the QuickStart server from the command line. Open a command prompt in the `ClaimCenter/bin` directory and type the following:

```
gwcc dev-start
```

An earlier topic showed an example startable plugin. See “Create an OSGi-compliant Class that Implements a Plugin Interface” on page 646. If you used that example, closely read the console messages during server startup. The example OSGi startable plugin prints messages during server startup. This proves that ClaimCenter successfully called your OSGi plugin implementation.

## Using Third-Party Libraries in Your OSGi Plugin

If your OSGi plugin code uses third-party libraries, there are two deployment options, depending on your type of third-party JAR file:

- **Embed inside your OSGi bundle** – You can embed any Java library in your OSGi plugin bundle. The library is not required to be an OSGi bundle, but OSGi third-party libraries are also supported. Deploy your library JAR in the module directory within the `inline-lib` subdirectory. See “Embed a Third-Party Java Library in your OSGi bundle” on page 649.
- **Deploy as a separate OSGi bundle (requires third-party JAR to support OSGi)** – If your library contains a properly-configured OSGi bundle with OSGi properties in the manifest, you can optionally deploy it as a separate OSGi bundle outside your main OSGi plugin bundle. For use within your OSGi plugin project, deploy your library JAR in the module directory within the `lib` (not `inline-lib`) subdirectory. See “Deploying a Third-party OSGi-compliant Library as a Separate Bundle” on page 649

### Deploying a Third-party OSGi-compliant Library as a Separate Bundle

If your library contains a properly-configured OSGi bundle with OSGi properties in the manifest, you can optionally deploy it as a separate OSGi bundle outside your main OSGi plugin bundle.

#### To use third-party OSGi-compliant libraries separate from your OSGi plugin bundle

1. Put any third-party OSGi JAR files in the `lib` (not `inline-lib`) folder inside your OSGi module in IntelliJ IDEA with OSGi Editor.
2. Write code that uses the third-party JAR.
3. Confirm there are no compile errors.
4. Open a command prompt at the root of your module and type:

```
ant install
```

The tool generates various messages, and concludes with:

```
BUILD SUCCESSFUL
```

The `ant install` script copies your bundle JAR files to the `ClaimCenter/deploy/bundles` directory.

### Embed a Third-Party Java Library in your OSGi bundle

You can embed any Java library in your OSGi plugin bundle. The library is not required to be an OSGi bundle, but OSGi third-party libraries are also supported. In some cases, you may require special configuration to change how the bundle manifests imports Java packages that your embedded libraries reference.

For example, suppose you want to use a third-party library that has 100 classes, although you only use 5 of them, and only some of the methods on those classes. If the classes and APIs that you use only rely on available and embedded libraries, there is no problem at run time.

It may be that some of the classes in your third-party library but that you do not use have dependencies on external classes. The library might use an API that relies on classes that are unavailable at run time. OSGi tries to avoid this risk by ensuring at server startup that all required classes. Even if you never call those APIs, there will

be errors on server startup because OSGi tries to verify all libraries are available, including ones you never called.

Fortunately, you can modify the configuration file to ignore unavailable packages during OSGi library validation phase on server startup. The following instructions include techniques to mitigate these problems.

#### To use third-party libraries in your OSGi plugin

1. Put any third-party JAR files in the `inline-lib` folder inside your OSGi module in IntelliJ IDEA with OSGi Editor.

2. Write code that uses the third-party JAR.

3. Confirm there are no compile errors.

4. Open a command prompt at the root of your module and type:

```
ant dist
```

The tool generates various messages, and concludes with:

```
BUILD SUCCESSFUL
```

5. Open the file `MODULE_ROOT/generated/META-INF/MANIFEST.MF`.

6. In that file, check that the import package (`Import-Package`) header includes no unexpected packages. All classes of the embedded libraries are copied inside your bundle such that all library classes become part of your bundle. This process this might cause the `Bnd` tool to generate unexpected imports to appear in the list.

7. If you see unexpected imports, change the import package instruction in the `bnd.bnd` file, rather than modifying the `MANIFEST.MF` file.

---

**WARNING** Never directly change the file `MANIFEST.MF`. It is auto-generated.

---

In the `bnd.bnd` file, set the `Import-Package` instruction to a comma-delimited list of Java packages to ignore by including each with the exclamation point (!) prefix. The exclamation point means “not” or “exclude”. Finally add an asterisk (\*) character as the last item in the list to indicate to import all other packages. For example, to ignore packages `javax.inject` and `sun.misc`, set the line as follows:

```
Import-Package=!javax.inject,!sun.misc,*
```

8. Deploy your OSGi plugin and test it. Look for server startup errors that look similar to:

```
ERROR Server.OSGi Could not start bundle messaging-OSGi-plugins from reference:  
file:/D:/GuidewireApp801/webapps/pc/bundles/messaging-OSGi-plugins-1.0.0.jar  
org.osgi.framework.BundleException: The bundle "messaging-OSGi-plugins_1.0.0 [12]" could not  
be resolved. Reason: Missing Constraint: Import-Package: javax.inject; version="0.0.0"
```

If there are errors, repeat this procedure and add more Java packages to the `Import-Package` line in `bnd.bnd`.

For more information about the `bnd.bnd` file, see “Advanced OSGi Dependency and Settings Configuration” on page 652.

#### Example of Embedding Third-Party Libraries in an OSGi Plugin

Suppose your OSGi plugin requires the Java library called Guava, specifically version 15. First, download `guava-15.0.jar` from the project web site. Next, move the `guava-15.0.jar` file to the `inline-lib` folder within your OSGi project within IntelliJ IDEA with OSGi Editor.

Next, write some Java code that uses this library. For example:

```
import com.google.common.escape.Escaper;  
  
...  
  
// use the class com.google.common.escape.Escaper in guava-15.0.jar  
Escaper escaper = XmlEscapers.xmlAttributeEscaper();  
  
s = escaper.escape(s);
```

From a command prompt, run the following command:

```
ant dist
```

That tool generates OSGi metadata and prints various messages. If successful, the final line is:

```
BUILD SUCCESSFUL
```

Open the file MODULE\_ROOT/generated/META-INF/MANIFEST.MF and check that Import-Package header does not include any unexpected packages. Our example file might look like the following:

```
Manifest-Version: 1.0
Service-Component: OSGI-INF/com.qa.MessageTransportOSGiImpl.xml
Private-Package: com.google.common.html,com.google.common.net,com.google.common.collect,
    com.google.common.primitives,com.google.common.base,com.google.common.escape,com.qa,
    com.google.common.base.internal,com.google.common.cache,com.google.common.eventbus,
    com.google.common.util.concurrent,com.google.common.hash,com.google.common.io,
    com.google.common.xml,com.google.common.reflect,com.google.common.math,
    com.google.common.annotations
Bundle-Version: 1.0.0
Tool: Bnd-1.50.0
Bundle-Name: messaging-OSGi-plugins
Bnd-LastModified: 1382994235749
Created-By: 1.7.0_45 (Oracle Corporation)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: messaging-OSGi-plugins
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation,javax.inject,sun.misc
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
```

Note the following line:

```
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation,javax.inject,sun.misc
```

By default, the packages javax.inject and sun.misc are unavailable at runtime. The package javax.inject is a JavaEE package that is not exported by default. If you install the generated OSGi bundle in its current form and start the server, the server generates the following error:

```
ERROR Server.OSGi Could not start bundle messaging-OSGi-plugins from reference:
    file:/D:/GuidewireApp801/webapps/pc/bundles/messaging-OSGi-plugins-1.0.0.jar
org.osgi.framework.BundleException: The bundle "messaging-OSGi-plugins_1.0.0 [12]" could not
    be resolved. Reason: Missing Constraint: Import-Package: javax.inject; version="0.0.0"
```

In this case, both problematic packages are optional.

In the bnd.bnd file, set the Import-Package instruction to a comma-delimited list of Java packages to ignore by including each with the exclamation point prefix. Finally, add an asterisks(\*) character as the last item in the list to indicate to import all other packages. For example, to ignore packages javax.inject and sun.misc, set the line as follows:

```
Import-Package=!javax.inject,!sun.misc,*
```

Run the "ant install" tool again. The scripts embed the guava JAR in your OSGi bundle.

Open the MANIFEST.MF file and notice two changes:

- An additional Ignore-Package instruction.
- The Import-Package instruction does not include the problematic packages.

For example:

```
Manifest-Version: 1.0
Service-Component: OSGI-INF/com.qa.MessageTransportOSGiImpl.xml
Private-Package: com.google.common.html,com.google.common.net,com.google.common.collect,
    com.google.common.primitives,com.google.common.base,com.google.common.escape,com.qa,
    com.google.common.base.internal,com.google.common.cache,com.google.common.eventbus,
    com.google.common.util.concurrent,com.google.common.hash,com.google.common.io,
    com.google.common.xml,com.google.common.reflect,com.google.common.math,
    com.google.common.annotations
Ignore-Package: javax.inject,sun.misc
Tool: Bnd-1.50.0
Bundle-Name: messaging-OSGi-plugins
Created-By: 1.7.0_45 (Oracle Corporation)
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Bundle-Version: 1.0.0
Bnd-LastModified: 1382995392983
Bundle-ManifestVersion: 2
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation
Bundle-SymbolicName: messaging-OSGi-plugins
```

With this change, the server now starts up with no OSGi class loading errors.

## Advanced OSGi Dependency and Settings Configuration

The IntelliJ IDEA with OSGi Editor application configures dependencies and several additional files related to module configuration and Ant build script. These files are created automatically.

You can edit these files directly in the file system if necessary without harming OSGi module settings. The only file that IntelliJ IDEA with OSGi Editor modifies is `build.properties`, to edit the bundle symbolic name and version.

### Dependencies

The IntelliJ IDEA with OSGi Editor auto-creates the following dependencies:

- `PROJECT/MODULE/inline-lib` directory – Directory for third-party Java libraries to embed in your OSGi bundle. See “Embed a Third-Party Java Library in your OSGi bundle” on page 649.
- `PROJECT/MODULE/lib` directory – Directory for third-party OSGi-compliant libraries to use but deploy into a separate OSGi bundle. See “Deploying a Third-party OSGi-compliant Library as a Separate Bundle” on page 649.
- `ClaimCenter/java-api/lib` – The ClaimCenter Java API files that the `regen-java-api` tool creates.

If you have any JAR files that you require, put them in the `lib` or `inline-lib` directories, as specified earlier. You do not need to modify any build scripts to add JAR files.

If you add dependencies in the IntelliJ project from other directories, change the associated `build.xml` (Ant build script) to include those directories.

### Build Properties

The `build.properties` file at the root directory of the module contains the bundle symbolic name, version, and the path to Guidewire product. From within IntelliJ IDEA with OSGi Editor, you can edit these fields but optionally you can directly modify this file as needed. The following is an example of this file:

```
Bundle-SymbolicName=messaging-OSGi-plugins
Bundle-Version=1.0.0
-gw-dist-directory=C:/ClaimCenter/
```

The Ant build script (`build.xml`) uses these properties.

### OSGi Bundle Metadata Configuration File

The OSGi bundle metadata configuration file is called `bnd.bnd`. The contents of this file configures how scripts in the IntelliJ IDEA with OSGi Editor application generate OSGi bundle metadata, such as the `MANIFEST.MF` file and other descriptor files.

By default, this file looks like:

```
Import-Package=*
DynamicImport-Package=
Export-Package=
Service-Component=*
Bundle-DocURL=
Bundle-License=
Bundle-Vendor=
Bundle-RequiredExecutionEnvironment=JavaSE-1.7
-consumer-policy=${range;[==,+)}
-include=build.properties
```

For the format of this file, see:

<http://www.aqute.biz/Bnd/Format>

However, for the `Bundle-SymbolicName` and `Bundle-Version` properties, you must set or change those settings in the `build.properties` file. Both the `bnd.bnd` and `build.xml` file use those properties from the `build.properties` file.

You may need to edit this file if you want to export some Java packages from your bundle, or you want to customize the `Import-Package` header generation.

#### Build Configuration Ant Script

The build configuration Ant script (`build.xml`) relies on properties from the files `build.properties` and `bnd.bnd`.

For advanced OSGi configuration, you can modify the `build.xml` build script.

## Updating Your OSGi Plugin Project After Product Location Changes

After you initially configure a project within IntelliJ IDEA with OSGi Editor, the project includes information that references the Guidewire product directory on your local disk. If you move your product directory, you must update your project with the new product location on your local disk. Similarly, if you upgrade your Guidewire product to a new version with a different install path, you must update the OSGi project.

#### To update your OSGi project after Product Location Changes

1. If you need to move your OSGi project on disk, quit IntelliJ IDEA with OSGi Editor and move the files on disk.
2. Launch IntelliJ IDEA with OSGi Editor from your new Guidewire product location. See “Launch IntelliJ IDEA with OSGi Editor” on page 645.
3. In IntelliJ IDEA with OSGi Editor, open your OSGi plugin project.
4. In IntelliJ IDEA with OSGi Editor, update the module dependency for your OSGi module:
  - a. Open the **Project Structure** window.
  - b. Click the **Modules** item in the left navigation.
  - c. In the list of modules to the right, under the name of your module, click **OSGi Bundle Facet**.
  - d. To the right of the **Guidewire product directory** text field, click the **Change** button.
  - e. Set the new disk path and click **OK**.
  - f. Click **OK** in the dialog. The tool updates IDE library dependencies and the `build.properties` file.
5. Test your new configuration. See “Compiling and Installing Your OSGi Plugin as an OSGi Bundle” on page 647.

