ClaimCenter Integration Guide

Release 6.0.8

Product Name: Guidewire ClaimCenter
Product Release: 6.0.8
Document Name: ClaimCenter Integration Guide
Document Revision: 05-February-2013

Contents

About This Document

Guidewire ClaimCenter integrates with a wide variety of external systems. Use external systems to define external business logic, link with legacy systems, use corporate authentication, link with messaging systems, and more.

This topic includes:

Intended Audience

This document is intended for integration developers who develop software to extend the internal logic of ClaimCenter or to connect ClaimCenter with external systems. For example, information in this guide can help you develop code to connect ClaimCenter to an external, possibly legacy, system that generates new claim numbers.

Assumed Knowledge

This document assumes that you are already familiar with ClaimCenter functionality and how to install, configure, and run ClaimCenter.

This document assumes you are an experienced computer programmer familiar with relational databases, XML, the Java language, and Javadoc-formatted documentation. Although some APIs are Java-specific, Java is not required for all programming interfaces. Some integration methods connect with external code using web services, which are language-neutral and platform-neutral thanks to the web services protocol *SOAP*. However, because many customers use Java for integration code, this document occasionally uses Java syntax to explain API usage.

Related Documents

ClaimCenter Java API Reference Javadoc – Documents the Java API for integration programmers. It contains API details such as classes, interfaces, method parameters, return values, and behavior of each method. See the *Integration Guide* for more details. The Java API Reference Javadoc includes:

- Java plugin interface definitions (also usable to write Gosu plugins)
- Details of ClaimCenter Java entities, including both the entity data and additional methods called domain methods

- General Java utility classes in a separate Javadoc directory.

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

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

ClaimCenter Configuration Guide – Describes how to configure ClaimCenter and includes basic steps and examples for implementing such configurations. This guide is intended for IT staff and system integrators who configure ClaimCenter for an initial implementation or create custom enhancements. This guide is intended as a reference, not to be read cover-to-cover.

Related Specifications

This document provides only a rough overview of standards used in the product and is not intended to be a full specification of any of these standards. ClaimCenter uses the following integration-related 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 web page `http://www.w3.org/XML`.
- **SOAP** – The web services request-and-response protocol based on XML and 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 at `http://www.w3.org/TR/soap`.
- **WSDL (Web Services Description Language)** – The web services API description language, which describes an API interface 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. Related to XML and SOAP. For more information, refer to the World Wide Web Consortium at `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 it generates a Java and Gosu language binding (interface libraries) for programmers connecting to ClaimCenter SOAP APIs. For more information, refer to the Apache web site at the page `http://ws.apache.org/axis`.

Conventions In This Document

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

Support

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

- At the Guidewire Resource Center – `http://guidewire.custhelp.com`
- By email – `support@guidewire.com`
- By phone – +1-650-356-4955

chapter 1

Integration Overview

ClaimCenter can integrate with a wide variety of external systems using a diverse toolbox of services and APIs that can link ClaimCenter with custom code and external systems. This topic is an overview to assist in planning integration projects for a ClaimCenter deployment. This topic discusses technical details critical to successful integration. For example, learn about the various integration types and technical concepts that you must understand before starting.

This topic includes:

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, although this is not generally recommended and you can do almost everything in Gosu. Gosu can even easily call out to legacy Java classes or other Java libraries. Plugins run directly in the same Java virtual machine as ClaimCenter. For more plugin information, see “Plugin Overview”, on page 101. That topic includes a complete table of all plugins in ClaimCenter. 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 acknowledgements. For details, see “Messaging and Events” on page 139.
  - **Authentication plugins** define 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 239.
  - **Document and form plugins** transfer documents to and from a document management system, and also help prepare new documents from templates. Additionally, there are Gosu APIs to create and attach documents. For details, see “Document Management”, on page 249.
  - Other plugins for specific application needs. For example, as generating a claim number (`IClaimNumGenAdapter`), or getting a policy related to a claim (`IPolicySearchAdapter`). For a summary of all plugins, see “Summary of All ClaimCenter Plugins” on page 106.
- **Templates.** Templates generate text-based formats containing a combination 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 more about templates, see “Data Extraction Integration”, on page 279.
- **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 423.

The following table compares the main integration methods.

Integration type	What you write	Description
Web service APIs	• Publishing APIs: writing Gosu classes, and marking them as published services. • Using APIs: writing Java code that calls out to the published SOAP APIs, both built-in APIs and custom-written APIs.	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.
Plugins	• Gosu classes that implement a Guidewire-defined interface. • Java classes that implement a Guidewire-defined interface.	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.
Messaging code	• Gosu code in the Event Fired rule set. . • Messaging plugins • Configure one or more messaging destinations in Studio to use your messaging plugins.	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).
Guidewire XML (GX) Models	• Using the GX model editor in STudio, you customize which properties in an entity (or other type) to export in XML. • Various integration code that uses the XSD it creates.	The GX model editor in Studio lets you create custom schemas (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 262 in the *Gosu Reference Guide*.
Templates	• Text files that contain small amounts of Gosu code	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.
Database import	• Custom tools that take legacy data and convert them into database tables of data.	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.

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 339
`config/metro`	Your Metropolitan (police report/inquiry) files. See “Metropolitan Reporting Bureau Integration”, on page 389
`config/web`	Your web application PCF files.
`config/logging`	Your logging configuration files. See “Logging”, on page 293.
`config/templates`	This contains two types of templates relevant for integration: • Plugin templates. These are Gosu templates that are necessary for a small number of plugin interfaces. These plugin templates extract important properties from entities and generate text that describes the results. When ClaimCenter needs to call the plugin ClaimCenter passes the results to the plugin as text-based parameters. • Optional Gosu templates that you can define for your messaging code. Use templates to define notification letters or other similar messages with much text but a small amount of Gosu code. There are built-in versions some of these templates. To modify the built-in versions, in Studio navigate to **Resources** → **Configuration** → **Other Resources** → **Templates**.
`config/docmgmt`	Document management files. See “Document Management”, on page 249.
`plugins`	Your Java plugin files. To add Java files for plugin support, see “Deploying Java Plugins” on page 115 and “Java Class Loading, Delegation, and Package Naming” on page 137. Also see “Java and Gosu” on page 101 in the *Gosu Reference Guide*. Your plugin files **must** be in appropriate subdirectories of the plugins directory that correspond to how you register the plugin, and consistent with the Java plugin loader rules. See “Plugin Overview” on page 101 for more information. This directory is within the `configuration` module but **not** in its `config` subdirectory. Register your plugin implementations in the Plugin registry in Studio. For more information, see “Plugin Overview”, on page 101 and “Using the Plugins Editor” on page 141 in the *Configuration Guide*. If you register a messaging plugin, you must register this information both of the following registries. • the plugin registry in the plugin editor in Studio • the messaging registry in the messaging editor in Studio. See “Messaging and Events”, on page 139 and “The Messaging Editor” on page 161 in the *Configuration Guide*.

There are other important directories **other** than files in the configuration module:

Directory	Description
`ClaimCenter/bin`	Command line tools such as `gwcc.bat`. Use this for the following integration tasks: • Regenerating the Java API libraries and SOAP API libraries. See “Regenerating the Integration Libraries” on page 17. • Regenerating the Data Dictionary. See “Data Dictionary Documentation” on page 20
`ClaimCenter/java-api`	Java API libraries and documentation. ClaimCenter scripts generate this directory. To regenerate these, see “Regenerating the Integration Libraries” on page 17.
`ClaimCenter/java-api/lib`	Java libraries for Java plugin development and also web service (SOAP) client integration development using Java.
`ClaimCenter/java-api/examples`	Example Java plugin examples.
`ClaimCenter/java-api/doc/api`	API documentation in Javadoc format for Java development
`ClaimCenter/soap-api`	SOAP API libraries and documentation. ClaimCenter scripts generate this directory. To regenerate these, see “Regenerating the Integration Libraries” on page 17.
`ClaimCenter/soap-api/doc/lib`	Java libraries for web service (SOAP) client development

Directory	Description
`ClaimCenter/soap-api/doc/api`	API documentation in Javadoc format for web service (SOAP) development
`ClaimCenter/soap-api/sampledata`	Sample data you can import using tools in `ClaimCenter/admin`
`ClaimCenter/soap-api/wsdl`	Generated WSDL files, only for unusual web services situations. Typically, you do not need these. Dynamically generated WSDL with full endpoint URLs and generated Java libraries usually suffice. For details, see “End-point URLs and Generated WSDL” on page 59.
`ClaimCenter/admin`	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). For details, see “Writing Command Line Tools to Call Web Services” on page 66.

Your code’s final location 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 remotely.

Regenerating the Integration Libraries

You must regenerate the Java API libraries and SOAP API libraries after making 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.

After 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 any 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 of the generated Java library documentation:

  ```
  ClaimCenter/java-api/doc/api
  ```

  The location for the libraries is:

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

- The location of the generated SOAP library documentation:

  ```
  ClaimCenter/soap-api/doc/api
  ```

  The location for the libraries is:

  ```
  ClaimCenter/soap-api/lib
  ```

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, bring up a command line window.
2. Change your working directory with the following command:

   ```
   cd ClaimCenter/bin
   ```

3. Type the command:

   ```
   gwcc regen-java-api
   ```

While regenerating the documentation for the Java or the SOAP files, as part of its normal behavior, the script display some warnings and errors.

Integration Documentation Overview

Use the *Integration Guide* for important integration information. However, also consulting other types of reference documentation during development:

1. API Reference Javadoc Documentation
2. 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 “Required Generated Files for Integration” on page 20. For writing Java plugins, also see “Java and Gosu” on page 101 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 the Integration Libraries” on page 17.

Find the interfaces for ClaimCenter plugins within the package `com.guidewire.cc.plugins.*`. Those interfaces are the requirements for **your** code to correctly implement a plugin interface.

SOAP API Reference Javadoc

The SOAP API Javadoc includes:

- The specification of the web service API interfaces, also known as the SOAP APIs.
- The details of SOAP entities, including the entity data only (get and set methods). For a high-level overview of entity differences and different entity access types, see the diagram in “Required Generated Files for Integration” on page 20.
- SOAP Java utility classes, such as `APILocator`.

- After regenerating these files, find it in `ClaimCenter/soap-api/doc/api/index.html`

  Note: To regenerate these files, see “Regenerating the Integration Libraries” on page 17.

The web service API publishers define the APIs in the web services description language (WSDL) and implemented using the SOAP protocol, and as such they are language-neutral. However, the Javadoc-based documentation provides developers using other programming languages the necessary information for understanding ClaimCenter APIs. For those who want to refer to the WSDL definition of ClaimCenter web services, ClaimCenter creates the WSDL in two ways:

- generated local WSDL files, as part of the `regen-soap-api` target.
- dynamically publishing WSDL for web services from the ClaimCenter web application.

The entities (Guidewire objects) described in that documentation are SOAP entities. SOAP entities are simple versions of the objects. They contain only the entity data accessed with getter and setter methods. SOAP entities do not include additional methods (*domain methods*) with complex logic such as entities accessed from the Gosu language, or from plugins implemented in Java.

For a high-level overview of entity differences and different entity access types, see the diagram in “Required Generated Files for Integration” on page 20. For writing Java plugins, also see “Java Entity Libraries Overview” on page 114 in the *Gosu Reference Guide*..

For more information, see “Web Service and SOAP Entity Overview” on page 26 and “Endpoint URLs and Generated WSDL” on page 59.

For web service API documentation, also refer to the *API Reference Javadoc* documentation under the `com.guidewire.cc.webservices.api` packages. Within that package, there are separate interface definitions for different logical groupings of the APIs.

To view information about data-related entities such as the `Claim` class, the `Policy` class, and the `Note` class, look in the `com.guidewire.cc.webservices.entity` package.

To find information about ClaimCenter enumeration classes, look in the SOAP Javadoc in the package `com.guidewire.cc.webservices.enumeration`.

For more information about web services, see “Web Services (SOAP)”, on page 25.

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 Gosu-like than using the Java generated Javadoc to understand the plugin interfaces. And this documentation includes more hyperlinks between objects than using the Gosu API Reference from within Studio.

For more information, see “Gosu Generated Documentation” on page 35.

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.

For example, in the SOAP documentation, first click on the package `com.guidewire.cc.webservices.api` to list the SOAP APIs.

Within that package, click on the `IClaimAPI` interface in the bottom left pane. The right-side pane now shows the documentation for `IClaimAPI`. The most important part of this is the **Method Summary** list, which shows all methods available to you. For example, within the `IClaimAPI` interface, the pane lists the `migrateClaim` method. Use it to add a claim to the ClaimCenter system.

If you do not know what package to look in, click the upper-left pane on the text **All Classes** and see the list of classes in the lower-left pane. Click on the class name in the lower-left pane. The right pane shows detailed information about the class including its methods.

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/dictionary/data/index.html
```

To regenerate the dictionary, open a DOS 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 using 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.

Required Generated Files for Integration

Depending on what kind of integrations you require, you must use required files. Use them to compile your Java code against, to import in a project, or to use in other ways.

There are several classes of 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.
- **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 entities. In some cases, special library files require programmatic access ClaimCenter entities. This section discusses this topic more later.

In all cases, ClaimCenter entities such as `Claim` contain data properties that can be manipulated or read using `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*. These are automatically accessible from Gosu, and can be accessed from Java using generated *entity libraries* that provide access to the full entities within the package `com.guidewire.cc.external.entity`.

In contrast, all web services APIs (*SOAP APIs*) use the *SOAP entities* as parameters and return values. SOAP entities are simple data-only versions of the Guidewire entities. They appear in the namespace (package),

`com.guidewire.cc.webservices.entity`. The SOAP entities do not expose the domain methods, for example, additional domain methods on a `Claim` object.

IMPORTANT Web service client code always use SOAP entities for parameters and return values. The SOAP entities are data-only (no methods other than getter and setter methods) defined in a different package from full entities.

The following table summarizes the differences in the entity libraries accesses:

Entities	Contains	Java package, if used
Full entities	Entity data plus additional domain methods.	`com.guidewire.cc.external.entity`
SOAP entities	Entity data	`com.guidewire.cc.webservices.entity`

The following table summarizes the different entity integration implications for each integration type:

Entity access	Description	Entities	Entity libraries required
Gosu plugins	Plugin interface defined in Gosu.	Full entities	None
Java plugins	Java code that accesses an entity associated with a Java plugin interface parameter or return value.	Full entities	Java entity 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 entities	Java entity libraries
SOAP APIs	Code that calls a ClaimCenter web services API over the SOAP protocol. This includes any code that is calling a SOAP API from within other Java code	SOAP entities	SOAP entity libraries, if from Java -- see “SOAP Entities and Using Languages Other Than Java” on page 21.

SOAP Entities and Using Languages Other Than Java

Guidewire recommends Java for ClaimCenter SOAP API development. If you implement SOAP APIs using Java, use the ClaimCenter-generated libraries. ClaimCenter uses the generated WSDL and WSDD files that describe the SOAP API interfaces to generate the generated Java libraries.

However, if you want to use Microsoft .NET or other non-Java languages, do not use these libraries directly to call SOAP APIs. Instead, use the generated ClaimCenter WSDL and WSDD.

If you need to use the WSDL directly, then the best way is to use the dynamically-generated WSDL available only while the product ClaimCenter web application runs. For example, view the WSDL for the `ILoginAPI` SOAP API with the following code:

```
http://localhost:8080/cc/soap/ILoginAPI?wsdl
```

Alternatively, you can regenerate the SOAP API libraries and use the WSDL and WSDD in the `soap-api/wsdl/api` subdirectory. Use these WSDL files with any language and development environment that can use WSDL and WSDD, including Microsoft .NET. However, you must customize the WSDL files to include the endpoint’s server URL, which is not automatically included.

For details of .NET usage of web services and other languages, see:

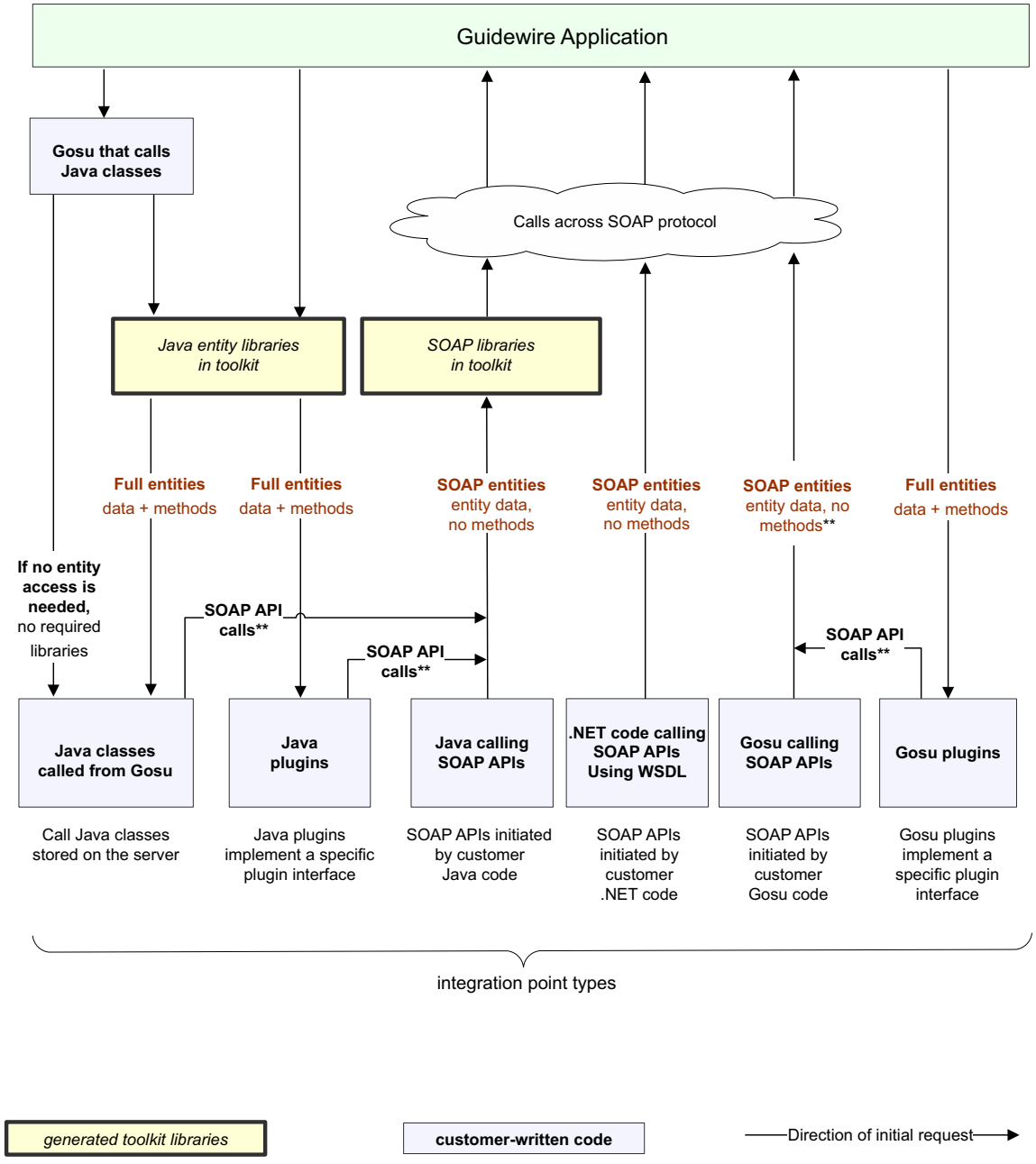
- “Calling Your Web Service from Microsoft .NET (WSE 3.0)” on page 45
- “SOAP From Other Languages, Including Java 1.4 & Non-.NET” on page 52.

Diagram of Entity Library Accesses and Differences

The following diagram illustrates the entity access differences between integration points. Notice that if you access SOAP APIs, use the SOAP entities used for parameters and return values only, even if you call SOAP from Java plugins or other Java code.

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.

Guidewire Entity Access



** Always avoid SOAP API calls that loop back to the **same** server from any plugin code or rule set execution, particularly if the API changes entity data. Contact Customer Support if you think you need to do this. Also note that calls to SOAP APIs always use SOAP entities, which have entity data but no domain methods. In cases where full entities are otherwise available, SOAP entities exist in a totally separate package hierarchy from regular “full” entities.

Required Generated Files for Each Integration Type

The following table represents the types of files or data used by each integration type.

Needed for this integration point....	Java utility JAR	GW entity JAR	GW plugin JAR	SOAP JAR	SOAP WSDL
Gosu plugin	--	--	--	--	--
Any other Gosu	--	--	--	--	--
Java class called from Gosu, not using entities	--	--	--	--	--
Java class called from Gosu, using entities	Yes	Yes	--	--	--
Java plugin	Yes	Yes	Yes	--	--
SOAP APIs, using Java	Yes	--	--	Yes	--
SOAP APIs called from languages other than Gosu or Java	--	--	--	--	Yes

The following table describes each of these files in detail. All files regenerate as part of the integration libraries regeneration process. The files include data model extensions in all entities.

General name	File name	Description
Java utility JAR	`gw-util.jar`	Java utility classes in the package `com.guidewire.*` and other packages
GW entity JAR	`gw-entity-cc.jar`	Java library required to access full entities from Java plugins or Java classes callable from entity libraries.
GW plugin JAR	`gw-plugin-cc.jar`	Java library required to access Java plugin interfaces.
SOAP JAR	`gw-soap-cc.jar`	Java library that simplifies web service (SOAP) API development in the Java language by providing easy-to-use interfaces generated as part of the SOAP API libraries from the WSDL/WSDD files. If you use SOAP APIs from Java, you must use these.
SOAP WSDL	`ClaimCenter/soap-api/wsdl/api/`*`APIInterfaceName`*`.wsdl`	The raw Web Services Description Language files that describe the ClaimCenter APIs.

See Also

- For more detailed information about regenerating integration libraries, see “Regenerating the Integration Libraries” on page 17.
- For more detailed information about web services and SOAP, see “Web Services (SOAP)” on page 25.
- For more information about plugins, see “Plugin Overview” on page 101.
- For more information about entity libraries, see “Java Entity Libraries Overview” on page 114 in the *Gosu Reference Guide*.

chapter 2

Web Services (SOAP)

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 another application 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.

Additionally, Gosu has native features to call out to remote web services published on another system.

This topic includes:

Web Service and SOAP Entity 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 3rd-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” on page 59.

Gosu natively supports web services in two different ways:

- **Publish your Gosu code as new web service APIs.** Write Gosu code that external systems can 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, see “Publishing a Web Service” on page 32 and “Writing Web Services that Use Entities” on page 35.
- **In your Gosu code, easily call web service APIs published by external applications.** 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, see “New Entity Syntax Depends on Context” on page 55.

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 “Built-in Web Services” on page 70.

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 Web Service” on page 32.

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 35. 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 Web Service" on page 32.

- **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 "Referring By Reference to an Entity By Public ID" on page 59. 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 testing a web service that reads entity data and commits entity changes to the database, see "Testing Your Web Service" on page 39.

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.

Entity Access

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*.

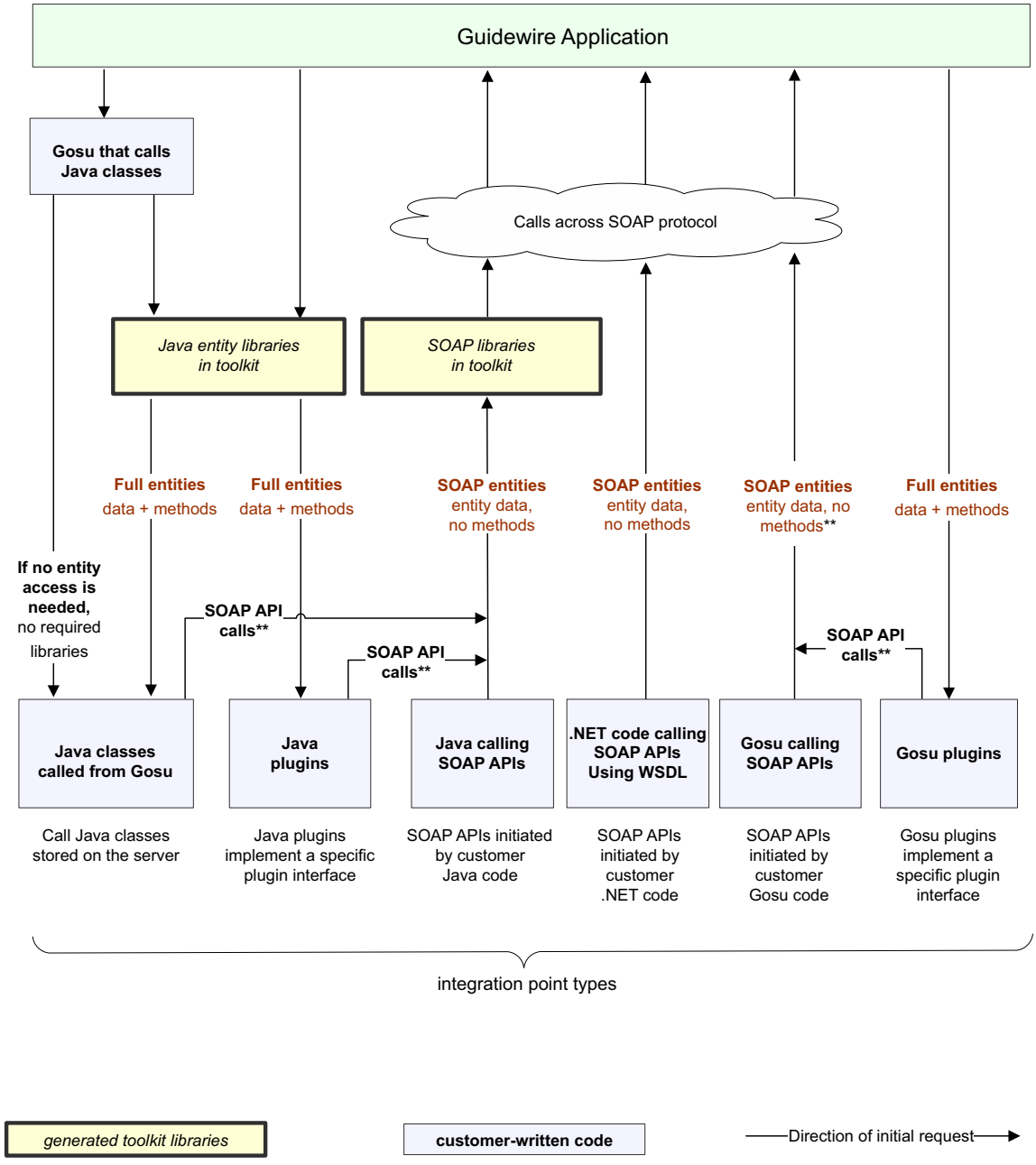
To import new records into ClaimCenter, your web services client code would create a new SOAP entity such as an instance of the `Address` class. Populate all relevant properties, taking care to populate all required properties.

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 55.

The following diagram describes the different entity access for each integration point. Note how the application uses library files in different contexts and how you access SOAP APIs and entities in different integration points.

Guidewire Entity Access



** Always avoid SOAP API calls that loop back to the **same** server from any plugin code or rule set execution, particularly if the API changes entity data. Contact Customer Support if you think you need to do this. Also note that calls to SOAP APIs always use SOAP entities, which have entity data but no domain methods. In cases where full entities are otherwise available, SOAP entities exist in a totally separate package hierarchy from regular “full” entities.

Which Objects and Properties Work With SOAP

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 "The ClaimCenter Data Model" on page 187 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.

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.

Limitations on What Kinds of Types Work with SOAP

For full specifications of entity classes, object properties, and typelist values, refer to the *API Reference* documentation and the *ClaimCenter Data Dictionary*. See "Integration Documentation Overview" on page 18.

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 231.
- **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 221.
- **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 and Interceptors”, on page 199.
- **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 178 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.

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 have an edge case where it is 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 as *entities*. This is a very different sense of the term *entities* used by Guidewire applications. Except for this topic about SOAP, all other Guidewire documentation uses the term *entity* to mean business data objects defined in the data model configuration files.

- 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 other languages might not represent SOAP entities as objects.

Publishing a Web Service

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* (a special line of text before the class definition) immediately before a Gosu class.

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 33.

As mentioned in "Web Service and SOAP Entity Overview" on page 26, 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 "Referring By Reference to an Entity By Public ID" on page 59.

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

@WebService
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.

You may want to some or all of following related sections:

- “Optional Web Service Publishing Options” on page 33.
- “Writing Web Services that Use Entities” on page 35.
- “Local SOAP Endpoints in Gosu” on page 75.
- “Regenerating the Integration Libraries” on page 17.
- For a detailed example of testing an web service that reads entity data and commits entity changes, see “Testing Your Web Service” on page 39.

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.

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 don't
    // 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)” on page 63,

Writing Web Services that Use Entities

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

IMPORTANT This section assumes the reader understands public IDs and their role in the ClaimCenter data model. For details, see “Public IDs and Integration Code” on page 57. That topic also covers information regarding special issues with writing SOAP client code using public IDs.

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 that would create memory errors for too many results. Follow the example of the error checking in the example that 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 don't
    // 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 Web Service” on page 39.

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

Guidewire provides APIs related to database transactions. You may need to use these APIs if you design SOAP APIs that take parameters that must commit to the database. You may need to use these APIs if your web service looks up entities in the database and makes entity changes. These APIs are documented fully in “Bundles and Transactions” on page 275 in the *Gosu Reference Guide*. This section shows a simple example. See that topic for additional details and code samples.

A *bundle* is a group of Guidewire entities, grouped together so they save to the database together. You can get the current transaction by calling `gw.transaction.Transaction.getCurrent()`.

The most important things to know about changing entity data in your web service as follows:

- When a web service client calls your 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, get its bundle as the *`entity`*`.bundle` property. However, generally speaking use `gw.transaction.Transaction.getCurrent()`.
- Any entities serialized into the 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 commits to the database after your 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 with `find()` statements, entities you find are initially read-only. You must add them to the current writable transaction to modify the entities. Add them by calling the following code:

  ```
  writableEntity = gw.transaction.Transaction.getCurrent().add(foundEntity)
  ```
- 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 282.

  In your block, create new entities with the bundle argument as an argument to the entity constructor:

  ```
  writableEntity = new Address(bundle)
  ```

  In your block, add existing entities to the new bundle with the `bundle.add(`*`entity`*`)` method:

  ```
  writableEntity = bundle.add(foundEntity)
  ```
- There may be cases where web service method parameters include entities you do not want to commit but you make other entity changes that must commit. If so, create an entirely new bundle and explicitly add entities to that group, as discussed in “Running Code in an Entirely New Bundle” on page 282. Simply let the application commit the new bundle created in `runWithNewBundle`. If you use `runWithNewBundle`, do **not** commit the default (current) bundle for the web service.

The following simple example web service takes an entity:

```
uses gw.transaction.Transaction

@WebService
class AddressExampleAPI {
  public function insertAddress(newAddress : Address) : Address {

   // commit all entities in the current bundle
   gw.transaction.Transaction.getCurrent().commit()

  // note: for SOAP API implementations, bundle includes incoming entities
  //       deserialized with the request.

   return newAddress
 }
}
```

WARNING Be extremely careful only to commit entity changes at appropriate times or serious data integrity errors occur. If an entity'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 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(userPublicID : String, roleIDs : String[]) : String {

    var user = User( userPublicID )
    if (user == null){
      throw new DataConversionException("No User exists with PublicID: " + userPublicID);
    }

    // add user to current bundle -- the current database transaction
    // this is always strictly required if you got the entity 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 with "new" to use a specific bundle
      var userRole = new UserRole();

      userRole.Role = role
      user.addToRoles(userRole);
    }

    // commit our bundle
    Transaction.getCurrent().commit();
    return userPublicID;
  }
```

For more bundle and transaction API information, see “Bundles and Transactions” on page 275 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 Web Service” on page 39.

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 Web Service

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.

IMPORTANT For important information about calling web services from Gosu, see “Calling Web Services from Gosu” on page 73.

You can call your API over the SOAP protocol by using the special `soap.local.api.*` namespace to find your API on the same server. Calling your own server using SOAP is good for testing, but **unsupported** for production code. See the warning later in this topic.

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") )

   // actually call the API
   var r = myAPI.echoInputArgs( "San Francisco" )

   // did it get the right answer?
   print(r)

   // Check your answers 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")
```

See “Using GUnit” on page 559 in the *Configuration Guide* for a detailed list of assertion methods available to you.

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 its 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 it,
    // 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.

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 #1
   outgoingPhones = new UserPhones()
   outgoingPhones.PublicID = userPublicID
   outgoingPhones.HomePhone = "home1"
   outgoingPhones.WorkPhone = "work1"
   myAPI.setUserPhones( outgoingPhones )

   // Check values for phone 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 #2
   outgoingPhones = new UserPhones()
   outgoingPhones.PublicID = userPublicID
   outgoingPhones.HomePhone = "home2"
   outgoingPhones.WorkPhone = "work2"
   myAPI.setUserPhones( outgoingPhones )

   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 Web Service from Java

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. This avoids incurring 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 tookit 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 a User's public ID based on its 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 244.

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 244.

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 62.

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("aapplegate"));

    // set the noteís 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 ClaimCenterÖ */}

// 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 “Referring By Reference to an Entity By Public ID” on page 59.

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`.

Dependent JARs for Java SOAP Clients in Eclipse

The following JAR files are required to compile Java SOAP clients in Eclipse:

- `ClaimCenter/java-api/lib/commons-discovery.jar`
- `ClaimCenter/java-api/lib/commons-logging-1.1.1.jar`
- `ClaimCenter/java-api/lib/gw-axis.jar`
- `ClaimCenter/java-api/lib/gw-util.jar`
- `ClaimCenter/java-api/lib/jaxrpc-noqname.jar`
- `ClaimCenter/java-api/lib/log4j-1.2.15.jar`
- `ClaimCenter/java-api/lib/mailapi.jar`
- `ClaimCenter/java-api/lib/wsdl4j.jar`

Calling Your Web Service from Microsoft .NET (WSE 3.0)

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 Web Service from Microsoft .NET (WSE 2.0)” on page 50.

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 help 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 244.

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.SoapttpClicentProtocol {
    ```
    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 redownloads the WSDL and regenerate the service. This redownloading removes your changes. 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 ISystemToolsAPI.ISystemToolsAPIService
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 ISystemToolsAPI.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. In other words, it returns an object of type `String[][]`. As Microsoft .NET imports the WSDL and generates the method signature in the native .NET environment for that language, it misconverts the type. The .NET environment inappropriately converts it to a single-dimensional array, such as `String[]` instead of `String[][]`.

To work around this issue, first import the WSDL into Visual Studio. Afterward, update the generated method signature to access the proper multi-dimensional object, such as `String[][]` instead of `String[]`.

Calling Your Web Service from Microsoft .NET (WSE 2.0)

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 Web Service from Microsoft .NET (WSE 3.0)" on page 45.

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.SoapttpClicentProtocol {
```

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 redownloads the WSDL 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 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. In other words, it returns an object of type `String[][]`. As Microsoft .NET imports the WSDL and generates the method signature in the native .NET environment for that language, it misconverts the type. It inappropriately converts it to a single-dimensional array, such as `String[]` instead of `String[][]`.

To work around this issue, first import the WSDL into Visual Studio. Afterward, update the generated method signature to access the proper multi-dimensional object, such as `String[][]` instead of `String[]`.

SOAP From Other Languages, Including Java 1.4 & Non-.NET

Previous sections discuss connecting to web services from Java 1.5 (using the included `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 62). 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 Web Service from Java” on page 42. 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>
```

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.

WARNING If you use SOAP APIs from Java code, entities you use with web services are always SOAP entities, not full entities. This is true even if calling web services hosted on the same machine from Java plugin code or Java code called from Gosu. For a diagram of entity access, refer to the diagram in “Web Service and SOAP Entity Overview” on page 26.

If you write Gosu or Java code with full entities that connects 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.

For a high-level overview of entity differences and different entity access types, see the diagram in “Required Generated Files for Integration” on page 20. For Java plugins, also see “Using Java Entity Libraries” on page 117 in the *Gosu Reference Guide*.

The following table summaries important differences in the way you create SOAP entities versus full entities.

Language	Entity	Context of Use	Code to create new entity
Gosu	**Full**	Most typical Gosu code, including writing your own web service in Gosu if your code is a SOAP server.	Simply use the `new` keyword: `var myInstance = new MyClass()` 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.
Gosu	**SOAP**	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.	Use the regular `new` keyword with the correct namespace: `var myInstance = new soap.`*`SERVICENAME`*`.entity.MyClass()` For more information about this syntax, see “Calling Web Services from Gosu” on page 73. For calling web services on the same server, see that same section. **WARNING:** In general, 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.

Language	Entity	Context of Use	Code to create new entity
Java	**Full**	Code in a Java plugin or from a Java class called from Gosu using the ClaimCenter entity libraries.	Use the entity creation utility `EntityFactory`: `EntityFactory.getEntityFactory().newEntity(MyClass.class);` For more information about `EntityFactory`, see "Java and Gosu" on page 101 in the *Gosu Reference Guide*

Java	**SOAP**
	(from external systems)
		External integration code written in Java using and compiling against Guidewire entity libraries. This is the typical case for Java code that calls SOAP. In this case your code is a SOAP client.	Simply use the `new` keyword:
			`import com.guidewire.cc.webservices.entity.MyClass;`
			`...`
			`binvoice = new MyClass();`

Java	**SOAP** (from same server)	Generally speaking, avoid writing Java code that accesses SOAP APIs on the same server. For example, from plugins or rule sets. This is **not** generally recommended. See the warning in the column to the right.	Same syntax as the previous row. This is much lower performance than normal native Java calls to entity domain methods. This may require copying many properties from SOAP entities to full entities, depending on what you are doing. **WARNING:** In general, 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.

From the perspective of Java code connecting to ClaimCenter web services through SOAP API generated libraries, the API class accessed through Java is always in the package `com.guidewire.cc.webservices.api.*`. This is independent of the original package of the Gosu class that implements the web service as defined within Studio.

On a related note, from the perspective of Gosu code that connects to an externally-hosted SOAP API, the API class is always in the package `soap.`*`SERVICENAME`*`.api`. This is independent of the original package of the class that implements the web service in the remote system.

Typecodes and 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 values that indicates that the claim is an auto claim, a worker's compensation claim, or a few other choices. For example, a claim object's loss type is represented within the typecode enumeration class `LossType`. In Java syntax, describe the loss types from your web service client code `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_married45joint 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 get the values. Simply use code such as LossType.TC_WC, not LossType.getTC_WC(...).

To convert an enumerated typelist values 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, the best way to determine the available set of typelist codes is the Data Dictionary documentation (see “Integration Documentation Overview” on page 18). This documentation includes typelist codes that you create after you run the dictionary regeneration scripts.

If you use an integrated development environment (IDE) that provides easy access to object properties and methods of objects, this improves development time accessing typecode values. For example, the IntelliJ IDE or Eclipse IDE has code completion features.

At run time, you can use an API to get the list of typelist codes for a typelist. This is helpful if you need to verify that a code is valid. See the “Mapping Typecodes to External System Codes” on page 81 for details about the ITypelistTools interface function getTypelistValues.

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 needed, you can write a new ClaimCenter SOAP API that generates unique identifiers in a sequence. For more information about the Gosu APIs, see “Creating Unique Numbers in a Sequence” on page 136.

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. However, if your API client code does not need to 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 by calling 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`. For more information about reference types, see "Referring By Reference to an Entity By Public ID" on page 59.

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. See "PublicIDPrefix" on page 54 in the *Configuration Guide* for details. 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.

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	`IClaimAPI.findPublicIdByClaimNumber`
`User`	user name	`IUserAPI.findPublicIdByName`
`Group`	group name	`IGroupAPI.findPublicIdByName`

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.

Referring By Reference to an Entity By 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)" on page 63.

Endpoint URLs and Generated WSDL

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 web page `http://www.w3.org/XML`.
- **SOAP** – The web services request-and-response protocol based on XML and 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 at `http://www.w3.org/TR/soap`.
- **WSDL (Web Services Description Language)** – The web services API description language, which describes an API interface 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. Related to XML and SOAP. For more information, refer to the World Wide Web Consortium at `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 it generates a Java and Gosu language binding (interface libraries) for programmers connecting to ClaimCenter SOAP APIs. For more information, refer to the Apache web site at the page `http://ws.apache.org/axis`.

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 the Integration Libraries” on page 17). 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 18.

To use WSDL files with Microsoft .NET, see “Calling Your Web Service from Microsoft .NET (WSE 3.0)” on page 45. 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 like during SOAP API library regeneration.

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

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 62.
- **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 62.

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 configured 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 authentication 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 239.

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 deterine 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 ContactCenter integration to load balance the connection to ContactCenter.

- From web service client code programatically 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 129 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.updateLoggingLevel`
- `ISystemToolsAPI.getActiveSessionData`
- `ILoginAPI` – for setting up or releasing a session with a specific server

SOAP Faults (Exceptions)

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. For more information and examples, see "Declaring Exceptions" on page 35. 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.

The package hierarchy exceptions appear differ depending on the programming context. The following table compares the syntax and package hierarchy for SOAP exceptions:

Context	Language	Root class for SOAP exceptions
Implementing a custom web service in Gosu	Gosu	`gw.api.webservice.exception.SOAPException`
Connecting as a SOAP client to a web service published by a Guidewire application.	Java (using the generated libraries)	`com.guidewire.cc.webservices.fault.SOAPException`
	Gosu	`soap.`*`SERVICENAME`*`.fault.SOAPException`, where *`SERVICENAME`* 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 `com.guidewire.cc.webservices.fault.SOAPException`.
Calling from Gosu as a client to a SOAP client of a web service published by an external system.	Gosu	`soap.`*`SERVICENAME`*`.fault.SOAPException`, where *`SERVICENAME`* is the name of your web service as registered in Studio

More detailed exceptions indicate the specific type of problem subclass from the three exception types already mentioned.

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`, rather than from.

Exception name (indenting shows subclass)	Description
`SOAPServerException`	The root exception for all server exceptions. See discussion earlier in this section.
→ `BatchProcessException`	An exception specific to batch process operations
→ `ServerStateException`	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.
`SOAPSenderException`	Root exception for all sender exceptions. See discussion earlier in this section.

Exception name (indenting shows subclass)	Description
→ `EntityStateException`	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.
→ `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
`SOAPTimeoutException`	The SOAP transaction timed out.

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 it detects 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 to Call Web Services

Guidewire provides a Java class to compile your code into a command line tool and 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. You can trigger 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

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
- provides 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
`-help`	Display option help	Display the available options and default values for this command line tool.
`-d`	Set a system property with *`name=value`* syntax	Set a system property as shown in this example: `-D javax.net.ssl.trustStore=C:/cmd-tools/lib/keystore`
`-server`	ClaimCenter server	The ClaimCenter server that is publishing web services. The Java class can override the method `getDefaultServerUrl` to define the default URL.
`-user`	ClaimCenter user name	The ClaimCenter user name on this server. The Java class can override the method `getDefaultUserName` to define the default URL.
`-password`	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};
```

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 `getProductCode`. 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 getProductCode() {
  return "cc";
}
```

IMPORTANT If you do not implement `getProductCode` 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"`.

If You Want to Call More than One Web Service

This class makes it simple to code tools that only need to communicate with one web service interface. You can use 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 implement in the superclass `CmdLineToolBase`. Do not attempt to use the `APILocator` class discussed in “Local SOAP Endpoints in Gosu” on page 75, since the command line tool does **not** support using `APILocator`.

IMPORTANT If your command line tool calls more than one ClaimCenter SOAP API get a handle to the desired API interface with the `CmdLineToolBase.getProxy(...)` method. Do not use the Java class `APILocator` within a tool that uses `CmdLineToolBase`.

Example Command Line Tool

The following example command line tool calls the web service defined in the “Publishing a Web Service” on page 32 and “Testing Your Web Service” on page 39. 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
```

If you want to use and test this example, first create the web service as shown in “Publishing a Web Service” on page 32. Then, regenerate the integration libraries as discussed in “Regenerating the Integration Libraries” on page 17. The generated libraries include your new web service `MyServiceAPI` in the Java JAR files.

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 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 Tool From 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"
```

For a better interface to your command line tool, create a wrapper for it. For example, for Windows use 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 changing 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.

Built-in Web Services

ClaimCenter API functions are grouped by purpose within various interfaces within the package `com.guidewire.cc.webservices.api`. The following table summarizes these interfaces:

Interface	Description
General web services	
`ISystemToolsAPI`	This interface provides a set of tools that are always available, even if the server is set to `DBMaintenance` run level. One of its most important tools is getting and setting the server's *run level*. This particular API provides the current server and schema versions. See “System Tools Web Services” on page 85
`IMaintenanceToolsAPI`	This interface provides a set of tools that are available only if the system is at Maintenance run level or higher. This particular API can start various background processes. You must poll the server later to see if the process failed or completed successfully. See “Maintenance Web Services” on page 84.
`IImportTools`	Imports system data from an XML file. You must **only** use this 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 “Importing Administrative Data” on page 83.
`IMessagingToolsAPI`	Various messaging methods. For example, retry a message or get the counts of pending or failed messages for an external system. See “Messaging and Events”, on page 139.
`ITypelistToolsAPI`	Tools for getting valid typecodes for a typelist and mapping between ClaimCenter internal codes and the codes for external systems. See “Mapping Typecodes to External System Codes” on page 81.
`IUserAPI`	Manipulates user data. See “User and Group Web Services” on page 86.
`IGroupAPI`	Manipulates group data. See “User and Group Web Services” on page 86.

Interface	Description
ITableImportAPI	Table-based import interface for high volume data import. This would be used typically for large-scale data conversions, particularly if migrating claims from a legacy system into ClaimCenter prior to bringing ClaimCenter into production. See “Importing from Database Staging Tables”, on page 423.
ITemplateToolsAPI	This interface includes methods that list the available Gosu templates and validate the syntax of the templates. See “Data Extraction Integration”, on page 279.
WorkflowAPI	Designate a workflow to immediately check if it can advance to its next step. See “Workflow Web Services” on page 86.
ILoginAPI	Log into the server if you use Microsoft .NET or other languages other than Gosu and Java 1.5. See “Calling Your Web Service from Microsoft .NET (WSE 3.0)” on page 45 for how to use it with .NET. In general, do not use this for Java development as it is better to use the APILocator Java class. For Java login, see “Calling Your Web Service from Java” on page 42. If you use more than one Guidewire application, do not use this to connect from Gosu language to the other server. For more information, see “Testing Your Web Service” on page 39.
IProfilerAPI	Sends information to the built-in system profiler. For more information, see “Profiling Web Services” on page 87.
ZoneImportAPI	Takes geographic zone data in comma separated value (CSV) format and populates staging tables with the data, in preparation for importing the data. You probably do not need to use web service directly. Instead, use the command line tool zone_import, which provides a simple interface for importing your CSV data that is already stored locally. See “Importing from Database Staging Tables”, on page 423.
ContactCenter-related web services	
IContactAutoSyncAPI	Address books use this web service for automatic contact autosync functionality. If a contact updates in an address book, the address book notifies ClaimCenter to pick up changes. ClaimCenter finds contacts to synchronize to new values in the external address book. If you use ContactCenter and have it installed, ContactCenter uses this automatically. See “Address Book Integration” on page 299.
Claim and exposure web services	
IClaimAPI	Manipulates claims and claim data. Methods add documents to a claim, create a new claim or FNOL (first notice of loss), and add activities to claims using an activity pattern as a template. See “Claim APIs” on page 89.
IClaimFinancialsAPI	Manipulates financials information associated with claims. Methods perform actions such as loading reserves, payments, and recoveries from an external system, or update status of a payment. See “Claim Financials Web Services” on page 200.
IExposureAPI	Various actions on exposures, such s adding documents and notes, reopening an exposure, and getting an exposure’s status. See “Exposure APIs” on page 96.
ISREEAuthenticationAPI	This is for Guidewire use only. This is part of the internal implementation to communicate between ClaimCenter and the optional reporting module for ClaimCenter. Do **not** directly use this web service interface. For more information about reporting, see *ClaimCenter Reporting Guide*

The full specifications of the web service API interfaces and their methods are documented in the SOAP API Reference Javadoc documentation. For more information about developer documentation, see “Integration Documentation Overview” on page 18.

chapter 3

Calling 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.

Calling External Web Services

You can write code that easily imports 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.

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.

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 145 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.

Note: 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. 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
`soap.`*`SERVICENAME`*`.*`	Root package hierarchy for the service
`soap.`*`SERVICENAME`*`.api.*`	The API classes provided by the remote service and specified in its WSDL file. Using SOAP terminology, these corresponds to *SOAP ports* defined by the service. Objects within this hierarchy are what you must instantiate to use the API.
`soap.`*`SERVICENAME`*`.entity.*`	The entity classes for this web service. This includes all objects, but does not include enumerations.
`soap.`*`SERVICENAME`*`.enum.*`	Enumerations for this web service.
`soap.`*`SERVICENAME`*`.fault.*`	SOAP faults, which are analogous to Gosu or Java exceptions.

IMPORTANT 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:

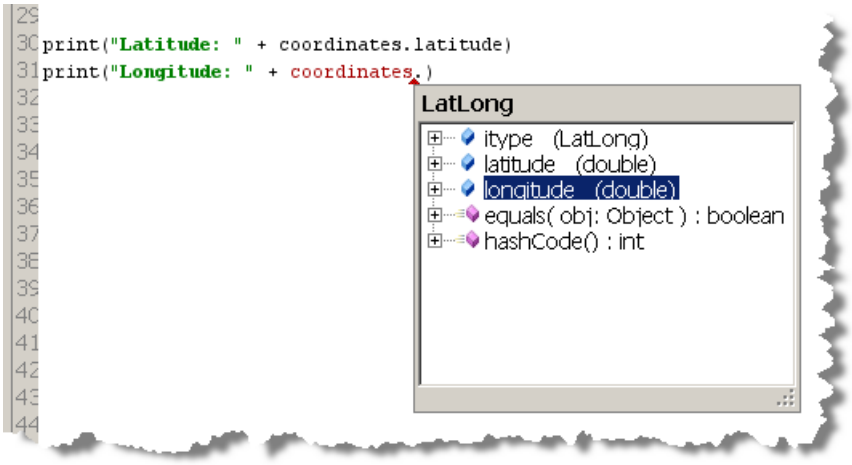
```
uses soap.AddressID.entity.FindOptions
uses soap.AddressID.entity.FindAddressSpecification
uses soap.AddressID.entity.Address
uses soap.AddressID.entity.FindRange

// 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 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.

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 Web Service” on page 39 shows how to use this feature to test your web service. It also shows how to authenticate to the server using handlers, a feature discussed in the following section.

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.

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 145 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 145 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 and Interceptors” on page 199 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 SOAP Endpoints in Gosu” on page 75. 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_USERNAME_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. This is similar to an interceptor, see “Annotations and Interceptors” on page 199 in the *Gosu Reference Guide*. 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:

```
api.Methodname()
```

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 about anonymous subclasses, see “Annotations and Interceptors” on page 199.

Calling 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 create 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 as the standard way to integrate with ICD-9 codes. This example demonstrates how web services might be used in application logic. You can write Gosu utility classes to prepare data for the service, or perform post-processing after getting results.

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 are null, 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 Tester

Launch the Gosu Tester 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 Tester:

```
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

chapter 4

General Web Services

This topic describes general-purpose web services, such as mapping typecodes general system tools.

IMPORTANT This topic relies on you to understand what web services are and how to call them from remote systems. Before reading this topic, read “Web Services (SOAP)”, on page 25.

This topic includes:

Mapping Typecodes to External System Codes

For an overview of typecodes, see “New Entity Syntax Depends on Context” on page 55.

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 is 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 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`:

```
tlAPI = (ITypelistToolsAPI) APILocator.getAuthenticatedProxy(ITypelistToolsAPI.class, url, "su", "gw");
...
typekeyString = tlAPI.getAliasByInternalCode("Contact", "ABC:system1", "ATTORNEY");
```

Using Gosu or Java to Translate Typecodes

Typecode translation may occur very frequently in Gosu or Java plugin code. Calling the SOAP API lowers server performance in such cases. For high performance from Gosu and Java, always use a related utility class called `TypecodeMapperUtil` that provides similar methods to the SOAP API such as `getInternalCodeByAlias`. For details of other plugin utilities, see "Java Plugin Utilities" on page 118.

From Java, translate a typecode with `TypecodeMapperUtil` with code such as:

```
TypeKey tk = TypecodeMapperUtil.getInternalCodeByAlias("Contact", "ABC:system1", "ATTORNEY");
```

From Gosu, translate a typecode with `TypecodeMapperUtil` with code such as:

```
var mapper = gw.api.util.TypecodeMapperUtil.getTypecodeMapper()

var mycode = mapper.getInternalCodeByAlias( "Contact", "ABC:system1", "ATTORNEY" )
```

Importing Administrative Data

ClaimCenter provides tools for importing and exporting data in XML using the import tools API (`IImportToolsAPI`) 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**.

To prepare data for XML import, or to understand exported XML data from the user interface, use the *XML Schema Definition* (XSD) file that defines the XML data format. ClaimCenter automatically generates this file as a result of the `regen-xsd` argument to the `gwcc` command. These generated XSD files automatically contain your data model extensions.

The application can generate the following XSD files:

```
ClaimCenter/soap-api/sampledata/cc_import.xsd
ClaimCenter/soap-api/sampledata/cc_entities.xsd
ClaimCenter/soap-api/sampledata/cc_typelists.xsd
```

Regenerate these files by calling the `regen-xsd` command using the `gwcc` command line tool. Be aware that regenerating the SOAP libraries with the `regen-soap-api` command does not regenerate the XSDs.

To regenerate XSDs

1. In Windows, bring up a command line window.
2. Change your working directory to:
   ```
   ClaimCenter/bin
   ```
3. Type the command:
   ```
   gwcc regen-xsd
   ```

The `IImportToolsAPI` 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 `IImportToolsAPI` 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 for more information about the straightforward CSV-related methods.

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 API (`IMaintenanceToolsAPI`) interface 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 `IMaintenanceToolsAPI` interface 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 ContactCenter servers.

If you use the `IMaintenanceToolsAPI` 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 `UIRunable` or `APIRunable`.

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("ActivityEscalation");
```

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("ActivityEscalation");
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("ActivityEscalation", 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.

IMPORTANT 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 ContactCenter and you use these APIs on a ClaimCenter server, it applies only to ClaimCenter not ContactCenter. Similarly, if you use these APIs on a ContactCenter server, it applies only to ContactCenter not ClaimCenter.

Marking Claims for Archive

To mark claims for archive, on the `IMaintenanceToolsAPI` interface, call the `scheduleForArchive` method. Internally, this just calls the `scheduleForArchive` on the `IClaimAPI` web service interface. It is provided on this web service interface as a convenience. This method is also triggerable using the `maintenance_tools` command line tool as the `-scheduleforarchive` option.

For usage of this SOAP API method and important warnings, see “Archiving and Restoring Claims” on page 94.

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 (`ISystemToolsAPI`) 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.

Getting and Setting the Run Level

The most important usage of the `ISystemToolsAPI` 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
Server not running	*No response to the HTTP request*
`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 `ISystemToolsAPI` 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();
```

User and Group Web Services

The user and group API interfaces (`IUserAPI` and `IGroupAPI`) provide methods to manipulate user and group information. For example, these APIs can add a user to a group, check whether a user exists, or find a public ID by name.

Important methods include `addUser` and `addGroup`. These add users and groups, respectively. Refer to the API Reference Javadoc for details.

Workflow Web Services

The web service API interface `IWorkflowAPI` 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. For more information about workflows, see *ClaimCenter Rules Guide* and *ClaimCenter Application Guide*.

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	IWorkflowAPI method	Description
Invoke a workflow trigger	`invokeTrigger`	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 typecode from the `WorkflowTriggerKey` extendible typelist. To check whether you can call this workflow trigger, use the `isTriggerAvailable` method in this interface (see later in this table). This method returns nothing.
Check whether a trigger is available	`isTriggerAvailable`	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 `invokeTrigger` method in this web services interface. This method takes a workflow public ID and a typecode from the `WorkflowTriggerKey` extendible typelist. It returns `true` or `false`.
Resume a single workflow	`resumeWorkflow`	Restarts one workflow specified by its public ID. This method sets the state of the workflow to `TC_ACTIVE`. This method returns nothing.
Resume all workflows	`resumeAllWorkflows`	Restarts all workflows that are in the error state. It is important to understand that this **only** affects workflows currently in the error state `TC_ERROR` or `TC_SUSPENDED`. The workflow engine subsequently attempts to advance these workflows to their next steps and set their state to `TC_ACTIVE`. For each one, if an error occurs again, the application logs the error sets the workflow state back to `TC_ERROR`. This method takes no arguments and returns nothing.
Suspend a workflow	`suspend`	Sets the state of the workflow to `TC_SUSPENDED`. If you must restart this workflow later, use the `resumeWorkflow` method or the `resumeAllWorkflows` method.
Complete a workflow	`complete`	Sets the state of a workflow (specified by its public ID) to `TC_COMPLETED`. This method returns nothing.

Profiling Web Services

From remote systems you can enable the or disable the ClaimCenter profiler system using the `IProfilerAPI` web service. The methods are fairly straightforward. The common arguments are:

- 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 indicates whether to enable (`true`) or disable (`false`) the profiler.
- 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 dmbs counters.

Each method takes a boolean value as the first parameter. Set that value to `true` to enable the profiler. Set that value 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.

chapter 5

Claim-related Web Services

This section talks about claim and exposure web services.

IMPORTANT This topic relies on the reader understanding what web services are and how to call them from remote systems. Before reading any sections in this topic, read “Web Services (SOAP)”, on page 25.

This topic includes:

Claim APIs

The claim API (`IClaimAPI`) interface manipulates claims. Important methods include:

- `addActivityFromPattern` - Adds an activity to a claim using an activity pattern as a template.
- `addDocument` - Registers a link in the claim file to a document stored in an external document repository. See “Document Management Overview” on page 249 for more details.
- `addNote` - Adds a note to a claim file.
- `addFNOL` - Creates a new claim as a first notice of loss (FNOL), the first time an insurance copy is notified about a claim. Note the difference between this and the `migrateClaim` method. See “Adding First Notice of Loss” on page 90.
- `migrateClaim` - Loads a claim that is new to ClaimCenter but that is not a new claim (FNOL). See “Migrating a Claim” on page 93.

These methods are for active claims only. If an external system posts any new updates on an *archived* claim, `IClaimAPI` methods throw the exception `EntityStateException`.

Adding First Notice of Loss

If you add a first notice of loss (FNOL) with `addFNOL`, ClaimCenter does the following steps:

1. Creates a new claim number (if one does not exist)
2. Creates a claim snapshot
3. Runs the Pre-setup rules on the claim and each exposure
4. Runs the segmentation rules on each exposure, then the claim
5. Runs the assignment rules on the claim, then each exposure
6. Runs the workplan rules on the claim, then each exposure (generating the workplan)
7. Runs the Post-setup rules on the claim and each exposure
8. Updates history timestamps (for client-added history events to be correctly stamped)
9. Sets the status of the claim and each exposure to `open`
10. Makes sure that each exposure has a valid claim order property set
11. Sets initial reserves
12. Commits the claim's data to the database, including all exposures, activities, and other objects creating during the request.
13. Commits the data to the database triggers Pre-update rules, double check validation, and run any appropriate Event Fired rule sets for new entities.

The following example adds a first notice of loss:

```
// create a new claim
Claim newClaim = new Claim();
newClaim.setPublicID("FNOL:AUTO3453454");
newClaim.setCoverageInQuestion(Boolean.FALSE);
newClaim.setDeductibleStatus(DeductibleStatus.TC_paid);
newClaim.setDescription("Insured vehicle proceeded from stop sign and 4 way intersection.");
newClaim.setFireDeptInfo("none filed");
newClaim.setHowReported(HowReportedType.TC_fax);
newClaim.setIncidentReport(Boolean.FALSE);
newClaim.setJurisdictionState(State.TC_CA);
newClaim.setLOBCode(LOBCode.TC_auto);
newClaim.setLossCause(LossCause.TC_vehcollision);

// set the loss date
Calendar loss = Calendar.getInstance();
loss.set(2003, 9, 9, 23, 24, 0);
newClaim.setLossDate(loss);

// set a loss location
Address lossLocation = new Address();
lossLocation.setAddressLine1("X of Maryland Avenue and Ellsworth Avenue");
lossLocation.setCity("San Mateo");
lossLocation.setState(State.TC_CA);
newClaim.setLossLocation(lossLocation);
newClaim.setLossType(LossType.TC_AUTO);

// set other claim properties
newClaim.setMainContactType(PersonRelationType.TC_self);
newClaim.setPoliceDeptInfo("none filed");
newClaim.setReportedByType(PersonRelationType.TC_self);
Calendar reported = Calendar.getInstance();
reported.set(2003, 9, 11, 6, 10, 0);
newClaim.setReportedDate(reported);
newClaim.setWeather(WeatherType.TC_CL);

// set Insured
Address insuredAddress = new Address();
insuredAddress.setAddressLine1("4205 Harris Avenue");
insuredAddress.setCity("Moss Beach");
insuredAddress.setState(State.TC_CA);
insuredAddress.setPostalCode("94001");
Person insuredPerson = new Person();
```

```
insuredPerson.setFirstName("Sarah");
insuredPerson.setLastName("Roth");
insuredPerson.setMiddleName("Elisabeth");
insuredPerson.setPrimaryAddress(insuredAddress);
insuredPerson.setWorkPhone("650-456-4444");

// set Vehicles
Vehicle vehicle = new Vehicle();
vehicle.setMake("Nissan");
vehicle.setModel("Sentra");
vehicle.setYear(new Integer(1997));
Vehicle vehicle2 = new Vehicle();
vehicle2.setMake("Ford");
vehicle2.setModel("Contour");
vehicle2.setYear(new Integer(1999));
VehicleIncident newVehicleIncident[] = new VehicleIncident[3];

//Add the new vehicleIncident
newVehicleIncident[0] = new VehicleIncident();
newVehicleIncident[0].setDriverRelation(PersonRelationType.TC_self);
newVehicleIncident[0].setVehicle(vehicle);
newVehicleIncident[0].setOwnersPermission(Boolean.TRUE);
newVehicleIncident[0].setDriverRelation(PersonRelationType.TC_self);
newVehicleIncident[0].setVehicleType(VehicleType.TC_listed);

// set Policy
Policy policy = new Policy();
policy.setPolicyNumber("11-1111");
policy.setPolicyType(PolicyType.TC_auto_per);
policy.setVerified(Boolean.TRUE);
Calendar policyEffective = Calendar.getInstance();
policyEffective.set(2003, 7, 1);
policy.setEffectiveDate(policyEffective);
Calendar policyExpiration = Calendar.getInstance();
policyExpiration.set(2004, 7, 1);
policy.setExpirationDate(policyExpiration);

// set Vehicles on the policy
VehicleRU vehicleRU = new VehicleRU();
vehicleRU.setVehicle(vehicle);
vehicleRU.setRUNumber(new Integer(1));
VehicleRU[] policyVehicles = new VehicleRU[]{policyVehicle};
policy.setRiskUnits(policyVehicles);
policy.setTotalVehicles(new Integer(1));
newClaim.setPolicy(policy);

// Add the new vehicleIncident
newVehicleIncident[1] = new VehicleIncident();
newVehicleIncident[1].setDriverRelation(PersonRelationType.TC_self);
newVehicleIncident[1].setVehicle(vehicle);
newVehicleIncident[1].setOwnersPermission(Boolean.TRUE);
newVehicleIncident[1].setDriverRelation(PersonRelationType.TC_self);
newVehicleIncident[1].setVehicleType(VehicleType.TC_listed);
newVehicleIncident[1].setDescription("front left side");

// set up one exposure
Exposure exposure1 = new Exposure();
exposure1.setClaimantType(ClaimantType.TC_insured);
exposure1.setClaimOrder(new Integer(1));
exposure1.setJurisdictionState(State.TC_CA);
exposure1.setLossParty(LossPartyType.TC_insured);
exposure1.setPrimaryCoverage(CoverageType.TC_COLL);
exposure1.setExposureType(ExposureType.TC_VehicleDamage);
exposure1.setIncident(newVehicleIncident[1]);
exposure1.setPublicID("FNOL:VD12354234");

// set up an address
Address claimantAddress = new Address();
claimantAddress.setAddressLine1("555 Richard Drive");
claimantAddress.setCity("Palo Alto");
claimantAddress.setState(State.TC_CA);
claimantAddress.setPostalCode("94303");
Person claimantPerson = new Person();
claimantPerson.setFirstName("Karl");
claimantPerson.setLastName("Randall");
claimantPerson.setPrimaryAddress(insuredAddress);

newVehicleIncident[2] = new VehicleIncident();
newVehicleIncident[2].setDriverRelation(PersonRelationType.TC_self);
newVehicleIncident[2].setVehicle(vehicle);
newVehicleIncident[2].setOwnersPermission(Boolean.TRUE);
```

```
newVehicleIncident[2].setDriverRelation(PersonRelationType.TC_self);
newVehicleIncident[2].setVehicleType(VehicleType.TC_listed);
newVehicleIncident[2].setDescription("unknown");

// set up another exposure
Exposure exposure2 = new Exposure();
exposure2.setClaimantType(ClaimantType.TC_veh_other_owner);
exposure2.setClaimOrder(new Integer(2));
exposure2.setJurisdictionState(State.TC_CA);
exposure2.setLossParty(LossPartyType.TC_third_party);
exposure2.setOtherCoverage(Boolean.TRUE);
exposure2.setOtherCoverageInfo("Allstate; 918123456");
exposure2.setPrimaryCoverage(CoverageType.TC_APD);
exposure2.setExposureType(ExposureType.TC_VehicleDamage);
exposure2.setIncident(newVehicleIncident[2]);
exposure2.setPublicID("FNOL:VD12354235");

// Create an array to hold the new exposures to set on Claim, and then set it...
Exposure[] exposures = new Exposure[]{exposure1, exposure2};
newClaim.setExposures(exposures);

// set Notes
Note note = new Note();
note.setBody("No Injured Parties reported at First Notice.");
note.setConfidential(Boolean.FALSE);
note.setSubject("First notice of loss");
note.setTopic(NoteTopicType.TC_fnol);
Note[] notes = {note};
newClaim.setNotes(notes);

// set Contacts
ClaimContactRole reporter = new ClaimContactRole();
reporter.setRole(ContactRole.TC_reporter);
ClaimContactRole maincontact = new ClaimContactRole();
maincontact.setRole(ContactRole.TC_maincontact);
ClaimContactRole insured = new ClaimContactRole();
insured.setRole(ContactRole.TC_insured);
ClaimContactRole driver = new ClaimContactRole();
driver.setRole(ContactRole.TC_driver);
ClaimContactRole claimantexposure1 = new ClaimContactRole();
claimantexposure1.setRole(ContactRole.TC_claimant);
claimantexposure1.setExposure(exposure1);
ClaimContactRole driverexposure1 = new ClaimContactRole();
driverexposure1.setRole(ContactRole.TC_driver);
driverexposure1.setExposure(exposure1);

ClaimContactRole[] insuredRoles = new ClaimContactRole[]{reporter,maincontact, insured,
  driver, claimantexposure1, driverexposure1};

ClaimContact claimInsuredContact = new ClaimContact();
claimInsuredContact.setContact(insuredPerson);
claimInsuredContact.setRoles(insuredRoles);

ClaimContactRole claimantexposure2 = new ClaimContactRole();
claimantexposure2.setRole(ContactRole.TC_claimant);
claimantexposure2.setExposure(exposure2);
ClaimContactRole driverexposure2 = new ClaimContactRole();
driverexposure2.setRole(ContactRole.TC_driver);
driverexposure2.setExposure(exposure2);

ClaimContactRole[] claimantRoles = new ClaimContactRole[]{claimantexposure2, driverexposure2};

ClaimContact claimClaimantContact = new ClaimContact();
claimClaimantContact.setContact(claimantPerson);
claimClaimantContact.setRoles(claimantRoles);

ClaimContact[] claimContacts = new ClaimContact[]{
claimInsuredContact, claimClaimantContact};

newClaim.setContacts(claimContacts);
newClaim.setIncidents(newVehicleIncident);

// call addFNOL
String newClaimPublicId = claimAPI.addFNOL(newClaim, null);
```

The `addFNOL` method includes a parameter for a `SynchStateData` object. Always pass `null` for that parameter. The `SynchStateData` is an artifact of previous releases of ClaimCenter and the current version does not support non-null values for that parameter.

Migrating a Claim

If you migrate a claim with `migrateClaim`, some normal processing steps for a new claim are unnecessary. For example, picking a claim number and assigning to an adjuster.

Instead, to migrate a claim ClaimCenter does only the following steps:

1. Sets the claim state to `open`
2. Assigns the claim to the appropriate user and group
3. Validates the claim at the `loadsave` level
4. Commits the claim's data to the database, including all exposures, activities, and other objects creating during the request
5. Commits the data to the database triggers Pre-update rules, double checks validation, and runs any appropriate Event Fired rule sets for new entities.

WARNING It is important that you **never** add new claims to ClaimCenter with the web service APIs while the Financials Calculations batch process is running. Doing so is dangerous. This applies to both `addFNOL` and `migrateClaim` methods. However, this batch process only runs manually while the server is in maintenance mode. For more information see "Batch Processes and Distributed Work Queues" on page 134.

There are two method signatures for the `migrateClaim` method. The simpler version sets the claim state to `open` by default. Another method signature takes an additional `ClaimState` parameter so as you migrate the claim, you can specify a claim state other than open.

The following example demonstrates migrating a claim:

```
// Create a new Claim SOAP entity
Claim newClaim = new Claim();
newClaim.setClaimNumber("1234");
newClaim.setPublicID("ABC:1234");

// create a new person
Person insuredPerson = new Person();
insuredPerson.setLastName("Smith");

// create a new policy
Policy newPolicy = new Policy();
newPolicy.setPolicyNumber("12345477-HO");
newPolicy.setPolicyType(PolicyType.TC_auto_per);

// assign a contact to a policy
ClaimContact claimContact = new ClaimContact();
claimContact.setContact(insuredPerson);
ClaimContactRole claimContactRoleData = new ClaimContactRole();
claimContactRoleData.setRole(ContactRole.TC_insured);
claimContact.setRoles(new ClaimContactRole[]{claimContactRoleData});
newPolicy.setContacts(new ClaimContact[]{claimContact});

newClaim.setPolicy(newPolicy);
newClaim.setLossType(LossType.TC_AUTO);

String claimPublicId = claimAPI.migrateClaim(minClaim, groupPublicId, userPublicId, null);
```

The `migrateClaim` method includes a parameter for a `SynchStateData` object. Always pass `null` for that parameter. The `SynchStateData` is an artifact from previous releases of ClaimCenter. This release does not support non-null values for that parameter.

Importing a Claim from XML

The `IClaimAPI` has a method called `importClaimFromXML` that imports the claim from XML format. If the format is the ACORD format, instead use the `importAcordClaimFromXML` method.

For details about this feature and how to customize ClaimCenter behavior, see "FNOL Mapper" on page 381.

Archiving and Restoring Claims

To schedule a claim for archiving, there are two methods in the `IClaimAPI` 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. Note that the archiving work queue is asynchronous so typically no claims are archived by the time this 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 workaround 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 example shows the by-public-ID variant:

```
String[] claimPublicIDs = { "abc:1234","abc:5678" };
claimAPI.scheduleForArchiveByPublicId(claimPublicIDs);
```

These methods return no value.

Restoring Archived Claims

To restore an archived claim, call the `IClaimAPI` web service method `restoreClaim`. 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.

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");
```

Activity APIs

In the `ClaimAPI` web service interface, there are several activity-related methods.

Adding Activities from Patterns

To create an activity from an activity pattern, call the `addActivityFromPattern` method. It attempts to generate an activity from the given pattern. The method takes the following two parameters:

- the claim public ID
- the activity pattern public ID

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 activity pattern's loss type must either be `null` or must match the claim's loss type.

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`, `Command`

ClaimCenter calculates the activity target using the following fields in the pattern:

- `targetStartPoint`, `TargetDays`, `TargetHours`, and `TargetIncludeDays`

The activity's escalation date 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 activity's claim ID is set to the given claim ID, and the exposure ID is set to `null`. The activity's previous user ID property is updated with the current (SOAP) user.

The newly created activity is then assigned 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");
```

There is a variant of this method that allows you to set arbitrary fields in the new activity. To do this, call the `addActivityFromPatternWithOverride` method instead of `addActivityFromPattern`. Pass an additional parameter containing array of field values to apply to the activity. These field values are in the form of `FieldValue` objects, which simply encapsulate `Field` and `Value` properties.

For example to set the activity priority (from Java):

```
FieldValue[] changeFields;
FieldValue priorityValue = new FieldValue();
priorityValue.setName("priority")
priorityValue.setValue("high");
changeFields[0] = priorityValue
publicID = claimAPI.addActivityFromPatternWithOverride("abc:123", "abc:emailnote1", changeFields);
```

Claimant Activities

You can add a claimant activity using the method `addClaimantActivityFromPattern`. It is similar to the previous methods for adding activities, but has an extra parameter. Pass the claimant public ID as the second parameter, in between the claim public ID and the activity pattern ID.

Skipping and Completing Activities

To complete an activity, call the `completeActivity` method. It simply takes a public ID for the activity. The claim must have the `VIEW_CLAIM`, `CREATE_ACTIVITY` permissions. For example (from Java):

```
claimAPI.completeActivity("abc:1234");
```

To skip an activity, call the `skipActivity` method. It simply takes a public ID for the activity. The claim must have the `VIEW_CLAIM`, `CREATE_ACTIVITY` permissions. For example (from Java):

```
claimAPI.skipActivity("abc:1234");
```

Both the `completeActivity` and `skipActivity` methods run the standard rule sets for skipped or completed activities, and set the `CloseDate`/`CloseUser` properties.

Exposure APIs

The exposure API (`IExposureAPI`) interface provides tools for manipulating exposures. Important methods include:

- `addDocument` - Adds a document to an exposure.
- `addNote` - Adds a note to an exposure.
- `closeExposure` - Close an exposure
- `reopenExposure` - Reopen an exposure
- `getExposureState` - Get the exposure's state, represented as a typekey from the `ExposureState` typelist.

These methods are straightforward. Refer to the SOAP API Reference Javadoc for details.

chapter 6

Servlets

You can define simple web servlets for your ClaimCenter application using the `@Servlet` annotation.

Using 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 from the Guidewire PCF entrypoints feature. You can define arbitrary code triggered from any URL as long as you can define a Gosu block that can determine from the URL whether it owns the request. There is no complex object serialization or deserialization such as is done in the SOAP protocol. The implementation uses the standard Java classes in the package `javax.servlet.http` to define the servlet request and response.

It is absolutely critical for you to understand the large security implications of using servlets. Extending `HttpServlet` provides absolutely no security for the servlet. For example, no authentication is necessary to execute it. If you want to implement your own authentication system, you can. For example, you can extend `AbstractGWAuthServlet` to translate the security headers in the request and authenticate with the Guidewire server. Alternatively, you can base your servlet on the class `AbstractBasicAuthenticationServlet`, which authenticates using HTTP BASIC authentication. This type of authentication might not be what you choose to implement, but it might be useful to see how the code is structured.

WARNING If you use servlets, there are very large security considerations. You must implement your own authentication system to protect information and data integrity. If you have questions about how to do this safely, contact Guidewire customer support. Be extremely careful with coding your servlets and consider all possible security implications.

To make a simple servlet entrypoint

1. Write a Gosu class that extends the class `javax.servlet.http.HttpServlet`.
2. Add the `@Servlet` annotation on the line before your class definition.

3. 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. Optionally your block can use a regular expression to define which URLs you support. To do this, call the `matches` method on the `String` type and pass the regular expression. For example:

   ```
   @Servlet( \ path : String ->path.matches( "/test(/.*)?" ) )
   ```

   This example servlet responds to URLs that start with the text "`/test/`" in the servlet query path. The servlet query path is every character after the computer name, the port, the web application name, and the word "`/servlet/`". In other words, your servlet gets the servlet URL substring identified as *`YOUR_SERVLET_STRING`* in the following URL syntax:

   ```
   http://localhost:8080/cc/service/YOUR_SERVLET_STRING
   ```

4. Override the `doGet` method to do your actual work. Your `doGet` method takes a servlet request object and a servlet response object, which are instances of `HttpServletRequest` and `HttpServletResponse`, respectively.

5. Do your desired work using the request object.

   Important properties on the 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 this Sun Javadoc page:

   ```
   http://java.sun.com/webservices/docs/1.6/api/javax/servlet/http/HttpServletRequest.html
   ```

6. Write an HTTP response using the servlet response object. For example, the following simple response sets the content MIME type, the status of the response (`OK`), and

   ```
   resp.ContentType = "text/plain"
   resp.setStatus(HttpServletResponse.SC_OK)
   ```

   If you want to write output text or other byte stream, use the `Writer` property that hangs off of the response object. For example

   ```
   resp.Writer.append("hello world output")
   ```

The following example responds to servlet URL substrings that start with the string `/test/`. If an incoming URL matches that pattern, the servlet simply echos back the `PathInfo` property of the response object, which contains the path. This example uses the `AbstractBasicAuthenticationServlet`, which ensures there is at basic HTTP authentication.

```
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 " + HttpServletRequest.RequestURI)
  print("Servlet test Query String " + HttpServletRequest.QueryString)

  resp.ContentType = "text/plain"
  resp.setStatus(HttpServletResponse.SC_OK)
  resp.Writer.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;
    }
  }
```

To test this, launch your application and go to this URL in your browser:

```
http://localhost:8080/cc/service/test/is/this/working
```

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:

- 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/servlet/test1/my_magic_subfolder_one_level_down
  http://localhost:8080/cc/servlet/test2/my_magic_subfolder_one_level_down
  http://localhost:8080/cc/servlet/test3/my_magic_subfolder_one_level_down
  ```

chapter 7

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, most of which are optional. You can write your own implementations of plugins in either Gosu or Java. For the complete list of all ClaimCenter plugins, see “Summary of All ClaimCenter Plugins” on page 106.

Related topics include:

- For interfaces, see “Interfaces” on page 191 in the *Gosu Reference Guide*.
- For writing Java code, see “Java and Gosu” on page 101 in the *Gosu Reference Guide*

IMPORTANT For additional Java information that you need to know to write Java plugins, see “Java and Gosu” on page 101 in the *Gosu Reference Guide*.

- For messaging plugins, see “Messaging and Events”, on page 139.
- For authentication plugins, see “Authentication Integration”, on page 239.
- For document and form plugins, see “Document Management”, on page 249.
- For other plugins, see “Other Plugin Interfaces”, on page 333.

This topic includes:

Overview of ClaimCenter Plugins

ClaimCenter *plugins* are mini-programs that ClaimCenter invokes to perform an action or calculate a result. For example, ClaimCenter calls a plugin to generate and return a new claim number. ClaimCenter calls a message transport plugin to send a message to an external system. In general, you can implement plugin interfaces in Gosu or in Java, although Guidewire strongly recommends writing plugin implementations in Gosu.

Plugins are a type of *interface*, which is a set of functions that are necessary for a specific task. The plugin interface defines a strict contract of interaction and expectation between two or more software elements. The code that implements the interface is responsible the for implementation details.

To take a real-world example of an interface, imagine a car stereo system. The buttons, such as for channel up and channel down, are the interface between you and the complex electrical circuits on the inside of the box. You press these buttons to change the channels. However, you probably do not care about the implementation details of how it performs those tasks hidden behind the solid walls of the television. If you replace your stereo, it likely has a similar set of buttons on the stereo. Since you interact only with the buttons and the output audio, if it implements the user interface and sends appropriate sounds to speakers, the internal details do not affect you. You do not care about the details of how the stereo *internally* handles the button presses for “channel up”, “channel down”, “volume up”, to perform essential functions.

Similarly, ClaimCenter defines plugin interfaces to support tasks like document management. If you want to integrate the application with a document management system, the plugin defines the agreement of how the application interacts with the document management system. The implementation details of document management are the responsibility of the code that implements the interface.

For more information about interfaces, see “Interfaces” on page 191 in the *Gosu Reference Guide*.

After you register your plugin class in the Studio plugin editor, ClaimCenter calls the plugin at the appropriate times in the application logic. The plugin performs some action or computation and in some cases returns results back to ClaimCenter.

For example, a method within the validation plugin checks an entity (such as a claim or an exposure) and returns either no errors or a list of errors.

The plugin itself might do most of the work or it might consult with other external systems. You must ensure that the plugin performs reasonably quickly. Plugins typically run while users wait for responses from the application user interface. Because of this, Guidewire strongly recommends that you carefully consider response time, including network response time, as you write your plugin implementations.

In some cases, ClaimCenter provides a built-in demonstration plugin or a default behavior if you do not implement the plugin interface. For instance, you can use Gosu-based validation rules for validation rather than registering a validation plugin.

Number generation is always different for each implementation, so ClaimCenter defines plugins for these calculations. You must register implementations of number generation plugins. Demonstration plugins are preinstalled for these plugin types for you to quickly run new installations of ClaimCenter. However, you **must** replace these with your own plugins before final deployment.

Similarly, you must register your own custom implementation for the policy search plugin. Your plugin must connect to your own policy administration system.

You can implement a plugin interface in the following ways:

- **Gosu plugins.** Guidewire encourages you to implement plugins in Gosu. You write a *Gosu class* that implements the plugin interface. Gosu provides the following advantages for plugin developers
  - Quick access to the full Gosu type system (no need to regenerate the Java API libraries)
  - Quick debugging cycles compared to Java (no need to regenerate the Java API libraries)
  - Native debugging access within Studio
  - Full access to language features like Gosu enhancements, Gosu blocks, and type inference. Some enhancements enhance Java types like lists and collections and give you concise powerful easy-to-read code.
  - Natively call web services on your legacy system directly from Gosu.
  - Any Gosu code can easily call out to Java types (classes, libraries) as needed and even call methods on Java types as if they were native Gosu objects.
  - Some application APIs use Gosu blocks.
  - Some application APIs use native Gosu types (Gosu classes, for example).
  - Some application APIs use generics. When implemented from Java plugins, ClaimCenter strips parameterization from the types as exposed through the external entity libraries. For example, `ArrayList<Address>` becomes simply `ArrayList`.
  - Gosu type inference allows concise easy-to-read code.
- **Java plugins.** You can also write any plugin in Java. If you write your plugin in Java, you must regularly regenerate the Java API libraries to compile against the latest libraries.

IMPORTANT For deployment information, see “Deploying Java Plugins” on page 115 in the *Integration Guide*. For additional Java information that you need to know to write Java plugins, see “Java and Gosu” on page 101 in the *Gosu Reference Guide*.

The following table compares the two types of plugin implementations you can create:

Plugin type	Description	For use with what plugins?	Documentation location

Gosu plugin	Gosu code that runs on each ClaimCenter server within a cluster. (Although messaging plugins run only on the batch server.) Use Guidewire Studio to edit the code. For more information, see “Deploying Gosu Plugins” on page 111.	All plugins. You can write any plugin implementation in Gosu. Anything you can do in Java you can do in Gosu. It is important to note that you can call out to Java classes or libraries easily from Gosu. For more information, see “Java and Gosu” on page 101 in the *Gosu Reference Guide*.	Use the Java plugin specification for the interface and then implement it as a Gosu class with those method signatures:
			`ClaimCenter/java-api/index.html`
			Alternatively, you could use the Gosu documentation:
			`ClaimCenter/build/gosudoc/index.html`
			For details, see “Gosu Generated Documentation” on page 35 in the *Gosu Reference Guide*.

Java plugin	Java code that runs on each ClaimCenter server within a cluster. (Although messaging plugins run only on the batch server.) Use your own Java IDE to edit the code. For more information, see “Deploying Java Plugins” on page 115.	All plugins. You can also create plugin implementations in Java. However, you cannot access the advanced features of Gosu such as blocks, collection enhancement methods, type inference, or native access to Guidewire business data objects. Also, you need to regenerate the Java libraries if you make certain types of data model changes.	The Java API Reference Javadoc: `ClaimCenter/java-api/index.html`

For the API reference documentation for plugin interfaces, refer to the rightmost column in the previous table.

For each plugin interface, Guidewire defines two interfaces defined in different package hierarchies:

- for access from Gosu code and all Guidewire internal classes, ClaimCenter generates what is called the *internal interface* for the plugin.
- for access from the Java language, ClaimCenter generates what is called the *external interface*. Using a separate external interface for Java plugin code reduces the reliability and support risks from Java plugins. For example, this limits changes of data corruption from calling internal methods on Java classes that are unsafe to call from your external code. A proxy code layer within ClaimCenter translates method calls and data between internal and external interfaces.

The plugin interfaces may include ClaimCenter entities, in which cases the external interface generates any data model extensions. After you change the data model, you must regenerate the Java API libraries and recompile your plugin source.

If you make minor changes to the data model during informal development, Java plugins might continue to work without regenerating the Java API libraries. However, this is unsafe. Always regenerate the Java API libraries and use the latest libraries to compile your plugin. If you do not use the latest libraries, serious errors and Java exceptions could occur at unexpected times.

WARNING For Java development, you must always regenerate the Java API libraries and compile plugins against the latest libraries for plugin testing and final deployment.

For more information about regenerating the Java API libraries, see “Regenerating the Integration Libraries” on page 17.

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.

Plugin Registration

Once you write your plugin code, register the plugin. You register plugins with a special editor within Studio. For more information about that user interface, see “Using the Plugins Editor” on page 141 in the *Configuration Guide*.

As you register plugins in Studio, the interface prompts you for a plugin interface name (in a picker) and in some cases additionally a plugin name.

The three interfaces for messaging plugins support multiple implementations of a single interface. For example, register multiple messaging plugins for the `MessageTransport` 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. Use that plugin *name* as you configure the messaging destination in the separate Messaging editor in Studio.

Similarly, the `IEncryption` plugin interface supports multiple implementations, although only in special circumstances. This new plugin handles property-level encryption. Normally, just use the default name `IEncryption`. If at later date you change your encryption algorithm, first register a regular encryption plugin with the new algo-

rithm. Next, register another with the name value `OldEncryption` to indicate that the upgrader must use that one to decrypt the data before reencrypting with the new encryption plugin.

Other than these exceptions, plugins must not have multiple classes registered for the same plugin interface.

If you write a Java plugin, you likely need to use the entity libraries to access entity data. See “Using Java Entity Libraries” on page 117.

If you want your Gosu plugin to call out to Java code, you can. You can use the features discussed in “Java and Gosu” on page 101 in the *Gosu Reference Guide* to call Java classes or libraries. You can even use Guidewire entities using the tools discussed in “Using Java Entity Libraries” on page 117 in the *Gosu Reference Guide*. However, do not register your additional Java code in these cases *as plugins* in the plugin registry in Studio.

Included Plugin Implementations

Note the following things about included plugin implementations:

- Some plugin interfaces include built-in plugins that implement the default behavior if you do not implement the plugin.
- Some plugin interfaces include built-in *demo plugins* intended for initial deployment only. For example, the claim number generator plugin.
- Some plugin interfaces include example plugins, which include source code and compiled class files. In general, examples are in `ClaimCenter/java-api/examples/`.... Some of the class files are pre-installed in `ClaimCenter/modules/configuration/modules/configuration/plugins/`.... For example, ClaimCenter includes an example message transport plugin that writes messages to the ClaimCenter console as if it were an external system.
- Some plugin interfaces communicate between Guidewire applications, such as between ClaimCenter and ContactCenter. If you use two or more Guidewire applications that have a built-in and supported integration, Guidewire provides Gosu plugins that communicate between them.
  - A built-in implementation of `IAddressBookAdapter` lets ClaimCenter communicate with ContactCenter.
  - A built-in implementation of `IPolicySearchAdapter` lets ClaimCenter communicate with PolicyCenter.

  For information about installing and configuring multiple applications to work together, see the *ClaimCenter Installation Guide*.

Configuring ClaimCenter for New Plugins

Register all your plugins in the Guidewire Studio plugin registry. The plugin registry elements tell ClaimCenter how to call your plugin. To use the plugin registry, you must know the plugin interface name, the class name of your implementing class, and any initialization parameters that you want to add. Additionally, you need to specify a plugin directory for any Java classes and library files, which is a subdirectory in the `ClaimCenter/plugins` directory.

In general, you cannot have multiple plugins registered for the same interface. Be sure to change the existing one or remove the previous one before registering your own plugin for that interface. However, messaging plugins can register multiple plugins for each plugin interface: `MessageRequest`, `MessageTransport`, and `MessageReply`.

For details, see “Using the Plugins Editor” on page 141 in the *Configuration Guide*.

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.

If errors occur for which no valid response seems appropriate at all, Java plugins or Gosu plugins must catch checked exceptions. Clean up anything necessary to clear up and then rethrow the exception as a run-time exception. The ClaimCenter server catches any run-time exception and **roll back** any related database transaction, and then displays an error in the application user interface.

In rare cases if an error occurs, a plugin can choose to return a valid but *empty* response rather than throwing an exception. This is best in some user interface contexts in which success is most important. For example, perhaps for some external lookup, you can return zero matches rather than throw exceptions, which rolls back changes to the database for the current action. Only use this non-error approach in cases in which you are sure it is appropriate. Consult the Guidewire Technical Support group if you have questions about this.

Quick Server Restarting and Testing Tips For Plugin Developers

Admin Tools For Reloading an Application

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
```

Mock Entities for Java Plugins

To help test Java code that uses entity libraries, ClaimCenter includes a utility class called `EntityMock` that generates a *mock* (fake) instance of a ClaimCenter entity. This allows you to unit test some types of code without loading the code as a real ClaimCenter plugin. If you confirm your code's basic logic, you can deploy the plugin in a live ClaimCenter server as a ClaimCenter plugin and change little code. For more information, see "Testing Your Entity Code in Java (EntityMock)" on page 125 in the *Gosu Reference Guide*.

Summary of All ClaimCenter Plugins

The following table summarizes the plugin interfaces that ClaimCenter defines.

Interface	Description
Authentication plugins (see "Authentication Integration", on page 239)	
`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.
`AuthenticationSource`	A marker interface representing an authentication source. 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.
`AuthenticationSourceCreator-Plugin`	Creates an authentication source (an `AuthenticationSource`) from an HTTP protocol request (from an `HTTPRequest` object). The authorization source must work with your registered implementation of the `AuthenticationServicePlugin` plugin interface.
`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.

Interface	Description
`ABAuthenticationPlugin`	Advanced authentication between ClaimCenter and ContactCenter, such as hiding/encrypting name and password information. See “ABAuthenticationPlugin for ContactCenter Authentication” on page 246.
Document management and note plugins (see “Document Management”, on page 249)	
`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.
`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.
`IDocumentMetadataSource`	Stores metadata associated with a document, typically to store it in a remote document management system; see `IDocumentContentSource` 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.
`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.
`IEmailTemplateSource`	Gets email templates. By default, ClaimCenter stores the email templates on the server but you can customize this behavior by implementing this plugin.
`IDocumentTemplateSerializer`	*Use the built-in version of this plugin using the “plugin registry” but generally it is best not implement your own version.* 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.
Messaging plugins (see “Messaging and Events”, on page 139)	
`MessageRequest`	Optional pre-processing of messages, and optional post-send-processing (separate from post-acknowledgement processing).
`MessageTransport`	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).
`MessageReply`	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.
`MessagingNotification`	Gets notification after any messaging destination (including built-in destinations such as email) suspends or resumes. This plugin gets notified after suspend and resume. In contrast, ClaimCenter calls the `suspend` and `resume` methods of messaging plugins before (not after) suspend and resume.

Interface	Description
Financials plugins	
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. See “Bulk Invoice Integration” on page 221.
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. See “Initial Reserve Initialization for Exposures” on page 236.
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 (`IExchangeRateSetPlugin`) interface, whose main task is to create `ExchangeRateSet` entities encapsulated in a `ExchangeRateSet` entity. See “Exchange Rate Integration” on page 236.
IBackupWithholdingPlugin	Handles backup withholding on checks. See “Deduction Calculations for Checks” on page 235.
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. See “Handling Other Deductions” on page 235.
Other ClaimCenter plugins	
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. See “Claim Number Generator Plugin” on page 333.
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. See “Policy Search Plugin” on page 449.
IAddressBookAdapter	Guidewire application interface to an address book application. Deployments that use Guidewire ContactCenter do not implement this plugin, but instead use the built-in plugin to connect to ContactCenter. For more information, see “Address Book and Contact Search Plugins” on page 301.
IContactSearchAdapter	Allows ClaimCenter to search an external database of people or companies for contact information. If a resulting contact connects to a claim, a snapshot of that contact stores locally in ClaimCenter. Provide search criteria in the ClaimCenter search screen and this plugin returns a list of contacts. The default behavior is to search the internal address book. Implement this to enable search of internal or external address books, or both. For more information, see “Address Book and Contact Search Plugins” on page 301.
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. See “Approval Plugin” on page 335.

Interface	Description
`IGeocodePlugin`	Geocoding support within ClaimCenter and ContactCenter. See “Geographic Data Integration”, on page 315.
Other platform plugins	
`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 *for development (non-production) use only.* It programmatically changes the system time to accelerate the perceived passing of time within ClaimCenter. **WARNING:** The testing clock plugin is for application development only. You must **never** use it on a production server. See “Testing Clock Plugin (Only For Non-Production Servers)” on page 336.
`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. See “Encryption Integration” on page 401.
`IBaseURLBuilder`	This plugin generates a *base URL* to use for web application pages affiliated with this application, given the HTTP servlet request URI (`HttpServletRequest`). See “Defining Base URLs for Fully-Qualified Domain Names” on page 334.
`ManagementPlugin`	The external management interface for ClaimCenter, which allows you to implement management systems such as JMX, SNMP, and so on. See “Management Integration”, on page 411.
`IScriptHost`	*You can use the built-in version of this plugin using the plugin registry APIs but do not attempt to implement your own version.* Evaluates Gosu code to provide context-sensitive data to other services, such as property data paths expressed as Gosu expressions. For example, this service could evaluate a String with contents `"policy.myFieldName"` at run time to retrieve and evaluate the data model extension property named `myFieldName`.
`IProcessesPlugin`	Creates new batch processes (new custom batch process subtypes). For more information, see “Custom Batch Processes” on page 325 in the *Integration Guide*.
`IStartablePlugin`	Creates new plugins that immediate instantiate and run on server startup. For more information, see “Startable Plugins” on page 128.
`IPreupdateHandler`	Implements your preupdate handling in plugin code rather than in the built-in rules engine. See “Preupdate Handler Plugin” on page 337.
`IWorkItemPriorityPlugin`	Calculate a work item priority. For more information, see “Work Item Priority Plugin” on page 336.

ContactCenter Notes

The table mentioned earlier describes plugins for ClaimCenter. The plugins for ContactCenter are the same except it does not include the plugins identified as “other ClaimCenter plugins” and “financials plugins”. In particular, the address book plugin and the contact search plugin are plugins for the ClaimCenter. The ContactCenter application does **not** define these plugin interfaces.

For related information, see “Address Book Integration”, on page 299. That topic includes a complete listing of the plugin interfaces defined within ContactCenter.

Plugin Implementation Overview

To implement a plugin, perform the following high-level steps:

1. Implement the interface for that plugin.

2. Update plugin templates (in some cases only) to prepare data for the plugin.
3. Configure the server for the new plugin.
4. Install and deploy the plugin, including any necessary library files.
5. Rebuild the application WAR file.

Step 1: Implement the Plugin Interface

Implement the interface for that plugin. For typical Java use, regenerate the Java API libraries and base a new class on the a generated Java interface, such as `IClaimNumGenAdapter`. Implement all public methods of the interface. Create as many other private methods and classes as you need to provide the logic of the interface, but ClaimCenter does not need to know anything about these.

The implementation varies based on the type of plugin:

- For Gosu plugins, implement a Gosu class that implements the interface. Create a new Gosu class in Guidewire Studio. See “Deploying Gosu Plugins”, on page 111.
- For Java plugins, implement a Java class that implements the interface. See “Deploying Java Plugins”, on page 115.

For example, to implement `IClaimNumGenAdapter` your class would look like the following:

```
public class ClaimNumGenTestAdapter implements IClaimNumGenAdapter {

  public ClaimNumGenTestAdapter() {
  }

  public String generateNewClaimNumber(String templateData) {
    // Generate new claim number here, perhaps with the sequence generator
    return com.guidewire.pl.plugin.util.SequenceUtil.next(100, 1, "myClaimNum")
  }

  public String generateTempClaimNumber(String templateData) {

    // use similar code as with generateNewClaimNumber , or a different algorithm if appropriate
    return com.guidewire.pl.plugin.util.SequenceUtil.next(100, 1, "myClaimNum")
  }
}
```

For a more detailed example of implementing this interface, see “Deploying Java Plugins” on page 115.

Step 2: Design or Update Plugin Templates (In Some Cases Only)

Some plugin interface methods have parameters that directly specify their data as simple objects such as `String` values. Other plugin methods require structured data such as a `Java.Map` object. Other plugin methods require a full ClaimCenter entity such as `Claim`. Parameter objects might point to other objects, resulting in a potentially large object graph.

However, some plugin methods take a single `String` that it must parse to access important parameters from entity. This text data is *template data* and some methods require it as a parameter called *`templateData`*. Template data is the output of a Gosu template called a *plugin template*.

This potentially-large `String` is a parameter to the plugin for the aforementioned interface methods that take template data as a parameter. This approach lets ClaimCenter pass the plugin all necessary properties in a large data graph but with minimal data transfer.

For details, see “Writing Plugin Templates in Gosu” on page 120.

Step 3: Configure the Server for the New Plugin

Configure the ClaimCenter plugin registry for your new plugin by adding an element to the plugin registry using a special plugin editor within Studio. For more information about that user interface, see “Using the Plugins Editor” on page 141 in the *Configuration Guide*.

You need to specify the plugin interface name and the class name of your plugin implementation.

The details for configuration vary slightly by plugin type, described in the following sections:

- “Deploying Gosu Plugins” on page 111
- “Deploying Java Plugins” on page 115
- “Accessing and Creating Guidewire Entities from Java” on page 118

Some plugins specify their own required or optional parameters, for example connection URLs or authentication information. Be sure to include all necessary parameters.

Note: If the plugin you write is a messaging plugin, you must also register it in the messaging registry in the messaging editor in Studio. For details, see “Using the Messaging Editor” on page 161 in the *Configuration Guide* for the editor or for more messaging information, see “Messaging and Events” on page 139.

Step 4: Deploy the Plugin and Libraries

Plugin deployment details vary greatly by plugin type, as described in the following sections:

- “Deploying Gosu Plugins” on page 111
- “Deploying Java Plugins” on page 115
- “Accessing and Creating Guidewire Entities from Java” on page 118

For Java plugins, move the desired Java `.class` files to the directory

`ClaimCenter/modules/configuration/plugins/`*`plugindir`*`/classes`

For Java plugins, remember to also copy necessary `.lib` files in that example’s `lib` folder to the directory:

`ClaimCenter/modules/configuration/plugins/`*`plugindir`*`/lib`

The *`plugindir`* value is the plugin directory that you specify in the plugin registry. For example, you might want to put messaging plugin classes in a `messaging` subdirectory of the plugins directory. To set this value, see “Using the Plugins Editor” on page 141 in the *Configuration Guide*.

For an extended discussion of Java class loader rules and class repositories, see “Java Class Loading, Delegation, and Package Naming” on page 137.

Step 5: Rebuild the Application WAR File

Follow the instructions in the *ClaimCenter Installation Guide* for application deployment details.

Using the Logging APIs

You might find it useful to log messages with the ClaimCenter logging APIs within plugin code to log any assumptions, progress, warnings, and errors. For detailed information about logging within integration code, see “Logging”, on page 293.

If your plugin deploys in a ClaimCenter server cluster, there must be plugin instances on every server in the cluster. You may want to consider including information within the logging information about which server within which the plugin is running. For more information about system properties that can be used to identify the server environment, see “Writing Plugin Templates in Gosu” on page 120.

Also, for messaging plugins, see “Saving the Destination ID for Logging or Errors” on page 184.

Deploying Gosu Plugins

You can write all your plugin implementations in Gosu. By using Gosu, you get full advantage of Gosu language features. For example, Gosu blocks, Gosu collection enhancement methods, native access to data model types,

intelligent code completion after you type the period (.) character, and type inference. These features help you quickly write concise and easy-to-maintain code.

Gosu plugins typically perform one or more of the following tasks in each method:

- Perform all a variety of actions in Gosu.
- Call out to a remote SOAP API.
- Call out to Java code, as discussed in “Java and Gosu” on page 101 in the *Gosu Reference Guide*.

Writing a Gosu Plugin

To write a Gosu plugin, write a Gosu class that implements the plugin interface and ensure that the class properly defines all required methods perform appropriate tasks. ClaimCenter calls your plugin at appropriate times in the business logic. From within Gosu, all plugin interfaces are in the package `gw.plugin.*`. As soon as you type “`gw.plugin.`” in your Gosu code, Studio displays your plugin interface choices.

IMPORTANT In Studio as you create your new class, create it within your normal package hierarchy such as `com.mycompany.plugins`. Do not create your own plugins in the built-in `gw.*` package hierarchy or the `plugins.*` package hierarchy.

Always add “`implements` *`INTERFACENAME`*” in your plugin class definition, for example:

```
package com.mycompany.plugins

class MyTransport implements gw.plugin.MessageTransport {
  ...
}
```

If your class does not properly implement the plugin interface, Studio alerts you to compilation errors. You must fix these issues.

The most common compilation issue is that if an interface’s method looks like properties, it must be implemented as a Gosu property. In other words, if the interface contains a method starting with the substring `"get"` or `"is"` and takes no parameters, define the method using the special Gosu property syntax. In contrast, 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 com.mycompany.plugins

class MyClass implements gw.plugin.ExamplePlugin {

  property get MyVar() : String {
  ...
  }
}
```

Using Parameters

Gosu plugin implementation can access parameters passed from the plugin registry in Studio. For example, the plugin registry could pass server names, port numbers, timeout values, or other settings you want to set from within the plugin registry. For more information about that user interface to add parameters in the plugin registry, see “Using the Plugins Editor” on page 141 in the *Configuration Guide*. For information about getting the parameters from your plugin, see “Getting Plugin Parameters from the Plugins Editor” on page 119.

Registering Your Plugin

Register plugins using a special plugin editor within Studio. For more information about that user interface, see “Using the Plugins Editor” on page 141 in the *Configuration Guide*.

You must specify the plugin interface name and the class name of your plugin implementation.

For example, `MyClaimNumGenerator` deploys within the Tomcat web application container at the path *`TOMCAT`*`/webapps/cc/modules/configuration/config/resources/classes/Plugins/MyClaimNumGenerator.gsm`.

Additionally, you must specify a plugin directory, which is a subdirectory in the `ClaimCenter/plugins` module where your class and library files reside.

Each plugin element can contain various extra parameters, which can be repeated as desired:

Example Gosu Plugin

The following example Gosu plugin defines a claim number generator and also uses a custom parameter called `myCustomParameterName`:

```
package Plugins;

uses java.io.*;
uses java.util.Properties;
uses java.util.Map;
uses com.guidewire.logging.LoggerCategory;
uses java.lang.System;
uses java.lang.Thread;

class MyClaimNumGenTestGosuPlugin
{
 private var _tmpDir    : String;
 private var _rootDir   : String;
 private var _logger    : LoggerCategory;
 private var _prefixNew = 5;

  function MyClaimNumGenTestGosuPlugin(params : Map)
  {
   _logger = LoggerCategory.PLUGIN;
   _logger.info("*** claimnumgen called *** " )

    if (params.containsKey("myCustomParameterName")) {
     var tmp1 = params.get("myCustomParameterName") as String;
     _prefixNew = tmp1.substring( 0, 1 );
    }
    else{
     // ...
}

}

function generateNewClaimNumber(templateData : String) : String
  {
    var nextNum = _prefixNew + gw.api.system.database.SequenceUtil.next(10, "ClaimNumGen")
    return nextNum
  }

function generateTempClaimNumber(templateData : String) : String
  {
    var nextNum = _prefixNew + gw.api.system.database.SequenceUtil.next(10, "TempClaimNumGen")
    return nextNum
  }
function cancelNewClaimNumber(templateData : String, claimNumber : String) : String
  {
   // perhaps do nothing. this is only if you need special handling to reduce “missing”
   // numbers already allocated to claims
  }
}
```

The simple example mentioned earlier uses a sequence generator to generate the number. A more real-world example would probably use the properties on the claim that passed to it in the method arguments. In this case, and some other plugins like it, the parameter is a block of text in a potentially-large `String` object called *`templateData`*. For more information about how to create this text and how to parse a `String` like this, see “Writing Plugin Templates in Gosu” on page 120.

This is an example message transport plugin:

```
uses java.util.Map;
uses java.plugin;

class MyTransport implements MessageTransport, InitializablePlugin {

   // note the empty constructor. The application now longer calls the constructor
```

```
    // with a map of parameters. If you do provide an empty constructor, the application
    // calls it as the app instantiates the plugin, which is before calling setParameters
    construct()
    {
    }

    function setParameters(map: java.util.Map)  {

     // 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()
    }
}
```

If Your Gosu Plugin Needs Other 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.

Place any class files in either of the two following locations for the root of your class hierarchy:

```
ClaimCenter/modules/configuration/plugins/shared/classes/...
ClaimCenter/modules/configuration/plugins/Gosu/classes/...
```

For example, suppose a class's fully-qualified name is `examples.plugins.messaging.MyMessageTransport`.

Place the `.class` files in the either of the following locations:

```
ClaimCenter/modules/configuration/plugins/shared/classes/examples/plugins/
     messaging/MyMessageTransport.class
ClaimCenter/modules/configuration/plugins/Gosu/classes/examples/plugins/
     shared/MyMessageTransport.class
```

If your plugin requires additional libraries, copy the library JAR files to either of the following directories:

```
ClaimCenter/modules/configuration/plugins/shared/lib/
ClaimCenter/modules/configuration/plugins/Gosu/lib/
```

Remember to use the `shared` library folder to store any Java library JAR files needed by your plugin class implemented in Java. This is particularly useful if multiple of your Java plugins need to use the same code.

IMPORTANT The instructions above only apply if the plugin interface itself is implemented by a Gosu class. If you implement the plugin interface in Java and that code needs additional Java files, the instructions for where to put files is slightly different. See "Deploying Java Plugins" on page 115.

See Also

- "Java Class Loading, Delegation, and Package Naming" on page 137
- "Java and Gosu" on page 101 in the *Gosu Reference Guide*
- "Java Entity Libraries Overview" on page 114 in the *Gosu Reference Guide*

Running Gosu Plugins in 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: Messaging plugins exist only on the batch server in the cluster.

Deploying Java Plugins

You can write your plugin in Gosu or in Java. If you write your plugin in Java, you must regularly regenerate the Java API libraries to compile against the latest generated Java libraries. To see a comparison of Gosu and Java for plugin development, see “Overview of ClaimCenter Plugins” on page 102.

Implement the plugin interface, which is always in one of the following the package hierarchies: `com.guidewire.cc.plugin.*` or `com.guidewire.pl.plugin.*`. The second of those two categories is for Guidewire platform-level plugins.

For example, a ClaimCenter claim number generator plugin would be implemented as a Java class that implements the interface `com.guidewire.cc.plugin.claimnumbergen.IClaimNumGenAdapter`

To implement a plugin in Java, start by creating a class that implements the desired interface. For example, to implement `IClaimNumGenAdapter` your class might look like the following:

```
package examples.plugins.ClaimNumGen;

import com.guidewire.cc.webservices.plugin.IClaimNumGenAdapter;

import java.util.Properties;
import java.io.ByteArrayInputStream;

public class ClaimNumGenTestAdapter implements IClaimNumGenAdapter {

  public ClaimNumGenTestAdapter() {
  }

  public String generateNewClaimNumber(String templateData) {

    // Use template data to populate claim number
    Properties claimProperties = new Properties();

    try {
      claimProperties.load(new ByteArrayInputStream(templateData.getBytes()));
    } catch (java.io.IOException IOE) {
    System.out.println("ClaimNumGenTestAdapter:  bad template data");
    }

    String policyNumber = claimProperties.getProperty("PolicyNumber");
    int policyLength = policyNumber.length();
    String claimPrefix = policyNumber.substring(policyLength-3, policyLength);

    java.util.Random generator = new java.util.Random(System.currentTimeMillis());
    int claimBody = generator.nextInt(100);

    String claimSuffix = new Long(System.currentTimeMillis()).toString();
    claimSuffix = claimSuffix.substring(claimSuffix.length()-6, claimSuffix.length());

    String claimNumber = claimPrefix + "-" + claimBody + "-" + claimSuffix;
    System.out.println("generated new ClaimNumber:  " + claimNumber);

    return claimNumber;
  }

public String generateTempClaimNumber(String templateData) {

   // use similar code as with generateNewClaimNumber mentioned earlier…
 }
public String cancelNewClaimNumber(String templateData) {

   // use similar code as with generateNewClaimNumber mentioned earlier…
```

```
  }
}
```

Note: If you implement your first claim number generator, or later change the claim number format later on during development, you must coordinate changes within the field validators for claim number. For more details, see “Claim Number Generator Plugin” on page 333.

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).
- If a method has containers (arrays, lists, or maps) as arguments or return values in Java classes, carefully read the documentation about special issues with containers. It is important to understand how ClaimCenter makes a copy of the container.

For more information about these issues, carefully read the topic “Java and Gosu” on page 101 in the *Gosu Reference Guide*.

IMPORTANT Read the topic “Java and Gosu” on page 101 in the *Gosu Reference Guide* for important information about writing Java code within ClaimCenter. In particular, that topic contains important information about working with entities and working with containers (arrays, lists, and maps) from Java code in ClaimCenter.

Initializing a Java Plugin

After you write your Java plugin, be sure to configure your ClaimCenter server and store the class and library files in the correct directory. Register plugins using a special plugin editor within Studio. For more information about that user interface, see “Using the Plugins Editor” on page 141 in the *Configuration Guide*.

You need to specify the plugin interface name and the class name of your plugin implementation.

WARNING Java classes that you deploy must **never** have a fully-qualified package name that starts with `"com.guidewire."` because that would interfere with Java class loading behavior. For related discussion, see “Java Class Loading, Delegation, and Package Naming” on page 137.

Additionally, you must specify a plugin directory, which is a subdirectory in the `ClaimCenter/plugins` module where your class and library files reside.

You can also add one or more optional parameters you can pass to a plugin at initialization in the form of name/value parameter strings. See “Getting Plugin Parameters from the Plugins Editor” on page 119.

Which Libraries to Compile Java Code Against

If you have not regenerated the Java API libraries since your last change to the data model, it might be necessary to regenerate them. For details, see “Regenerating the Integration Libraries” on page 17.

ClaimCenter creates the library JAR files at the path:

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

Compile against the `.jar` files in that directory. The Guidewire-specific libraries have names that begin with `"gw-"`. For more information about working with Guidewire entities from your Java code, see “Using Java Entity Libraries” on page 117 in the *Gosu Reference Guide*.

It is important to note that you must **copy** the `gw-entity-cc.jar` file to the following directory for ClaimCenter to find it at run time:

```
ClaimCenter/modules/configuration/plugins/shared/lib
```

IMPORTANT Compile your Java code against the files in `ClaimCenter/java-api/lib/`. Those are the only supported versions of Guidewire Java libraries, including the generated files from Guidewire entities. Do not rely on Java libraries in internal modules or in other directories in the product hierarchy. However, you **must** copy the `gw-entity-cc.jar` file to your configuration directory, as discussed earlier, every time after you regenerate the libraries.

Where To Put Your Java Class and Library Files for Java Plugins

For your Java class that implements the plugin, place class files within the configuration environment within:

```
ClaimCenter/modules/configuration/plugins/plugindir/classes
```

The placeholder *`plugindir`* represents the *plugin directory name* in the plugin registry for that plugin. If the plugin registry does not define a plugin directory for that plugin, or if class files are not there, the application looks in the directory called `shared`. In other words, the application looks in the location:

```
ClaimCenter/modules/configuration/plugins/shared/classes
```

Next, put any additional Java library files (JAR files) that your code needs in the `lib` subdirectory:

```
ClaimCenter/modules/configuration/plugins/plugindir/lib
```

If the plugin registry does not define a plugin directory for that plugin, or if library files are not there, the application looks for classes in the directory called `shared`. In other words, the application looks in the location:

```
ClaimCenter/modules/configuration/plugins/shared/lib
```

For example, suppose a class's fully-qualified name is `examples.plugins.messaging.MyMessageTransport` and its plugin registration specified the plugin directory (the `plugindir` attribute) as the value `messaging`.

Place the `.class` files in the either of the following locations:

```
ClaimCenter/modules/configuration/plugins/messaging/classes/examples/plugins/
    messaging/MyMessageTransport.class
ClaimCenter/modules/configuration/plugins/messaging/classes/examples/plugins/
    shared/MyMessageTransport.class
```

If your plugin requires additional libraries, copy the library JAR files to either of the following directories:

```
ClaimCenter/modules/configuration/plugins/messaging/lib/
ClaimCenter/modules/configuration/plugins/shared/lib/
```

Remember to use the `shared` library folder to store any Java library JAR files needed by multiple plugins implemented in Java.

For more details of class loading, see "Java Class Loading, Delegation, and Package Naming" on page 137.

IMPORTANT The instructions above only apply if the plugin interface itself is implemented by a Java class. If you implement the plugin interface in Gosu and that code needs additional Java files, the instructions for where to put files is slightly different. See "If Your Gosu Plugin Needs Other Java Classes and Library Files" on page 114.

As mentioned earlier in this topic, if your Java plugin uses Guidewire entities at all, you must also copy the `gw-entity-cc.jar` file to the directory:

```
ClaimCenter/modules/configuration/plugins/shared/lib
```

If you have not regenerated the Java API libraries since your last change to the data model, it might be necessary to regenerate them, depending on your changes. For details, see “Regenerating the Integration Libraries” on page 17.

IMPORTANT Read the topic “Java and Gosu” on page 101 in the *Gosu Reference Guide* for important information about writing Java code within ClaimCenter.

Parameterization of Types Stripped from Java External Entities

If any plugin definition uses parameterized types, ClaimCenter strips parameterization from those types when generating external entities in the Java entity libraries. This applies to both method parameters and return types

For example, if a plugin interface returns the type `List<User>`, in Java it simply returns the type `List`.

Temporarily Disabling a Java Plugin

By default, ClaimCenter call Java plugins in the plugin registry at the appropriate time in the application logic. To temporarily or permanently disable a plugin in the registry, add the `disabled` attribute set to the value `true`.

Java Plugin Utilities

Accessing and Creating Guidewire Entities from Java

To access or create Guidewire entities such as `Claim` from Java, you access special entities through a special interface generated in the Java API libraries. ClaimCenter generates *entity library* classes in the JAR file `gw-entity-cc.jar`. Compile your Java plugins against this generated library.

Read the “Java Entity Libraries Overview” on page 114 in the *Gosu Reference Guide* for important information about how information passes between Java code you write and internal ClaimCenter code. See “Java Entity Libraries Overview” on page 114 in the *Gosu Reference Guide*.

In some cases, your plugin code may need to create new entities. For example, use the following Java expression to get a new `Claim` entity from Java:

```
myClaim = EntityFactory.getEntityFactory().newEntity(Claim.class)
```

For information about this class, see “Creating Entities from Java” on page 117 in the *Gosu Reference Guide*.

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.

The primary API to this system is the SOAP API `ITypelistAPI`, which you configure using an XML file. However, because typecode translation may occur very frequently in Java plugin code and calling the SOAP API sometimes may result in lower server performance. For higher performance, Java plugins can use a utility class called `TypecodeMapperUtil`. It has similar methods to the SOAP API, such as `getInternalCodeByAlias`.

For example, to translate a typecode with `TypecodeMapperUtil`, use code such as:

```
TypeKey tk = TypecodeMapperUtil.getInternalCodeByAlias("Contact", "ABC:system1", "ATTORNEY");
```

To modify the typelist conversion configuration file or to convert typecodes from Gosu or Java, see “Mapping Typecodes to External System Codes” on page 81.

Mock Entities for Java Plugins

To help you test your own Java code that uses the entity libraries, ClaimCenter includes a utility class called `EntityMock`. It generates a *mock* (fake) instance of a ClaimCenter entity. In some cases, this allows unit test programs and easier debugging without loading the code as a real ClaimCenter plugin. After you test and confirm the code’s logic, deploy the plugin in a live ClaimCenter server as a ClaimCenter plugin without changing code. For more information, see “Testing Your Entity Code in Java (EntityMock)” on page 125.

Dependent JARs for Java Plugins in Eclipse

The following JAR files are required to compile Java plugins in Eclipse:

- `ClaimCenter\java-api\lib\gw-gosu-core-api.jar`
- `ClaimCenter\java-api\lib\gw-entity-cc.jar`
- `ClaimCenter\java-api\lib\gw-plugin.jar`

The following JAR file is required to deploy Java plugins in Eclipse

- `ClaimCenter\java-api\lib\gw-entity-cc.jar`

Getting Plugin Parameters from the Plugins Editor

In the Studio plugins editor, you can add one or more optional parameters to pass to your plugin during initialization. For example, you could use the Plugins 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:

```
class MyDocumentProduction implements IDocumentProduction {
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 appplication calls setParameters
   construct() {
   }

   function setParameters(map: java.util.Map)  {

    // access values in the MAP to get parameters defined in plugin registry in Studio

    _servername = map.MyServerName

   }

  // 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()
    }
}
```

Built-in Property in the Map: Root Directory

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`.

Built-in Property in the Map: Temp Directory

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`.

Writing Plugin Templates in Gosu

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.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 291 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 (policy type, existence of coverage) 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.getBytes()));
} 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/adapters/`.... 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 `Assignment_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.

The Plugin Registry

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

var plugin = Plugins.get( IBillingSummaryPlugin )

try{
  plugin.updateAccountBillingSettings( accountNumber, this )
}catch(e){
  e.printStackTrace()
  throw new DisplayableException(e.Message)
}
```

Alternatively, you can request a plugin by name. 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`:

```
uses gw.plugin.Plugins

var plugin = Plugins.get( "MyPluginName" )

try{
  plugin.updateAccountBillingSettings( accountNumber, this )
}catch(e){
  e.printStackTrace()
  throw new DisplayableException(e.Message)
}
```

Is Enabled

From Gosu you can use the Plugins class to determine if a plugin is enabled in Studio given its interface name or its plugin name that you specified in Studio. Simply call the `isEnabled` method on the `Plugins` class and pass either interface name type or a plugin name `String` value. For example:

```
uses gw.plugin.Plugins

var billingSummaryEnabled = Plugins.isEnabled( IBillingSummaryPlugin )
```

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 and Static Variables

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.

Note: 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 “Event and Messaging Flow”, on page 143. 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 the 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 311.

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 an 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 Your Plugins to Support Server Clusters” on page 127). 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 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 124
- "Using Synchronized Methods (Java Only)" on page 125
- "Using Java Synchronized Blocks of Code (Java only)" on page 125

For thread safety issues in Gosu, see "Gosu Static Variables and Gosu Thread Safety" on page 124.

Additionally, note some similar issues related to multi-server (cluster) plugin design in "Design Your Plugins to Support Server Clusters" on page 127.

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 describes 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 other information about concurrency, see "Concurrency" on page 311.

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 other information about concurrency, see "Concurrency" on page 311.

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 plugin registry in Studio. For more information about plugin parameters, see "Deploying Java Plugins" on page 115.

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 during the time Java is constructing it.

For important other information about concurrency, see "Concurrency" on page 311.

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 125.

Both approaches protects 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 other information about concurrency, see “Concurrency” on page 311.

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 a 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 and Static Variables” on page 123.

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 and Static Variables” on page 123 to be thread safe.

For important other information about concurrency, see “Concurrency” on page 311.

Design Your Plugins 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. For more information, see “Event and Messaging Flow”, on page 143.

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 and Static Variables” on page 123 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 other information about concurrency, see “Concurrency” on page 311.

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 editor as discussed in “Using the Plugins Editor” on page 141 in the *Configuration Guide*.

For more information about `env` and `serverid` settings in general, see “Using the registry Element to Specify Environment Properties” on page 18 in the *System Administration Guide*.

Gosu plugins can query system properties using the methods `getEnv`, `getServerId` and `isBatchServer`, all on the `ServiceInfoSource` 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.domain.ServiceInfoSource.getEnv("gw.cc.env")
var serverID = gw.api.domain.ServiceInfoSource.getServerId()
var isBatchServer = gw.api.domain.ServiceInfoSource.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");
```

For more information about managing clusters, see the *ClaimCenter System Administration Guide*.

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, startable plugins only operate on the batch server.

IMPORTANT Startable plugins run only on the batch server.

Startable plugins have some similarities with the messaging reply plugin, in the sense that 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 (SOAP)" on page 25.

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 editor in the following location **Configuration** → **Plugins** → **gw** → **api** → **startable** → **IStartablePlugin**. Right-click on that interface name and choose **Add implementation...**

Note: 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 325.

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 inline function. (See “Gosu Blocks” on page 213.) The Gosu block you pass to the `execute` method takes no arguments.

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 130.

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. See “Bundles and Transactions” on page 275 for more information about working with bundles.

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. See “If You Define Your Startable Plugin In Java” on page 135.

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 “Returning Query Results” on page 143 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 to writable bundle + save result. The original entity is in a readonly bundle
        // in query results. The add method adds to your writable bundle so you can modify it.
        var e_in_new_bundle = bundle.add(e)

        // modify properties as desired on the result of bundle.add(e)
        e_in_new_bundle.Department = "Example of setting a property on a writable entity."
       }

    })
  //Note: You do not need to commit the bundle here. The execute method commits after your block runs.
  }
```

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
  }
}
```

JMS Queue Startable Plugin Example

The following is a more complex example that creates a JMS queue and listens on it:

```
package gw.api.startableplugin
uses gw.api.startable.IStartablePlugin
uses gw.api.startable.StartablePluginCallbackHandler
uses javax.jms.QueueReceiver;
uses javax.jms.MessageListener;
uses javax.jms.Message;
uses java.lang.RuntimeException
uses gw.api.startable.StartablePluginState

class JMSStartablePlugin extends JMSExampleBase implements IStartablePlugin, MessageListener
{
  var _state = StartablePluginState.Stopped;
  var _callback : StartablePluginCallbackHandler;
  var _queueReceiver : QueueReceiver = null;

  construct()
  {
  }

  override property get State() : StartablePluginState
  {
    return _state
  }

  override function initQueues() : void
  {
    // Create reply queue and set up a receiver
    _queueReceiver = createQueueReceiver(SEND_QUEUE_NAME, this);

    try {
      // Specify this object as the message listener on the queue
      _queueReceiver.setMessageListener(this);
    } catch (e) {
      throw new RuntimeException("Failed to set listener for queue receiver: " + e.getMessage(), e);
    }
  }

  override function resetQueues() : void
  {
    _queueReceiver = null;
  }

  override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) : void
  {
    _callback = cbh
    _state = Started
    _callback.log( "***Starting JMS." )
    init()
    _callback.log( "***Started JMS." )
  }
```

```
  override function stop( isShuttingDown: boolean ) : void
  {
    _callback.log( "***Stopping JMS." )
    shutdown()
    _callback.log( "***Stopped JMS." )
    _state = Stopped
  }

  override function shutdown() : void
  {
    try {
      // Prevent further message notifications on the callback thread
      if (_queueReceiver != null) {
        _queueReceiver.setMessageListener(null);
      }
    } catch (jmse) {
      _callback.log("Got error trying to shutdown: " + jmse.toString());
    }

    super.shutdown();
  }

  override function onMessage( messageReceived : Message )
  {
    var messageBody = messageReceived.getStringProperty( BODY_PROPERTY_NAME );
    _callback.log( "***Received message with body: " + messageBody + "." )
    _callback.execute( \ -> {

      // IMPORTANT: Change this find query to find whatever data you want
    _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 to writable bundle + save result. The original entity is in a readonly bundle
        // in query results. The add method adds to your writable bundle so you can modify it.
        var e_in_new_bundle = bundle.add(e)

        // modify properties as desired on the result of bundle.add(e)
        e_in_new_bundle.Department = "Example of setting a property on a writable entity."
       }

    })

} )
  }

}
```

This example uses another base class that implements much of the JMS-specific logic, as follows:

```
package gw.api.startableplugin

uses com.guidewire.logging.LoggerCategory;

uses javax.jms.ExceptionListener;
uses javax.jms.JMSException;
uses javax.jms.MessageListener;
uses javax.jms.QueueConnection;
uses javax.jms.QueueConnectionFactory;
uses javax.jms.QueueReceiver;
uses javax.jms.QueueSender;
uses javax.jms.QueueSession;
uses javax.jms.Session;
uses javax.naming.Context;
uses javax.naming.InitialContext;
uses javax.naming.NamingException;
uses java.util.Hashtable;
uses java.lang.RuntimeException

abstract class JMSExampleBase implements ExceptionListener
{
  protected static final var JMS_URL : String = "tcp://localhost:3035/";
  protected static final var CONNECTION_FACTORY_NAME : String = "ConnectionFactory";
  protected static final var SEND_QUEUE_NAME : String = "jms/exampleQueue";
  protected static final var BODY_PROPERTY_NAME : String = "body";
  protected static final var _logger : LoggerCategory =
       new LoggerCategory(LoggerCategory.MESSAGING, "examples");
```

```
 private var _queueConnection : QueueConnection = null;
 private var _queueSession : QueueSession = null;

 protected function init() {
   // Initialize factory.
   var factory : QueueConnectionFactory;
   try {
     factory = getQueueConnectionFactory();
   } catch (e : NamingException) {
     throw new RuntimeException("Error initializing queue connection factory.
please ensure that your config properties are correct and that your JMS Server is running.", e);
   }

   // Init connection and session.
   try {
     initQueueConnectionAndSession(factory);
   } catch (e: JMSException) {
     throw new RuntimeException("Error initializing queue connection and session", e);
   }

   // Init queues.
   initQueues();
 }

 public function shutdown() {
   closeJMSConnection();
   resetQueues();
 }

 /**
  * ****** ExceptionLister ***********************************
  */
 public override function onException(jmsException : JMSException) {
   _logger.error("JMS Exception: " + jmsException.toString());
 }

 /**
  * Initialize queues.
  */
 protected abstract function initQueues();

 /**
  * Reset the queues, ie. set them to null.
  */
 protected abstract function resetQueues();

 /**
  * @param payload payload
  * @return the JMS message
  * @throws JMSException
  */
 protected function createJMSMessage(payload : String) : javax.jms.Message {
   var jmsMessage = _queueSession.createMessage();
   jmsMessage.setStringProperty(BODY_PROPERTY_NAME, payload);
   return jmsMessage;
 }

 /**
  * Create a queue receiver.
  * @param queueReceiverName the name of the queue receiver
  * @param listener the listener for the queue receiver
  * @return queue receiver
  */
 protected function createQueueReceiver(queueReceiverName : String, listener : MessageListener)
       : QueueReceiver {
   try {
     var replyQueue = _queueSession.createQueue(queueReceiverName);
     var receiver = _queueSession.createReceiver(replyQueue);
     receiver.setMessageListener(listener);
     return receiver;
   }  catch (e : JMSException) {
     // Close the JMS connection since we could not create the queue sender.
     closeJMSConnection();
     throw new RuntimeException("Failed to initialize receive queue", e);
   }
 }

 /**
  * Create a queue sender.
  * @param queueSenderName the name of the queue sender
```

```
   * @return queue sender
   * @throws InitializationException if a JMSException occurs
   */
  protected function createQueueSender(queueSenderName : String) : QueueSender {
    // Create send queue and set up a sender
    try {
      var sendQueue = _queueSession.createQueue(queueSenderName);
      return _queueSession.createSender(sendQueue);
    } catch (e : JMSException) {
      // Close the JMS connection since we could not create the queue sender.
      closeJMSConnection();
      throw new RuntimeException("Failed to initialize send queue", e);
    }
  }

  /********* Private Methods ***********************************/

  /**
   * Create Queue connection factory given the JMS url and the factory name.
   *
   * @return a queue connection factory
   */
  private function getQueueConnectionFactory() : QueueConnectionFactory {
    // Setup relationship to JMS server
    var properties = new Hashtable<String, String>();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, "org.exolab.jms.jndi.InitialContextFactory");
    properties.put(Context.PROVIDER_URL, JMS_URL);

    // Get the JNDI server
    var jndiContext = new InitialContext(properties);

    // Create a connection to the JMS Server
    return jndiContext.lookup(CONNECTION_FACTORY_NAME) as QueueConnectionFactory;
  }

  /**
   * Initialize the queue connection and session.
   */
  private function initQueueConnectionAndSession(factory : QueueConnectionFactory) {
    // Get connection and start it.
    _queueConnection = factory.createQueueConnection();
    _queueConnection.start();
    _queueConnection.setExceptionListener(this);

    // Create a non-transaction session that requires the client to explicitly
    // acknowledge the messages received.
    _queueSession = _queueConnection.createQueueSession(false, Session.CLIENT_ACKNOWLEDGE);
  }

  /**
   * Close JMS connection.  That takes care of the session and queue.
   */
  private function closeJMSConnection() {
    if (_queueConnection != null) {
      try {
        _queueConnection.close();
      } catch (jmse : JMSException) {
        throw new RuntimeException(jmse);
      } finally {
        _queueConnection = null;
      }
    }
  }
}
```

To test this JMS startable plugin, you can use the following class that generates messages on the queue:

```
package gw.api.startableplugin

uses gw.testharness.TestBase
uses gw.api.system.server.Runlevel
uses gw.testharness.RunLevel
uses gw.testharness.DoNotRunInHarness
uses gw.testharness.ProductUnderTest
uses javax.jms.QueueSender;
uses javax.jms.JMSException;
uses java.lang.RuntimeException
uses gw.api.util.DateUtil

@RunLevel(Runlevel.NONE)
class JMSClientTest extends TestBase
```

```
{
  public function testClient() {
    var client = new JMSClient();
    client.init();
    client.send("Hello world @ " + DateUtil.currentDate());
  }

  private static class JMSClient extends JMSExampleBase {
    private var _queueSender : QueueSender = null;

    public function send(message : String) {
      _logger.info("JMSMessageTransport sending message: " + message);
      try {
        // Create the JMS message.
        var jmsMessage = createJMSMessage(message);

        // Send it to the JMS Server
        _queueSender.send(jmsMessage);

      } catch (jmse : JMSException) {
        // Assume all JMS errors are unrecoverable
        throw new RuntimeException("Error while sending jms message:" + jmse.getMessage(), jmse);
      }
    }

    override protected function resetQueues() {
      _queueSender = null;
    }

    override protected function initQueues() {
      _queueSender = createQueueSender(SEND_QUEUE_NAME);
    }
  }
}
```

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 Inner Classes Implementing an Interface” on page 186 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 "Deploying Java Plugins" on page 115
- If your main startable plugin class is a Gosu class, see "If Your Gosu Plugin Needs Other Java Classes and Library Files" on page 114

Persistence and Startable Plugins

Your startable plugin can manipulate Guidewire entities. If your startable plugin needs to maintain state for itself, do one of the following:

- create your own custom entities to track your internal state information
- use the system parameter table for persistence

Do Not Call Local SOAP APIs From Plugins

In general, 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 the `soap.local.*` Gosu APIs, Java code that uses the SOAP generated libraries, and any Studio-registered web services that call the same server as the client.

Creating Unique Numbers in a Sequence

Typical ClaimCenter implementations need a way to reliably create unique numbers in a sequence for records. 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 the following parameters:

- A `String` with up to 256 characters that uniquely identifies the sequence. This is the sequence key (*`sequenceKey`*).
- A value that be the sequence number **only** if a number for this sequence has never before been requested. This is the minimum value (*`minValue`*).

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(1, "seqID")
```

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.

In Java, use code like the following:

```
nextNum = com.guidewire.pl.plugin.util.SequenceUtil.next(1, "seqID")
```

Java Class Loading, Delegation, and Package Naming

IMPORTANT Read the topic “Java and Gosu” on page 101 in the *Gosu Reference Guide* for important information about writing Java code within ClaimCenter. In particular, that topic contains important information about working with entities and working with containers (arrays, lists, and maps) from Java code in ClaimCenter.

Java Class Loading Rules

IMPORTANT Read the topic “Java and Gosu” on page 101 in the *Gosu Reference Guide* for important information about writing Java code within ClaimCenter. In particular, that topic contains important information about working with entities and working with containers (arrays, lists, and maps) from Java code in ClaimCenter.

Loading Your Java Classes (Local Loading)

IMPORTANT Read the topic “Java and Gosu” on page 101 in the *Gosu Reference Guide* for important information about writing Java code within ClaimCenter. In particular, that topic contains important information about working with entities and working with containers (arrays, lists, and maps) from Java code in ClaimCenter.

Delegate Loading

IMPORTANT Read the topic “Java and Gosu” on page 101 in the *Gosu Reference Guide* for important information about writing Java code within ClaimCenter. In particular, that topic contains important information about working with entities and working with containers (arrays, lists, and maps) from Java code in ClaimCenter.

Java Class Repository Listing

IMPORTANT Read the topic “Java and Gosu” on page 101 in the *Gosu Reference Guide* for important information about writing Java code within ClaimCenter. In particular, that topic contains important information about working with entities and working with containers (arrays, lists, and maps) from Java code in ClaimCenter.

chapter 8

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 new check in ClaimCenter, your messaging code notifies a corporate financials system or notify 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.

IMPORTANT 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 101. For the complete list of all ClaimCenter plugins, see “Summary of All ClaimCenter Plugins” on page 106.

For financials-specific events and messaging, read **this** topic for messaging concepts and implementation details. Next, refer separately to “Financials Integration”, on page 195 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 a plugin name. Use that plugin *name* as you configure the messaging destination in the Messaging editor in Studio.

This topic includes:

Messaging Overview

To understand this topic, it may help to get a high-level overview of terminology and an overview of events in ClaimCenter. The following table summarizes terminology regularly used in this topic:

Term	Description
event	An abstract notification of a change in ClaimCenter that might be interesting to an external system. For example, adding, changing, or removing a Guidewire entity. For example, adding a new financial transaction on a claim.
message	Information to send to an external system in response to an event. ClaimCenter can send one or more (different) messages to each destination interested in an event. In addition to an ID and some status information, each message has a payload, which is the main data content of the message.
messaging destination	An external system to which to send messages. Register your messaging destinations in Studio and specify which classes implement your messaging plugins for that destination. Register exactly one destination for each external system. Each destination registers a list of events for which it wants notifications. Each destination may listen for different events compared to other destinations.
acknowledgement	A response to a ClaimCenter message that indicates successful message processing. This could be a *positive acknowledgement* (Ack) that means that the external system processed the message successfully. It could also be a negative acknowledgement (Nack) that means the external system could not handle the message due to some error.
safe ordering	ClaimCenter always sends messages associated with claims ordered by claim for each messaging destination. These are *safe-ordered* messages. There are other types of messages, such as Catastrophe, that are *non-safe-ordered* messages. ClaimCenter waits for an acknowledgement from each destination before processing the next safe-ordered message for a claim. In contrast, ClaimCenter sends non-safe-ordered messages for each destination in send, or creation, order and does *not* wait for acknowledgements before sending the next one. For details, see “Message Ordering and Multi-Threaded Sending” on page 170.

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 and destination setup.** At application startup, ClaimCenter checks its configuration information and constructs abstract *destinations*, each of which represents an external system that can receive a

message. As part of this destination setup, 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. **System events trigger destination events.** Events trigger after data changes. For example, if you change data in the user interface, if an outside system calls a web service that changes data, or a time-delayed event changes data. The system event represents the changes to application data as the entity commits to the database.
3. **Event Fired rule set runs and your rules create messages.** ClaimCenter runs the Event Fired rule set for each event name that triggers (and runs multiple times for an event anme if multiple destinations listen for it). Your rules can choose to generate new messages. Messages have a text-based *message payload*. In the earlier 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 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 the earlier example, the checking printing destination might take an XML payload and submit it to a JMS queue to notify an external system about the check.
5. **ClaimCenter waits for an acknowledgement.** The check printing system replies with an acknowledgement to the destination after it processes the message. In turn, the destination informs ClaimCenter of the acknowledgement. In the earlier example, the check printing system might send an acknowledgement (reply) message on a reply queue once it had printed the check. In turn, the destination would notify ClaimCenter in an *acknowledgement* that the check message succeeded (or failed). The code that submits the acknowledgement might also make changes to application data, such as updating a property on a `Claim` entity.

ClaimCenter makes no assumptions about the final message format or system. Instead ClaimCenter relies on the messaging plugins that represent the destination to respond with an appropriate *acknowledgement*. The acknowledgment indicates that the destination system handled the message. A *negative acknowledgement* indicates failure on the receiving system's end of the communication.

The bulk of integration code using events and messaging is designing Event Fired rules to create message payloads and writing a `MessageTransport` plugin to send it. There are two other types of messaging plugins that are optional, which this topic discusses later.

Once you have written your code that implements your messaging plugin, register your new messaging plugins in Studio. As you registering new plugins in Studio, Studio prompts you for a plugin interface name (in a picker) and in some cases a plugin name. Use that plugin name to configure the messaging destination in the Messaging editor to register each destination.

For more information about registering a plugin, see "Using the Messaging Editor" on page 161 in the *Configuration Guide*. For more information about plugins in general, see "Plugin Overview", on page 101.

For more information about the Studio Messaging Editor, see "Using the Messaging Editor" on page 161 in the *Configuration Guide*. Also see "Message Destination Overview" on page 148 later in this topic.

ClaimCenter does not assume any specific type of transport. Destinations can deliver the message any way they want, including but not limited to the following:

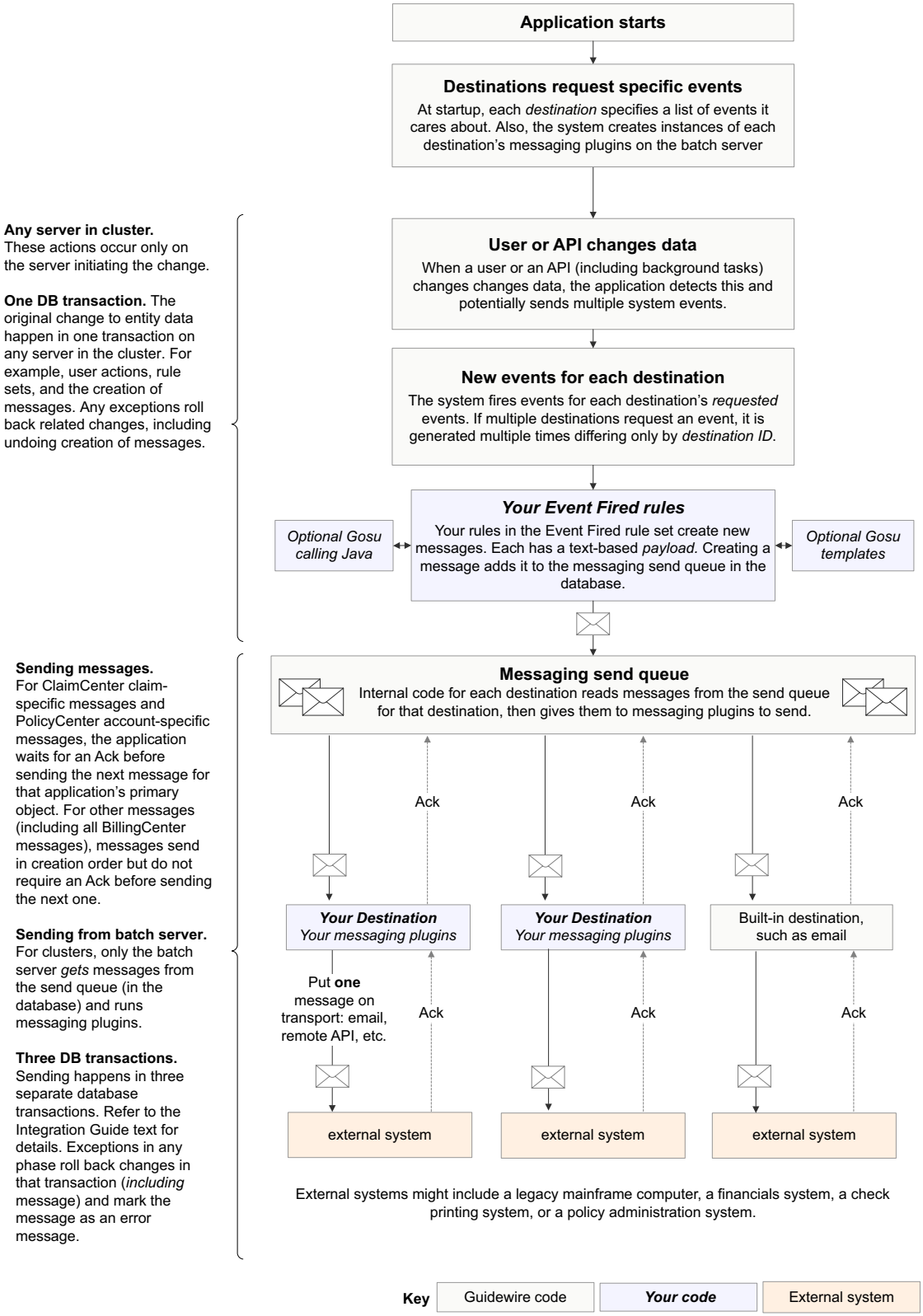
- **Submit the message to a guaranteed-delivery messaging system.** For instance, send to a Java Messaging Service (JMS) queue implemented with IBM WebSphere MQ middleware.
- **Submit the message using remote API calls.** Use a web service (SOAP) interface or a Java-specific interface to send a message to an external system.
- **Save to special files in the file system.** Message "sending" could be implemented by writing data to local text files that are read by nightly batch processes.

- **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 convenient for simple notifications but might not be appropriate for systems that rely on guaranteed delivery and message order.

Event and Messaging Flow

The following diagram illustrates the chronological flow 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 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 call) changes something.** An action in the application user interface, an API call, a batch process, and even some background processes can change 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.

   Note: It is critical to understand that the change does not fully commit to the database until step 5. Any exceptions that occur before step 5 undoes 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 154.

3. **ClaimCenter generates (and duplicates) destination 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 another destination. To change this list, review the Messaging editor in Studio for each destination.

4. **ClaimCenter invokes business rule sets for each event to generate messages.** ClaimCenter calls the Event Fired rule set for each destination/event pair. Event-handling rule sets rules check the event name and the messaging destination 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 165. Your rules can use the following techniques:

   - **Optionally export an entity to XML using Guidewire-standardized XSD format.** 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 168.

IMPORTANT The Guidewire XML (GX) modeler is a powerful tool for integrations. With it, you can 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. For import or export, GX models allow you to work with XML naturally in Gosu as objects and properties, rather than as text data. See “Creating XML Payloads Using Guidewire XML (GX) Models” on page 168.

- **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 "Java and Gosu" on page 101 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 Fields" on page 174 for details.

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See "Entity Data Restrictions in Messaging Rules and Messaging Plugins" on page 147.

5. **New messages add to the send queue.** After all rules finish running, 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. In other words, 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 message writer's assessment of safe ordering of messages, discussed in step 6.

6. **On the batch server, each messaging destination thread pulls messages from the send queue.**

   Each destination reads messages from the send queue and dispatches them in the appropriate order to the messaging plugins to send. Messages that are claim-related always send in *safe order*. Safe ordering guarantees there is never more than one message sent at one time for each claim/destination pair.

   IMPORTANT For details of how ClaimCenter retrieves messages and orders them for sending, see "Message Ordering and Multi-Threaded Sending" on page 170.

   If the messaging destination sends a message, it calls the `send` method of the destination's message transport plugin. The message transport plugin is responsible for sending the message in whatever native transport is appropriate. For example, it might submit a message to a queue, call a SOAP API, send SQL to a database, or write a file for later batch processing. ClaimCenter makes no assumptions about the native messaging format. The `send` method can throw exceptions to indicate that a retryable error occurred (try again later) or a non-retryable error occurred (stop messages to this destination).

   WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See "Entity Data Restrictions in Messaging Rules and Messaging Plugins" on page 147.

7. **Acknowledging messages.** Some destination implementations know success or failure immediately during sending. For example, a messaging transport plugin might calls 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 a JMS messaging queue that confirmed 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, submit an error, also called a negative acknowledgement or *Nack*. You can submit an Ack or Nack in several places. For synchronous sending, submit it in your `MesssageTransport` 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 195.

8. **A message acknowledgement for safe-ordered messages permit sending the next related message.**

   If ClaimCenter gets an acknowledgement for a safe-ordered (claim-related) message, it potentially changes what messages are now sendable. If there are other messages for that destination in the send queue for the same claim, ClaimCenter soon sends the **next** message for that claim.

IMPORTANT For details of how messages are retrieved and ordered, see “Message Ordering and Multi-Threaded Sending” on page 170.

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. Entity changes in these code locations do not rerun validation or pre-update rules, so you must not make changes that might require those rules to run again.

Also, entity changes in messaging plugins do not trigger concurrent data exceptions except in some special cases. See “Messaging Database Transactions During Sending” on page 154. To avoid data integrity issues with concurrency, you must avoid changing data in these code locations.

Specific rules for Event Fired rules and messaging plugin implementations:

- For properties, only set messaging-specific properties and properties that users can never modify from the user interface, even indirectly.
- It is dangerous and unsupported to add new or delete objects in these code locations, even indirectly through other APIs. The only exception is that you create new `Message` objects from Event Fired rules.
- Do not call business logic APIs that might change entity data, since indirectly they could change, add, or remove entity instances in violation of the earlier rules.

WARNING For customer data integrity and server reliability, you must follow these restrictions regarding what data you can change in Event Fired rules and messaging plugin implementations.

Messaging Database Transactions During Message Creation

All steps up to and including adding messages to the send queue (in the list earlier in this section, step 2 through step 5) occur in one database transaction. This is the same database transaction that triggered the event. The database transaction rolls back if any of the following occur:

- exceptions in rule sets before message creation
- exceptions after message creation but before commit the bundle to the database
- errors committing the bundle to the database (this bundle includes new messages)

If any of these errors occur, ClaimCenter rolls back **all** messages added to the send queue in that transaction.

Additionally, there are special rules about database transactions during message sending at the destination level. See “Messaging Database Transactions During Sending” on page 154.

Important Notes About ClaimCenter Clusters

If you run ClaimCenter in a cluster, be aware that step 1 through step 5 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 6 through step 8), 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.

IMPORTANT For ClaimCenter clusters, only the batch server actually uses your messaging plugins.

Do Not Call SOAP APIs on the Same Server From Messaging Plugins

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 In general, 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 can receive a message, you must define an abstract *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.

Note: Choose carefully how you 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**. See “Message Ordering and Multi-Threaded Sending” on page 170.

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 178.

- **Message transport.** (Required) The main task of a destination is to send messages to an external system. Its underlying protocol might be a JMS 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 179.
- **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 180.

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 161 in the *Configuration Guide*.

Note: For more information about plugins, see “Plugin Overview”, on page 101.

To create new destinations, configure the server’s 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.
- The amount of time in milliseconds (`initialretryinterval`) after a retryable error to retry a sending a message.
- The number of retries (`maxretries`) to attempt before the retryable error becomes *non-retryable.*
- 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.
- 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)*"`.
- A 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.

IMPORTANT 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.

- 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 reminder of the poll interval. If the current round of sending takes longer than the poll interval, than the thread does not sleep at all and continues to the next round of querying and sending. See "Message Ordering and Multi-Threaded Sending" on page 170 for details on how the polling interval works.

  If your performance issues primarily relate to many messages per claim per destination, then the polling interval is the most important messaging performance setting.
- Polling interval. To send messages associated with a claim (*safe-ordered* messages), ClaimCenter can create multiple sender threads for each messaging destination to distribute the workload. These are threads that actually call the messaging plugins to send the messages. Use this field to configure the number of sender threads for safe-ordered messages. This setting is ignored for non-claim-specific messages, since those are always handled by one thread for each destination. If your performance issues primarily relate to many messages but **few messages 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 170.
- 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 148.

Manage these settings in Guidewire Studio in the Messaging editor. See "Using the Messaging Editor" on page 161 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 Editor" on page 141 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see "Using the Messaging Editor" on page 161 in the *Configuration Guide*. Use the plugin name to configure the messaging destination in the Messaging editor in Studio to register each destination.

To write your messaging plugins, see "Implementing Messaging Plugins" on page 177.

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See "Entity Data Restrictions in Messaging Rules and Messaging Plugins" on page 147.

Sharing Plugin Classes Across Multiple Destinations

It is common to implement messaging plugin classes that multiple destinations share. For example, a JMS message queue transport plugin might manage the transport layer for multiple destinations that use JMS as its 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 thread safe, 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 184.

Handling Acknowledgements

Due to differences in external systems and transports, there are two basic approaches for handling replies. ClaimCenter supports both types of acknowledgements, although in different ways:

1. **Synchronous acknowledgement (at the transport level).** 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 an acknowledgement or error. In this case, acknowledgement is straightforward using code like `message.reportAck(...)` and `message.reportError(...)`.
2. **Asynchronous acknowledgement.** Some transports might finish an initial process such as submitting a message on a JMS 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 JMS 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.

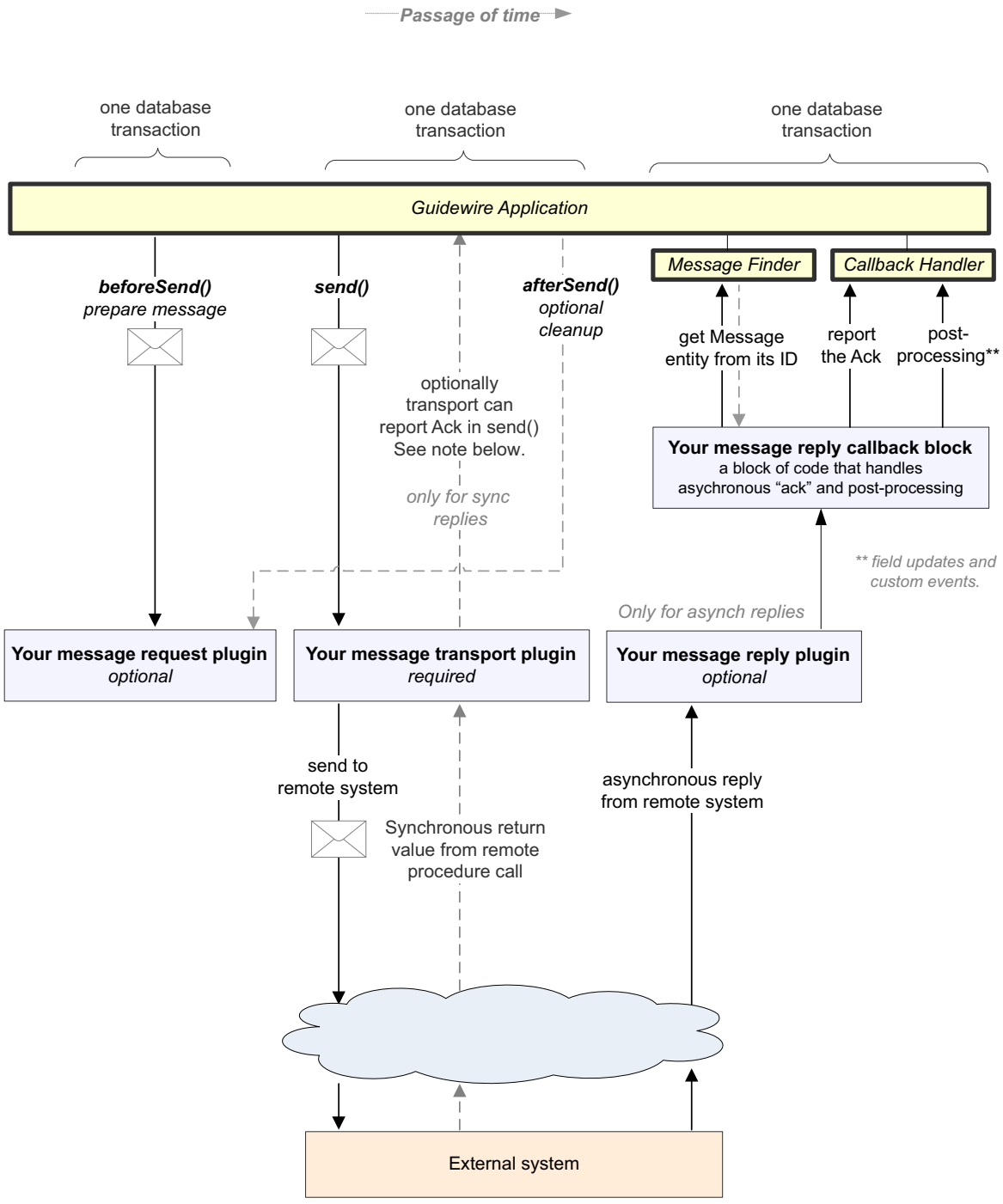
For more information about these objects and how to implement them, see "Implementing a Message Reply Plugin" on page 180.

An alternative to this approach is for the external system to call a ClaimCenter web service (SOAP) API to acknowledge the message. For example, the following Java example acknowledges the message:

```
api = (IMessagingToolsAPI) APILocator.getAuthenticatedProxy(IMessagingToolsAPI.class, url, "su", "gw");
Acknowledgement ack;

ack = new Acknowledgement();
ack.setMessageID("ABC-1234");
ack.setError(true);
ack.setRetryable(false);

api.acknowledgeMessage(ack);
```

The following diagram illustrates the destination-related plugins, associated components, along with the chronological flow of actions between elements in the system.

Messaging Plugin Flow



Guidewire code

Your Code

External 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.

No Message Ack, Error, or Skip Methods in Rule Sets

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 methods only within messaging plugins. This prohibition also applies to Event Fired rule set execution.

Messaging Database Transactions During Sending

As mentioned in "Messaging Database Transactions During Message Creation" on page 147, 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.

At the start of each transaction, ClaimCenter locks the message and the primary object of the messages (if a primary object exists). This ensures that the application does not try to acknowledge a skipped message or similar conditions.

IMPORTANT See earlier in this section for a diagram of the timing of messaging database transactions.

If the message is associated with a claim, during all of these operations you can request that ClaimCenter optionally lock the primary entity (the `Claim`) at the database level. This can reduce some problems in edge cases in which other threads try to modify objects associated with this claim.

ClaimCenter checks the `config.xml` parameter `LockPrimaryEntityDuringMessageHandling`. If it is set to `true`, ClaimCenter locks the primary entity during the following operations:

- during send
- during message reply handling
- while marking a message as skipped

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 this email API, see the *ClaimCenter Rules Guide*. ClaimCenter always creates the built-in email destination, independent of your configuration settings. To configure this built-in destination, see *ClaimCenter Configuration Guide*.

Additionally, ClaimCenter includes a built-in destination to communication 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 339.

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 389.

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 mesage 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 *`Entityname`*`Added` events and *`Entityname`*`Changed` 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.

IMPORTANT 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 the claim or exposure is ready for ISO.

List of Messaging Events in ClaimCenter

ClaimCenter generates events on an object basis. The key is ensuring that each event provides enough context about the object to be useful. For some objects (for example, claim, exposure, or note), this is easy – the object is self-contained, and clearly related to a specific claim.

However, there are subobjects in the ClaimCenter data model that are not easy to interpret. For example, the `Address` subobject is common and changing it is common. However, it is not useful to know a particular address changed unless you know its role in the claim. Without this additional context, you cannot do something useful with the event.

For example, was it a claim's loss location? Was it the claimant's temporary address?

In ClaimCenter, top-level objects include claims and exposures. Note that every exposure belongs to a claim, which is the primary object in ClaimCenter for safe-ordering. Most other subobjects in the claim have a `Claim` property that points to the claim.

The application reports adds, removals, and changes for most top level objects, and many of their subobjects too. The destination can typically do something useful with the notification if you figure out the context of the top-level object the subobject is associated with.

In ClaimCenter, most top-level objects include enough information to identify the claim. However, the following top-level objects are not claim-specific: contacts, users, and groups.

ClaimCenter triggers events for most top-level objects if an entity is added, removed (or retired), or if any property on a top-level object changes. For example, selecting a different claim type on any claim generates a *changed* event on the claim.

ClaimCenter reports a change to the top-level object if a foreign key reference to a subobject changes but **not** if data **on the subobject** itself changes. (There is an exception in special cases, described later.)

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.

You can view a full list of ClaimCenter events in the SOAP API Javadoc as enumerations within the `GW_Events` class. For instance, the following constant represents an event that one claim changed: `com.guidewire.cc.webservices.enumeration.GW_Events.ClaimChanged`.

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`	`ActivityAdded`
	`ActivityChanged`
	`ActivityRemoved`
		Standard events for the `Activity` entity. `ActivityChanged` indicates that an activity changed, including marking the activity completed or skipped.

Assignable Entities:
• `Claim`
• `Exposure`
• `Activity`
• `Matter`
	`AssignmentAdded`
	`AssignmentChanged`
	`AssignmentRemoved`
		Assignment events for assignable entities. The assignabled entity is the root entity for the event. If an assignment added to this entity, `AssignmentAdded` triggers. If the entity previously had an assignment and now has no assignment, `AssignmentRemoved` triggers. If assignment data such as assigned user, group, date changed, `AssignmentChanged` 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 *valiation level*, 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 *in error* 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 “Resyncing Messages” on page 185.
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. CoverageAddded, 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: • If it is important to send claims and exposures to external systems only after reaching a specific *valiation level*, Event Fired rules decide whether to ignore or log that event. • 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. • The ExposureChanged event triggers if a claim is closes or reopens. • If your implementation does not allow users to remove exposures once added, the ExposureRemoved event may never trigger.
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.

Entity	Events	Description
MetroReport	MetroReportAdded MetroReportChanged MetroReportRemoved	Standard events for MetroReport entities. For more information about the optional Metropolitan service, see “Metropolitan Reporting Bureau Integration”, on page 389.
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: • 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. • The PolicyChanged event triggers after a policy or any subobject updates. This includes property, vehicle, stat code, but not coverages. • ClaimCenter sends these events independent of the claim valiation level of the associated claim. Filter (ignore) events as needed • Changes to a property on a Coverage entity triggers the PolicyChanged event in addition to the CoverageChanged event.
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.
General-purpose events		
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 processHistory.CompletionTime property for the date-stamp in a datetime object. See “Batch Processes and Work Queues” on page 129 in the *System Administration Guide*.
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.
Administration events		

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.
Financial transaction events		
BulkInvoice	BulkInvoiceAdded BulkInvoiceChanged BulkInvoiceRemoved	Standard events for the BulkInvoice entity.
	BulkInvoiceStatusChanged	The bulkinvoice.status property changed. See "Financials Integration", on page 195 and "Claim Financials Web Services" on page 200 for details.
Check	CheckAdded CheckChanged CheckRemoved	Standard events for the Check entity.
	CheckStatusChanged	The check.status property changed. See "Financials Integration", on page 195 and "Check Integration" on page 203 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 195 and "Reserve Transaction Integration" on page 220 for details.
AuthorityLimitProfile	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 195 and "Reserve Transaction Integration" on page 220 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.

Entity	Events	Description
	RecoveryReserveStatusChanged	The recoveryreserve.status property changed. See “Financials Integration”, on page 195 and “Recovery Reserve Transaction Integration” on page 217 for details.
Recovery	RecoveryAdded RecoveryChanged RecoveryRemoved	Standard events for recovery transactions.
	RecoveryStatusChanged	The check.status property changed. See “Financials Integration”, on page 195 and “Recovery Transaction Integration” on page 218 for details.
Transaction	TransactionAdded TransactionChanged TransactionRemoved	Standard events for Transaction, which is the supertype of other transactions.
Contacts and address book events		
ABContact	ABContactAdded ABContactChanged ABContactRemoved	Standard events for ABContact entities. (ContactCenter 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.
CompanyVendor	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.

Entity	Events	Description
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 *EntityName*Removed 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* that 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 can be retired. Refer to the *Data Dictionary* for details.

Triggering Custom Events

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. For more information about the entity libraries, see “Java Entity Libraries Overview” on page 114 in the *Gosu Reference Guide*.

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 ContactCenter, this also affects ContactCenter's entity validation for events such as the `ResyncABContact` 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, alternate 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 ContactCenter) 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 to 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 an 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.

IMPORTANT The web services interface `IImportToolsAPI` and the corresponding `import_tools` command line tool do not generate events after loading data. 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. 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 “Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 147.

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 168.

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 `EventFired` 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, you can add it as a Gosu Library within Guidewire Studio. Your messaging code can call the shared logic from each rule that needs it. For more information about developing custom libraries in Guidewire Studio, see the *ClaimCenter Rules Guide*.

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)
```

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 167.

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 Fields” on page 174.

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 Entity Data Changed in a Bundle” on page 285 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)
}
```

IMPORTANT For a complete list of Gosu methods related to finding what changed, see “Determining What Entity Data Changed in a Bundle” on page 285 in the *Gosu Reference Guide*.

No Message Ack, Error, or Skip Methods in Rule Sets

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 methods only within messaging plugins. This prohibition also applies to Event Fired rule set execution.

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 account

// 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 with a Gosu Template

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 %>
```

For much more information about using templates, see “Gosu Templates” on page 291.

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 Gosu templates, see “Gosu Templates” on page 291.

Note: For information on the separate legacy Gosu template format (the `.gs` file format), see “Gosu Templates” on page 54

Setting a Message’s Root Object

From the Gosu environment, the `messageContext.Root` property specifies the *root object* (primary object) for an event. Typically this is also the root object for the message generated for the event, and as such that is the default for this property.

However, it does not have to be the case. For example, suppose Event Fired rules for a `ClaimValid` event creates messages for the claim and a message for each note on the claim. The root object for each note message might be the note, not the claim.

This distinction is not critical for most implementations. However, ClaimCenter treats claim-related messages differently from cross-claim messages, which are not sent in a strict order. This is called *safe-ordering*. For details about how message ordering happens, see “Message Ordering and Multi-Threaded Sending” on page 170.

To ensure a message is safe-ordered with the claim, set the message’s root object to a claim-related object. Typically, this means a claim or an entity that has a `Claim` property that points to the claim.

Set the message’s `Root` property as follows in your Event Fired rules:

```
myMessage.MessageRoot = myClaim
```

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.**

IMPORTANT For instructions on using the GX modeler, see "The Guidewire XML (GX) Modeler" on page 262 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 "Java and Gosu" on page 101 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 where 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 176.

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 ClaimCenter version 6.0.8, messages can contain up to one billion characters.

Message Ordering and Multi-Threaded Sending

This section explains how ClaimCenter manages the order in which messages are sent and in which cases ClaimCenter sends messages in multiple threads.

Safe-ordered Messaging

Safe ordering is a messaging feature in ClaimCenter and PolicyCenter:

- In ClaimCenter, messages associated with a claim always send ordered by claim for each messaging destination. These are *safe-ordered* messages. Messages associated with entities not tied to a specific claim, such as messages relating to a `Catastrophe` entity, are called *non-safe-ordered* messages.
- In PolicyCenter, messages associated with a account always send ordered by account for each messaging destination. These are *safe-ordered* messages. Messages associated with entities not tied to a specific account are called *non-safe-ordered* messages.

The application waits for an acknowledgement before processing the next safe-ordered message for that primary object (claim for ClaimCenter, account for PolicyCenter) 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 claim message before receiving any messages for adding notes or other additional objects to that claim. If the external system rejects the first new claim message with an error, you must not send further messages at all.

Even if the transport layer guarantees delivery order, it is unsafe for ClaimCenter to send the second message before confirming the first message succeeded. Doing otherwise could cause very difficult error recovery problems.

This feature 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 use Guidewire ContactCenter, ContactCenter also supports safe ordering of messages associated with each `ABContact` entity. This means that ContactCenter only allows one message per `ABContact`/destination pair in flight at any given time. For more ContactCenter integration information, see “Address Book Integration” on page 299.

BillingCenter does not need and thus does not use safe-ordered messaging. Effectively all BillingCenter messages are non-safe-ordered in this special sense of the phrase *safe-ordered*. BillingCenter messages have separate threads for each destination and each sends messages in send order (the `Message.SendOrder` property). You can think of send order as the order of message creation.

Non-safe-ordered Messages

Some messages link to objects you can use across multiple claims. These cross-claim messages are *non-safe-ordered messages*.

Non-safe-ordered messages send as soon as possible in the order that Event Fired rules generated them. ClaimCenter does not wait for acknowledgements from prior messages before sending the next one.

IMPORTANT For non-safe-ordered messages, ClaimCenter sends them in order for each destination, but does **not** wait for acknowledgements before sending the next one.

For example, ClaimCenter sends a new catastrophe code message immediately. ClaimCenter never waits for acknowledgements for other messages before sending this message.

For ClaimCenter, messages for the following events are non-safe-ordered messages:

- `Group` events
- `User` events
- `Catastrophe` events
- Contact-related events for `Contact` entities, but **not** `ClaimContact` or `ClaimContactRole` which represent a change in the contact's *relationship* to the claim.

For ContactCenter, messages for the following events are non-safe-ordered messages:

- `Group` events
- `User` events

If Multiple Events Fire, Which Message Sends First?

Events trigger based on the order in each destination's configuration. In general, messages send in the order of message creation in Event Fired rules. Rules create messages for an event prior to any messages for the next listed event. This means that for any destination, the order of messages for a single event primarily is the event name order for the destination in the Messaging editor in Studio. For messages with the same event name and destination, the message order is simply the order that your Event Fired rules create the messages.

In typical deployments, the event name order in the destination setup is very important. Carefully choose the order of the event names in the destination setup. Remember that changes to the ordering of the event names change the order of the events in Event Fired rules. Such changes can produce a radical effect in the behavior of Event Fired rules if they assume a certain event order.

However, because ClaimCenter supports safe-ordering of messages related to a claim, the actual ordering algorithm is more complex. Refer to "Message Ordering and Multi-Threaded Sending" on page 170 for details.

For more information about destination setup, see "Message Destination Overview" on page 148.

WARNING The event name order in the destination setup is important. 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. Order changes could change message order that force major changes in required messaging logic.

To implement messaging destinations, see "Implementing Messaging Plugins" on page 177. Manage this information in Guidewire Studio in the Messaging editor. See "Using the Messaging Editor" on page 161 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 Editor" on page 141 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see "Using the Messaging Editor" on page 161 in the *Configuration Guide*.

Message Ordering and Multi-Threading 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 during each round of sending. This is called the *chunk size*. Configure the chunk size in the messaging destination configuration editor with the **Chunk Size** field. By default, this value is set to 100,000, which typically includes all messages currently in the send queue that can be sent. You can also change the polling interval in the messaging destination configuration editor with 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:

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 claim-specific messages, 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 for a specific claim part of this list. If a message for that claim is sent but unacknowledged, it does not appear in this list yet. This enforces the rule that no more than one message can be in-flight for each claim per destination.

      The query ensures that no two messages are in flight (sent but not yet acknowledged) for the same destination for the same claim. So, if there are 100 messages for one claim, the query only reads and dispatches **one** of those messages (out of a possible 100) to the destination subthreads.

   b. Next, ClaimCenter queries for non-claim-specific messages, 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.

WARNING 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. Thus, the downstream system probably needs ClaimCenter to send the catastrophe message before the claim message that references the catastrophe.

3. For each destination, the worker thread iterates through all non-safe-ordered messages for that destination, sending one at time in a single thread to the messaging plugins.

   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.

IMPORTANT Assigning the number of sender subthreads for a destination affects only the safe-ordered messages for that destination. All non-safe-ordered messages always send in a single thread for each worker, which ensures that the `SendOrder` property dictates the send order for non-safe-ordered messages.

If the message is associated with a claim, during messaging operations you can optionally lock the primary entity (the `Claim`) 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 claim.

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.

IMPORTANT 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.

First, suppose there are two messaging destinations and 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.

IMPORTANT If your performance issues primarily relate to **many messages per claim** 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.

IMPORTANT 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.

Thread-Safe Messaging Plugins

You must write all your messaging plugin code *as thread-safe*. This means that you must be extremely careful about static variables and any other shared memory structures. For more information, see “Concurrency” on page 311 in the *Gosu Reference Guide*.

This is true even if the **Number Sender Threads** setting is set to 1.

How Sending Errors Affect Ordering

If ClaimCenter receives a *negative acknowledgement* (*Nack*) for a message, ClaimCenter marks the message entity with an error flag and saves the error message.

If the message with the error is a safe-ordered message, ClaimCenter blocks sending any more messages for that claim/destination pair until the problem is resolved or the claim is resynced.

Errors for *non-safe-ordered* (cross-claim) messages, for example new/changed catastrophes, do **not** hold up other messages. However, errors in non-safe-ordered messages may cause errors at the destination.

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.

Late Binding Fields

In most cases, capture information for the message payload at the time the Event Fired rule set runs. For example, for an `ClaimChanged` event, typical messages contain the new information for the account as of the time the event triggers and the Event Fired rule set runs. This often a good thing. 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 allowing *late binding* of properties in the message payload. You can designate certain properties for late binding so they calculate values immediately before the messaging transport sends the message.

As a general rule, Guidewire recommends exporting properties regularly (early bound) unless you have a specific reason why late binding is critical for that message and destination.

For new entities, ClaimCenter entity's *public ID property* is typically sent late bound. In typical installations, 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 within the message. Your `MessageRequest` plugin code or `MessageTransport` plugin code can directly find the message root object such as a `Claim` 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 `MessageRequest` might do this like the following in its `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`:

```
function beforeSend(m : Message) {
  var c = m.MessageRoot as Claim
  var s = org.apache.commons.lang.StringUtils.replace(m.getPayload(), "<AAAA>",c.SomeProperty)
  return s
}
```

This example assumes that the message contains the string `"<AAAA>"` as a special marker in the message text.

Message Sending Errors

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:

- **Retryable errors during send().** Sometimes a message transport plugin has a send error and expects it to be temporary. If the message transport plugin throws a retryable exception from its `send` method, ClaimCenter retries after a delay time. The delay time is an exponential wait time (*backoff time*) up to a wait limit specified by each destination. ClaimCenter halts sending messages all messages for that destination pair during that retry delay. After the delay reaches the wait limit, the retryable error becomes an unrecoverable error. Refer to the following discussion for unrecoverable errors.
- **Unrecoverable errors during send().** This is the status of a message if the destination has an error that the application does not expect to be temporary. Unrecoverable errors include:
  - the `send` method throwing an exception.
  - a retryable error that has reached its retry limit
  - other unexpected exceptions during the `send` method.

  The destination suspends sending for all messages for that destination until one of the following is true:
  - an administrator retries the sending the manually and it succeeds this time
  - an administrator removes the message
  - it is a safe-ordered ClaimCenter message (claim-related) and an administrator resyncs the claim.
  - In ContactCenter, if it is a safe-ordered ContactCenter contact message and an administrator resyncs the contact.

Errors conditions that occur later:

- **A negative acknowledgement (Nack).** A destination might get an error reported from the external system (database error, file system error, and delivery failure) and special intervention might be required to fix it.

  For safe-ordered messages, ClaimCenter stops sending messages for this claim/destination pair until the error clears through the administration console or automated tools.

  For safe-ordered messages, ClaimCenter stops sending messages for this account/destination pair 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 claim/destination pair until it receives an Ack or Nack.

For ClaimCenter administrative tools that monitor and recover from messaging errors, see "The Administration User Interface" on page 190.

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.

Submitting Acks and Errors and Duplicates from Your Messaging Plugin

To submit an acknowledgement or a negative acknowledgement from a Java messaging plugin, use the `Message` methods based on the success or failure:

- If successful, call `reportAck()`.
- If message had a retryable error, call `reportError()`. There is an optional method signature that takes a `Date`. Use this alternate signature to automatically retry a message at a later time. This is equivalent to using the application user interface to in the Administration tab at that specified time, and select the message and click Retry.
- If the message had a non-retryable error, call `reportNonRetryableError()`.
- If the message was a duplicate, call `reportDuplicate()` on the message history entity. The `MessageFinder` interface has methods to allow the reply plugin to find message history entities given the original message ID or the sender reference ID.

  If your duplicate detection code runs in the `MessageTransport` plugin, you can use standard database query code to find the original message. From within a `MessageReply` plugin, use the `MessageFinder` object to find messages during asynchronous message reply handling. The `MessageFinder` methods are as follows:

  - `findHistoryByOriginalMessageID(int originalMessageId)` - find message history entity by original message ID
  - `findHistoryBySenderRefID(String senderRefID, int destinationID)` - find message history entity by sender reference ID

  After you find the original message in the message history table, report the duplicate message by calling `reportDuplicate` on the original message.

If you use asynchronous message replies using the `MessageReply` plugin, you must take additional steps. You must get a reference to the associated `Message` entity from your plugin so that you call these `Message` methods. For more information about this topic, see "Implementing a Message Reply Plugin" on page 180.

Submitting Acks and Errors Directly from External Systems

If you want to acknowledge the message directly from the external system, use the SOAP API `IMessagingTools.acknowledgeMessage(ack)`. First, you create an `Acknolwedgement` SOAP entity. If you detect problems with the message, set the following properties:

- **Error.** Set the `Acknowledgement.error` property to `true` if there was an error.
- **Duplicate.** If you detect the message is a duplicate, set the `Acknowledgement.duplicate` property to `true`. Also set `Acknowledgement.error` to `true`.

- **Retryable.** If the error is retryable, set the `Acknowledgement.retryable` property to `true` and `Acknowledgement.error` to `true`. A message could be retryable if the error at the destination system is temporary. For instance, if the record to be updated is locked, it is a temporary error. If the message has a retry error, then you can request a retry through the administrative user interface.

Reporting Errors and Retrying Messages from SOAP APIs

The web service interface `IMessagingToolsAPI` contains methods to retry messages with retryable errors.

See the API Reference Javadoc documentation for the `IMessagingToolsAPI` methods `retryMessage` and `retryRetryableErrorMessages`, the latter of which optionally limits retry attempts to a specified destination. You can only use the `IMessagingToolsAPI` interface if the server's run mode is set to `multiuser`. Otherwise, all these methods throw an exception.

If the acknowledgement is positive (an *Ack*), then ClaimCenter marks the message as successfully sent. If the message was associated with a claim, ClaimCenter sends the **next** message for that claim/destination pair, if any such message exists in the send queue.

As part of an *Ack*, the destination can ask ClaimCenter to update properties on the objects that generated the message. For example, this could be used to set the public ID of the exposure after the `ExposureValid` event processes.

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 the current exposure with the name "`expo1`". In the acknowledgement, the destination would indicate that the `publicID` of object `expo1` must be updated to "`123-45-4756:01`" to indicate the first exposure on claim # "`123-45-4756`". 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. Later sections explain more about how to save the name during message creation and how to use it to construct an acknowledgement.

Implementing Messaging Plugins

There are three types of messaging plugins. See the following sections for details:

- "Implementing a Message Request Plugin" on page 178
- "Implementing a Message Transport Plugin" on page 179

- “Implementing a Message Reply Plugin” on page 180

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Entity Data Restrictions in Messaging Rules and Messaging Plugins” on page 147.

For all three types, your implementations of messaging plugins 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. If you do not already have this defined explicitly, you must also implement `InitializablePlugin` or the application does not initialize your messaging plugin.

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 {
```

To conform to this new interface, add 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

var myVar = map["MyKeyName"]
}
```

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.

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 178.

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 methods `reportError` and `reportNonRetryableError`. 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 148.

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 180 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 {

   // note the empty constructor. The application now longer calls the constructor
   // with a map of parameters. If you do provide an empty constructor, the application
   // calls it if making an instance of the plugin, which is before calling setParameters
   construct()
   {
   }

   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 183.

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 to notify 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 classes and interfaces called `MessageFinder`, `PluginCallbackHandler`, and the inner interface called `PluginCallbackHandler.Block`.

A message finder (`MessageFinder`) object is built-in object that returns a `Message` object from its `messageID` or `senderRefID`, using its `findById` or `findBySendRefId` method, respectively. During the message reply plugin initialization phase, ClaimCenter calls the message reply plugin’s `initTools` method with an instance of `MessageFinder`. Save this instance of `MessageFinder` in a private variable within the plugin instance.

A plugin callback handler (`PluginCallbackHandler`) is an object provided to a 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.

Your implementation of this plugins 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.

For important information about using message finders to submit errors and duplicates, see “Error Handling in Messaging Plugins” on page 183.

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.

There are two ways of doing this:

- **Using MessageReply.** If you use the simple `MessageReply` interface, you must implement the `initTools` method which gets these objects as parameters. The plugin can store these values in private properties such as `_pluginCallbackHandler` and `_messageFinder`.
- **Using MessageReplyBase.** Alternatively, you can base your implementation on `MessageReplyBase`, which implements the `MessageReply` interface. `MessageReplyBase` automatically implements the `initTools` method and stores those values in private variables `_pluginCallbackHandler` and `_messageFinder`.

Message Reply Callbacks

To acknowledge the message asynchronously, the message reply plugin uses its reference to the `PluginCallbackHandler`. The plugin passes 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 also has an `add` method that can mark a ClaimCenter entity as potentially modified so it can commit to the database with the Ack. Call the `add` method with an entity to ensure that changes to that entity commit to the database in the correct database transaction. This adds the entity to the correct *bundle*. For more about bundles, see “Bundles and Transactions” on page 275 in the *Gosu Reference Guide*.

Your code in `PluginCallbackHandler.Block` must perform the following steps:

1. Get a `Message` object from methods on the plugin’s `MessageFinder` instance, described earlier in this section.
2. Call methods on the `Message` to signal acknowledgement or errors. For example, report success with `message.reportAck()`. If there are errors or duplicates, use `message.reportError`, `message.reportNonRetryableError`, and `message.reportDuplicate`. For more information about acknowledgements, see “Submitting Acks and Errors and Duplicates from Your Messaging Plugin” on page 176.
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.

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 “Java Entity Libraries Overview” on page 114 in the *Gosu Reference Guide*.

Example Message Reply Plugin

The following code illustrates a simple abstract message reply plugin that listens for an acknowledgement to a recent message send request. This example plugin registers an incoming message queue from which it pulls

future responses from the remote destination system. If the plugin detects a response from this queue, it acknowledges the message with the `message.reportAck()` method.

```
package examples.plugins.messaging;

import com.guidewire.cc.external.entity.Message;
import com.guidewire.cc.plugin.messaging.MessageFinder;
import com.guidewire.cc.plugin.messaging.MessageReply;
import com.guidewire.cc.plugin.messaging.MessageReplyBase;
import com.guidewire.pl.plugin.PluginCallbackHandler;

import java.util.Map;

/**
 * Sample implementation of the MessageReply plugin interface that shows
 * how to acknowledge a message from an asynchronous reply.  Registers
 * a listener to be called whenever a message posts to
 * a simulated asynchronous queue.  If your listener get a message,
 * use the PluginCallbackHandler and MessageFinder instances to
 * look up and acknowledge the message.
 */
public class MessageReplyImpl extends MessageReplyBase {

  public void setParameters(Map params) {
  }

  public void shutdown() {
  }

  public void suspend() {
  }

  public void resume() {
    // Start listening on the queue.
    QueueSimulator.getInstance().listen(new MyQueueListener());
  }

  public void setDestinationID(int destinationID) {
  }


  /**
   * Implementation of QueueListener that listens for messages to be posted
   * to the queue and acks them.
   */
  private class MyQueueListener implements QueueListener {

    /**
     * Called if a message inserts in the queue.  Looks up the message
     * and acks it using the PluginCallbackHandler instance supplied while
     * creating and initializing the plugin.
     *
     * @param messageID
     */
    public void responseReceived(final int messageID) {

      // IMPORTANT: this method relies on the private variables _messageFinder
      //            and _messageFinder, which are set by the superclass MessageReplyBase

      PluginCallbackHandler.Block block = new PluginCallbackHandler.Block() {
        public void run() throws Throwable {
          Message message = _messageFinder.findById(messageID);
          message.reportAck();
        }
      };

      try {
        _pluginCallbackHandler.execute(block);
      } catch (Throwable throwable) {
        throwable.printStackTrace();  // In the real world, log this at least.
      }
    }
  }
}
```

You can find the complete sample code (including the queue simulator) in the product at the path:

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
```

This example includes the files `QueueSimulator.java`, `MessageReplyImpl.java`, `MessageRequestImpl.java`, `QueueListener.java`, and `MessageTransportImpl.java`.

Exception Handling

For details of exceptions and handling suspect/shutdown in messaging plugins, see “Error Handling in Messaging Plugins” on page 183.

Error Handling in Messaging Plugins

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.
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.

IMPORTANT All Gosu or Java exceptions mark the message a retryable during the `send` method or from the related `beforeSend` and `afterSend` methods. All exceptions at this time imply retryable errors. However, the distinction between retryable errors and non-retryable errors still exists if submitting errors later on 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 but it can retry more times, call `message.reportError(...)`. If there is an error for the message and retrying would not help, call `message.reportNonRetryableError(...)`. See "Submitting Acks and Errors and Duplicates from Your Messaging Plugin" on page 176.

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 and Errors and Duplicates from Your Messaging Plugin" on page 176.

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 JMS 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.

WARNING A messaging plugin `suspend`, `shutdown`, and `resume` method must not use messaging web service APIs that suspend or resume messaging destinations.

Resyncing Messages

Most ClaimCenter implementations use the messaging system to synchronize ClaimCenter data with claim data in an external system.

In rare cases some messaging integration condition might file, 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 the case that the external system has incorrect or incomplete information. For this type of situation, ClaimCenter provides a programming hook called a *resync event* (a resynchronization event) to recover from such messaging failures.

The resync can be triggered from the administration user interface or programmatically using the `IMessagingToolsAPI` interface's `resyncClaim` method.

As a result of a resync request, ClaimCenter triggers the resync event (`ClaimResync`). Your messaging destination can listen for this even in Event Fired business rules.

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 claim.

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 elements of the claim that are out-of-sync (or might be) 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. In this case, resend the entire series of messages. It may be necessary to add custom properties to various elements so you can help the external system track its synchronization state. If you can determine that you only need to resend a subset of messages, you can choose to generate elements of the claim only for those not in sync.

In the second approach, Gosu rules examine the failed message and all queued and unsent messages for the claim for a specific destination. Your rules must determine which messages must be regenerated. Instead of examining the entire claim, consider only 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.

If you create new messages, the new messages queue in creation order, independent of the order of original messages. This might be a different order than they originally sent data to 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 (Nack).
- `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 message status values are as follows:

Message status value	Meaning	Can appear during resync
1	Pending send	Yes
2	Pending ack	Yes
3	Error	Yes
4	Retryable error	Yes
10	Acknowledged	No
11	Error cleared and skipped	No
12	Error retried and skipped	No
13	Skipped	No

As indicated in the previous table, status values 10 and higher do not actually appear during resync. However, use this table for reference for debugging and analyzing the status property in `Message` and `MessageHistory` tables.

Cloning New Messages From Pending Messages

As mentioned earlier, you can clone a new message from a pending message that you retrieved with the code `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 method copies a message into a new message and returns the new message. If desired, you can modify the new message within the 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.

Resync in ContactCenter

If you use Guidewire ContactCenter, be aware that ContactCenter supports *resync* features for the `ABContact` entity. To detect resyncing of an `ABContact` entity, your destination can 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 ContactCenter with the external system. For more ContactCenter integration information, see “Address Book Integration” on page 299.

How Resync Affects Pre-Update and Validation

Be aware that claim 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 `ResyncClaim` event (triggered from a claim resync from the user interface), no entity data inherently changes due to this event.

This affects any other custom event firing through the `addEvent` entity method.

If you use ContactCenter, this also affects ContactCenter’s entity validation for events such as the `ResyncABContact` event.

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 the 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 included in the Java plugin libraries that can map the message payload using text substitution. The utility scans the message to find any strings surrounded by delimiters you define, and allows you to substitute any value. A typical usage is looking up a value in a table and recoding it.

IMPORTANT The utility class is in the package `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, use the 2-character strings "`**`" and "`**`" as your delimiters.
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 `StringSubstitution.Exchanger` class that knows what to look for and what to substitute. For example, the exchanger class might use its own look-up table or look in a properties file.
4. Initialize the `StringSubstitution` class with your delimiters and your `Exchanger` class
5. Call the utility's `substitute` method 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 170.

- **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 resyncs the message's related claim.

- **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 is claim-related, it is *safe-ordered*. This type of error blocks sending other messages for that claim/destination pair.

  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 processes. 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 (Nack).** Message sending for that message has failed.

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 resyncing to handle this case. All related pending and failed messages drop and resend. For more information, see “Resyncing Messages” on page 185.

If you also use Guidewire ContactCenter, note that it also supports resync.

For more information about acknowledgements, see “Reporting Acknowledgements and Errors” on page 176.

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 is 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 initializes each destination again.

Messaging tool actions such as `suspend`, `resume`, `retry`, and others can trigger from the **Administration** interface and also from the web services `IMessagingToolsAPI` interface. These messaging tools require the server’s run level to be `multiuser`.

For more information about administration features, see the *ClaimCenter System Administration Guide*.

SOAP API Interfaces for Error Handling

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 services API interface, `IMessagingToolsAPI`, which lets an external system remotely control the messaging system:

- `retryMessage` – Same as using the ClaimCenter user interface.

- `skipMessage` – Same as using the ClaimCenter user interface.
- `retryRetryableErrorMessages` – Retries all *retryable* messages for a destination. This might make sense if the destination was temporarily unavailable and is now back on-line. You can also specify a maximum number of times to retry any given message so that messages which continue to fail do not retry endlessly.
- `purgeCompletedMessages` – Deletes completed messages 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 muse this method periodically to purge old messages. This prevents the database from growing unnecessarily large.

  IMPORTANT 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.

- `suspendDestination` – Tells ClaimCenter to stop 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 to shut down the destination system to halt sending during processing of a daily batch file.
- `resumeDestination` – Tells ClaimCenter to start trying to send messages to the destination again. If suspend released some resources, reclaim these resources in the this resume method. For example, reconnect to a message queue.
- `getMessageStatisticsForSafeOrderedObject` – Returns information similar to that viewable in the user interface for a claim: the number of messages failed, retryable, in-flight, and unsent for a claim/destination pair.

Second, these same API methods (except for `getClaimMessageStatistics`) are available using the `messaging_tools` command line tool. Within the administrative environment, the following command shows the syntax:

```
messaging_tools -help
```

Messaging tool actions such as `suspend`, `resume`, `retry`, and others can be triggered from the Administrator interface and also from the web services `IMessagingToolsAPI` interface. These messaging tools can only be used if the server run mode is `multiuser`.

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 141 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** → **Plugins** → **gw** → **plugin** → **messaging** → **emailMessageTransport.** In the parameters area of the pane, click the **Add** button. Add the parameter `UseMessageCreatorAsUser` and set it to `true`.

Messaging Transport Examples

There are several messaging examples included in the Java API examples in the following directory:

```
ClaimCenter/java-api/examples/messaging
```

These examples include:

- **Console message transport.** An extremely simple transport that writes the message text payload to the ClaimCenter console window. Use this 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.
- **JMS message transport.** A JMS transport that works with most web application hosts.
- **Basic message transport, reply, and request.** A simple message transport. This includes a "fake" queue called a *queue simulator* that pretends to be a queue like a JMS queue.

To use these, follow the instructions in the "Plugin Overview", on page 101.

The following overview lists the steps you must perform to use these examples:

1. If you want to make changes to the Java files before installing them, recompile them after making changes.
2. Move the desired Java `.class` files to the directory:
   ```
   ClaimCenter/modules/configuration/plugins/messaging/classes
   ```
3. Move any associated `.lib` files in that example's `libs` folder to the directory:
   ```
   ClaimCenter/modules/configuration/plugins/messaging/libs
   ```

IMPORTANT Some examples have critical library files. For example, the JMS message transport example requires its JMS-specific library files to the configuration environment `plugins` folder as described in the step 3. The JMS example requires OpenJMS version 0.7.5, and other versions may require modifications to the Java code.

4. Register each Java plugin in the plugin registry. Manage this information in Guidewire Studio in the Messaging editor. See “Using the Plugins Editor” on page 141 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 Editor” on page 141 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 161 in the *Configuration Guide*.

   Specify the plugin directory as `messaging` and specify any parameters. The JMS example requires the following connection parameters:

   Set the following `JMSTransport` parameters

   - The `url` parameter must have queue URL value such as `"rmi://localhost:1099/"`
   - The `connectionFactoryName` parameter must have the value `"JmsQueueConnectionFactory"`
   - The `sendQueue` parameter must have the value `"outbound"`

   Set the following `JMSMessageReply` parameters:

   - The `url` parameter must have queue URL value such as `"rmi://localhost:1099/"`
   - The `connectionFactoryName` parameter must have the value `"JmsQueueConnectionFactory"`
   - The `replyQueue` parameter must have the value `"outbound"`

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.

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 Editor” on page 141 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 161 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.

chapter 9

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.

IMPORTANT This topic heavily relies on terminology and concepts documented within "Messaging and Events", on page 139. That earlier topic documents the messaging system, including events, messages, destinations, messaging plugins, and acknowledgements. If you have not yet read that earlier topic fully, do so before reading this topic.

This topic includes:

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 139.

The *Entityname*`StatusChanged` 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 196 and "Message Acknowledgement-based Status Transitions" on page 198.

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 198.
- **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 203. 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 the *ClaimCenter 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 "Do Not Call SOAP APIs on the Same Server From Messaging Plugins" on page 148. 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)
  ```

  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 change the status *directly.*

  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 directly. 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.

Message Acknowledgement-based Status Transitions

As mentioned in “How Do Status Transitions Happen?” on page 196, **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	`check.acknowledgeSubmission()`	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	`check.acknowledgeTransfer()`	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	`recovery.acknowledgeVoid()`	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	`payment.acknowledgeRecode()`	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	`transaction.acknowledgeSubmission()`	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	`bulkInvoice.acknowledgeSubmission(message)`	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.

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 196.

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 ensure that 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 “Event and Messaging Flow” on page 143 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 203
- “Payment Transaction Integration” on page 212
- “Recovery Reserve Transaction Integration” on page 217
- “Recovery Transaction Integration” on page 218
- “Reserve Transaction Integration” on page 220

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 193 for details.

Claim Financials Web Services

The claim financials API (`IClaimFinancialsAPI`) interface manipulates checks, transaction sets, and other financial information attached to claims.

Check Update Status SOAP APIs

An important method in the `IClaimFinancialsAPI` interface 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.

To update check status from SOAP APIs, see “Update Check Status” on page 209.

Check Void and Stop SOAP APIs

To void or stop a check from SOAP APIs, see “Voiding and Stopping Checks” on page 209.

Add Claim Financials SOAP APIs

There are two methods in the `IClaimFinancialsAPI` interface for adding claim financials to a claim.

Both methods take the financials information in the form of a subclass of `TransactionSet` that corresponds to the desired data:

- **Checks.** For checks, create a `CheckSet` object, which includes a list of checks, a list of check groups, and a set of associated reserves (`CheckSetReserve` objects). Check sets can include recurring checks.
- **Recoveries.** For recoveries, create a `RecoverySet` object, which is mostly a container for a one-item array of recovery (`Recovery`) objects. Refer to the SOAP API Reference Javadoc for details.
- **Reserves.** For checks, create a `ReserveSet` object, which is mostly a container for an array of reserve (`Reserve`) objects.

For more details on the specific transaction properties, refer to the *SOAP API Reference Javadoc* for details.

To submit new financials that have been processed neither in ClaimCenter or an external system yet, use the `addClaimFinancialsWithValidation` method. It triggers transaction validation, duplicate checking, approval, and submission.

This method runs validation twice: both before submitting for approval, and also after committing to the database. If you are trying to integrate financial information for later editing and processing, use this input method because it parallels the steps in the user interface.

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 (the 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 situations in which ClaimCenter provides only *read-only* financial information entered from another system, use the `addClaimFinancials` method. Despite the lack of the phrase “with validation” in the method name, this method **does** trigger validation. Never attempt to import financials that would fail validation. However, 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`.

Both of these methods can also take a set of new `Document` objects to attach to the claim. Encapsulate the documents within a `TransactionSetDocument` object. You can attach an array of an array of `TransactionSetDocument` objects to a `TransactionSet` subclass using the method on `TransactionSet` called `setDocuments`.

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 this method, call the `IClaimAPI` method `getClaimInfo` with `null` as the second parameter and get the archived state:

```
archiveState = myClaimAPI.getClaimInfo("ABC:12345", null).getArchiveState();
```

Additionally, you can use the `reopenClaim` method of `IClaimAPI` to reopen the claim if necessary, which would allow you to add financials to it.

Multicurrency with New Financials

If you import transactions with the `IClaimFinancialsAPI` method `addClaimFinancials`, there are two ways to import transactions.

There are special method signatures of `addClaimFinancials` and `addClaimFinancialsWithValidation` that accept an exchange rate parameter and a comment about the exchange rate:

```
String addClaimFinancials(TransactionSet tSet, BigDecimal rate, String exchangeRateDescrip)
String addClaimFinancialsWithValidation(TransactionSet tSet, BigDecimal rate,
   String exchangeRateDescrip)
```

If you use any of the other alternative method signatures, all transactions in a transaction set **must** have the same currency and must use the same exchange rate. See "Multiple Currencies" on page 171 in the *Application Guide* for more information about multicurrency and "Exchange Rate Integration" on page 236 for more information about specifying exchange rates. These methods throw an exception if you try to add a check with payments of differing currencies. Also, the exchange rate must not be set to `null`.

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 save such changes.

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 "Do Not Call SOAP APIs on the Same Server From Messaging Plugins" on page 148. 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.

Multicurrency Foreign Exchange Adjustment SOAP APIs

There are two `IClaimFinancialsAPI` 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.

For a check, use the method:

```
applyForeignExchangeAdjustmentToCheck(String checkId, BigDecimal newClaimCurrencyAmount)
```

For a payment, use the method:

```
applyForeignExchangeAdjustmentToPayment(String paymentId, BigDecimal newClaimCurrencyAmount)
```

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.

ClaimCenter does not support foreign exchange adjustments for multi-payee checks or payments. However, joint-payee checks can have adjustments.

Note: See "Multiple Currencies" on page 171 in the *Application Guide* for more information about multicurrency and "Exchange Rate Integration" on page 236 for more information about specifying exchange rates.

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 "Do Not Call SOAP APIs on the Same Server From Messaging Plugins" on page 148. 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.

Getting Financials By Cost Type, Cost Category, and Cache Type

The `IClaimFinancialsAPI` web services API interface extracts financials information for a set of claims or exposures based on a financials cache type, cost type, and cost category. Not all financials values map to a specific cost category or cost type. If you need the overall information for a claim or exposure, pass `null` for those parameters.

To extract claim financials, use the `getClaimAmounts` method, which takes claims (as claim numbers), as well as a `FinancialsCacheType`, a `CostCategory`, and `CostType`.

To extract claim financials, use the `getExposureAmounts` method, which is identical except takes a list of exposures instead of claims.

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 "Do Not Call SOAP APIs on the Same Server From Messaging Plugins" on page 148. 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.

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 **New Claim 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.** You can specify that a check print and process **outside** the ClaimCenter system. 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.

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. In the reference implementation, manual checks do not need approval but you can customize this behavior.	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. The tasks trigger at 12:01am and 5:01pm by default. For a check to issue on the same day as its creation, ClaimCenter holds the check in this state temporarily until the background processing task runs. The schedule for the background processing task must correspond to your work schedule so that same-day checks created late in the day do not wait overnight to submit.	Yes	--
`requesting`	The standard check request is ready to be sent to an external system.	Yes	--

Check status	Meaning	Std	Man
`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 check notification is ready to be sent to an external system.	--	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	--
	→ `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
	→ *object deleted*	A user deletes a check in the user interface.	Yes	Yes
`awaitingsubmission`	→ `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	--
	→ *object deleted*	A user deletes a check in the user interface.	Yes	--
`requesting`	→ `requested`	ClaimCenter received an acknowledgement for the associated message for the `requesting` status. This transition **requires** you to call `check.acknowledgeSubmission()` in your message Ack code in your messaging plugins.	Yes	--
	→ `pendingtransfer`	A user attempts to transfer a check.	Yes	--

Check status	Can become status	How it changes	Std	Man
	→ 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 denyCheck (either SOAP API method or 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. This transaction only works for single checks. The check cannot be part of a multipayee/grouped check, nor a BulkInvoiceItem check.	Yes	Yes
	→ issued	*This transition is disallowed.*	Yes	--
	→ cleared	*This transition is disallowed.*	Yes	--
pendingvoid	→ voided	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check voided.	Yes	Yes
	→ issued	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check issued. The void was unsuccessful.	Yes	Yes
	→ cleared	From SOAP API or domain method, calling updateCheckStatus for a check status change that indicates the check cleared. The void was unsuccessful.	Yes	Yes
	→ stopped	*This transition is disallowed.* To simulate this transition: **1.** first transition from PendingVoid → Issued using the method updateCheckStatus (either SOAP API method or domain method). **2.** Request stop using the stopCheck() method (again from either SOAP API or domain method. **3.** Finally use updateCheckStatus again to transition from PendingStop -> Stopped.	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	*This transition is disallowed*. To simulate this transition: **1.** first transition from PendingStop → Cleared using the method updateCheckStatus (either SOAP API method or domain method). **2.** Request void using the voidCheck() method (again from either SOAP API or domain method. **3.** Finally use updateCheckStatus again to transition from PendingVoid → Voided.	Yes	Yes

Check status	Can become status	How it changes	Std	Man
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	--
	→ denied	For single standard checks, this transition works from the check method denyCheck (either SOAP API method or domain method). For single manual checks, this transaction works **only** from the check method denyCheck called from messaging plugins. It does **not** work from the SOAP API version of this method. This transaction only works for single checks. The check cannot be part of a multipayee/grouped check, nor a BulkInvoiceItem check.	Yes	Yes
	→ pendingvoid	A user requests a check void.	Yes	--
	→ pendingstop	A user requests a check stop.	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 a manual check only, 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

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). For single manual checks, this transaction works **only** from the check method denyCheck called from messaging plugins. It does **not** work from the SOAP API version of this method. This transaction only works for single checks. The check cannot be part of a multipayee/grouped check, nor a BulkInvoiceItem check.	Yes	Yes
	→ pendingvoid	A user requests a check void.	--	Yes
	→ pendingstop	A user requests a check stop.	--	Yes
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 198

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 SOAP API interface IClaimFinancialsAPI method updateCheckStatus.

> **Note:** For more information about using the IClaimFinancials web service interface, see “Claim Financials Web Services” on page 200.

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 196 and “Do Not Call SOAP APIs on the Same

Server From Messaging Plugins” on page 148 for more information.

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 “Do Not Call SOAP APIs on the Same Server From Messaging Plugins” on page 148. 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.

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 203.

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 212.

Updating Status from SOAP

The SOAP API interface `IClaimFinancialsAPI` method `updateCheckStatus` method takes the following:

- a check ID
- a check number
- a 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” on page 200.

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 196.

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

An important method in the `IClaimFinancialsAPI` interface 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 id (optional)
- check number (optional)
- the issue date (optional)
- the status of the check (required), as a `TransactionStatus` typecode
- a mapping of additional fields on the check to update as part of the status change (optional).

To omit any optional arguments, pass `null` for that argument.

For example:

```
claimfinancialsAPI.updateCheckStatus("abc:1234", "1024", null, TransactionStatus.TC_REQUESTED, null);
```

If you want to update the additional fields, pass an array of `FieldValue` objects for the last parameter. They encapsulate name/value pairs for properties to set on the `Check` entity instance.

To void or stop a check, see “Voiding and Stopping Checks” on page 209.

For more information about how a check’s status affects check and payment status, see the following topics:

- “Check Integration” on page 203
- “Payment Transaction Integration” on page 212,
- “Check Status Transitions” on page 207

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 “Do Not Call SOAP APIs on the Same Server From Messaging Plugins” on page 148. 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.

Voiding and Stopping Checks

ClaimCenter includes APIs to void or stop a check. They are available in two ways:

- The `IClaimFinancialsAPI` 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 SOAP API (in the IClaimFinancialsAPI 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 SOAP APIs are called from external systems.

The SOAP API version takes a check public ID:

```
claimfinancialsapi.voidCheck("abc:12345");
```

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 210.

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 SOAP API (in the IClaimFinancialsAPI 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 SOAP APIs are called from external systems.

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` is `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 SOAP API (in the `IClaimFinancialsAPI` web service) and also as a domain method on a `Check` entity instance. However, the arguments are slightly different due to how SOAP APIs are called from external systems.

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 210.

The SOAP API version takes a check ID:

```
claimfinancialsapi.stopCheck("abc:12345");
```

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.

An example 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.

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 the place to include the check amount. In other words, which one of the following:

- in Future Payments (tomorrow or later)
- in 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
`null`	An internal initial status.	Yes	Yes
`pendingapproval`	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
`rejected`	The approver did not approve the associated check.	Yes	Yes
`awaitingsubmission`	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 203 for more details.	Yes	--
`submitting`	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
`submitted`	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
`pendingvoid`	Void request for the associated check is ready to be sent to an external system.	Yes	Yes
`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

Payment status	Can change to status	How it changes	Std	Man
	→ `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 `check.acknowledgeSubmission()` 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
	→ `pendingvoid`	Either of the following: • A user attempts to void an associated check in the user interface. • Some code calls the `voidCheck` method (available as a SOAP API or as a domain method on a check). See “Voiding and Stopping Checks” on page 209	Yes	Yes
	→ `pendingstop`	Either of the following: • A user attempts to stop an associated check in the user interface. • Some code calls the `stopCheck` method (available as a SOAP API or as a domain method on a check). See “Voiding and Stopping Checks” on page 209	Yes	Yes
	→ `pendingstransfer`	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 215.	Yes	Yes

Payment status	Can change to status	How it changes	Std	Man
submitted	→ pendingvoid	Either of the following: • A user attempts to void an associated check in the user interface. • Some code calls the voidCheck method (available as a SOAP API or as a domain method on a check). See "Voiding and Stopping Checks" on page 209	Yes	Yes
	→ pendingstop	Either of the following: • A user attempts to stop an associated check in the user interface. • Some code calls the stopCheck method (available as a SOAP API or as a domain method on a check). See "Voiding and Stopping Checks" on page 209	Yes	Yes
	→ pendingstransfer	A user requests a check transfer to another claim.	Yes	Yes
	→ 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 215.	Yes	Yes
pendingvoid	→ voided	SOAP API for a check status change that indicates successful voiding.	Yes	Yes
	→ stopped	SOAP API for a check status change that indicates the check stopped. The void was unsuccessful.	Yes	Yes
	→ 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
	→ voided	SOAP API for a check status change that indicates the check voided. The stop was unsuccessful.	Yes	Yes
	→ 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 check.acknowledgeTransfer() 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 216	Yes	Yes
pendingrecode	→ recoded	ClaimCenter received an acknowledgement for the associated message for the pendingrecode status. This transition **requires** you to call check.acknowledgeRecode() in your message Ack code in your messaging plugins. For more information, see "Integration Events For Payment Recoding" on page 215.	Yes	Yes
recoded	→ pendingvoid	Either of the following: • A user attempts to void an associated check in the user interface. • Some code calls the voidCheck method (available as a SOAP API or as a domain method on a check). See "Voiding and Stopping Checks" on page 209	Yes	Yes

Payment status	Can change to status	How it changes	Std	Man
	→ pendingstop	Either of the following: • A user attempts to void an associated check in the user interface. • Some code calls the stopCheck method (available as a SOAP API or as a domain method on a check). See "Voiding and Stopping Checks" on page 209	Yes	Yes
transferred	*no transitions possible*	*no transitions possible.* For related information, see "Integration Payment Events for Check Transfer" on page 216.	--	--
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 198.

Note: 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" on page 200.

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* is 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 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 217.

The following table lists events that trigger during recoding:

Root object	Event name	Meaning in this context
Original payment	`PaymentChanged`	The payment `status` property changed. You can catch this event, but you can also use the special `PaymentStatusChanged` event listed later in this table.
	`PaymentStatusChanged`	The payment `status` property changed, specifically `payment.status = pendingrecode`. Use the `Payment` domain properties `Onset` and `Offset` to get the associated payments.
New offset payment	`PaymentAdded`	New offset payment. Within business rules, check `payment.RecodingOffset` to determine if this is the offset payment. Call `payment.PaymentBeingOffset` 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 `payment.RecodingOnset` to determine if this is the onset payment. Call `payment.OriginalPayment` 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 148 and the *ClaimCenter 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. Cal the method `payment.acknowledgeRecode()`. See in "Message Acknowledgement-based Status Transitions" on page 198 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 203.

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 217. 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 198. 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.

Recovery reserve status	Meaning
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	→ submitting	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.
	→ pendingapproval	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.
pendingapproval	→ submitting	The approver approves the recovery reserve and no additional approval is necessary.
	→ rejected	The approver rejects the recovery reserve.
	→ *object deleted*	A user deletes the recovery reserve in the user interface.
rejected	→ *object deleted*	A user deletes the recovery reserve in the user interface.
submitting	→ submitted	ClaimCenter received an acknowledgement for the associated message for the submitting status. This transition **requires** you to call transaction.acknowledgeSubmission() in your message Ack code in your messaging plugins.
submitted	*no transitions possible*	*no transitions possible*

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 198

Recovery Transaction Integration

The following table lists the status codes and meanings for recovery transactions:

Recovery status	Meaning
null	An internal initial status.
submitting	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.
submitted	The recovery sent to an external system (and acknowledged).
pendingvoid	The check void request is ready to be sent to an external system.
voided	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	→ submitting	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.
	→ pendingapproval	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.
submitting	→ submitted	ClaimCenter received an acknowledgement for the associated message for the submitting status. This transition **requires** you to call recovery.acknowledgeSubmission() in your message Ack code in your messaging plugins.
	→ pendingvoid	A user attempts to void a recovery.
submitted	→ pendingvoid	A user attempts to void a recovery.
pendingvoid	→ voided	ClaimCenter received an acknowledgement for the associated message for the pendingvoid status. This transition **requires** you to call recovery.acknowledgeVoid() in your message Ack code in your messaging plugins.
voided	*no transitions possible*	*no transitions possible*
pendingapproval	→ submitting	The approver approves the recovery and no additional approval is necessary.
	→ rejected	The approver rejects the recovery
rejected	→ *object deleted*	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 198

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
`null`	An internal initial status.
`submitting`	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 `submitting` status.
`submitted`	The reserve was sent to an external system (and acknowledged).
`pendingapproval`	The reserve saved in ClaimCenter but is pending approval.
`rejected`	The approver did not approve this reserve.
`awaitingsubmission`	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
`null`	→ `submitting`	New reserve 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*).
	→ `pendingapproval`	New reserves make this transition if they require approval due to approval rules and authority limits.
	→ `awaitingsubmission`	The reserve acts as either a non-eroding offsetting reserve or a zeroing offsetting reserve for a payment.
`submitting`	→ `submitted`	ClaimCenter received an acknowledgement for the associated message for the `submitting` status. This transition **requires** you to call `reserve.acknowledgeSubmission()` in your message Ack code in your messaging plugins.
`submitted`	*no transitions possible*	*no transitions possible*
`pendingapproval`	→ `submitting`	The approver approves the recovery and it does not require further approval.
	→ `rejected`	The approver rejects the reserve.
	→ *object deleted*	A user deletes the recovery in the user interface.
`rejected`	→ *object deleted*	A user deletes the recovery in the user interface.
`awaitingsubmission`	→ `submitting`	If the reserve is a non-eroding offsetting reserve or zeroing offsetting reserve and the payment that it offsets transitions to the `submitting` state.
	→ *object deleted*	Deletion automatically occurs if something deletes the associated payment or if the reserve's payment changes such that the offsetting reserve is unnecessary. Examples of the latter case: • A *non-eroding* payment changes to be an *eroding* payment. • A *current day* payment that exceeds reserves has its owning check rescheduled to be a *future check*. • The payment amount changes such that it no longer exceeds reserves, or exceeds reserves by a different amount.

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 198.

Bulk Invoice Integration

ClaimCenter now contains support for *bulk invoices*. 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 to 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 optional web service APIs a rental car company 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 **Validate** button to validate the invoice. This validate step ensures that the bulk invoice does not violate your business logic requirements for bulk invoices.

> **IMPORTANT** 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:

- **Basic claim number validity.** In the user interface, the user interface code checks basic claim number validity. The web service method `IBulkInvoiceAPI` method `addItems` 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 `IBulkInvoiceAPI` method `addItems` 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 method `IBulkInvoiceAPI` method `submitBulkInvoice` verifies this automatically.

Once 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. See the *ClaimCenter Rules Guide* for details.

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 in Gosu. Use this for testing purposes only. Access the sample Gosu plugin by navigating in Studio in the **Resources** panel at the path **Classes** → **Plugins** → **bulkinvoice**.

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

ClaimCenter includes a web services API interface called `IBulkInvoiceAPI` that allows integrations to submit bulk invoices directly from external systems. For example, an associated rental car company could directly submit bulk invoices to ClaimCenter from their systems using these integration APIs. `IBulkInvoiceAPI` methods can create and submit `BulkInvoice` entities, as well as add, update, and delete `BulkInvoiceItem` entities.

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 domain method versions of all these APIs. If you have additional questions, contact Custom Support.

Many returned entities contain foreign key IDs rather than direct subobject links to associated entities. For example, calling the `getItems` method returns an array of bulk invoice item (`BulkInvoiceItem`) entities. Each bulk invoice item has a `BulkInvoice` property, which is the public ID of the bulk invoice (`BulkInvoice`) that contains it. To get a reference to the `BulkInvoice` entity to read properties on it, make a separate call to the `IBulkInvoiceAPI` interface method `getBulkInvoice`.

Similarly, the bulk invoice item links to the claim and exposure with public IDs. This public ID approach simplifies API responses and makes network access efficient. To call `IBulkInvoiceAPI` with bulk invoice items, define links to the `Exposure` and `Claim` on a `BulkInvoiceItem` with `setClaim` and `setExposure` methods. Both methods take a public ID string argument.

The `IBulkInvoiceAPI` interface includes the following methods:

- `addItems` - Adds `BulkInvoiceItem` entities to a `BulkInvoice` entity
- `createBulkInvoice` - Creates a new unapproved `BulkInvoice` entity with a status of `BulkInvoiceStatus.draft` and commits the bulk invoice.

- `deleteItems` - Deletes `BulkInvoviceItem` entities from a `BulkInvoice`.
- `getBulkInvoice` - Searches and returns a specific `BulkInvoice` specified by public ID.
- `getItems` - There are two versions of this method. One of them gets all `BulkInvoiceItem` entities with the specified claim ID linked to the specified `BulkInvoice`. The other gets all `BulkInvoiceItem` entities with the specified claim ID and payment amount linked to the specified `BulkInvoice`.
- `placeDownstreamHoldOnInvoice` - Sets the status of the `BulkInvoice` to be `onhold`.
- `requestBulkInvoice` - Requests that the specified `BulkInvoice` escalate for submission to the downstream system. Ordinarily the scheduled batch process `bulkinvoicesescalation` handles escalation, but it is possible that you need a `BulkInvoice` to escalate prior to this process running. If the scheduled send date of the `BulkInvoice` is the current day, this action causes the `BulkInvoice` and associated checks to move to a state of `requesting` then `requested`. If the scheduled send date is not today then no action occurs for this invoice.
- `stopBulkInvoice` - Stops the specified `BulkInvoice`.
- `submitBulkInvoice` - Submits the specified `BulkInvoice` for approval.
- `updateBulkInvoiceStatus` - Updates the status of a `BulkInvoice` to `issued`, `cleared`, `voided`, or `stopped`, and/or to set the check number or issue date of the bulk check.
- `updateItems` - Updates `BulkInvoiceItem` entities for a given `BulkInvoice`.

  WARNING Before you call the `updateItems` method, always fully populate all fields on the `BulkInvoiceItem` entities that you pass to this method. Any `null` values always overwrite non-`null` values on existing items. Similarly, if you add an array extension property to the `BulkInvoiceItem` entity, this method must fully populate the array or your array extension property elements disappear during the update.

- `validateBulkInvoice` - Validates a `BulkInvoice`.
- `voidBulkInvoice` - Voids a `BulkInvoice`.

Refer to the SOAP API Javadoc Documentation for details of each method.

Typical Bulk Invoice API Usage

Suppose you want to create a new bulk invoice and submit it. A typical flow would be:

1. Create a new `BulkInvoice` entity.
2. Fill in properties in the `BulkInvoice` entity and fill in the following properties:

- `CheckNumber` - the check number
- `RequestingUser` - the requesting user
- `SplitEqually` - (Boolean) whether to split items equally
- `PublicID` - bulk invoice public ID
- `InvoiceNumber` - invoice number
- `Payee` - the payee contact entity. Using Bulk Invoices requires using a contact system and the payees must be in the contact system such as ContactCenter. For the `Payee` property, ClaimCenter needs the public ID of the contact. To safely get the public ID:

  a. first ensure that the contact is in the contact system. If you use ContactCenter, use the ContactCenter web services 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*. This ID is the native address book ID for the contact. This is different from the public ID.

  b. Populate a local version of a ClaimCenter SOAP `Contact` entity using the *address book link ID*.

  c. Call the ClaimCenter web service API that takes the link ID and returns a contact entity: `IBulkInvoiceAPI.createContactByLinkId(`*`contactLinkID`*`)`.

d. Get the public ID from the result of that method, and set the `Payee` to that value.

3. all the `createBulkInvoice` method to create a new unapproved `BulkInvoice`.
4. Call the `addItems` method to add new `BulkInvoiceItem` entities to the `BulkInvoice` entity.
5. After you add all items add to the bulk invoice, call the `validateBulkInvoice` to validate the `BulkInvoice`.
6. If there were no validation errors, call the `submitBulkInvoice` method to submit the bulk invoice.

For example, the following shows how you might create and submit a new simple bulk invoice in Java using the web service APIs:

```
// Create a new SOAP entity locally for the Bulk Invoice entity
binvoice = new BulkInvoice();

// set the total
binvoice.setBulkInvoiceTotal(new BigDecimal(1234));

// In this example, get a contact by "public ID".
// If using ContactCenter, instead you'd get a contact by its ContactCenter "link ID"
// using IBulkInvoiceAPI.createContactByLinkId(...)

// figure out which Contact will be the Payee.
// get the Address Book "link ID" for the contact.
//The contact MUST already be in the contact system.
// You must create the record in the address book before submitting the bulk invoice.
// Do whatever you need to get or create contact that is linked to AddressBook
Contact payeeContact = MyUtils.getMyPayee();

// get the Address Book Link ID for that contact, and ask ClaimCenter to get the payee
Contact biPayee = bulkInvoiceAPI.createContactByLinkId( payeeContact.getAddressBookUID() );

// set public ID with the PublicID property from the result of bulkInvoiceAPI.createContactByLinkId()
binvoice.setPayee( biPayee.getPublicID() );

// set other properties on the bulk invoice
binvoice.setRequestingUser("demo_sample:1");
binvoice.setSplitEqually(new Boolean(false));
binvoice.setPublicID("ABC:invoice1");
binvoice.setInvoiceNumber("invoice-1231");
binvoice.setCheckNumber("check-546");

// create the bulk invoice in the ClaimCenter system
BulkInvoice bin = bulkInvoiceAPI.createBulkInvoice(binvoice);

// Create a new array with one bulk invoice item for the bulk invoice
BulkInvoiceItem[] binvoiceItem = new BulkInvoiceItem[1];
binvoiceItem[0] = new BulkInvoiceItem();
binvoiceItem[0].setClaim("demo_sample:1");
binvoiceItem[0].setExposure("demo_sample:20002");
binvoiceItem[0].setCostType(CostType.TC_aoexpense);
binvoiceItem[0].setCostCategory(CostCategory.TC_other);
binvoiceItem[0].setPaymentType(PaymentType.TC_partial);
binvoiceItem[0].setStatus(BulkInvoiceItemStatus.TC_draft);
binvoiceItem[0].setAmount(new BigDecimal(5));
binvoiceItem[0].setPublicID("ABC:item123");

//set the nonEroding to true, so Remaining Reserves still to be the same
binvoiceItem[0].setNonEroding(new Boolean(true));

// Add the item to the bulk invoice using the SOAP API (in real code, catch exceptions...)
bulkInvoiceAPI.addItems(binvoice.getPublicID(), binvoiceItem);

// validate (in real code, catch exceptions...)
bulkInvoiceAPI.validateBulkInvoice(binvoice.getPublicID());

// submit the invoice (in real code, catch exceptions...)
bulkInvoiceAPI.submitBulkInvoice(binvoice.getPublicID());
```

After creating the bulk invoice, you can call APIs such as these if you have special properties to update. You can design your own SOAP APIs to update properties on existing `BulkInvoice` entities within the ClaimCenter database.

Post Submission Actions On the Bulk Invoice

The bulk invoice lifecycle is not necessarily complete after it submits to the downstream system. Possible updates can occur either through the ClaimCenter user interface, or through the SOAP API. For all SOAP API methods, refer to the SOAP API Javadoc Documentation for details.

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 232

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 `IBulkInvoiceAPI.placeDownstreamHoldOnInvoice()` method. Refer to the Java API Javadoc Documentation for more information about this method.

Once a *downstream hold* is on an invoice, only the following actions are possible:

- The bulk invoice can void.
- The bulk invoice can stop.
- The bulk invoice can resubmit as is.

Stopping a Bulk Invoice

A submitted bulk invoice can stop if it is considered stoppable by the system. Stopping a bulk invoice has the following effects:

- The status of the invoice transitions to `pendingstop`
- The status of every invoice item that has an associated check transitions the status Pending Stop
- Every associated check stops. This is the same effect as if the check stops individually through the user interface.

Once a bulk invoice transitions to `pendingstop`, it can only transition to Stopped status using a call to `IBulkInvoiceAPI.updateBulkInvoiceStatus(...)` method. 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, you can use the `IBulkInvoiceAPI.updateBulkInvoiceStatus()` method or the domain method `bulkInvoice.updateBulkInvoiceStatus()` to transition the invoice back to `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-stopped. This is the same behavior as if a regular un-bulked check stops but then the stop attempt fails.

Voiding a Bulk Invoice

A submitted bulk invoice can void if it is considered voidable by the system. Voiding a bulk invoice has the following effects:

- The status of the invoice transitions to `pendingvoid`.
- The status of every invoice item that has an associated check transitions to `pendingvoid`.
- Every associated check voids. This has the same effect as if the check voids 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 `IBulkInvoiceAPI.updateBulkInvoiceStatus(...)` method. For important warnings about usage and limitations with calling SOAP APIs, refer to "How Do Status Transitions Happen?" on page 196.
- a call from your messaging plugins to `bulkInvoice.updateBulkInvoiceStatus(...)`. For important warnings about usage and limitations with calling status-updating domain methods, refer to "How Do Status Transitions Happen?" on page 196.

In both cases, if the bulk invoice did not *already* void, using this method to transition to the `voided` status generates an error.

If the void on the bulk invoice is unsuccessful, 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.

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 processes all 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 creates 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 asychronous 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
`null`	An internal initial status for a new `BulkInvoice` entity.
`draft`	Status for initial editing of an invoice, or after invalidation due to changes.
`inreview`	Bulk Invoice requires approval and awaits action by the assigned approver.
`pendingbulkinvoiceitemvalidation`	The approver approved the bulk invoice. Individual bulk invoice items are now pending validation and undergoing final processing.
`invalidbulkinvoiceitems`	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.
`awaitingsubmission`	Bulk invoice awaits submission to the downstream system.
`requesting`	The bulk invoice queued for submission to the downstream system.
`requested`	The bulk invoice sent successfully to the downstream system.
`issued`	The bulk invoice's associated bulk check issued.

Bulk invoice status	Meaning
cleared	The bulk invoice's associated bulk check cleared.
rejected	The assigned approver rejected the bulk invoice.
pendingvoid	The bulk invoice voided, Confirmation of the void is pending.
pendingstop	The bulk invoice stopped. Confirmation of the void is pending.
voided	The bulk invoice successfully voided.
stopped	The bulk invoice successfully stopped.
onhold	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

Bulk invoice status	Can change to status	How it changes
null	→ draft	New `BulkInvoice` entities created.
draft	→ inreview	This transition occurs if both of the following occur: • a bulk invoice submits either through the ClaimCenter user interface or using the `IBulkInvoiceAPI.submitBulkInvoice()` method • the defined bulk invoice approval rules require it to escalate for approval.
	→ pendingbulkinvoiceitemvalidation	This transition occurs if both of the following occur: • a bulk invoice submits either through the ClaimCenter user interface or using the `IBulkInvoiceAPI.submitBulkInvoice()` method • the defined bulk invoice approval rules do **not** require it to escalate for approval.
inreview	→ pendingbulkinvoiceitemvalidation	This transition occurs if you approve a bulk invoice through the ClaimCenter user interface.
	→ rejected	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	This transition occurs after all approved bulk invoice items successfully validate, and a check automatically creates for them and the approver approves the check.
	→ invalidbulkinvoiceitems	This transition occurs if the processing of the bulk invoice's items complete and one or more items are now `notvalid` or the approver rejects one or more associated checks.
invalidbulkinvoiceitems	→ pendingbulkinvoiceitemvalidation	This transition occurs if the bulk invoice is resubmitted without first changing such that it becomes `draft` status again and does not require approval.
	→ draft	This transition occurs if you edit the bulk invoice in such a way that it no longer passes validation.

Bulk invoice status	Can change to status	How it changes
	→ inreview	This transition occurs if the bulk invoice resubmits and requires approver (and if it does not become draft status).
awaitingsubmission	→ draft	This transition occurs if you edit the bulk invoice such that it no longer passes validation.
	→ requesting	This transition occurs for bulk invoices with a scheduled send date of the current day if the bulkinvoicesescalation 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.
requesting	→ requested	ClaimCenter received an acknowledgement for the associated message for the submitting status. This transition **requires** you to call bulkInvoice.acknowledgeSubmission() in your message Ack code in your messaging plugins.
	→ issued	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status issued.
	→ cleared	This transition can occur through the calling updateBulkInvoiceStatus 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 IBulkInvoice.stopBulkInvoice() method.
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the IBulkInvoice.voidBulkInvoice() method.
	→ onhold	This transition occurs if the Bulk Invoice API places a downstream hold on the invoice, using a call to the placeDownstreamHoldOnInvoice( ) method.
requested	→ issued	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status issued.
	→ cleared	This transition can occur through the calling updateBulkInvoiceStatus method from SOAP API or domain method to change to status cleared.
	→ pendingstop	This transition occurs of the bulk invoice stops either through the ClaimCenter user interface, or using a call to the IBulkInvoice.stopBulkInvoice() method.
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the IBulkInvoice.voidBulkInvoice() method.

Bulk invoice status	Can change to status	How it changes
	→ onhold	This transition occurs if the Bulk Invoice API places a downstream hold on the invoice, using a call to the placeDownstreamHoldOnInvoice( ) method.
cleared	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the IBulkInvoice.voidBulkInvoice() method.
issued	→ cleared	This transition can occur through the calling updateBulkInvoiceStatus 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 IBulkInvoice.stopBulkInvoice() method.
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the IBulkInvoice.voidBulkInvoice() method.
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 the bulk invoice resubmits either through the ClaimCenter user interface or using the IBulkInvoiceAPI.submitBulkInvoice() method, and bulk invoice approval rules require escalation for approval.
	→ pendingbulkinvoiceitemvalidation	This transition occurs if the bulk invoice resubmits (in the user interface or through IBulkInvoiceAPI.submitBulkInvoice() and also the bulk invoice approval rules do not require it to escalate for approval.
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.

Bulk invoice status	Can change to status	How it changes
	→ 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.
	→ 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 IBulkInvoice.stopBulkInvoice() method.
	→ pendingvoid	This transition occurs if the bulk invoice voids either through the ClaimCenter user interface, or using a call to the IBulkInvoice.voidBulkInvoice() method.
	→ requesting	This transition occurs if the invoice resubmits either through the ClaimCenter user interface or using the IBulkInvoiceAPI.submitBulkInvoice() method.

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 198

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`.

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 `IBulkInvoiceAPI.updateBulkInvoiceStatus()`
- from your messaging code, call the domain method `bulkInvoice.updateBulkInvoiceStatus()`

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 196.

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 196.

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
`null`	An internal initial status for a new `BulkInvoiceItem` entity not yet committed to the database.
`draft`	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.
`approved`	The approver approved the bulk invoice item and is valid for processing.
`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.

Bulk invoice item status	Meaning
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:

Bulk invoice item status	Can change to status	How it changes
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 bulkInvoice.acknowledgeSubmission() 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 IBulkInvoice.stopBulkInvoice() method.
	→ pendingstop	This transition occurs if the owning bulk invoice voids either through the ClaimCenter user interface, or using a call to the IBulkInvoice.voidBulkInvoice() method.
	→ pendingtransfer	This transition occurs if the check associated with the bulk invoice item transfers from the ClaimCenter user interface.
submitted	→ pendingvoid	This transition occurs if the owning bulk invoice stops either through the ClaimCenter user interface, or using a call to the IBulkInvoice.stopBulkInvoice() method.

Bulk invoice item status	Can change to status	How it changes
	→ pendingstop	This transition occurs if the owning bulk invoice voids either through the ClaimCenter user interface, or using a call to the IBulkInvoice.voidBulkInvoice() method.
	→ pendingtransfer	This transition occurs if the check associated with the bulk invoice item transfers from the ClaimCenter user interface.
inreview	→ draft	This transition occurs if the bulk invoice item changes in such a way that the owning bulk invoice no longer passes validation.
notvalid	→ draft	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 pendingbulkinvoiceitemvalidation status.
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 check.acknowledgeTransfer() in your message Ack code in your messaging plugins.
Voided	→ 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.

Bulk Invoice Batch Processes

ClaimCenter has two bulk invoice batch processes: bulkinvoicesworkflow and bulkinvoicesescalation. For more information, see the *ClaimCenter 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.

IMPORTANT For more information about using and deploying plugin templates, see “Writing Plugin Templates in Gosu” on page 120 in the *Integration Guide*.

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 `InitialReservesAdapter` 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 in Gosu” on page 120.

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.

IMPORTANT For more information about multicurrency, see “Multiple Currencies” on page 171.

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, 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 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 use the rate you explicitly specified.

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

The 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 129 and also using web services in “Maintenance Web Services” on page 84.

For more information about multicurrency, see “Multiple Currencies” on page 171 in the *Application Guide*.

chapter 10

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.

IMPORTANT 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 101. For the complete list of all ClaimCenter plugins, see “Summary of All ClaimCenter Plugins” on page 106.

This topic includes:

Overview of User Authentication Interfaces

There are three mechanisms to “log in” to ClaimCenter:

- Logging into the web application user interface, as a standard user.

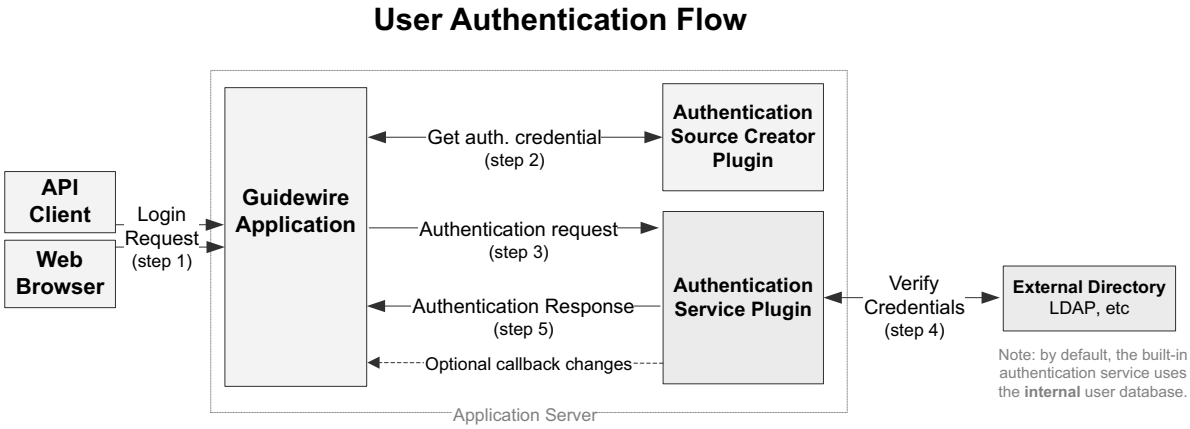
- Logging into the server through Guidewire Studio.
- From a SOAP API call, logging in as a user with SOAP API privileges.

Authentication plugins must handle **all three types of logins**.

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:

1. **An initial login request.** User authentication requests can come from web browsers and web service API calls (including command line tools). If the specified user is not currently logged in, ClaimCenter attempts to log in the user.
2. **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 into a structured credential called an *authentication source*. This plugin creates the *authentication source* from information in `HTTPRequests` from browsers and returns 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 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");
   ```

3. **The server requests authentication by using an authentication service.** ClaimCenter passes the *authentication source* to the *authentication service*, which is responsible for determining whether or not to permit the user to log in.
4. **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. **The 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 gives the user access to the system. If rejected, ClaimCenter redirects the user to a login page to try again or returns 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 cannot use ClaimCenter web service (SOAP) APIs from within authentication plugins.

WARNING Do not attempt to use ClaimCenter web service (SOAP) APIs from within authentication plugins because doing so requires authentication.

Getting Username and Password From Built-in PCF Login Page Using Attributes

In the reference ClaimCenter implementation, login-related PCF files set the username and password as attributes. 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 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;

    // 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;
  }
}
```

Getting Username and Password From Headers from 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("Invald 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 for this example, refer to

```
ClaimCenter/java-api/examples/plugins/authenticationsourcecreator/
```

User Authentication Service Plugin

An authentication service plugin (`AuthenticationServicePlugin`) 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.

You can use the following mechanisms to log in to ClaimCenter:

- Log into the web application user interface, as a standard user.
- Log into the server through Guidewire Studio.
- From a SOAP API call, log in as a user with SOAP API privileges.

The `AuthenticationServicePlugin` **must** handle all three types of logging in. For Guidewire Studio users and SOAP API calls, the credentials passed to this plugin is 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.** An authentication plugin 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 by 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 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 `ShouldSynchUserRolesInLDAP`. 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` method call to the configured JAAS provider. The provider in turn calls back to the `JAASCallbackHandler` 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.

SOAP API Limitations

You cannot use ClaimCenter web service (SOAP) APIs from within authentication plugins.

WARNING Do not attempt to use ClaimCenter web service (SOAP) APIs from within authentication plugins because doing so would require authentication.

SOAP API User Permissions and Special-Casing Users

Guidewire recommends creating a separate 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 that this user have few permissions or no permissions in the web application user interface.

From an authentication service plugin perspective, 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 'Authenticate' Method

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.check(_handler != null ? null : "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 "The Messaging Editor" on page 161 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 (not the plugin registry), there is a `sessiontimeoutsecs` parameter that configures how long much 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 by using the API, for example, do not have an HTTP session, only an application session.

The following is an example of this session timeout parameter:

```
<security sessiontimeoutsecs="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 username and password substitutes into the database configuration file anywhere that `${username}` or `${password}` are found in the database parameter elements.

> **Note:** *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 by using the standard `setParameters` method of `InitializablePlugin`. For more information, see "Deploying Gosu Plugins" on page 111.

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 “The Messaging Editor” on page 161 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.

ABAuthenticationPlugin for ContactCenter Authentication

Typical authentication from ClaimCenter to ContactCenter uses the name and password defined in the `IAddressBookAdapter` plugin in Studio for the ClaimCenter application. See “Installing and Integrating ContactCenter with QuickStart” on page 12 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 ContactCenter version is called `ABAuthenticationPlugin`.

IMPORTANT To use standard authentication to connect ClaimCenter to ContactCenter, do **not** implement `ABAuthenticationPlugin`. Instead, define the username and password in the `IAddressBookAdapter` web service configuration, as described in "Installing and Integrating ContactCenter with QuickStart" on page 12.

To use custom authentication from ClaimCenter to ContactCenter

1. Write a `ABAuthenticationPlugin` plugin implementation with the same design as described in the previous section about database authentication plugins, "Database Authentication Plugins" on page 245.
2. In Studio, register the `ABAuthenticationPlugin` plugin implementation as a Gosu plugin. See "Plugin Overview", on page 101 for more information.
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 ContactCenter (`IAddressBookAdapter`) detects the `ABAuthenticationPlugin`. The plugin ignores the web service default connection authentication information (in the Studio user interface). Instead, ClaimCenter calls your `ABAuthenticationPlugin` to get the user name and password.

For more information about implementing this plugin, see "Database Authentication Plugins" on page 245. The `ABAuthenticationPlugin` plugin uses the same approach as the database authentication plugins.

chapter 11

Document Management

ClaimCenter provides a user interface and integration APIs for creating documents, downloading documents, and generating automated form letters. These APIs include the ability to integrate the ClaimCenter document user interface to a separate corporate document management system that can store the documents and optionally the document metadata.

IMPORTANT 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 101. For the complete list of all ClaimCenter plugins, see "Summary of All ClaimCenter Plugins" on page 106.

This topic includes:

Document Management Overview

The ClaimCenter user interface provides a **Documents** section within the claim file. This user interface section lists documents such as letters, faxes, e-mails, and other attachments that a document management system (DMS) contains.

For example, suppose that business rules decide to send a notification fax to an insured customer. ClaimCenter can save a copy of the fax as a document and attach it to the claim file for later reference. Documents could also include incoming paper or images such as witness reports, photographs, scans of police reports, or signatures.

ClaimCenter provides interfaces for generating documents (document publishing / creation) and interfaces for interacting with an external document repository.

Production

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*, which means the document contents are returned immediately from the creation methods)
- *asynchronous production*, which means 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 production modes:

- raw document contents (an *input stream*)
- a web page containing an ActiveX control
- Jscript code to run on the user's machine. Use JScript for Windows client-side document production, such as for Microsoft Word documents, Microsoft Excel documents, and client-side PDF creation.
- a URL that can display the content (perhaps in an Active/X control) from a local content store.

These settings are defined as the `ResponseType` property within `DocumentContentsInfo`, which is the return result from synchronous production methods. Possible response types include `DOCUMENT_CONTENTS`, `HTML_PAGE`, `JSCRIPT`, or `URL`.

The details of these plugins are discussed in "Document Production" on page 252. For locations of templates and descriptors, see "TemplateSource Reference Implementation" on page 266.

After document production, storage plugins store the document in the document management system. ClaimCenter provides a user interface for the user to edit the new document, discussed further in "User Editing and Uploading" on page 251.

Storage of Contents and Metadata

Your document storage plugins provide ClaimCenter an interface to the document storage and retrieval APIs for your corporate document management system. From a user perspective, these plugins allow ClaimCenter users to view and print documents from a central document repository.

ClaimCenter separates document storage/retrieval tasks into two separate components based on the type of information, and this information is managed by two separate plugin interfaces.

- One plugin handles *document metadata.* Metadata might include document names, MIME types, permissions information, and the set of files available on the document management system.
- Another plugin handles *document content*. Content might be a fax image, a Microsoft Word file, a photograph, a PDF representation of an outgoing email, or any other attachment that represents incoming or outgoing documents.

Just as in document production, there are several document retrieval modes:

- raw document contents (an *input stream*)
- a web page containing an ActiveX control

- Jscript code to run on the user's machine. Use JScript for Windows client-side document production, such as for Microsoft Word documents, Microsoft Excel documents, and client-side PDF creation.
- a URL that can display the content (perhaps in an Active/X control) from a local content store.

The details of these plugins are discussed in "Document Storage" on page 270.

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 the 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** document to the document management system without bringing documents into application server memory. For more information, refer to "Asynchronous Document Storage" on page 276.

IMPORTANT For maximum responsiveness with a slow external document storage system, use asynchronous document storage.

User Editing and Uploading

You can edit documents (whether they are new files or other files) using the ClaimCenter user interface in the document file. After you click **Edit** in the document file user interface, ClaimCenter retrieves the file from the document content storage system and delivers it to the your local system.

After you edit the file with a local application, you must upload the document from the local system to ClaimCenter. An included web browser Active/X control called `GuidewireDocumentAssistant` makes this process easier for users. Although this control is optional, it simplifies document uploading by enabling the user's web browser to find the file to upload because it is in a known local temporary directory. If the Active/X control is not used, a file can still upload to ClaimCenter. However, the user must manually browse the local file system to find the file on the local disk in the web browser application.

The user interface code, which you can customize, controls whether you can edit a document. To prevent some users from editing documents, or to make some documents editable but not others, edit the PCF files. Specifically, PCF files that include document management features must provide new logic to hide the **Edit** button if appropriate. See the *ClaimCenter Configuration Guide* and the PCF Reference Guide for more details about PCF configuration.

Included Plugins

ClaimCenter includes a simple reference implementation that can serve as a placeholder for a document management system but does not provide full document management system functionality. Instead, these plugins simply store documents on the ClaimCenter server's file system and provides them to callers on request. The metadata for the files are stored in the ClaimCenter database. This reference implementation is intended as an example only and is not meant to replace a full-featured document management system.

There are also built-in versions of document production plugins (implementations of `IDocumentProduction`) for common file types, such as Gosu templates, PDF files, and Microsoft Word files. You can use these built-in plugins or selectively override them with your own plugins that you register with ClaimCenter.

The details of the included plugins are discussed in “Built-in Document Production Plugins” on page 267.

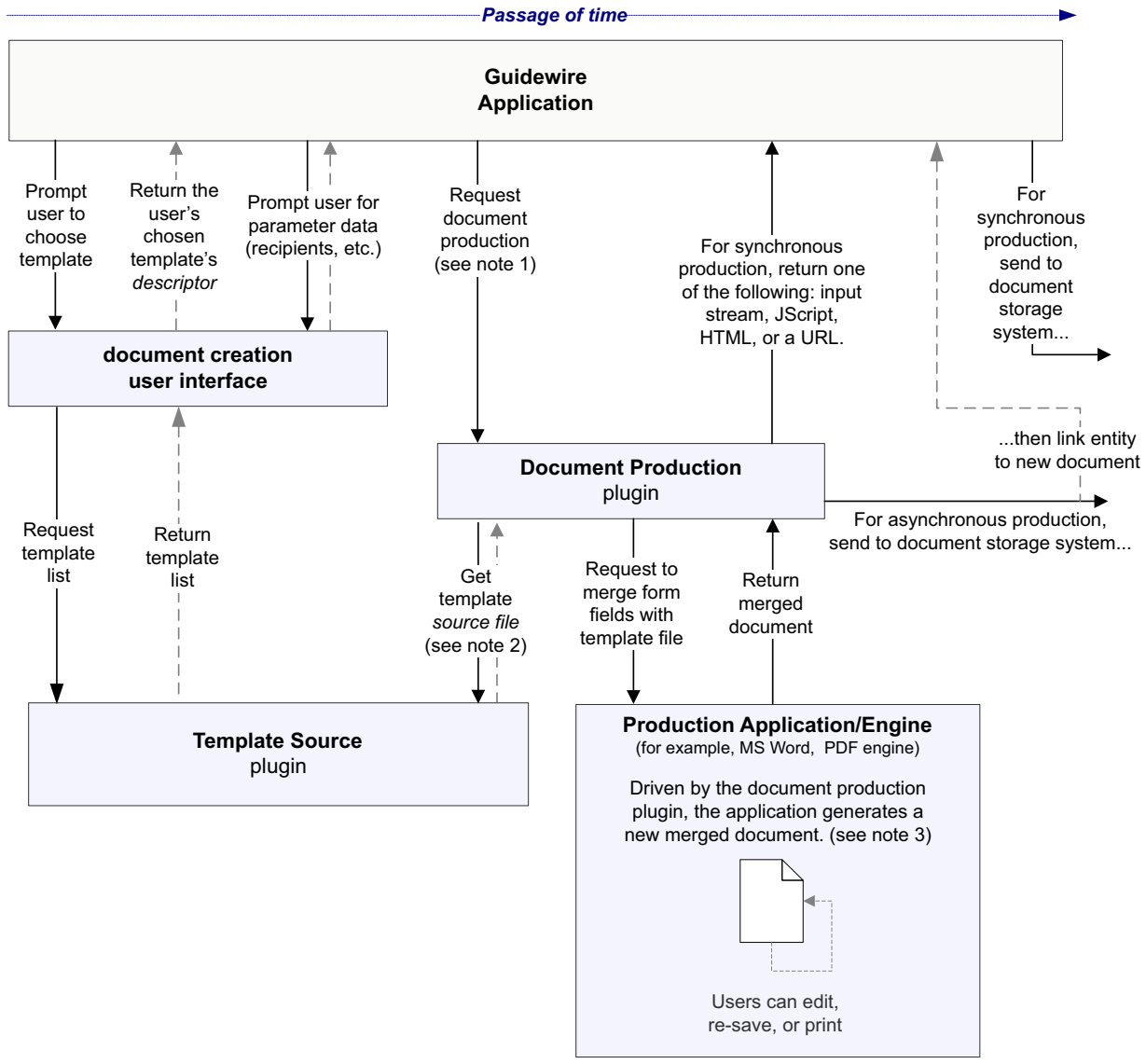
IMPORTANT The built-in document storage plugins are for demonstration only. For maximum document data integrity and document management features, you must use a complete commercial document management system (DMS). Implement new `IDocumentMetadataSource` and `IDocumentContentSource` plugins to communicate 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 the DMS or ClaimCenter built-in metadata storage, but not both. Avoid duplicating metadata in the ClaimCenter database itself for production systems.

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 diagram illustrates the interactions between ClaimCenter, the various services, an editor, and an external document management system to manage the generation of a form or letter. The Gosu-initiated automatic document creation would begin at the step labelled “Request document production”.

Typical Document Production Flow (Manual)



Notes

1. A *document production plugin* is chosen from values in the *template descriptor*, which could request a specific template handler string (which maps to a document production plugin based on settings in config.xml) or let one be chosen based on the specific file's MIME type (also configured in config.xml).

2. The document production plugin gets a template descriptor and the parameter data values, and is responsible for requesting the actual template file (such as the basic Microsoft Word file or the form letter PDF) from the *template source plugin*.

3. Document production may be on-server, but may be client-side production if the document production plugin generates JScript code to generate a file locally on a user's PC. Some built-in templates do client-side production: Word files, Excel files, and client-side PDF. For client-side production, the user then must upload the file afterward.

The chronological steps are as follows:

1. ClaimCenter invokes the *document creation user interface* to let the user 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 the user 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 chosen template's *template descriptor* information to ClaimCenter.
6. ClaimCenter prompts 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 the user enters 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 file's MIME type. The document production plugin gets a template descriptor and the parameter data values.
8. The document production plugin asks the template source for the actual template file.
9. The document production plugin takes whatever steps are necessary to launch an editor application or other "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 documents, Microsoft Excel documents, and client-side PDF document production. The plugins return JScript code to the client which calls the local application to open the file and merge in necessary fields.
10. If the document was created from the user interface, ClaimCenter automatically downloads the file to the user's local machine if the configuration parameter `AllowActiveX` is enabled. The file opens in the appropriate application on the user desktop. The user can print the file, and if the editor application supports editing, users can optionally change it and resave 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 the user's PC, the user uploads the completed local merged document (or lets the user choose a different document to upload). The user can then specify some additional parameters about the document. If the configuration parameter `AllowActiveX` is enabled, the upload step by the Guidewire Active/X control `GuidewireDocumentAssistant` supplies the location of the local merged document. If that parameter is not enabled or the control is not installed, then the user must manually browse 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 Storage" on page 270.

The steps earlier describe the reference implementation, but many parts can be customized.

IMPORTANT The user interface flow can work differently by customizing the user interface PCF configuration files and in some cases customizing the document-related plugins.

Document Production Plugins

As mentioned in “Document Production” on page 252, ClaimCenter supports *synchronous production* (the document contents return immediately) and *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 for and retrieving templates describing the document to be created. This 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 imlementation to set parameters in the default implementation.

The `IDocumentProduction` plugin has two main methods: `createDocumentSynchronously` and `createDocumentAsynchronously`.

- The `createDocumentSynchronously` method returns document contents as:
  - raw document contents (an *input stream*)
  - a web page containing an ActiveX control
  - Jscript code to run on the user’s machine. The Jscript code response type is useful for client-side document product, such as the built-in production plugins for Microsoft Word documents, Microsoft Excel documents, and client-side PDF creation.
  - a URL that can display the content (perhaps in an Active/X control) from a local content store.

  These settings are defined as the `ResponseType` property within `DocumentContentsInfo`, which is the return result from synchronous production methods. Possible response types include `DOCUMENT_CONTENTS`, `HTML_PAGE`, `JSCRIPT`, or `URL`.

  For synchronously-created documents, 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 asynchronously-created documents, your `createDocumentAsynchronously` method must put the newly created contents into the DMS. Next, your external system can use SOAP APIs to add the document (see the `IClaimAPI` interface) to notify ClaimCenter that the document now exists.

  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 documents, Microsoft Excel documents, and client-side PDF document production. The plugins display JScript code to the client which calls the local application to open the file and merge in necessary properties.

Depending on whether the client machine has Acrobat Viewer or full Acrobat, the user may or may not be able to save the resulting `.pdf` file. To save PDF changes, the full version of Adobe Acrobat is required, otherwise the user can just print the merged document.

For more information about the built-in production plugins, see "Built-in Document Production Plugins" on page 267.

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

The document production configuration is defined in the Plugins editor in Studio for the definition of the `IDocumentProduction` plugin. In the plugin definition, it includes parameters (as elements) that define a *template type* and a `IDocumentProduction` plugin implementation that handles production for that type of template. In this context, a *template type* is either a MIME type or an custom (arbitrary) *template production type* text value that you provide.

IMPORTANT For information about, see "Using the Plugins Editor" on page 141 in the *Configuration Guide*

Create new template types by adding parameters to this list, then providing a new `IDocumentProduction` implementation. Finally, deploy the class files in `ClaimCenter/modules/configuration/plugins/document/classes` and necessary libraries in `ClaimCenter/modules/configuration/plugins/document/lib`.

A typical document production configuration includes references to the built-in `IDocumentProduction` implementations for common file types. For example, for parameter `application/msword` it has the value `com.guidewire.cc.plugin.document.internal.WordDocumentProductionImpl`. Refer to Studio for the built-in settings for other production types.

ClaimCenter determines which `IDocumentProduction` implementation to create a document with the following procedure:

1. First ClaimCenter looks for an entry in this list that matches the template's document production type stored in its `documentProductionType` property.
2. If no match was found yet, ClaimCenter tries to find the template's `mimeType` as a listed template type in this list. The example earlier lists MIME types not document production types.
3. If no match was found yet, document creation fails.

If desired, you can also replace the default dispatching implementation with your own `IDocumentProduction` implementation.

Note also that ClaimCenter supports both server-side PDF generation (which is the default) and client-side PDF generation. The client-side implementation can be enabled by changing `ServerSidePDFDocumentProductionImpl` in the configuration under the parameter with the name set to `"application/pdf"` to the client side built-in plugin value:

```
com.guidewire.cc.plugin.document.internal.ClientSidePDFDocumentProductionImpl
```

Also, be aware that the `ServerSidePDFDocumentProductionImpl` uses the `IPDFMergeHandler` plugin interface.

For more details on the built-in classes, see the “Built-in Document Production Plugins” on page 267.

Adding a custom MIME type such that it is recognized by ClaimCenter requires several steps:

1. In Guidewire application’s `config.xml` file, add the new MIME type to the `<mimetypemapping>` section. The information in this section consists of four items:
   - The name of the MIME type, which is the same as the identifying string, such as `"text/plain"`.
   - The file extensions to use for the MIME type. If more than one extension applies, separate extensions with a pipe symbol (`"|"`). ClaimCenter uses this information to map both from the MIME type to the file extension, and from the file extension to the MIME type. To map from the MIME type to the file extension, ClaimCenter uses the first extension in the list. To map from extension to type, ClaimCenter uses the first `<mimetype>` entry containing that extension.
   - The image file path for documents of this MIME type. At runtime, this image must be in the *`SERVER`*`/webapps/cc/resources/images` directory.
   - A human-readable description of the MIME type.
2. Add the MIME type to the configuration of the application server, if required. This varies based on the details of the web application server configuration. For Apache Tomcat, configure MIME type information in the Tomcat `web.xml` configuration file. Within that file’s `<mime-mapping>` tags, ensure that the MIME type you need already exists correctly in this list, or add it.

Server-side PDF Licensing

In the ClaimCenter reference implementation, the server-side PDF generation software is made by a company called Big Faceless Org (BFO). Without a license, the generated documents contain a large watermark `"DEMO"` on the face of each generated page, rather than preventing document creation entirely. To remove the “demo” watermark, you must obtain a license key through Guidewire Customer Support.

All server-side PDF generation licenses are licensed *per server CPU*, not per server.

With your copy of ClaimCenter, you are entitled to **four CPU** licenses for PDF generation. 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` with your request for a PDF generation 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`.

Configuration

The default configuration appears in the Plugins registry in Studio.

IMPORTANT For information about using the plugins editor, see “Using the Plugins Editor” on page 141.

Set the Java class to `com.guidewire.pl.plugin.document.internal.BFOMergeHandler` and the following two parameter values:

- Parameter `BatchServerOnly` set to `true`
- Parameter `LicenseKey` set to your server key. The `LicenseKey` value is empty in the default shipped configuration, since you must acquire your own BFO licenses from customer support. The `BatchServerOnly` parameter determines whether PDF generation happens on each server or solely on the batch server.

ClaimCenter uses this plugin interface only for templates that are configured to use server-side PDF generation in the `IDocumentProduction` configuration. In other words, only templates set to use the class `ServerSidePDFDocumentProductionImpl`. Client-side PDF document production never uses the BFO library.

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 “TemplateSource Reference Implementation” on page 266.

Template Descriptor Fields : 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()`.

Template Descriptor Fields : 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 comapared 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 : 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 evaluations to an array, typically an array of Guidewire entities although that is not required. Also be aware that 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
EntityName	The name of any ClaimCenter entity such as `Claim`, `Exposure`, or `Activity`. 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.
`Entity`	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 `"Entity"`. This is useful for heterogeneous lists of objects.
TypekeyName	The name of any ClaimCenter typekey such as `"YesNo"`.
`string`	A case-sensitive all-lower-case string to appear in the user interface as a single line of text. The `<DefaultObjectValue>` tag must be present, and its contents indicates the *default* text for this context object. Ignores the `<PossibleObjectValues>` tag.
`text`	Case sensitive, must be all lower case. Appears in the user interface as several lines of text. The `<DefaultObjectValue>` tag must be present, and its contents indicates the default text for this context object. If you use this, ClaimCenter ignores the `<PossibleObjectValues>` 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 : Field Names 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 258. 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 : Context Objects” on page 260).

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 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 a 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: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 *ClaimCenter 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.

TemplateSource Reference Implementation

The ClaimCenter reference implementation keeps a set of document templates on the ClaimCenter application server. It initially resides in the directory:

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

At run time, the source directory typically is at the path:

```
SERVER/webapps/cc/modules/configuration/config/resources/doctemplates
```

However, for more security, move this template to a location outside of the `webapps/cc` directory. See the *ClaimCenter Configuration Guide* for details on how to change this.

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: • how to search and find this template • form fields that define what information populate each merge field • form fields that define context objects • form fields that define root objects to merge

To set up a new form or letter, you need to create both files and deploy them to the `templates` directory (mentioned earlier) on the web server.

For details on the XML formatting of the template descriptor, see “Document Template Descriptors” on page 258 and “XML Format of Built-in IDocumentTemplateSerializer” on page 262.

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)	There are two types of Acrobat document production. One type is server-based, and requires a license to some server-based software that works with ClaimCenter. Another type is client-based PDF production. Client-based PDF production requires a full Acrobat license for Windows. In both cases, users can view PDF files with the free Acrobat Reader application. However, client-based modification of the PDF template or of PDF files on the desktop requires the full version of Acrobat. See "Server-side PDF Licensing" on page 257 for details.
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 `FormField` 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 279. For help on Gosu syntax, see the *Gosu Reference Guide*.
Inetsoft reports	ClaimCenter includes a built-in integration with the Inetsoft Reporting software. That built-in integration includes a document production plugin that can generate a document from the Inetsoft reports.

Document Templates from Reports

You can generate document templates from InetSoft Reports. for more information, see *ClaimCenter Reporting Guide*.

Generating Documents from Gosu

ClaimCenter can generate PDF 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 Microsoft Word template, Excel templates, or client-side PDF creation. In all these cases, the document production plugin returns JScript that runs on the user's computer to generate the new document locally. While business rules run, it is possible no user is logged into 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, most Word documents convert to the text-based RTF format and many Excel templates can convert to CSV files. Therefore, most Microsoft documents can convert into equivalent Gosu templates that can generate text. 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 "Document Production" on page 252. 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`.

The new document uses the latest in-memory entity at the time the rule **runs**, not the persisted stored database version of the entity.

IMPORTANT The new document always uses the latest in-memory entity at the time the rule runs. Remember that this may be different from the persisted version.

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 transac-

tion 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. On a related note, see “Cached Document UIDs” on page 269.

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)
```

Note: For more examples of creating documents from Gosu, see “Document Creation” on page 151 in the *Rules Guide*.

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 information on these the `DocumentProduction` APIs, see the *ClaimCenter Rules Guide*.

Cached Document UIDs

If a new document is created and an error occurs within the same database transaction, the error typically ensures the database transaction rolls 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 document’s location in the external system. This information is stored in the `Document` entity in ClaimCenter in the same transaction so in this case was not committed to the database. The new document in the external system is effectively *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.

Document Storage

ClaimCenter separates document storage and retrieval tasks into two separate components that are managed by two separate ClaimCenter plugin interfaces:

- **Document metadata handling.** A *document metadata source* plugin manages the document metadata such as the file name, the file's MIME type, and the set of files available on the document management system. This type of plugin would be based on `IDocumentMetadataSource`.
- **Document content handling.** A *document content source* plugin stores and retrieves the document content from the document management system. This type of plugin would be based on `IDocumentContentSource`. Even though document content is the primary role of this plugin, this plugin also uses and optionally stores some document metadata (as discussed later in this topic).

ClaimCenter also provides document-related Gosu APIs so that business rules can add new documents to certain types of objects (including a claim). For more information about these Gosu APIs, see the *ClaimCenter Rules Guide*.

ClaimCenter also provides a web services API for adding a reference to a document to certain types of objects (including a claim). This allows some external process and document management systems to work together to inform ClaimCenter after creation of new documents related to a claim. For example, in paperless operations, new postal mail might come into a scanning center to be scanned and identified with a claim, and then is loaded into an electronic repository.

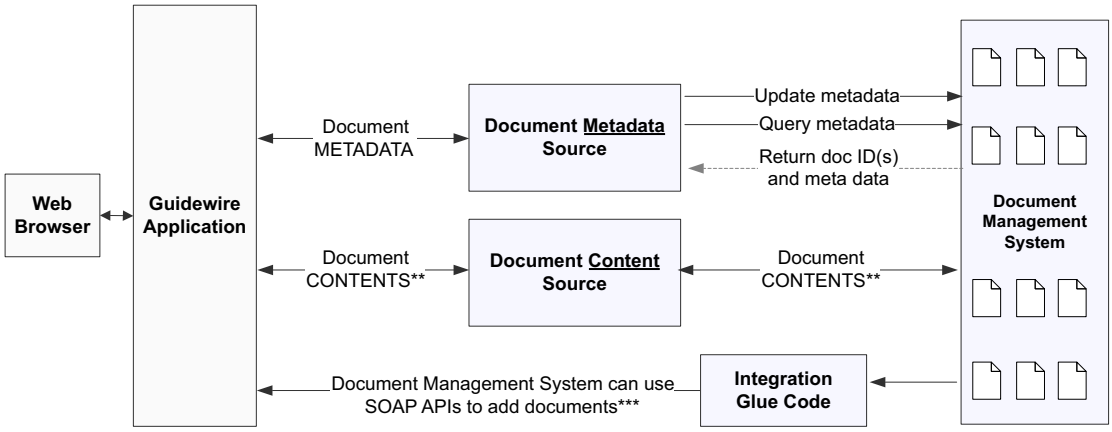
The repository could then notify ClaimCenter of the new document and create a *new mail activity* to review the new document. Similarly, after sending outbound correspondence (outbound e-mails or faxes), ClaimCenter can add a reference to the new document. After the external document repository saves the document and assigns an identifier to retrieve it if needed, ClaimCenter stores that document ID.

The `IClaimAPI` interface includes several methods to add documents to ClaimCenter objects:

- `addDocument` - add document to a claim
- `addClaimantDocument` - adds a document to a claimant
- `addClaimContactDocument` - add document to a claim contact
- `addExposureDocument` - add document to an exposure
- `addMatterDocument` - add document to a legal matter

The following diagram summarizes these key plugins and interfaces related to document storage:

Document Storage Architecture



** The “document contents” could be any of the following: (1) complete data of the full document’s contents, which could potentially be very large; (2) HTML contents that reference an ActiveX control; (3) Jscript to be executed on the client machine; (4) a URL to a content store on intranet/Internet.

*** If a Document Management System (DMS) adds a document on its own, the DMS will notify the Guidewire application of the new document’s document ID and its metadata.

Note: For details of the web services APIs, refer to the SOAP API Reference Javadoc in the package `com.guidewire.cc.webservices.api`. Specifically, refer to the `IClaimAPI` method `addDocument`, as well as the definition of the `Document` SOAP entity.

Implementing a Document Content Source

Document storage systems vary quite a bit in terms of how documents transfer or display in the user interface. To support this variety, ClaimCenter multiple document retrieval modes called *response types*:

- The document contents as an *input stream*, which can be raw binary data
- A web page containing an ActiveX control
- Some Jscript code to run on the user’s machine
- A URL to a page that can display the document (perhaps using an ActiveX control) from a local content store

To implement a new document content source, there are several methods to implement in the `IDocumentContentSource` plugin interface. The following sections describe the methods your plugin must implement.

Adding Documents (And Optionally, Metadata)

A new document content source plugin must fulfill an “add document” request from ClaimCenter by implementing the `addDocument` method. This method adds a new document object to the repository, taking as inputs the document metadata in an `Document` object and the document contents as an `InputStream`.

This method may optionally also store the metadata contained in the `Document` entity. Independent of whether the document stores the metadata, the plugin must update metadata properties within the `Document` object. Specifically, it must set an implementation-specific document universal ID (`DocUID`) and the modified date (`DateModified`). Changes that the method makes to the document metadata in the `Document` object persist in the ClaimCenter database.

Typically the method returns `false`, indicating that ClaimCenter stores the metadata using the separate plugin interface specifically for document metadata. The method returns `true` if (and only if) it already stored (or promises to store) the document metadata in the `Document` entity in addition to storing the document contents. ClaimCenter includes a simple default behavior for storing document metadata in the local database. Important limitations with this approach are discussed in "Included Document Storage Plugins" on page 275.

If the plugin encounters errors during storage before returning from this method, the method throws an exception. If document storage errors are detected later, such as for a asynchronous document storage, the document content storage plugin must handle errors in an appropriate fashion. For example, the plugin could send administrative e-mails, or it could create new activities using SOAP APIs to investigate document creation 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.

Retrieving Documents

A new document content source plugin must fulfill a "retrieve document" request from ClaimCenter by implementing the `getDocumentContentsInfo` method. This method takes a `Document` object as a parameter and the plugin can simply use the `docUID` property within the `Document` to identify the target document contents within the repository. However, the plugin could use additional data model extensions on `Document` if desired. The plugin can read the `Document` entity parameter but must not modify it.

The `getDocumentContentsInfo` method also includes a Boolean parameter *`includeContents`*. This parameter is `true` if the caller wants the result object to include an input stream with the contents of the document. This is `false` in situations in which only the metadata about the document contents (like the MIME type or the type of contents) is required. In this case, the overhead of actually transmitting the document contents is unnecessary.

The plugin must return its results within a container object called `DocumentContentsInfo`, which is really a simple container containing the following information:

- **Response type.** An enumeration indicating the type of the contents returned:
  - raw document contents (`DOCUMENT_CONTENTS`)
  - a web page containing an ActiveX control (`HTML_PAGE`)
  - Jscript to be executed on the client machine (`JSCRIPT`)
  - A URL (`URL`) to a local content store to display the content, perhaps in an ActiveX control
- **MIME type.** For all response types other than `URL`, the MIME type describes the type of contents (such as `application/msword`) or the MIME type of the HTML document (`text/html`).
- **Hidden target frame as Boolean.** For all response types other than `DOCUMENT_CONTENTS`, the Boolean value `TargetHiddenFrame` indicates ClaimCenter to open the web page in a hidden frame within the user's web browser. For example, use this if the document repository is naturally implemented as a small web page that contains JavaScript. That JavaScript might opens another web page as a popup window that displays the real document contents.
- **Contents as input stream.** The document contents in the form of an input stream, which is raw data as an `InputStream` object. In the case where the *`includeContents`* parameter is `false`, as discussed earlier in this section, set the `inputStream` property in the returned `DocumentContentsInfo` to `null`.

WARNING It is important to note that ClaimCenter calls some methods in a `IDocumentContentSource` plugin twice for each document: `isDocument` and `getDocumentContentsInfo`. Ensure that your plugin supports this design. For the `getDocumentContentsInfo` method, the *`includeContents`* parameter is set to `false` on the first call and true on call to the method. It is important that your plugin ensures that for these pair of calls, your plugin **must** return a `DocumentContentsInfo` with the same type. Mixing these result is unsupported and results in in undefined behavior.

Checking for Document Existence

A new document content source plugin must fulfill a “document existence check” request from ClaimCenter by implementing the `isDocument` method. This method takes a `Document` object as a parameter and the plugin might just use the `docUID` property within the `Document` to identify the target document within the repository. However, the plugin could use additional data model extensions on `Document` if desired. The `Document` entity parameter must not be modified.

WARNING It is important to note that ClaimCenter calls some methods in a `IDocumentContentSource` plugin twice for each document: `isDocument` and `getDocumentContentsInfo`. Ensure that your plugin supports this design.

Removing Documents

A new document content source plugin must fulfill a “remove document” request from ClaimCenter by implementing the `removeDocument` method. This method takes a `Document` object as a parameter and the plugin might just use the `docUID` property within the `Document` to identify the target document within the repository. However, the plugin could use additional data model extensions on `Document` if desired.

This method is called to notify the document storage system that the `Document` entity which is passed in is about to be removed from metadata storage. It is up to the `IDocumentContentSource` plugin implementation whether or not to delete the content, retire it, or another action. Other `Document` entities may still refer to the same chunk of content.

WARNING Choose carefully whether to delete the content, retire it, or another action. Other `Document` entities may still refer to the same actual content.

Be careful writing your document removal code. If the `removeDocument` implementation does not handle metadata storage, then some server problem might cause removal of the metadata to fail after removing document contents.

Return `true` if the method removed the document metadata from the metadata storage and no further action is required by the application. Otherwise, return `false`.

Updating Documents (And Optionally, Metadata)

A new document content source plugin must fulfill a “update document” request from ClaimCenter by implementing the `updateDocument` method and a new input stream (`InputStream`) for the contents. This method takes document metadata in a `Document` object and the new document contents as a new `InputStream`.

Whether or not the metadata is stored, the DMS system must update any appropriate properties, including version information (if any) and `DateModified`. The DMS must update the same set of properties as in the plugin’s `addDocument` implementation. Any metadata to describe version-related information is optional but recommended. The plugin implementation is required to handle this internally. The built-in `Document` entity does not contain version information. Your plugin implementation may create or set version-related properties on the `Document` entity. As with all data model extensions, ClaimCenter persists those changes to the `Document` entity.

IMPORTANT If the document storage system stores version information, the plugin can update version properties in `Document` during document update. There are no built-in versioning properties or behavior built-in automatically. However, changes to the `Document` entity during update persists.

Implementing a Document Metadata Source

Document metadata can optionally be handled by the *document source content plugin* so that it handles the content and the metadata. Alternatively, use the document metadata source plugin interface to implement systems that split document *metadata* from the document *content storage*. For more information about the docu-

ment source content plugin, see “Implementing a Document Content Source” on page 271.

It is important that in your `IDocumentMetadataSource` plugin implementation that you do not track documents using the document’s `Id` property (`Document.Id`). Each time the document exists it has a different `Id` value. Instead, use the public ID property (`Document.PublicID`).

WARNING It is critically important that you always track documents using the `PublicID` property, not the `Id` property.

To create a new `Document` object in Java, use the `EntityFactory` class discussed in “Java Entity Utility APIs” on page 121 in the *Gosu Reference Guide*. For related information about the Java *entity libraries*, see “Java Entity Libraries Overview” on page 114 in the *Gosu Reference Guide*.

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);
```

To implement the metadata plugin separately, you can implement the `IDocumentMetadataSource` plugin. This plugin’s required methods include:

- `saveDocument` - Persist document metadata in a `Document` object to the document repository.
- `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.

For more details, refer to the API Reference Javadoc documentation for `IDocumentMetadataSource`.

For more information about the included reference implementation, see “Included Document Storage Plugins” on page 275.

Included Document Storage Plugins

ClaimCenter includes simple reference implementations for the `IDocumentContentSource` plugin which does not provide full document management system functionality. Instead, this plugin simply stores documents on the ClaimCenter server's file system. This reference implementation is intended as an example only and is not meant to replace a full-featured document management 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. No built-in plugin exists for the `IDocumentMetadataSource` interface. The default behavior described earlier is simply the internal ClaimCenter behavior if you do not register a `IDocumentMetadataSource` plugin implementation.

IMPORTANT The built-in document storage plugins are for demonstration only. 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.

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 associated with	Directory naming	Example
Claim	`Claim` + *`claimNumber`*	`/documents/Claim02-02154/`
Exposure	*`claimDirectory`*`/Exposure` + *`exposurePublicID`*	`/documents/Claim02-02154/ExposureABC:01/`
Claimant	*`claimDirectory`*`/Claimant` + *`claimantPublicID`*	`/documents/Claim02-02154/ClaimantABC:99/`

Note: 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** document 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:

- **Asychronous 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 asychronous 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.

IMPORTANT The built-in document storage plugins are for demonstration only. 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 Asychronous Document Storage

To configure asychronous 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. Instad, 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 `SynchedContentSource` 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 `TrySynchedAddFirst` 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.

Rendering Arbitrary Input Stream Data Such as PDF

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 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.

chapter 12

Data Extraction Integration

ClaimCenter provides several mechanisms that generate messages, forms, and letters with ClaimCenter data into custom text-based formats. If an external system needs to get information about a claim from ClaimCenter, it can send requests using the Internet or intranet over the HTTP protocol to the ClaimCenter server.

This topic includes:

Why Templates are Useful for Data Extraction

Incoming data extraction requests includes what template to use and what information the requestee must pass 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 also 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.

Possible output formats might include:

- A plain text document with `name=value` pairs.
- A structured 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. If a template includes Gosu code, the template can dynamically generate output as needed.

Templates with Gosu code provide several advantages over a fixed pre-defined format:

- You can specify the required output properties so you can send as few properties as you want. In a fixed format that, all properties on all subobjects might be required to support unknown use cases.

- Responses can match the calling system's native format by customizing the template. This avoids additional code to parse and convert fixed-format data.
- Templates can directly generate HTML for custom web page views into the ClaimCenter database. Generate HTML either within ClaimCenter or from linked external systems such as Intranet pages that want to query ClaimCenter and display the results within its own user interface.

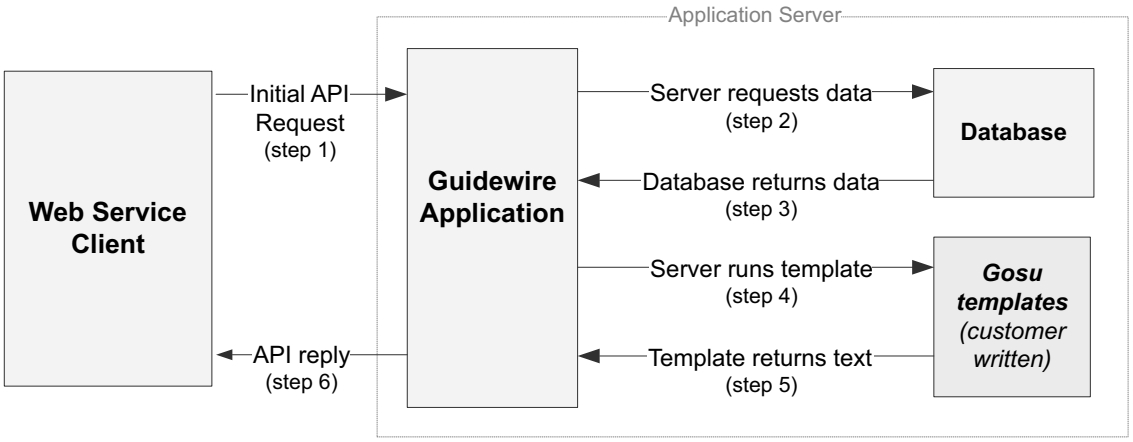
In ClaimCenter 6.0.8, there are two basic mechanisms for data extraction:

The major techniques for ClaimCenter data extraction:

- **PCF template page data extraction.** ClaimCenter PCF page configuration files include a special element called `<TemplatePage>`, which includes an element called `<Template>` which can include HTML and Gosu code. To make the template available from outside users, you must configure entry points, which create an externally-viewable URL that triggers this page. For details, see "PCF Template Page Data Extraction Overview" on page 280.
- **Design custom web services for data extraction.** Write custom web services that address each specific integration point in your network. Remember the following design principles:
  - Consider using different web service interfaces for each integration point.
  - Name your methods in your API appropriate for the integration point, rather than a general purpose template API. For example, if the template is for generating notification emails, name your method `getNotificationEmailText`.
  - Design your APIs to use method arguments with types specific to each integration point.
  - In your implementation, you can use Gosu templates. 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 API.

This diagram and the step-by-step descriptions after the diagram explain this process:

Web Service Data Extraction Flow



PCF Template Page Data Extraction Overview

To extract data from ClaimCenter, a calling system makes an HTTP request to the ClaimCenter server. ClaimCenter reads the request parameters, retrieves the necessary data, passes the data to the template engine for processing and formatting, and finally returns the template results to the calling system.

Make a New PCF Template Page

The first step is to design a new page configuration file with a `<TemplatePage>` element, which includes an element called `<Template>`. The `<Template>` element contents must contain a template of HTML or other plain text format with embedded Gosu code. Embed Gosu code in the same way you embed Gosu code into a normal a Gosu template.

For more information about the `<TemplatePage>` element, see the *ClaimCenter Configuration Guide*.

Register a New Entry Point

To make the page accessible to users with its own URL, create a ClaimCenter *entry point*. For details on creating entry points, see the *ClaimCenter Configuration Guide*.

Given this entry point you can then access the template page with authentication using a URL of the form:

```
http://server:port/cc/MyTemplatePage.do?loginName=abc&loginPassword=gw
```

Or, with additional text-based custom parameters:

```
http://server:port/cc/MyTemplatePage.do?loginName=abc&loginPassword=gw&claimID=cc:123
```

Data Extraction Gosu Template Integration

IMPORTANT This section is an overview as it relates to integration only. For the Gosu language and full Gosu template definitions, see “Gosu Templates” on page 291 in the *Gosu Reference Guide*. In particular, that section goes into detail about how to **pass parameters** to Gosu template files.

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 claim as a root object for a template, you could design different templates to export policy data such as:

- A list of all notes on the claim
- A list of all open activities on the claim for a person
- A summarized view of a claim for an agent

To provide programmatic access to ClaimCenter data, ClaimCenter uses the Gosu engine, the basis of Guidewire Studio business rules. This general-purpose template engine allows you to embed Gosu code that generates dynamic text from the results of the Gosu code. Simply wrap all Gosu code (even partial expressions) with `<%` and `%>` characters for a Gosu block and `<%=` and `%>` for a Gosu expression whose return result to export.

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`-- the loss date
- `claim.LossType.Code` -- the code for a typelist value
- `claim.LossType.Name` -- the text description of the typelist value
- `claim.Policy.PolicyNumber` -- the policy number (different from its public ID)

The simplest Gosu expressions simply extract data object properties such as `claim.ClaimNumber`. For instance, suppose you want to export the claim number from a claim’s *public ID* string. 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 HO-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. That is why 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`.

For More Template Information

To learn more about Gosu templates, see “Gosu Templates” on page 291 in the *Gosu Reference Guide*.

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();
```

For information about calling Java from Gosu, see “Java and Gosu” on page 101 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 typelist mapping tools. ClaimCenter provides access to this using the `$typecode` object:

```
typecode.getAliasByInternalCode("LossType", "ns1", claim.LossType)
```

This `$typecode` API works **only** within Gosu templates, not Gosu in general.

There is a Gosu API called `TypecodeMapperUtil` you can use from Gosu. These APIs (including web services you can call from external systems) are discussed in “Mapping Typecodes to External System Codes” on page 81.

chapter 13

Archiving Integration

ClaimCenter supports archiving a claim as a serialized stream of data. You can choose to store the data in a file, in a document storage system, or in a database (as a single large binary object). The data format is a XML document.

IMPORTANT This topic discusses *plugins*. Plugins are software modules that ClaimCenter calls to perform an action or calculate a result. For information on plugins, see "Plugin Overview", on page 101. For the complete list of all ClaimCenter plugins, see "Summary of All ClaimCenter Plugins" on page 106.

This topic includes:

Overview of Archiving Integration Flow

ClaimCenter supports archiving a claim as a serialized XML document. You can choose to store the data in a file, in a document storage system, or in a database as a single large binary object.

The high-level integration flow for archiving storage:

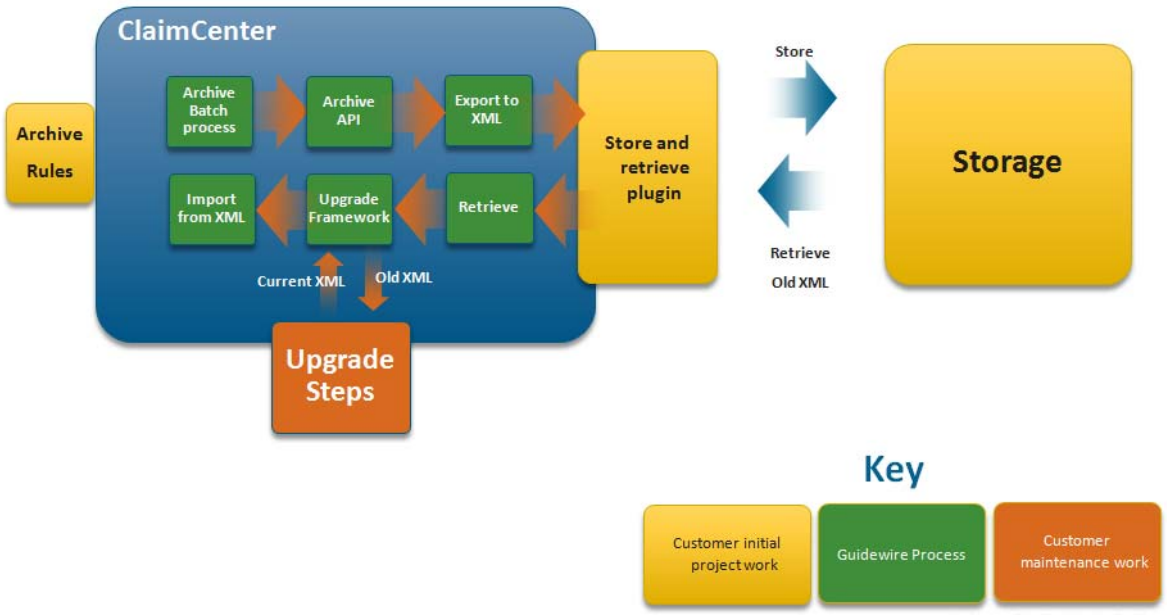
1. The ClaimCenter archiving batch process runs.
2. ClaimCenter encodes the claim into an in-memory XML representation.
3. ClaimCenter serializes the XML data and calls a customer-written plugin that sends archive data to an external system.

4. 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 restore.
2. ClaimCenter finds the associated info object (`ClaimInfo`) for the claim.
3. Based on information in the info object such as the claim number, ClaimCenter tells the archiving storage plugin to retrieve the archived data.
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 Restored Data” on page 292
6. ClaimCenter converts the temporary restored entity 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.



The Archive Source Plugin

Customers using archiving must implement the `IArchiveSource` plugin interface. The responsibility of this plugin is to actually store and retrieve archive data from the backing store, which typically is on a different physical computer. The backing store could be anything that you want. Common storage choices are:

- Files in a file system
- A full document management system containing all the data
- A database as a large binary object containing the XML document

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.

IMPORTANT Guidewire provides an archive source plugin implementation for **demonstration** purposes only. Do not use this in a production system. You must implement an archiving plugin that meets your specific business needs.

The superclass of the built-in plugin implementation is the class `gw.plugin.archiving.ArchiveSource`.

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 (`RootInfo`).

A root info object encapsulates a summary of a root entity (in ClaimCenter, the root entity is always a `Claim`), including its archiving status. `RootInfo` is an interface. In ClaimCenter, the concrete class that implements this is the entity called `ClaimInfo`. See the Data Dictionary for `ClaimInfo` for more information.

IMPORTANT In the APIs or the documentation where you see the type `RootInfo`, the concrete class for ClaimCenter is the entity called `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 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 error conditions.

For example, the plugin methods `storeFinally` is always called outside the main database transaction for the storage request. If the storage request fails, all ClaimCenter data changes do not commit to the database. However, any changes you make during the call to `storeFinally` will persist to the database.

WARNING Be extremely careful that you understand potential failure conditions and the relationship between each archive method and database transactions.

For details for each plugin method, refer to the reference information, grouped by purpose of the methods:

- "Archive Plugin Storage Methods" on page 289
- "Archive Plugin Retrieve Methods" on page 291
- "Archive Plugin Utility Methods" on page 291

Archiving Storage Integration

ClaimCenter implements archive integration as a work queue.

Archive Writers and Workers

The batch process called Archiving Item Writer finds `Claim` objects that are potentially eligible for archiving. The batch process creates a work item as a row in the database for every `Claim` object that is eligible for archiving.

It is possible that the archive store is unavailable due to network problems or configuration issues. The archive source plugin returns its status in its `getStatus` method. If the archive store is unavailable for any reason, then the writer work process logs an information message but does **not** queue any claims for archiving.

The archive worker process performs the following steps to process the archive items:

1. Each worker process performs additional eligibility checks on each work item. Some of these checks are internal to ClaimCenter. The Default Group Claim Archiving Rules rule set contains additional eligibility checks that you can enable or disable, delete, or modify to meet your business requirements.
2. 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 `RootInfo` object (an instance of a class that implements the `RootInfo` interface). A `RootInfo` object encapsulates information about a single `Claim` its archive-related status information. The concrete class in ClaimCenter is the entity called `ClaimInfo`. In the base configuration, this method is empty.

   IMPORTANT If you need to add or modify properties on `ClaimInfo` in addition to properties mentioned in *step b*, set these properies in your `updateInfoOnStore` method.

   d. The worker *tags* the root object (the claim) specified in the work item. The process of tagging includes setting the `ArchivePartition` property on the root object to a non-null value.
   e. The worker recursively tags all entities in the `Claim` domain graph whose parent object was tagged.
   f. The worker internally generates an XML document for the archived `Claim` and its subobjects.
   g. The worker serializes the XML document into a stream of bytes as an `java.io.InputStream` object.
   h. The worker deletes claim graph data.

   IMPORTANT It is critical to understand that ClaimCenter deletes claim graph data from the database at this step but does **not** delete the info object. The info objects remain in the database for all archived claims.

   i. The worker calls the archive plugin `store` method. The first argument to the method is the input stream that contains the serialized XML document. The second argument is the `RootInfo` object that encapsulates archive-related properties of the claim. In a real production system, the archive plugin would send the data to an external system along with any related metadata from the `RootInfo` object.
3. The worker commits the bundle. 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.
4. 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.

See also

- "Batch Processes and Distributed Work Queues" on page 134 in the *System Administration Guide*
- "Configuring Distributed Work Queues" on page 133 in the *System Administration Guide*
- "Archiving and Restoring Claims" on page 94 in the *Integration Guide*

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.

IMPORTANT To view the **Archive Info** page, you must set the `config.xml` configuration parameter `ArchiveEnabled` to `true`.

Archive Plugin Storage Methods

ClaimCenter calls the following `IArchiveSource` methods during the archive store process:

- `prepareForArchive(info : RootInfo)`
- `updateInfoOnStore(info : RootInfo)`
- `store(graph : InputStream, info : RootInfo)`
- `storeFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)`

prepareForArchive(info : RootInfo)

The method call for `prepareForAchive` 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.

Note: In the demonstration plugin implementation, this method does nothing.

updateInfoOnStore(info : RootInfo)

ClaimCenter calls this method inside the archiving transaction. This enables you to make additional updates on `RootInfo` objects (`ClaimInfo` objects in ClaimCenter). For example, use this location 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.

Note: In the demonstration plugin implementation, this method does nothing.

store(graph : InputStream, info : RootInfo)

ClaimCenter calls this 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 in order for the plugin to insert or update additional reference information to help with restore. For ClaimCenter, the concrete class for `RootInfo` is the entity called `ClaimInfo`.

If your plugin is not able 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.

The cause parameter contains a list of `String` objects that describe the cause of any failures.

It is important to be careful about what kinds of work you do in the the `storeFinally` method to properly handle error conditions. If the `storeFinally` method fails, such as a `RuntimeException` exception, the unwanted file might not be deleted but the `RootInfo` table was already updated. In such a situation, there is no rollback or recovery that the application can do. Therefore, you must design your code not to do substantial operations inside the `storeFinally` call.

WARNING You must **not** do substantial operations inside the `storeFinally` call.

Archive Retrieval Integration

The retrieval process moves data from an archive store back into the main database. It is up to you provide the necessary hooks to the restore 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 undergoes the following stages:

1. ClaimCenter calls `IArchiveSoure.retrieve`:
   - ClaimCenter passes this method a reference to a `RootInfo` object. Your plugin implementation is responsible for returning an `InputStream` that provides the claim graph in the XML form.
   - ClaimCenter recreates the referenced entity using the `InputStream` object.
   - ClaimCenter does **not** commit the bundle at this time.
2. ClaimCenter clears the data on `ClaimInfo` that it copied from the `Claim` object during the initial archive process.
3. ClaimCenter creates any data that relate to the restore. For example, notes and history events.
4. ClaimCenter calls `IArchiveSource.updateInfoOnRetrieve`. *In the demonstration plugin*, this method sets the `DateEligibleForArchive` field to a date based on the `DaysRetrievedBeforeArchive` configuration parameter. In general, 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.
5. ClaimCenter commits the bundle.

6. ClaimCenter calls `IArchiveSource.retrieveFinally`. It is up to you to decide what additional handling the restored claim requires, if any. For example, you can use this method to perform final clean up on files in the archive store.

   IMPORTANT Guidewire does **not** recommend, nor does it support, the use of this method to make changes to a claim after you retrieve it. Any attempt to commit these changes in the `retrieveFinally` method invokes the Preupdate and Validation rules. This is extremely undesirable. As ClaimCenter commits the main transaction for the retrieve process, it does not run these rules. Therefore, it is possible, for example, to retrieve a claim that does not pass validation rules. Committing additional changes in `retrieveFinally` could fail validation, causing only those changes to be rolled back, not the entire restore.

   Note: 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. To view the **Archive Info** page, you must set configuration parameter `ArchiveEnabled` to `true` in `config.xml`.

See also

- "Archiving Claims" on page 665 in the *Configuration Guide*
- "Archiving and Restoring Claims" on page 94 in the *Integration Guide*

Archive 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>)`

retrieve(info : RootInfo) : InputStream

The archive process passes the `RootInfo` object to the plugin. Your implementation of this plugin must return an `InputSteam` object that has the content of the document. 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)`.

retrieveFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)

The retrieval process calls this as the last retrieval step, outside of the restore transaction. The *`finalStatus`* parameter tells if the restore was successful. Use this plugin method to perform other actions on the storage. For example, deleting the file, or marking its status.

Archive 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)`

Your plugin must implement all of these methods. Refer to the demonstration plugin for guidance.

updateInfoOnDelete(info : RootInfo) : List<Pair<IEntityType, List<Key>>>

The delete process calls this method within the transaction in which it deletes the `ClaimInfo` object and related entities from the active database. This is part of the purge process during archiving.

If your archive plugin creates extension entities in the `updateInfoOnArchive` method that link to the `ClaimInfo` object, you must return these objects from `updateInfoOnDelete`. This is important because you may have created other objects that ClaimCenter cannot discover by looking at the foreign key references on the `ClaimInfo` 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` must contain the relevant public ID values for each instance of that entity type.

delete(info : RootInfo)

Your plugin implementation must delete the document identified by `RootInfo` from the backing store. The delete process calls this method *after* ClaimCenter commits the transaction in effect during `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 Restored Data

After the archiving plugin retrieves archived data, ClaimCenter deserializes the data into an in-memory object that represents the data graph. The application automatically runs upgrade trigger steps on the data. ClaimCenter loads the data into an object called an archived entity, represented by the interface `IArchivedEntity`. Any 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 match the current data model of the application. If this process fails, the restoring action (un-archiving) fails. If it succeeds, the new data is committed to the main system database. ClaimCenter also links the restored data with its info object (`ClaimInfo`).

IMPORTANT For more about writing upgrade triggers, see “Using the IDatabaseUpgrade Plugin” on page 56 in the *Upgrade Guide*

chapter 14

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. It is fully configurable at runtime using external configuration files. The ClaimCenter logging system is not limited to integration development. However, there are special "hooks" into the logging system that are particularly useful for integration developers, and these are discussed in this topic.

This topic includes:This topic contains:

See also

- For general information about ClaimCenter logging configuration and the `logging.properties` file, see "Configuring Logging" on page 35 in the *System Administration Guide*.

Logging Overview For Integration Developers

Logging Elements

There are three basic components to the logging system in ClaimCenter:

1. **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`.
2. **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.
3. **Code that triggers logging messages.** Many built-in parts of ClaimCenter can log information, warnings, errors, or arbitrary debugging information. In addition, integration developer code can log anything it wants

and can log information, warnings, errors, or arbitrary debugging information. However, just because code exists that triggers logging does not mean that those messages are written to files. The logging properties file specifies what to actually do with any specific logging information.

Logging Types: Category-based and Class-based

It is important to understand that there are two ways to configure your logging with ClaimCenter:

- **Category-based logging.** Guidewire strongly recommends that you use the hierarchical abstract category system that Guidewire defines for logging. This avoids dependencies on specific Java classes, and lets logging happen independent of Java packages.
- **Class-based logging.** If you have legacy Java code that uses `log4j` class-based logging, it might be easier to use *class-based logging*. However, migrating your code to category-based logging integrates your code with Guidewire pre-defined logging categories. Use logging categories to support quick and easy configuration of log levels, log file locations, and other logging settings.

The Logging Properties File

The *logging properties file* (`logging.properties`) configures (1) what to log and (2) where to log it. The logging properties file is not in a Guidewire-specific format. Instead, it is in the `log4j` format, which is defined by the Apache project for the `log4j` system. This topic only briefly mentions the details of how to configure `log4j` files such as these.

For detailed information about this format, refer to these Apache project `log4j` documentation pages:

```
http://logging.apache.org/log4j/docs/manual.html
http://logging.apache.org/log4j/docs/documentation.html
```

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.appender.PluginsLog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.PluginsLog.File=C:/Guidewire/ClaimCenter/logs/plugins.log
log4j.appender.PluginsLog.DatePattern = .yyyy-MM-dd
log4j.appender.PluginsLog.layout=org.apache.log4j.PatternLayout
log4j.appender.PluginsLog.layout.ConversionPattern=%-10.10X{server} %-4.4X{user} %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.appender.PluginsLog=org.apache.log4j.DailyRollingFileAppender
```

The line after that specifies the location of the log:

```
log4j.appender.PluginsLog.File=C:/Guidewire/ClaimCenter/logs/plugins.log
```

For more information about `log4j` file format options, refer to the documentation at the URLs listed earlier in this section.

Logging Categories for Integration

The list is hierarchical. Enabling a log file appender for the category `log4j.category.plugin` enables `log4j.category.plugin.IValidationAdapte`. 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 class that are pre-defined for common use. For details, see “Category-based Logging” on page 295.

The logging categories pre-defined for ClaimCenter are:

```
log4j.category.Plugin
log4j.category.Plugin.IApprovalAdapter
log4j.category.Plugin.IAssignmentAdapter
log4j.category.Plugin.IValidationAdapter
log4j.category.Plugin.IAddressBookAdapter
log4j.category.Plugin.IContactSearchAdapter
log4j.category.Plugin.ICustomPickerAdapter
log4j.category.Plugin.IPolicyRenewalAlertAdapter
log4j.category.Plugin.IInitialReserveAdapter
log4j.category.Plugin.ISegmentationAdapter
log4j.category.Plugin.IClaimNumGenAdapter
log4j.category.Plugin.IPolicySearchAdapter
log4j.category.Plugin.ICustomConditionAdapter
log4j.category.Plugin.IDeductionAdapter
log4j.category.Messaging
log4j.category.Messaging.Event
log4j.category.Profiler
log4j.category.Configuration
log4j.category.Server
log4j.category.Server.runlevel
log4j.category.Server.database
log4j.category.Server.profiler
log4j.category.Server.batchprocess
log4j.category.Server.cluster
log4j.category.Server.workflow
log4j.category.Server.ConcurrentDataChangeException
log4j.category.UserInterface
log4j.category.Application
log4j.category.Application.financials
log4j.category.Studio
log4j.category.Security
log4j.category.Security.login
log4j.category.RuleEngine
log4j.category.Integration.messaging.ISO
```

For the most accurate logging category list for your version of ClaimCenter, call the SOAP API `ISystemToolsAPI.getLoggingCategories()` to return the complete list of logging category strings.

Logging APIs For Java Integration Developers

Category-based Logging

ClaimCenter provides an API to the `log4j`-based logging system using the `LoggerCategory` class. This class is available to your Java code or your web services API client code.

For your Java plugins, the 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 **server's** application logging properties. If you are trying to set up logging within a Java plugin, skip ahead to "Category-based Logging" on page 296.

> **Note:** 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.

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 server (web application)

logger.properties text file. If you want to load a properties text file similar to logger.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 LoggerFactory class's configure method.

The following is an example of this approach for web services API code:

```
// Set up the logging category 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.

The PluginLoggerFactory class was removed, as it is no longer required due to the changes mentioned above.

For more information, refer to the API Reference Javadoc for com.guidewire.logging.LoggerFactory.

Category-based Logging

To use the LoggerCategory class, you first need an instance of the LoggerCategory class.

The easiest way to get an instance is to use the static instances of this class predefined for common top level categories, such as PLUGIN and API. You can access these as properties of the LoggerCategory class. For instance, LoggerCategory.PLUGIN refers to the static instance of the class 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 class LoggerCategory like this:

```
private LoggerCategory _logger = null;
```

Then, at runtime, get an instance of the LoggerCategory object. For example, use the built-in static instances:

```
_logger = LoggerCategory.PLUGIN;
```

You can now write to this logger object with LoggerCategory 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 an filename path for the log file output.

By default, ClaimCenter initially 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 programmatically override the logger's logging level using the setLevel method.

If you want to use a category for which there is no static instance, use one of two alternate constructors for the `LoggerCategory` class. One constructor creates a new root logging category. Another constructor that defines a new subcategory under another logging category.

The most common approach is to create a category as a subcategory of an existing category:

```
LoggerCategory logger = new LoggerCategory(LoggerCategory.PLUGIN, "IApprovalAdapter");
logger.info("My info mbpessage here")
```

Alternatively, create a new root category:

```
LoggerCategory logger = new LoggerCategory("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, such as the following:

```
log4j.category.IApprovalAdapter=DEBUG, MySpecialLog
log4j.additivity.MySpecialLog=false
log4j.appender.MySpecialLog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.MySpecialLog.File=c:/gwlogs/messaging.log
log4j.appender.MySpecialLog.DatePattern = .yyyy-MM-dd
log4j.appender.MySpecialLog.layout=org.apache.log4j.PatternLayout
log4j.appender.MySpecialLog.layout.ConversionPattern=%-10.10X{server} %-4.4X{user} %d{ISO8601} %p %m%n
```

If you created a new root category, replace `IApprovalAdapter` in the example above with your new category.

The following contains more information about the differences between various log levels.

- **INFO.** Information messages are intended to convey a sense of correct system operation. Typical messages might include “Component XYZ has started”, “A user entered the system.”, or “A user left the system”. In general, do not use this level.
- **WARN.** Warning messages indicate a potential problem. Typical messages might include “Special setting XYZ was not found, so the default value was used”, or “A plugin call took 90+ seconds. Why is the system is slow?”.
- **ERROR.** Error messages indicate a definite problem. Typical messages might include “A remote system has refused a connection to a plugin call”, and “Operation XYZ cannot be completed even with a default.”.
- **TRACE.** Trace messages provide flow of control logging. Typical messages might include “Calling plugin”, or “Returned from plugin call”.
- **DEBUG.** Debug messages are intended to test a provable and specific theory with regard to some system malfunction. These messages need not be details but include information that would be understandable by an administrator. For example, dumping the contents of an XML tag or short document is acceptable. However, exporting a large XML document with no line breaks is usually not appropriate. Typical debug messages might include:

  ```
  Length of Array XYZ = 2345.
  Now processing record with public ID ABC:123456.
  ```

Class-based Logging (Not Generally Recommended)

Instead of using abstract logging categories to identify related code, you can use a class’s fully-qualified name. Using this approach, a class **name and package** defines the hierarchy you use to define logging configuration settings. To do class-based logging, use the `Logger` class rather than `LoggerCategory`.

> **WARNING** You can configure logging based on class name and package hierarchies. However, Guidewire strongly recommends a category-based approach using the `LoggerCategory` class discussed in the previous section. For more information, see “Category-based Logging” on page 295.

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. For related info, see "Setting Up a Logger Factory (SOAP Client Code)" on page 295.

To use class-based logging in your code

1. If the code is non-plugin code, first configure a logger factory. See "Setting Up a Logger Factory (SOAP Client Code)" on page 295 for an example.
2. Configure the logger in the `logger.properties` file using the class name instead of the category name.
3. Use an `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.getInstance().getLogger(MyJavaClassName.class);
```

And then you can send logger messages with it, with similar methods as in `LoggerCategory`:

```
_logger.info("Setting up logger ...");
```

Dynamically Changing Logging Levels

System Tools API Methods for Logging

You can dynamically update the logging level for a specific logger category or a class-based logger using web service APIs. You can do this without having to redeploy the ClaimCenter application. Simply supply a `String` representation of the logger category (for category-based logging) or the Java class name (for class-based logging) and call the SOAP API `ISystemToolsAPI` interface method `updateLoggingLevel`. For more information about web service APIs, see "Web Services (SOAP)", on page 25. (Additionally, you can call the SOAP API `ISystemToolsAPI.getLoggingCategories()` to return the complete list of logging category strings.)

Command Line Tools for Logging

To make an immediate logging change in the ClaimCenter server without having to change the logging properties file and redeploying the application, use the `system_tools` command-line utility options `-updatelogginglevel`. For more information about these command line tools, see the *ClaimCenter Configuration Guide*. (Additionally, you can use the `-logger` cats to return the complete list of logging category strings.)

chapter 15

Address Book Integration

The Guidewire ClaimCenter application optionally integrates with the centralized address book application Guidewire ContactCenter, which is built on the Guidewire platform and is included with ClaimCenter.

IMPORTANT Refer to *Guidewire Contact Management Guide* and "Management and Configuration Resources" on page 19 in the *Contact Management Guide* for more details about address book configuration.

This topic describes the following topics related to address books:

- ContactCenter integration APIs
- Differences between integrating external systems with ContactCenter, compared to integrating external systems with ClaimCenter.
- ClaimCenter plugins that allow a third-party address book to be the ClaimCenter address book instead of using ContactCenter.
- ClaimCenter plugins for external contact searches

IMPORTANT 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 101.

This topic includes:

ContactCenter Integration Overview

In many ways, ContactCenter is like ClaimCenter. They are both complete applications built on the Guidewire platform. ContactCenter has its own versions of:

- integration libraries regeneration scripts:
  - `gwab.bat regen-java-api`
  - `gwab.bat regen-soap-api`
- plugin interfaces
- web service (SOAP) APIs published from the ContactCenter application
- messaging events
- destination plugins
- Java API Reference Javadoc
- SOAP API Reference Javadoc

Specific details of ContactCenter integration points are mostly discussed in this topic, although there areadditional notes in other parts of the documentation.

This topic discusses specific ContactCenter integration topics in the following sections:

- “ContactCenter Entities and Properties” on page 306
- “ContactCenter Web Services Overview” on page 305
- “ContactCenter-specific Web Services” on page 308
- “ContactCenter Messaging Events, by Entity” on page 312

If you use both ContactCenter and ClaimCenter and you integrated them, they work together to implement the ClaimCenter address book. For example, within the user interface of the ClaimCenter application, users can click on the **Address Book** tab to view and search an address book.

In this case, ClaimCenter requests address book information across the network using *plugins* to catch outgoing requests from ClaimCenter, and this plugin calls ContactCenter’s web service APIs.

To change or add properties to contact-related entities, extend or update the data model in both ClaimCenter and ContactCenter and ensure that information flows correctly between the two applications. See the *ClaimCenter Configuration Guide* regarding extending the data model. For information about translating the contact-related properties between contact records within ClaimCenter and the address book, see the *Guidewire Contact Management Guide* regarding synchronization and address book property translation.

If desired, replace ContactCenter with another address book application. This might be necessary if you have a large central corporate address book that cannot yet be transitioned to Guidewire ContactCenter. To do this, see “ClaimCenter Address Book Plugin” on page 302.

Some entire sections of ClaimCenter functionality are not present in any form within the ContactCenter application:

- Notes
- Documents
- Administrative groups
- Activities
- Assignment

There are no ContactCenter entities, plugins, or APIs associated with these objects or features.

Plugins Defined Within ContactCenter

The only supported plugin interface within ContactCenter itself is `IValidationAdapter`, which is an optional mechanism to validate `ABContact` entities rather than using the Validation rule set within ContactCenter. A minimal validation response that shows no errors or warnings, would look like this:

```
ValidationResponse response = new ValidationResponse();

response.setValidationCommand(ValidationCommand.GW_VALIDATION_COMPLETE);
response.setGeneralValidationData(new GeneralValidationData[0]);
response.setFieldValidationData(new FieldValidationData[0]);

return response;
```

To return errors, call the `setFieldValidationData` and `setGeneralFieldValidationData` methods of `response`.

Address Book and Contact Search Plugins

There are two ClaimCenter plugins related to address books and contacts, with different behavior.

- **Contact search plugin.** The contact search plugin (`IContactSearchAdapter`) implements external contact searches that originate within the user interface from *contact pickers*. For example, click on a claim, then select **Loss Details** → **Edit**, and you see a **Reported By** contact picker. If you click a contact picker and select **Search**, then additional registered contact search systems appear in the picker popup in some cases. Specifically, more systems appear if another system is registered in a special typelist that controls the behavior of contact search. See “The ClaimCenter Contact Search Plugin” on page 303.
- **Address book plugin.** The address book plugin (`IAddressBookAdapter`) results are what users see in the **Address Book** tab. That page and its subpages are the main interface to an external address book. An address book plugin must be able to implement contact searches, which returns more data than returned from the contact search plugin.

  Note: For ClaimCenter, this is what ContactCenter is implementing as a remote system during ClaimCenter-ContactCenter communication. If you use Guidewire ContactCenter, do **not** implement the address book plugin.

Generally speaking, the address book plugin has more features than the contact search plugin:

- The address book plugin can push data updates back to the address book.
- The address book plugin can look up and retrieve a contact by a single ID alone. This allows address bookyou to design efficient requests to get a single record.

However, each result from contact search searches are typically have more complete data in the first round of results compared to a standard address book search:

- An address book search returns summaries only in an `ContactSearchResult` entity. Typically an *additional* request would get the contact’s details by its unique ID. If a user performs an address book search followed by a contact edit, ClaimCenter makes several data requests to the plugin:

  1. The original contact search, which uses the address book plugin.
  2. If you edit the contact in the user interface, ClaimCenter gets the item by its unique ID using the address book plugin using the plugin method `retrieveContact`. Address book plugin searches returns a partial view of a contact. It returns only the properties required to display the contact to the user in a list, plus the address book unique ID property called `AddressBookUID`.

     For more information about this ID, see “ContactCenter Link IDs, and Comparison to Other IDs” on page 307.
  3. If the user updates the record and submits the changes, the application calls the address book plugin `submitUpdates` method.

- A contact search returns a whole contact record, not merely a summary. This returns an `ExtContactSrchResult` entity. Contact search is a one-step process. The plugin simply returns the contact search results. There is no additional record retrieval request from a summary, nor a possibility for record update from these results.

Although address book plugin searches return a partial view of a contact, that only applies to searches implemented by the `searchContact` method. The other address book plugin methods return the entire contact, including the methods `findDefinitiveMatch`, `findDefinitiveMatches`, and `retrieveContact`.

ClaimCenter Address Book Plugin

The address book plugin `IAddressBookAdapter` queries an address book application for information about vendors or other contacts. The data returned by this plugin is visible within the ClaimCenter interface in the **Address Book** tab. However, ClaimCenter user interface code controls the appearance of the data in the address book. For a comparison of the address book plugin and the contact search plugin, see “Address Book and Contact Search Plugins” on page 301.

By default, ClaimCenter implements this plugin included in the product that connects to the Guidewire ContactCenter application. If you use ContactCenter, do not implement a custom version of the `IAddressBookAdapter` plugin.

Your implementation of `IAddressBookAdapter` connects with your external address book application. It searches the address book and converts internal contact records to an external address book format (and back again).

This plugin’s API Reference Javadoc typically identifies contacts by their *address book unique IDs*, which are a contact’s ID defined by the current address book. In this context, the address book refers to the external system that the `IAddressBookAdapter` plugin interface represents.

If you use ContactCenter, the address book unique IDs correspond to a ContactCenter internal link ID. For important information about link IDs and related terminology, see “ContactCenter Link IDs, and Comparison to Other IDs” on page 307.

Implementing the ClaimCenter Address Book Plugin

The address book plugin’s basic functions are:

- Connect with the external address book.
- Search the address book.
- Convert internal `Contact` record data properties to the external address book contact format.
- Convert external address book contact records to a ClaimCenter `Contact` with appropriate subobjects.
- Notify the external address book system of added, deleted, and updated contact records.

The most important method in this plugin interface is the `retrieveContact` method, which must return the `Contact` record with the given address book ID. The data returned by this method determines the synchronization status with the ClaimCenter local copy of the contact. ClaimCenter determines whether a contact is in sync with the address book by ClaimCenter comparing local contact data to the contact returned by this plugin interface’s `retrieveContact` method.

The `retrieveContact` method must be prepared to retrieve any “related” contact records for that contact, if the `relationshipsToInclude` parameter is non-`null`. For example, the caller might want to know about all parent-child relationships or the employers or employees of that contact. If `relationshipsToInclude` is non-`null` and there are such contact records, this method returns the related contacts. The related contacts are arrays of `ContactContact` entities hanging off of `Contact`, within the `Contact` properties `TargetRelatedContacts` and `SourceRelatedContacts`. See “Retrieving Contacts and Their Relationships” on page 308 for a discussion about the difference between source relationships and target relationships.

The `submitUpdates` method must handle creating new records, updating data properties, and deleting records from the address book. Instead of handling a single operation at one time, this method gets an `UpdateBatch`

entity, which allows multiple actions of mixed type (create/update/delete) to occur as one transaction. An `UpdateBatch` entity encapsulates a list of actions, referred to in Guidewire terminology as *operations*. The granularity of the operations are not limited to an entire contact record. You can get an individual property or a subobject (such as an `Address`) in the graph of objects that describe a complete contact.

For more information about the format of `UpdateBatch` objects, see "Creating, Updating, and Deleting Records" on page 309, which describes the corresponding API for ContactCenter that parses the `UpdateBatch` entities.

The `findDefinitiveMatch` method must search for a contact that definitively matches the given contact data. This method takes a `Contact` object and returns a `ContactFindMatchResult` object.

The `findPotentialMatches` method must search for all contacts that potentially match the given contact data. The method takes a `Contact` and a `ContactSearchResultsSpec` and returns a `ContactSearchResult`.

The `searchContact` method must for all contacts that match the given search criteria. It takes a `ContactSearchCriteria` and `ContactSearchResultSpec` object and returns a `ContactSearchResult` object. Refer to the API Reference Javadoc documentation for details of these classes.

See *ClaimCenter Installation Guide* for details for installation.

Other Considerations If Using a Address Book Other than ContactCenter

If you want to integrate with a custom address book (such as a legacy corporate address book) rather than Guidewire ContactCenter, you must do the following:

- **Implement the address book plugin.** The main task for you to replace ContactCenter is to implement the address book plugin, described earlier in this section.
- **Change or add properties in the ClaimCenter data model as necessary.** If ClaimCenter preserves, views, or allows users to edit new address book properties, see the *ClaimCenter Configuration Guide* to extend the ClaimCenter contact data model.
- **Change or add contact relationships if necessary.** If there are new or different contact relationships, such as guardian-ward relationships, grandparent-grandchild relationships, extend the ClaimCenter contact relationship data model accordingly. Specifically, you might need to change the file `contactrelationship.xml` and the typelists `ContactRel` and `ContactBidiRel`. Edit these typelists in Studio. See the *Guidewire Contact Management Guide*.

It is important to understand that the address book plugin `retrieveContact` method determines the synchronization status with the ClaimCenter local copy of the contact. Be sure that it returns the most recent version of the contact information in a consistent and predictable way. If necessary, you can *exempt* local contact properties and/or remote address book contact properties from consideration during synchronization. See the *ClaimCenter Configuration Guide* for details.

> **IMPORTANT** To ensure accurate contact synchronization, you must implement the address book plugin `retrieveContact` method. It must return a contact in a consistent way and exempt properties that are temporary or other properties that might interfere with synchronization.

To translate the contact-related properties between contact records within ClaimCenter and the address book, see the *ClaimCenter Configuration Guide* regarding Address Book synchronization and address book property translation.

The ClaimCenter Contact Search Plugin

The contact search plugin (`IContactSearchAdapter`) searches additional contact systems for searches that originate in the user interface from contact pickers. ClaimCenter allows users to choose between searching "the address book" or searching an additional contact repository. The contact search plugin receives the search criteria from the contact search user interface page. Next, it makes the remote search request to an external system and

returns the results as a list. For a comparison of the address book plugin and the contact search plugin, see “Address Book and Contact Search Plugins” on page 301.

The contact search plugin is called only if you define an additional external contact system search instead of “the address book”. In this sense, “the address book”is ContactCenter only if ContactCenter is used with ClaimCenter. If you implemented the separate address book plugin, `IAddressBookAdapter`, the address book would be your address book.

The contact search system is controlled by a typelist called `ContactSearchType`, which defines a set of external systems to search, including the “address book”. In the reference implementation of ClaimCenter, the `ContactSearchType` typelist has one entry, which represents “the address book”. To add additional external contact search systems, modify the file at the location: `ClaimCenter/modules/configuration/config/typelists/ContactSearchType.xml`.

IMPORTANT To implement the contact search plugin successfully, you must also add an entry to the `ContactSearchType` typelist for each external system.

If you add other entries to the `ContactSearchType` typelist, then ClaimCenter does two things:

- ClaimCenter changes its user interface to show a drop down menu in contact pickers to display other contact search systems other than **Address Book**.
- If the user chooses anything other than **Address Book** in that picker, ClaimCenter calls the contact search plugin and requests the results. In this case, ClaimCenter does not call the main address book (ContactCenter, or a your own address book plugin) for the search. ClaimCenter only searches one system for any search.

If there are entries in this list, then a **Search Type** drop-down list appears on the **Address Book Search** page. Your typelist could look like this for two external search options:

```
<typecode id="1" code="externalA" name="System A"
   desc="Search system A" priority="2"/>
<typecode id="2" code="externalB" name="System B"
   desc="Search system B" priority="3"/>
```

In this example, a user would see three search options: **Address Book**, **System A**, and **System B**. The main address book search option already exists with priority level 1, which indicates the top of the sort order in the visible picker. However, you can choose to override the priority of the internal typecode so that it does not have priority 1.

The contact search plugin interface has only one method, which is called `searchContacts`. The method returns a `ExternalContactSearchResultSet` object, which encapsulates:

- an array of `Contact` objects (in the `results` property)
- the total number of results (in the `totalResultCount` property)
- a flag in the `isMoreResults` property that indicates whether additional results exist but were not returned. Set this if the setting for the maximum results to return is lower than the number of actual results.

You can determine how many contacts to return. Get that number, stop getting more contacts, and then set the `isMoreResults` flag to `true`. Guidewire strongly recommends the plugin return at least one page worth of results so that the user interface looks appropriate. The number of items on one screen is specified in the address book search list view PCF file `AddressBookSearchLV.pcf`.

WARNING Do **not** rely on the `MAX_CONTACT_SEARCH_RESULTS` property that is visible in the contact search plugin interface Javadoc to define the maximum number of results. That property will go away in future releases. Instead, as the plugin implementor, you determine how many contacts to return before stopping.

If you have multiple external systems, it is important to understand that ClaimCenter calls this one plugin method `searchContacts` your one plugin instance for all external systems. Your method check which external system to search by checking the value of the `SearchType` property. Check that property and run the correct search request.

That property contains a reference to the typecode instance of the `ContactSearchType` typecode, not the `String` representation of the code or the name.

It is important to understand that ClaimCenter searches only **one** plugin for any individual contact search. That search searches only one external system per search. One external system matches a single entry in the `ContactSearchType` typelist. Typically this represents only one actual remote server. In theory, your plugin could search multiple actual servers systems (at a network level) for a single entry in the `ContactSearchType`.

To compare the contact search type in the safest way, compare the typecode value with using the `equals` method with the static instance of the typecode on the typelist. For example, the following Java code checks the contact search type value to see if it is the typecode `EXTERNALA`:

```
ExtContactSrchResult searchContacts(ContactSearchCriteria searchCriteria) {
  if (ContactSearchType.EXTERNALA.equals(searchCriteria.getSearchType())) {

    // Query the specific external system that represents externalA
    ...
  }
}
```

Address Book Large Object Issues

The ContactCenter `ABContact` entities can be quite large, especially if you extended the entity with more properties. After ContactCenter returns `ABContacts` returned through the integration to ClaimCenter as SOAP objects, ClaimCenter must convert them to contacts in ClaimCenter.

Large objects can cause performance issues both at the SOAP level and also during the conversion process. The built-in integration between ClaimCenter and ContactCenter includes a search configuration file that filters down the size of the objects for the results.

In addition, make sure that your searches and search critieria only return parts of objects that they need. Do not return the full entity graph. Returning the full entity graph (including associations) can cause performance issues.

If you only need a small subset of the information from a contact (or another address book), do not use the built-in APIs. Instead, write your own web service in ContactCenter to return only the minimal information.

IMPORTANT Refer to *Guidewire Contact Management Guide* and “Management and Configuration Resources” on page 19 in the *Contact Management Guide* for more details about address book configuration.

ContactCenter Web Services Overview

Interface	Description
`IContactAPI`	Manipulates contact records. See the section “ContactCenter-specific Web Services” on page 308.
`IExportToolsAPI`	Exports data from administrative tables in an XML format to transfer it to another database. For example, this would support moving users, groups, regions from a test system to a production system. See “Importing Administrative Data” on page 83.
`IImportTools`	Imports system data from an XML file. This is **only** supported for administrative database tables (entities such as `User`) because this system does not perform complete data validation tests on imported data. See “Importing Administrative Data” on page 83.
`ILoginAPI`	Authenticate web service (SOAP) APIs to the ContactCenter server from langauges other than Java. The `ILoginAPI` interface is not recommended for Java development because the recommended Java approach is to use the Java class `APILocator`. For Java login, see “Local SOAP Endpoints in Gosu” on page 75. For non-Java login, see “Calling Your Web Service from Microsoft .NET (WSE 3.0)” on page 45.

Interface	Description
IMaintenanceToolsAPI	This interface provides a set of tools that are available only if the system is at run level Maintenance or higher. Use this API to kick off various background processes. If you call this API, the method return value means only that the request is received. You must poll the server later to see if the process completed successfully or failed. See “Maintenance Web Services” on page 84.
IMessagingToolsAPI	Various messaging methods, such as retrying a message or getting the counts of pending and failed messages for an external system. See “Messaging and Events”, on page 139.
ISystemToolsAPI	This interface provides a set of tools that are always available, even if the server is set to DBMaintenance run level. This particular API provides the current server and schema versions.
ITableImportAPI	Table-based import interface for high volume data import. This would be used typically for large-scale data conversions, particularly for migrating records from a legacy system into ContactCenter prior to bringing ContactCenter into production. See “Importing from Database Staging Tables”, on page 423.
ITypelistToolsAPI	Provides tools for getting the list of valid typecodes for a typelist and for mapping between ContactCenter internal codes and codes in external systems.

In particular, be aware that ContactCenter does **not** implement the following ClaimCenter interfaces:

- ITemplateToolsAPI - For ClaimCenter details, see “Data Extraction Integration”, on page 279.
- IUserAPI - For ClaimCenter details, see “User and Group Web Services” on page 86.
- IGroupAPI - For ClaimCenter details, see “User and Group Web Services” on page 86.

ContactCenter Entities and Properties

ContactCenter Entities

The ContactCenter entities include versions of entities that are similar to entities in ClaimCenter. For example, the ClaimCenter entity called Contact has a similar corresponding ContactCenter entity called ABContact.

An important entity for ContactCenter is the ABContact record, which is the main type of record for ContactCenter. The ABContact entity has the direct subclasses ABCompany, ABPerson, and ABPlace, and each of those three entities has specialized versions (for example, ABCompany has a vendor subclass called ABCompanyVendor). this subtype hierarchy parallels the Contact subtype hierarchy.

> **Note:** The ContactCenter application has entities like ABContact and the ClaimCenter contact address entities such as Contact and Address to support the ContactCenter’s own user and group table.

Important ContactCenter entities include the entities listed in the following table:

Entity or class	Description
ABContact	The ContactCenter version of the Contact entity.
ABContactFindMatchResult	One or more item summaries that are a result from a definitive match search, which returns at most one ABContact. This is not used for other types of searches.
ABContactSearchCriteria	Search criteria for request.
ABContactSearchResult	Results from a search.
ABContactSearchResultSpec	Specifies a search result’s control parameters.
ABContactUpdateResult	Encapsulates the results of an update request.
ABContactWithTravelInfo	A contact record that includes navigation information such as maps and/or complete driving directions.

ContactCenter Link IDs, and Comparison to Other IDs

Many entities defined for the ContactCenter application include a *link ID*, which is a special internal property that acts as the primary key for the ContactCenter application. It is similar to the *public ID* property (`publicID`) in the sense that the property can be queried for, or read, to identify a record uniquely. However, after the record is created and committed to the ContactCenter database, the link ID **cannot** change.

This different from the public ID property, which can be updated at any time as long as that property's uniqueness and maximum length requirements are satisfied. For more information about the public ID property, see "Public IDs and Integration Code" on page 57.

If an external system creates a new ContactCenter entity that requires a link ID, the external system can specify a specific link ID in the new object. For example, the external system could set the link ID to the same text as the public ID string. If the link ID is not provided with a new entity that requires one, ClaimCenter creates a link ID automatically. In all cases, after a new entity commits to the database, the link ID cannot change.

You can retrieve the link ID from an entity using the `getLinkID` method on an entity with link IDs. You can also get the public ID of an entity based on its link ID using the ContactCenter web services API:

```
IContactAPI.getPublicIdFromLinkId(publicID).
```

Although the term *link ID* is used for interaction with the ContactCenter application, APIs that work with alternative address book applications see address book unique IDs (UIDs). These are effectively the same IDs. Which name is used depends on the type of API. For example, the `IAddressBookAdapter` plugin interface relates to alternative address books separate from ContactCenter, that interface refers to this ID as *address book unique IDs* (UIDs).

If you extend the data model of ContactCenter with entirely new entities, the new entities do **not** have link IDs even if they have public IDs. However, ContactCenter `IContactAPI` methods that require a link ID permit the public ID for custom entities (entities you add to the data model), which never have link IDs. For example, suppose you create a new entity type `MyEntity` and one of these entities has the public ID `ab:1234`. If you call the following method, it returns the expected external entity even though you do not pass a real link ID:

```
IContactAPI.getPublicIdFromLinkId("ab:1234", "MyEntity");
```

Because the public ID acts like a link ID for ContactCenter entities of custom types (entities you add to the data model), Never change public IDs for such entities. If you changed public IDs, this breaks all links from client applications to those entities.

The following table compares the various types of IDs related to contacts:

Type of ID	Description
Link ID	The name for ContactCenter's internal ID for each entity
Public ID	The standard Guidewire public record ID that can be changed as needed. For more information on these IDs, see "Public IDs and Integration Code" on page 57. However, for new your own custom ContactCenter entities that never have link IDs, you can use the public ID with `IContactAPI` instead of link ID. For example, in the `getPublicIdFromLinkId` method. In such cases, if you change the public ID of these entities, do not expect links from client applications to be maintained correctly.
Address book unique ID (UID)	Within ClaimCenter APIs, the name for an address book implementation's internal ID. If using ClaimCenter and ContactCenter together, an address book UID and a link ID refer to the same ID. If you implement your own external address book using the address book plugin, then the address book UID refers to an object's internal ID within that external address book application.
UpdateBatch Temp Public ID	If adding new address book entities, a UpdateBatch Temp Public ID is used only with `UpdateBatch` objects within a single API call. It is similar to a public ID except temporary within one API call. After the API processes one `UpdateBatch` object, this ID value is forgotten and no longer used. See "Creating, Updating, and Deleting Records" on page 309.

ContactCenter-specific Web Services

The ContactCenter application includes a web services interface called `IContactAPI`. This interface includes several methods to search for contacts, to return contact data, and to update records from other applications. This web services API is used by the inter-application communication plugin, which is included with ClaimCenter to support the **Address Book** tab and other parts of the user interface.

However, you can use this API interface to load data from other data sources, to query contacts, or to update contacts. For example, you could write a command line tool that allows an external system to add new contacts based on new business data from another part of the company.

The features of the `IContactAPI` interface include the following, discussed in following sections:

- "Retrieving Contacts and Their Relationships" on page 308
- "Getting a Contact-related Record Public ID from Its Link ID" on page 311
- "Creating, Updating, and Deleting Records" on page 309
- "Finding Contact Unique Matches of a Contact" on page 311

Retrieving Contacts and Their Relationships

If you want to retrieve properties from contact records, write a custom web services API (a SOAP API) that extracts the desired subset of entity contact information. For example, from a public ID, the API could extract properties from an `ABContact` record and return it from ContactCenter to the SOAP client.

For typical cases, return only certain properties or subobjects, such as the properties to display or edit the record. Design your integration point accordingly to return only the necessary data, which reduces bandwidth and server resources.

In some cases, you might want to create new entities or Gosu classes to encapsulate your custom data.

If retrieving contact records, you can optionally retrieve related contacts, referred to generally within Guidewire applications as *relationships*. For example, to retrieve a contact's parent/guardian contact information or employer contact information, specify these types of relationships. Finally, design web services for each integration point to return that information if it exists.

To use relationship retrieval, you must understand the difference between what Guidewire calls *source* and *target* relationships. These terms are ways of using the relationship identification codes for relationships such as *parent/guardian* and *employer*, but also specify the directionality of the relationship.

The relationship codes are defined in the `ContactRel` enumeration. For example, it includes the relationship code `TC_employer`. Suppose you want the record for someone named John Smith. If you want to return John Smith's employers, specify `TC_employer` as a *target relationship*. If you want to return John Smith's employees, specify `TC_employer` as a *source relationship*.

A *target relationship* describes the relationship if the `ContactRel` enumeration description fits it into the sentence with the following structure:

> "I want to retrieve a contact and the _____ of the contact, if any is found."

For example, for the employee relationship (`employer`), the sentence would be:

> "I want to retrieve a contact and the **employer** of the contact, if any is found."

If your desired request, the meaning is the opposite of that relationship (such as employee instead of employer), specify the relationship as a *source relationship*.

While retrieving a contact record, you might want both directions of relationships. For example, you might want to simultaneously retrieve all of a company's employees and the company's primary contact. Use the `ContactRel.EMPLOYER` source relationship and the `ContactRel.PRIMARYCONTACT` target relationship.

However, it is important that each integration point have custom SOAP APIs that returns only the desired relationships. Do not allow your API to attempt to serialize (flatten) a potentially huge graph of inter-related entities over the SOAP protocol. If you write a SOAP API that returns return all relationships or returns an entire `ABContact` entity with all relationships, that can cause server memory and performance problems.

WARNING If extracting contact information using custom SOAP APIs, write your API to only return the required properties and the required relationships. If you return an entire `ABContact` or unneeded related contacts, SOAP serialization may trigger server memory problems or client memory problems.

Creating, Updating, and Deleting Records

Your integration code can modify data, create records, or delete records using the web services API. The `IContactAPI` interface method `submitUpdates` takes an `UpdateBatch` object, which encapsulates a list of actions, referred to as *operations*. The granularity of the operations are not limited to an entire contact record. You can get or set an individual property or a subobject (such as an `Address`) extracted from the graph of objects that describe a complete contact.

There are three operations, which are defined in the following table along with the operation entity to use:

Operation	Parameters and meaning	Use this entity
Create	Create a new object of type *`typeName`* and assign it a UpdateBatch Temp Public ID.	`CreateUpdateOp`
Delete	Remove object with public ID *`publicID`* from the database	`DeleteOp`
Set property	Set property *`fieldName`* on object *`ID`* to value *`newValue`*. The ID in this case can be either a UpdateBatch Temp Public ID (in this UpdateBatch only) or an actual public ID of an existing object committed to the database. During the processing of the operations, the system checks to see if it is a UpdateBatch Temp Public ID defined in this UpdateBatch object. Otherwise, ClaimCenter assumes this value is a public ID.	`FieldChangeUpdateOp`

An important concept to grasp before using this API is the UpdateBatch Temp Public ID. This identifier is used with the `CreateUpdateOp` to represent a new entity, and then later used by the set property operation `FieldChangeUpdateOp` to refer back to the new entity. This is a different ID from the persistent (saved) entity properties *public ID* and the *link ID*. The UpdateBatch Temp Public ID is only used to refer to new objects from "property change" operations within the same `UpdateBatch` set of operations.

To submit a list of operations to ContactCenter:

1. **Create an UpdateBatch object.**
   - Create an `UpdateBatch` object to encapsulate the operations.
2. **Create an operation object for each action.**
   - To create a new object, create a `CreateUpdateOp` entity.
   - To set a property, create a `FieldChangeUpdateOp` entity.
   - To delete an existing contact record, create an `DeleteOp` entity.
3. **Add operation objects to the UpdateBatch object.**
   - For each type of operation (set property, create, or delete), store the entities into separate arrays by type. Link each array to the `UpdateBatch` object using `UpdateBatch` methods that take array parameters: `setCreateUpdateOps`, `setDeleteUpdateOps`, and `setFieldChangeUpdateOps`.
4. **Submit the UpdateBatch object.**
   - Call `IContactAPI.submitUpdates(...)` with the `UpdateBatch` object. All operations execute as one requests as one database transaction. However, if creating objects and updating properties, imagine the

following order of actions. First, deletions happen. Then new objects are created. Finally, property updates occur on existing or created objects.

5. Process the results.

- The result of submitting the `UpdateBatch` object is an `ABContactUpdateResult` object, which includes a `getResultType` method that you can call to get the result type.
- If the request failed due to validation errors, then the result type is `ABContactUpdateResultType.ERROR_VALIDATION_MESSAGES` and you can get the specific validation errors using `getValidationResult()`.
- If another result type is returned, see the SOAP API Reference Javadoc for other result type meanings.

IMPORTANT If desired, you can check the return results of the `submitUpdates` method and find out the mapping between UpdateBatch Temp Public ID and the final link IDs for new entities. Call the `getTempToPermEntries` method (get the `TempToPermEntries` property) on the return result of `submitUpdates`. It contains the mappings **from** the UpdateBatch Temp Public ID (the temporary IDs) **to** the new ContactCenter link IDs (the permanent IDs). Each item in the `TempToPermEntries` array contains a `PermId` property and a `TempId` property containing the permanent ID and the temporary ID, respectively.

An example of creating a contact is listed in the following example. To fill in the property details of the new contact, you must use *property change operations* after the *create operation*:

```
// Create arrays to hold your new objects
UpdateBatch ub [] = new UpdateBatch[1];
CreateUpdateOp cuo [] = new CreateUpdateOp[1];
FieldChangeUpdateOp fc[] = new FieldChangeUpdateOp[1];

// Create a new “create” operation to create the new contact
cuo[0] = new CreateUpdateOp();

// Specify details of the create operation
cuo[0].setObjectUId(random1);
cuo[0].setPublicID("ext1:publicID-1");
cuo[0].setEntityTypeName("ABCompany");

// Specify details of the new object in a “property change” operation
fc[0] = new FieldChangeUpdateOp();
fc[0].setEntityTypeName("ABCompany");
fc[0].setField("TaxID");
fc[0].setValue("111-12-1234");
fc[0].setObjectUId("myTempID1");

// Create a new UpdateBatch object
ub[0] = new UpdateBatch();
ub[0].setCreateUpdateOps(cuo);
ub[0].setFieldChangeUpdateOps(fc);

// init
ABContactUpdateResult result = null;

// Call myContactAPI.submitUpdates to submit the request
try {
  result = myContactAPI.submitUpdates(ub[0]);

  if (result.getResultType() == ABContactUpdateResultType.GW_SUCCESS) {
    // post processing...
    }
   else {
    //error handing...
   }
 }
catch (Exception e) {
  ...
}
```

Replacing Existing Records

To fully replace an existing contact, delete the old entity using a delete operation (`DeleteOp`) and then a new create operation (`CreateUpdateOp`). These can be included in the same `UpdateBatch` request, which can then submitted using the `IContact.submitUpdates( )` method.

Finding Contact Unique Matches of a Contact

There are two methods in the `IContactAPI` interface that finds contact matches. You can choose whether to search for multiple *potential matches* or to search for an definitive (unique) match from a set of one or more defined properties.

For example, to attempt a definitive match of a company from its tax ID, you could use code like this:

```
// Create new ABCompany and set a property with the search properties
ABContact companyToFind = new ABCompany();
ABCompany.setTaxID("79-0103590");

// Search for a definitive match in the ContactCenter database
ABContactFindMatchResult result = contactAPI.findDefinitiveMatch(ABCompany);

// Convert ABContactFindMatchResult to an ABContact object
ABCompany resultCompany = (ABCompany) result.getABContact();
```

In contrast, to return multiple potential matches for specific property data, use the `findPotentialMatches` method. It takes a contact (`ABContact`) and an object that defines the search parameters (`ABContactSearchResultSpec`).

The following example of `findPotentialMatches` finds potential matches of people with a certain name:

```
// Define generate parameters for the search
ABContactSearchResultSpec spec = new ABContactSearchResultSpec();
spec.setIncludeTotal(true);
spec.setMaxResults(1000);

// Define the search criteria
ABPerson searchPerson = new ABPerson();
searchPerson.setFirstName("Andy");
searchPerson.setLastName("Applegate");
ABContactSearchResult result = contactAPI.findPotentialMatches(searchPerson, spec);
ABContactWithTravelInfo[] searchResults = result.getResults();

for (int i = 0; i < searchResults.length; i++) {
  firstName = ((ABPerson) (searchResults[i]).getABContact()).getFirstName();
  lastName = ((ABPerson) (searchResults[i]).getABContact()).getLastName();
  // get other properties of interest here...

  // [PRINT / PROCESS / DISPLAY the result as desired...]
}
```

Remember to set the `maxResults` property on the `ABContactSearchResultSpec` before requesting the search. There is no default maximum. The API returns the lesser of the number of actual results and the value of `maxResults`.

The total number of results is only included if you set the `includeTotal` flag on the `ABContactSearchResultSpec` passed into the query.

You can choose to set `maxResults` to 0 to return no results. This is useful in rare cases if your integration code needs to know the total number of results but does not care about individual result details at this time. You can get the total results in the contact search results entity. In this case, be sure to set the `includeTotal` flag on the `ABContactSearchResultSpec` passed to the request.

Getting a Contact-related Record Public ID from Its Link ID

Several methods exist to look up records from the ContactCenter link ID, an important property that is discussed in “ContactCenter Link IDs, and Comparison to Other IDs” on page 307. To get the public ID of the contact identified by the given link ID, use the `getPublicIdFromLinkId` method.

If the record is an `ABContact` object, you need only to pass the link ID string. If it is another type of object, there is another method signature for the `getPublicIdFromLinkId` method. It takes an entity type name parameter, such as the `String` value `"Address"`.

Searching for Multiple Contacts

To search for multiple matches in ContactCenter, design your own custom web service in ContactCenter and return only the exact data that you require. Contacts can be large objects, especially if you use the relationships feature. Be sure you transfer only the minimal amount of data necessary.

If you have several use cases for contact searches with different data to transfer, Guidewire strongly advises writing multiple integration points. You can define multiple custom web services in ContactCenter as needed. See “Web Services (SOAP)”, on page 25 for details.

ContactCenter Messaging Events, by Entity

Entity	Events	Description

`Activity`	`ActivityAdded`
	`ActivityChanged`
	`ActivityRemoved`
		Standard A/C/R events for root entity `Activity`. The `ActivityChanged` event triggers if the activity updates, including if a user marks it as completed or skipped.

`Assignment`	`AssignmentAdded`
	`AssignmentChanged`
	`AssignmentRemoved`
		Standard A/C/R events for root entity `Assignment`.

`Document`	`DocumentAdded`
	`DocumentChanged`
	`DocumentRemoved`
		Standard A/C/R events for root entity `Document`. Some implementations do not let users remove documents once added, so the remove event may not trigger.

`Note`	`NoteAdded`
	`NoteChanged`
	`NoteRemoved`
		Standard A/C/R events for root entity `Note`.

`Organization`	`OrganizationAdded`
	`OrganizationChanged`
	`OrganizationRemoved`
		Standard A/C/R events for root entity `Organization`.

Administration events		

`Group`	`GroupAdded`
	`GroupChanged`
	`GroupRemoved`
		Standard A/C/R events for root entity `Group`.

`GroupAssignmentState`	`GroupAssignmentStateAdded`
	`GroupAssignmentStateChanged`
	`GroupAssignmentStateRemoved`
		Standard A/C/R events for root entity `GroupAssignmentState`.

`Role`	`RoleAdded`
	`RoleChanged`
	`RoleRemoved`
		Standard A/C/R events for root entity `Role`.

`User`	`UserAdded`
	`UserChanged`
	`UserRemoved`
		Standard A/C/R events for root entity `User`. For the `UserChanged` event, only changes made directly to the user record trigger this event, not changes to roles or group memberships. A change to the user contact record, for example a phone number, triggers a `ContactChanged` event.

Contacts and address book events		

`ABAdjudicator`	`ABContactAdded`
	`ABContactChanged`
	`ABContactRemoved`
		Standard A/C/R events for root entity `ABContact`.

`ABAttorney`	`ABContactAdded`
	`ABContactChanged`
	`ABContactRemoved`
		Standard A/C/R events for root entity `ABContact`.

Entity	Events	Description
ABAutoRepairShop	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABCompany	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABCompanyVendor	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABDoctor	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABLawFirm	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABContact	ABContactAdded ABContactChanged ABContactRemoved	Contact entities only exist as subtypes of Contact, such as Person. Those subtypes generate standard A/C/R events for root entity ABContact.
	ABContactResync	An ABContact resynced. Resend all related messages to external systems as appropriate. For related information, see "Message Ordering and Multi-Threaded Sending" on page 170 and "Resyncing Messages" on page 185.
ABLegalVenue	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABMedicalCareOrg	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABPerson	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABPersonVendor	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABPlace	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
ABUserContact	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity ABContact.
Company	ContactAdded ContactChanged ContactRemoved	Standard A/C/R events for root entity Contact.
CompanyVendor	ContactAdded ContactChanged ContactRemoved	Standard A/C/R events for root entity Contact.
LegalVenue	ContactAdded ContactChanged ContactRemoved	Standard A/C/R events for root entity Contact.
Place	ABContactAdded ABContactChanged ABContactRemoved	Standard A/C/R events for root entity Contact.
Person	PersonAdded PersonChanged PersonRemoved	Standard A/C/R events for root entity Contact.
PersonVendor	PersonVendorAdded PersonVendorChanged PersonVendorRemoved	Standard A/C/R events for root entity Contact.
UserContact	ContactAdded ContactChanged ContactRemoved	Standard A/C/R events for root entity Contact.

The entities `Activity`, `Document`, and `Note` exist in ContactCenter, and therefore their events automatically exist. However, ContactCenter does not currently use these entities, so these events never trigger in the current ContactCenter release.

ABContact Message Safe Ordering

ContactCenter supports safe ordering of the `ABContact` entity. This means that ContactCenter only allows one message per `ABContact`/destination pair "in flight" (sent but not acknowledged) at any given time. Messages of this type are referred to as *safe-ordered messages*. If an event generates three safe-ordered messages related to one `ABContact` entity, ContactCenter sends the first message immediately but not send the second message. It sends the second message after it receives an acknowledgement from the destination about the first message. Because ContactCenter manages events for many claims, ClaimCenter can have many messages in-flight at a time, and even multiple messages to one destination, but only one for each `ABContact`/destination pair.

For more information about safe ordering as it relates to claims, see "Message Ordering and Multi-Threaded Sending" on page 170

ABContact Message Resync

ContactCenter also supports resyncing a contact (the `ABContact` entity) to an external system, similar to how a claim can be resynced. If an `ABContact` entity resyncs, a destination could listen for the `ABContactResync` event. If that event triggers, the Event Fired rules that process it could resend important messages to external systems for this entity to synchronize with the external system.

For more information about message resyncing as it relates to claims, see "Resyncing Messages" on page 185.

ContactCenter Callbacks into ClaimCenter

Address books can use automatic contact autosync functionality. If a contact updates in the address book, the address book notifies ClaimCenter to pick up the change. It finds the contacts to synchronize to the new values stored in the external address book. If you use ContactCenter also, ContactCenter uses this API automatically.

This communication happens through the ClaimCenter web service API interface `IContactAutoSyncAPI`.

You can call it from an external address book (or other application) using code such as:

```
addressBookUID = "abc:12345";
contactAutoSyncAPI.autoSyncContactWithABUID(addressBookUID);
```

chapter 16

Geographic Data Integration

Guidewire ClaimCenter and Guidewire ContactCenter provide a user interface and 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 ContactCenter support routing services, such as getting a map of an address and getting driving directions between two addresses, provided the addresses have been geocoded already.

This topic includes:

For general information about implementing plugins, see "Plugin Overview" on page 101.

Geocoding Plugin Integration

Guidewire ClaimCenter and Guidewire ContactCenter use the Guidewire `GeocodePlugin` 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.

The `GeocodePlugin` interface also 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 determine whether the registered implementation supports routing services and reverse geocoding.

How ClaimCenter Uses Geographic Data

ClaimCenter uses geocode information for actions within rules and other geographic searches. The default configuration and implementation supports these high-level 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
- **ContactCenter address book searches (proximity search of vendors).** If both ClaimCenter and ContactCenter are installed, you can search for nearby service providers in the address book based on geographic proximity. This feature requires that:
  - ClaimCenter and ContactCenter have the geocoding plugin enabled.
  - ContactCenter contains geocoded addresses in its database.

To support user assignment and searching using geocoding, ClaimCenter must install and register a `GeocodePlugin` implementation for an external geocoding service. To support address book searches that use geocoding, both ClaimCenter and ContactCenter must install and register a `GeocodePlugin` implementation.

IMPORTANT To support address book searches with geocoding, configure and install both ClaimCenter and ContactCenter, linking the two applications. See the *ClaimCenter Installation Guide* for details.

What the Geocode 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 Geocode 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 in the user interface, such as proximity searches, the user interface blocks until the plugin responds.

ClaimCenter also supports batch asynchronous geocoding using a distributed system that allows all servers in the cluster to perform geocoding in the background while other tasks happen. This *distributed work queue system* is especially useful after an upgrade with new addresses to geocode. You can use the work queue infrastructure for this background task, 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. See the *ClaimCenter System Administration Guide* to configure work queues such as the `Geocode` and `ABGeocode` work queues. The `Geocode` and `ABGeocode` work queues use only the geocoding method of the `GeocodePlugin` interface. Register and enable the plugin correctly, and the distributed work queue system handles background geocoding automatically.

The Built-in Bing Maps Geocode Plugin

ClaimCenter includes a fully functional and supported implementation of the geocoding plugin to connect to the Microsoft Bing Maps Geocode Service. For details about installing and configuring Bing Maps support, see “Using the Geocoding Feature” on page 22 in the *System Administration Guide*.

Steps to Deploy a Geocode Plugin Overview

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 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 318.

If you want to support proximity searches of vendors within the ContactCenter application, you must repeat this step in ContactCenter Studio.

IMPORTANT If you use proximity searching of vendors, remember to repeat this for ContactCenter.

Step 2: Register the Plugin Implementation in Studio

1. 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 support proximity searches of vendors, which uses the ContactCenter application, you must repeat this step in ContactCenter Studio.

IMPORTANT If you use proximity searching of vendors, remember to repeat this for ContactCenter.

Step 3: Enable the User Interface for Desired Geocoding Features

You also need to 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/ pickers. For example:

```
<param name="UseGeocodingInPrimaryApp" value="true"/>
```

If you support proximity searches of vendors (which uses the ContactCenter application) and want to support the user interface for geocoding, set `UseGeocodingInAddressBook` parameter. Set this parameter in each application

that requires support. For example, to support the user interface in **both** ClaimCenter and ContactCenter, set this in both ClaimCenter and ContactCenter.

```
<param name="UseGeocodingInAddressBook" value="true"/>
```

Writing a Geocoding Plugin

To support a 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 Geocode 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. To support geocoding, implement the required method `geocodeAddressBestMatch`.

   It accepts an address and returns the address with latitude and longitude coordinates assigned.

   See "Geocoding an Address" on page 319.

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 320.

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.

   See:

   - “Supporting Multiple Address Corrections with a List of Possible Matches” on page 320
   - “Retrieving Overview Maps” on page 321
   - “Adding Segments of the Journey with Optional Maps” on page 322
   - “Getting a Map For an Address” on page 322
   - “Getting an Address from Coordinates (Reverse Geocoding)” on page 323

Geocoding an Address

The *geocoding plugin* (`GeocodePlugin`) interface has one main method for geocoding that is required, the `geocodeAddressBestMatch` method. If that is the only method that you override from `AbstractGeocodePlugin`, your plugin does not support optional features such as address correction. A Guidewire application communicates with the geocoding plugin through `Address` entities. Each `Address` entity contains standard `Address` properties such as *`address`*`.AddressLine1` and *`address`*`.Country`.

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.

ClaimCenter also geocodes in the background with a batch process that calls this plugin as necessary to geocode an address. For more information about how to configure the batch process, See the “Using the Geocoding Feature” on page 22 in the *System Administration Guide*.

The `geocodeAddressBestMatch` method takes an address (`Address`) entity and returns another address with an appropriate `GeocodeStatus` value. The `GeocodeStatus` property contains a status from the `GeocodeStatus` typelist. Values in this typelist include `exact`, `failure`, `street`, `postalcode`, or `city`. For more information, see “Geocoding Status Codes” on page 323. If geocoding was successful, also set the `Latitude` and `Longitude` properties in the address.

Do **not** modify the incoming address parameter. You can clone it with `address.clone()` to make a copy, or can create a new address with the Gosu code “`new Address()`”.

For example, create a new address with exact geocoding status and explicit coordinates:

```
a = new Address()
a.GeocodeStatus = GeocodeStatus.TC_EXACT
a.Latitude = 42.452389
a.Longitude = -71.375942
```

In a real implementation, your code gets the coordinate values from an external service such as a web service rather than setting them explicitly.

If the geocoding service modified other properties as part of correction or clarification, set those properties as well. All blank properties in a returned address are assumed to be blank purposely.

For example, address properties in return data might be unknown or inappropriate for that geocode status. If the geocode status represents the weighted center of a city, the street address might be blank because the 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.

The returned addresses might be variants of an address. For example, the street address “123 Main Street” might match the geocoding data for “123 North Main Street” and “123 South Main Street”, each with different coordinates. The geocoding service might return both results for the user to select among. Differences might be due to differences in geocoding data (`Street` versus `St`), rather than actual mistakes. Similarly, some services automatically remove Suite/Apartment/Floor numbers from addresses, or other changes. For the `geocodeAddressBestMatch` method, return only the *best* match. In contrast, you can return multiple results in a different plugin method, see “Supporting Multiple Address Corrections with a List of Possible Matches” on page 320.

Supporting Multiple Address Corrections with a List of Possible Matches

If your geocoding service can provide a list of potential addresses with address corrections, implement the `geocodeAddressWithCorrections` method. Additionally, implement the `pluginSupportsCorrections` method and return `true` to tell ClaimCenter that your implementation supports this feature.

IMPORTANT The default configuration of ClaimCenter never calls the methods `pluginSupportsCorrections` or `geocodeAddressWithCorrections`. If you modify the application to implement corrections behavior, your new PCF code would call the plugin methods `pluginSupportsCorrections` and `geocodeAddressWithCorrections` to connect to the external service.

In contrast to the `geocodeAddressBestMatch` method, the `geocodeAddressWithCorrections` method returns a list of addresses rather than a single address. Despite what the name might imply, both methods can return address corrections or clarifications, or leave some properties blank if they were not used to generate the coordinates. (See earlier in this section for a discussion of address correction). However, the system calls this method if the context can handle a list of corrections. For example, a user interface context might support a user choosing among geocoding results to identify the truly intended address.

The result list must be a standard `List` (`java.util.List`) that contains only `Address` entities. This type of object can be described in Gosu using the generic syntax `List<Address>`. For more information, see “Gosu Generics” on page 221 in the *Gosu Reference Guide*.

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, it implements this behavior for you.

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 323.

Getting Driving Directions

ClaimCenter and ContactCenter 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 221 in the *Gosu Reference Guide*.

Geocoding Status Codes

Geocoding services typically provide a set of geocoding *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`, `postal code`, 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
none	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.
failure	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.
exact	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match exactly the complete address.
street	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 postalcode, depending on the complete address and the length of the street.
postalcode	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.
city	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.

chapter 17

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 “Startable Plugins” on page 128.

This topic includes:

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 318 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 BatchProcess-Base" on page 327.

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 BatchProcessBase

Each batch process must have an implementation of the `BatchProcess` interface, which defines the contract with ClaimCenter. The easist 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 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 325.

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 282 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 327.

> **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 requets 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 overriden 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 328.

Exclusive

The property getter for the `Exclusive` poperty 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 `incrementOperationsFailed`.

- `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` typecode. 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 282 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 ENTRYPOINT 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 282 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 `IMaintenanceToolsAPI` 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 318 in the *Configuration Guide*.

chapter 18

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:

See also

- For general information about plugins, see "Plugin Overview", on page 101.
- For geocoding plugins, see "Geographic Data Integration", on page 315
- For messaging plugins, see "Messaging and Events", on page 139.
- For authentication plugins, see "Authentication Integration", on page 239.
- For document and form plugins, see "Document Management", on page 249.

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 in Gosu" on page 120.

For an example implementation of this plugin:

- ...as a Gosu plugin, see "Deploying Gosu Plugins" on page 111.
- ...as a Java plugin, see "Deploying Java Plugins" on page 115.

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 293 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.

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

   }
```

Approval Plugin

The approval plugin has to answer two questions. First, does the set of transactions need approval? Second, if so, who needs to 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`.

Testing Clock Plugin (Only For Non-Production Servers)

Testing ClaimCenter behavior can be challenging, particularly testing complex behavior over a long span of time during multiple billing cycles. To help testing, you can programmatically change the system time to accelerate the perceived passing of time. Do this by implementing the `ITestingClock` plugin for development testing only. For example, 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`.

To store artificial time values, this plugin can create or read one or more instances of the `TestingClockContainer` entity. Guidewire includes this entity as a shell to contain artificial time values to use with this plugin. The `TestingClockContainer.CurrentTime` property must contain a non-null time value that can be read or updated. You could use this entity type in any way desired to store time values. One possibility is to create a single `TestingClockContainer` that contains an offset time value to add to the actual (true) time each time ClaimCenter asks for the current time.

Time must always increase (not go back in time) or else the behavior of ClaimCenter could be highly unpredictable. However, even if only moving the clock forward in time, you must only use this plugin on **non-production** systems.

WARNING The `ITestingClock` plugin is intended for testing only on development (non-production) servers. Registering this plugin on production servers is **unsupported** and **dangerous**.

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.

Preupdate Handler Plugin

If you want to implement your pre-update handling in plugin code rather than in the built-in rules engine, register an implementation of the `IPreUpdateHandler` plugin interface. If this plugin is implemented and the server `config.xml` parameter `UseOldStylePreUpdate` is set to `false`, ClaimCenter uses this plugin instead of the validation-graph based approach.

In some cases, using this plugin instead of the validation-graph approach results in higher performance.

Your plugin implementation must implement one method, which is `executePreUpdate`.

It 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.

For Java plugins, these appear as three methods: `getInsertedBeans`, `getUpdatedBeans`, and `getRemovedBeans`.

For an overview of preupdate rules, see “Preupdate” on page 70 in the *Rules Guide*

Default Plugin Implementation

ClaimCenter includes a built-in implementation `gw.plugin.preupdate.impl.PreUpdateHandlerImpl`.

By default, this collects all inserted and updated beans. Next, ClaimCenter executes the pre-update rules logic for each one. Any exception cause the bounding database transaction to roll back, effectively undoing the update.

ClaimCenter includes a default implementation of this plugin that calls the pre-update rules with added or updated entities. It also calls pre-update rules with any `Account` or `Job` with changes to associated `UserRoleAssignment` entities.

PreUpdateUtil

It is possible to maintain some preupdate code in this plugin but still call preupdate rules in some cases. ClaimCenter includes the `PreUpdateUtil` class in the `gw.api.preupdate` package to let you call preupdate rules from your plugin. The `PreUpdateUtil` class provides a single method, `executePreUpdateRules`. That method executes preupdate rules on an entity if it has an associated preupdate rule set. It does nothing if a preupdate rule does not exist.

chapter 19

Insurance Services Office (ISO) Integration

The Insurance Services Office (*ISO*) provides a service called ClaimSearch which 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 filename 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 ClaimCenter generated XML payload, discussed in "ISO Payload XML Customization" on page 370.

You may wish also to refer to the ISO ClaimSearch web site:

```
http://www.iso.com/index.php?option=com_content&task=view&id=701&Itemid=543
```

This topic includes:

ISO Integration Overview

ClaimSearch helps detect duplicate and fraudulent insurance claims. After an insurance company enters a claim, the company 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 may 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.

Choose Either Claim-based Messaging or Exposure-based Messaging

You can choose whether ClaimCenter sends claims or exposures to ISO. This is controlled by the `ISO.properties` file property `ClaimLevelMessaging`. By default, this is set to exposure messaging. You can keep using exposure-based ISO support if you prefer. If you ever switch from exposure-based ISO messaging to claim-based ISO messaging, ISO continues to track your old submissions of ClaimCenter exposures in the ISO system. ClaimCenter continues to maintain exposure ISO-related properties and handles match reports that ISO sends later based on exposures submitted before the switch to claim-based ISO 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 old submissions of ClaimCenter exposures in the ISO system. ClaimCenter continues to maintain the following properties on an exposure:

- `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 (that is, the `ISOSendDate` is non-null), ClaimCenter handles ISO replies on that claim only using 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 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 tell ClaimCenter to use claim level messaging for this claim. It returns false to indicate exposure-based messaging. The built-in implementation uses the algorithm described in the previous paragraph. 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 claimust 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 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 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. This required a data model change for the `ISOMatchReport` entity to be two separate entities: `ClaimISOMatchReport` for claims and `ExposureISOMatchReport` for exposures.

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` properties is `null`. For `ExposureISOMatchReport`, the `ISOExposure` property points to the exposure and the `ISOClaim` properties 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.

Note: 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.

  For more information, see “ISO User Interface” on page 365. To enable the ISO user interface (as well as the other components), see “Enabling ISO Integration” on page 345.
- **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 360.
- **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 busi-

ness requirements, such as custom exposure types. For more information about payload generation functions, see “ISO Messaging Destination” on page 361 and “ISO Payload XML Customization” on page 370.

Note: 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 363 and “ISO Match Reports” on page 373.
- **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 363. 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 373.
- **ISO properties files.** There are additional configuration options in the ISO property files (see “ISO Properties File” on page 365)
- **ISO mapping files.** There are additional configuration options in the mapping files for coverages and typecodes (see “ISO Type Code and Coverage Mapping” on page 369)
- **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 67 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 filename 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 370.

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
```

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 346.

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 possible 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 accomodate new exposure types
- the ISO properties files, especially to map settings related to the `ClaimContact` entity.

  Note: 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 370
- “ISO Exposure Type Changes” on page 374
- “ISO Properties File” on page 365
- “ISO Type Code and Coverage Mapping” on page 369

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 test system	https://clmsrchwebsvct.iso.com/ClaimSearchWebService/XmlWebService.asmx
The live system	https://clmsrchwebsvc.iso.com/ClaimSearchWebService/XmlWebService.asmx

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
`SubmitToISO`	`SubmitToISO` checks the XML and queues it up for addition to the ISO database. After the request processes, ISO sends an asynchronous response.
`SubmitTestToISO`	`SubmitTestToISO` 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. `SubmitTestToISO` 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 345.

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 in **Resources**, click **Messaging**.
2. Click the row with ID 66 and the name `iso`.
3. Check the **Enabled** checkbox.
4. Within Studio, click the **Plugins** node and expand it.
5. Click the **isoTransport** node.
6. Check the **Enabled** checkbox.

For more information about messaging plugins, see “Messaging and Events”, on page 139.

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
- Step 2: Team Resources
- Step 3: Requesting ISO Testing and Production Server Access
- Step 4: ISO Configuration and Integration Testing
- Step 5: ISO Integration Data Migration
- Step 6: ISO Production Testing, Including Connectivity Testing
- Step 7: ISO Production Deployment

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 358.

Be sure you get contact information for people at ISO, for both administrative and technical contacts.

IMPORTANT Begin the ISO membership process **immediately**. Also be sure to get contact information at ISO, in addition to your Professional Services contacts at Guidewire, to help test your implementation.

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 358.

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 358.

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 365
- “ISO Type Code and Coverage Mapping” on page 369
- “ISO Payload XML Customization” on page 370
- “ISO Match Reports” on page 373

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 potentialy 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 379.

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 349.

WARNING You must 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 349.

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*).

WARNING 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 365.

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 exose 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 effecient 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 359 and "Proxy Servers", on page 415.

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 a 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 send 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.

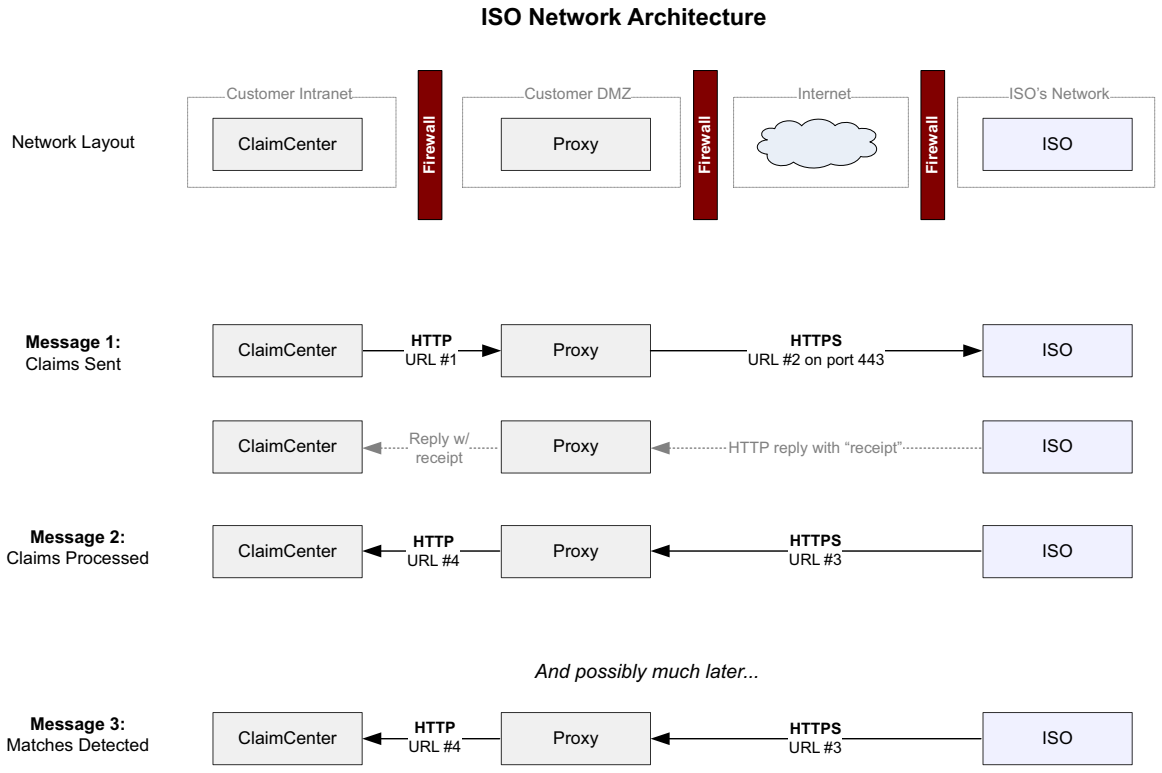
  Validation URL. Use `https://clmsrchwebsvc.iso.com:443/ClaimSearchWebService/TestUtility.htm` if calling ISO for programmatic validation. ISO tests that validity of your XML message and authentication without submitting the claim to any ISO database. Although there is an immediate reply to the request 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 the your proxy server in the DMZ. The port number is configurable. The format of the URL is `http://CCServer:CCport/WebApp/ISOReceive` where `WebApp` is the name of the web application, and is typically "`cc`". The immediate HTTP reply to this request does not contain significant information.

The typical ISO network setup, the 3 ISO messages, and the 4 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 uninvolved.

Remember to configure your firewall and proxy servers accordingly.

IMPORTANT For ClaimCenter clusters, the batch server is the only server that attempts to interacts 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 asychronous message reply. For more information about message reply plugins and overall messaging flow, see “Messaging Overview” on page 140.
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.

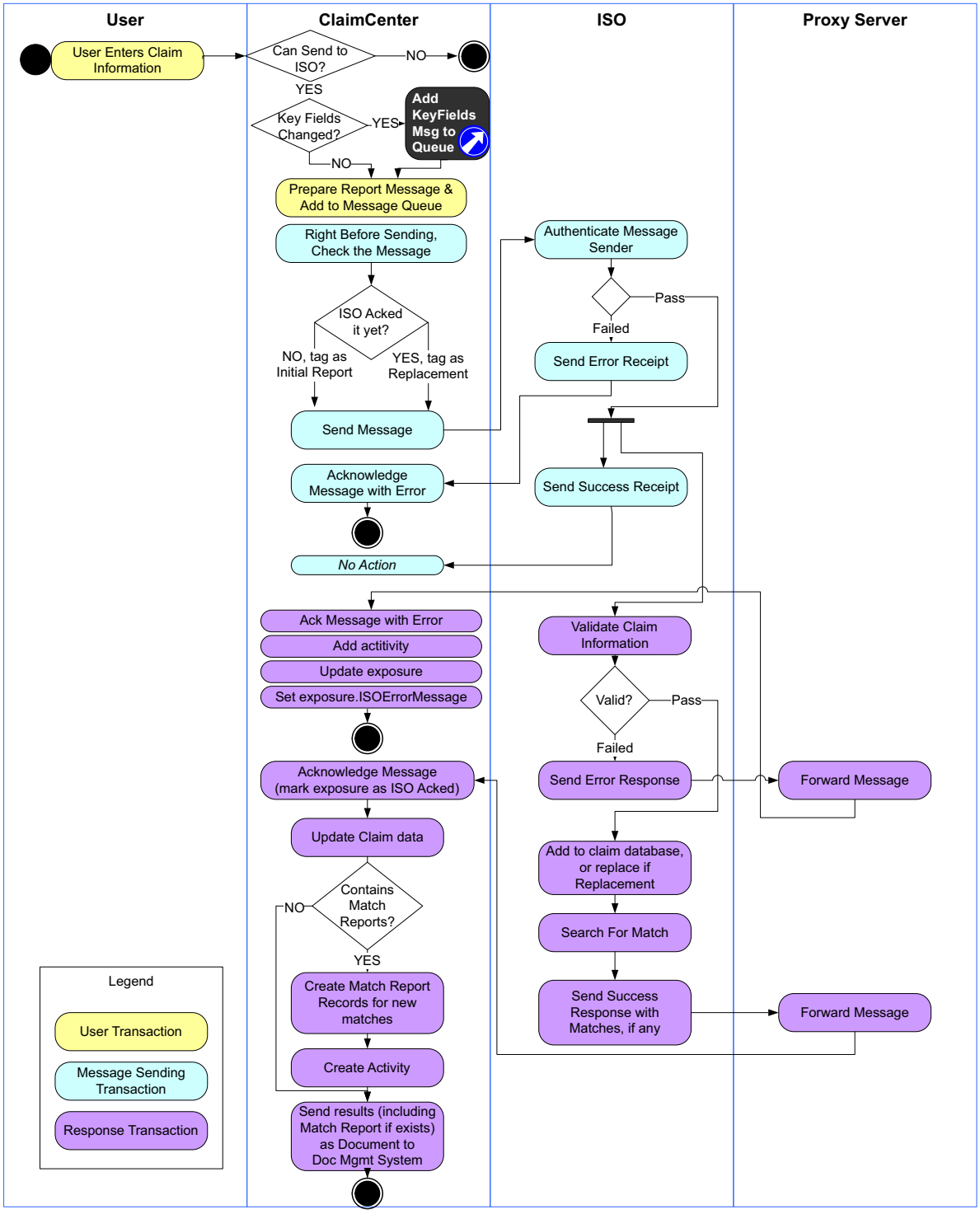
After the key field update message is handled another "replace" message is sent to get any new match results.

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



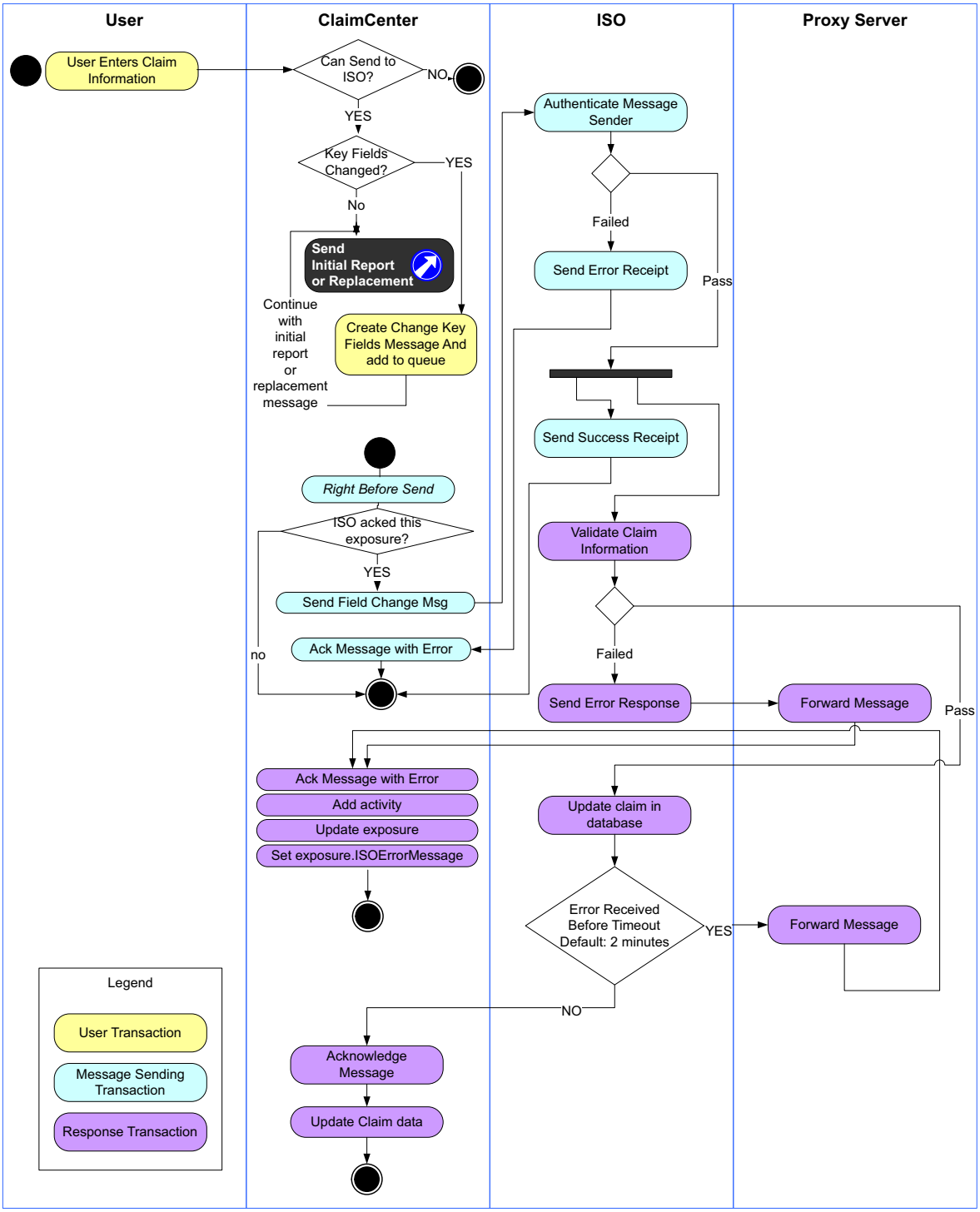
The following diagram describes the activity during a key fields change message. A key fields change message is a special type of message that ClaimCenter sends to ISO. For claim-based ISO messaging, ClaimCenter sends a

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

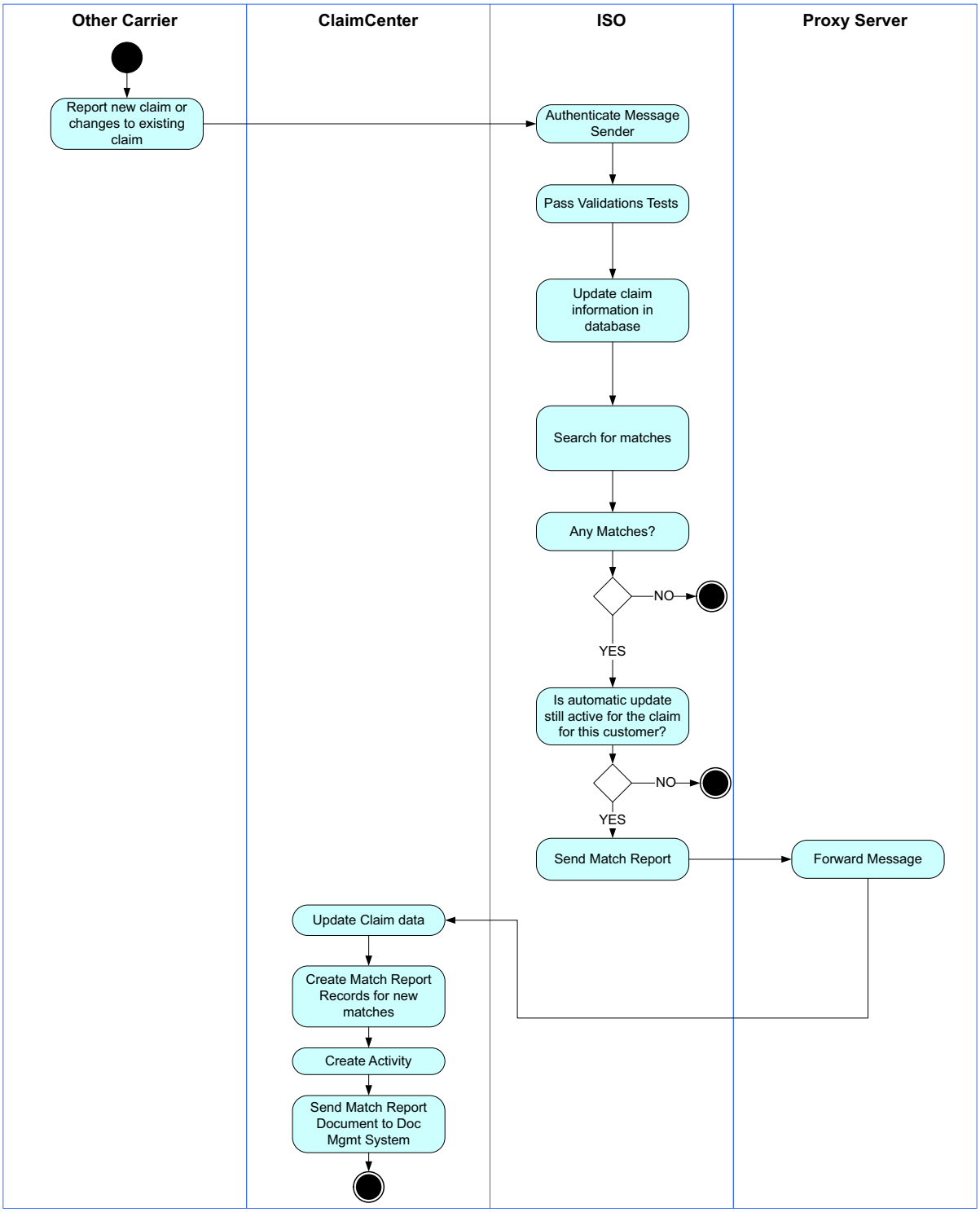
thinks ISO knows about the claim or exposure, then it sends the key fields change message. Otherwise, ClaimCenter ignores the key fields message and does not send the message to ISO.

ISO Key Fields Changed Activity



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 349.

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 348. 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 348.

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.

IMPORTANT For more details, see “Proxy Servers”, on page 415.

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 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 mesaging, a notable requirement in validation is that the claim desription 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.

IMPORTANT If you use exposure-level messaging, exposure validation logically depends on the claim’s validity. If the claim is not valid for ISO, the exposure cannot be valid for ISO. You must continue to enforce this in your customized validation rules.

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” on page 79 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 370. 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.

IMPORTANT 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 345), 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 370.

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.

IMPORTANT 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 348.

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 a 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.

IMPORTANT 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 asychronous message reply. For more information about message reply plugins and overall messaging flow, see “Messaging Overview” on page 140.

ISO Properties on Entities

ClaimCenter ncludes 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	ISOSendDate	date	The last time the exposure was sent to ISO.
Claim, Exposure	ISOReceiveDate	Boolean	The last time a response was received by ISO for this exposure.
Claim, Exposure	ISOKnown	Boolean	A flag whether this exposure been successfully received by ISO.
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 374 and “ISO Payload XML Customization” on page 370. This property is viewable in the user interface for administrative users, as described further in “ISO User Interface” on page 365.
Claim, Exposure	ISOMatchReports	ISOMatchReport[]	An array of zero, one, or more ISO match reports for this exposure.

The following are fields of special interest 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 typelist	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.

Note: 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	`config/web/pcf/claim/exposures/iso/ISODetailsDV.pcf`
ISO list view for displaying a list of match reports	`config/web/pcf/claim/exposures/iso/ISOMatchReportsLV.pcf`
ISO detail for displaying an individual match report	`config/web/pcf/claim/exposures/iso/ISOMatchReportDV.pcf`

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 sees 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 374 and "ISO Payload XML Customization" on page 370.

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 363.

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.

Note: 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
`ISO.AgencyId`	Set to the `AgencyID` provided by ISO. For claim-level ISO messaging, you can instead specify this at the claim level.
`ISO.ConnectionURL`	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 `http://` *`proxyDomainOrIP:portnumber`*. Using the syntax of the Apache configuration file example file, use the URL `http://$DMZProxyDomainName:$DMZProxyPortA`
`ISO.CustLoginId`	Set to the login ID provided by ISO
`ISO.CustPswd`	Set to the ISO customer password provided by ISO
`ISO.RequireSecureServer`	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 349, 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

Property	Description
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 1_request.xml, 1_receipt.xml and 1_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 also in 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
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 349
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 372.
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 373.

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 =
ISOClaimParty.PT =
ISOClaimParty.TN =
ISOClaimParty.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:

```
ISOClaimSearch.ContactRole.Witness = witness
ISOClaimSearch.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 369.

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`.

Note: 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
`LossType`	ISO `Loss Type` code
`ExposureType`	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,gl` 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.

IMPORTANT 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 370.

For additional type code mapping information relating to the ClaimCenter `ClaimContact` entity, see "Converting from ClaimContact (ClaimCenter) to ClaimParty (ISO)" on page 368.

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 346 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()`.

IMPORTANT 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 374.

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.

IMPORTANT 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.

Exporting ISO Rules From Guidewire Studio

You might want to export ISO-related rules from the ClaimCenter sample data/rule sets. For this situation, you can select a specific subset of data (such as ISO-related rules) to export. Later, you can import this data. See *ClaimCenter Rules Guide*.

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 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 349

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 367.

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 integration, see “Document Management”, on page 249.

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 superclass 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 `GWExposureISOEnhancement` to customize the `ISOLossSectionType` enhancement property. This defines the mapping of exposure type to loss section Gosu class.	See "ISO Payload XML Customization" on page 370
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 `null` 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 370.
Update ISO user interface	If you use exposure-level messaging only, change the PCF page definition (user interface definition) for the **Exposure Details** 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 365.
Update typecode mapping	Update the typecode mapping file with your data model changes.	See "ISO Type Code and Coverage Mapping" on page 369.
Update coverage mapping	Update the coverage mapping file with your data model changes.	See "ISO Type Code and Coverage Mapping" on page 369.
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 360.

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 370.

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 374.

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 67 in the *System Administration Guide*.

ISO-Specific Error Codes

Error code	Description
UA0103 (Policy)	Policy Number (PolicyNumber) missing or invalid.
UA0104 (ClaimsOccurrence)	Policy Type (LOBCd) missing or invalid.
UA0109 (ClaimsOccurrence)	Claim Number (ItemIdInfo/InsurerId) missing or invalid.
UA0110 (ClaimsOccurrence)	Date of Loss (LossDt) missing or invalid.
UA0119 (ClaimsOccurrence)	Location of Loss State (Addr/State) missing or invalid.
UO0101 (ClaimsParty)	Role Code (ClaimsPartyInfo/ClaimsPartyRoleCd) missing or invalid.
UO0103 (ClaimsParty)	Business Name (CommlName) missing or invalid.
UO0104 (ClaimsParty)	Last Name (PersonName/Surname)missing or invalid.
UO0105 (ClaimsParty)	First Name (PersonName/Surname) missing or invalid.
UO0115 (ClaimsParty)	Address Line 1 (Addr/Addr1) missing or invalid.
UO0117 (ClaimsParty)	Address (Addr/City) missing or invalid.
UO0118 (ClaimsParty)	Address (Addr/StateProvCd) missing or invalid.
UC0106 (AdjusterPartyInfo)	Loss Type (LossCauseCd for Casualty Claim) missing or invalid.
UC0107 (AdjusterPartyInfo)	Coverage Type (CoverageCd for Casualty Claim) missing or invalid.
UC0108 (ClaimsInjuredInfo)	Alleged Injuries/Property Damage (ClaimsInjury/InjuryNatureDesc) missing or invalid.
UP0106 (AdjusterPartyInfo)	Loss Type (LossCauseCd for Property Claim) missing or invalid.
UP0206 (AdjusterPartyInfo)	Loss Type (LossCauseCd for Boat or Mobile Equipment) missing or invalid.
UP0207 (AdjusterPartyInfo)	Coverage Type (CoverageCd for Boat or Mobile Equipment) missing or invalid.
UP0210 (Watercraft)	Boat Year (ItemDefinition/ModelYear) missing or invalid.
UP0211 (Watercraft)	Boat Make (ItemDefinition/Manufacturer) missing or invalid.
UP0217 (Watercraft)	Boat HIN (ItemDefinition/SeraialIdNumber) missing or invalid.
UP0226 (ItemInfo)	Mobile Equipment Year (ItemDefinition/ModelYear) missing or invalid.
UP0227 (ItemInfo)	Mobile Equipment Make (ItemDefinition/ManufacturerCd) missing or invalid.
UP0228 (ItemInfo)	Mobile Equipment Model (ItemDefinition/ModelCd) missing or invalid.
UP0230 (ItemInfo)	Mobile Equipment VIN (SerialIdNumber) missing or invalid.
UP0236 (RecoveryInfo)	Date of Recovery (RecoveryDt) missing or invalid.
UP0237 (RecoveryInfo)	Recovery Agency (RecoveryAgencyRef - requires ClaimsParty for Recovery Agency as well) missing or invalid.
UP0242 (RecoveryInfo)	Condition of Recovered Vehicle (RecoveryStatusCd) missing or invalid.
UV0106 (AdjusterPartyInfo)	Loss Type (LossCauseCd for Vehicle Claim) missing or invalid.
UV0107 (AdjusterPartyInfo)	Coverage Type (CoverageCd for Vehicle Claim) missing or invalid.
UV0108 (AutoLossInfo)	Vehicle Year (VehInfo/ModelYear) missing or invalid.
UV0109 (AutoLossInfo)	Vehicle Make (ManufacturerCd) missing or invalid.
UV0116 (AutoLossInfo)	VIN (VehInfo/VehIdentificationNumber) missing or invalid.
UV0142 (RecoveryInfo)	Date of Recovery (RecoveryDt) missing or invalid.
UV0143 (RecoveryInfo)	Recovery Agency (RecoveryAgencyRef - requires ClaimsParty for Recovery Agency as well) missing or invalid.
UV0148 (RecoveryInfo)	Vehicle Recovery Condition (RecoveryStatusCd) missing or invalid.
UV0905 (SalvageInfo)	Date of Salvage (SalvageDt) missing or invalid.
UV0909 (SalvageInfo)	Owner Retaining Salvage Indicator (OwnerRetainingSalvageInd) missing or invalid.

Error code	Description
UV0911 (SalvageInfo)	Buyers Business Name missing or invalid. The Buyer is reported in its own ClaimsParty aggregate with a BuyerRef in the SalvageInfo (CommlName is missing or invalid.)
UV0912 (SalvageInfo)	Buyers Last Name missing or invalid. The Buyer is reported in its own ClaimsParty aggregate with a BuyerRef in the SalvageInfo (PersonName/Surname is missing or invalid.)
UV0913 (SalvageInfo)	Buyers First Name missing or invalid. The Buyer is reported in its own ClaimsParty aggregate with a BuyerRef in the SalvageInfo (PersonName/GivenName is missing or invalid.)
UV0914 (SalvageInfo)	UV01 (AutoLossInfo) Must Proceed the UV09 (SalvageInfo)
US0102 (Policy)	Policy Number (PolicyNumber) missing or invalid.
US0103 (ClaimsOccurrence)	Claim Number (ItemIdInfo/InsurerId) missing or invalid.
US0104 (ClaimsOccurrence)	Date of Loss (LossDt) missing or invalid.
US0202 (Policy)	Policy Number (PolicyNumber) missing or invalid.
US0203 (ClaimsOccurrence)	Claim Number (ItemIdInfo/InsurerId) missing or invalid.
US0204 (ClaimsOccurrence)	Date of Loss (LossDt) missing or invalid.
US0206 (com.iso_Update)	Business Name (com.iso_OriginalFields/CommercialName) missing or invalid.
US0207 (com.iso_Update)	Last Name (com.iso_OriginalFields/PersonName) missing or invalid.
US0208 (com.iso_Update)	First Name (com.iso_OriginalFields/PersonName) missing or invalid.
US0303 (ClaimsOccurrence)	Claim Number (ItemIdInfo/InsurerId) missing or invalid.
US0304 (ClaimsOccurrence)	Date of Loss (LossDt) missing or invalid.
US0306 (com.iso_Update)	Business Name (com.iso_OriginalFields/CommercialName) missing or invalid.
US0307 (com.iso_Update)	Last Name (com.iso_OriginalFields/PersonName) missing or invalid.
US0308 (com.iso_Update)	First Name (com.iso_OriginalFields/PersonName) missing or invalid.
US0310 (AdjusterPartyInfo)	Coverage Type (CoverageCd) missing or invalid.
US0311 (ClaimsPayment)	Claim Status (ClaimsPaymentCovInfo/ClaimStatusCd) missing or invalid.
US0312 (ClaimsPayment)	Estimate Amount (ClaimsPaymentCovInfo/PaymentAmt) missing or invalid.
US0313 (ClaimsPayment)	Reserve Amount (ClaimsPaymentCovInfo/PaymentAmt) missing or invalid.
US0314 (ClaimsPayment)	Amount Paid to Date (ClaimsPaymentCovInfo/PaymentAmt) missing or invalid.
US0402 (Policy)	Policy Number (PolicyNumber) missing or invalid.
US0403 (ClaimsOccurrence)	Claim Number (ItemIdInfo/InsurerId) missing or invalid.
US0404 (ClaimsOccurrence)	Date of Loss (LossDt) missing or invalid.
US0406 (ClaimsParty)	Business Name (GeneralPartyInfo/NameInfo/CommlName) missing or invalid.
US0407 (ClaimsParty)	Last Name (GeneralPartyInfo/NameInfo/Surname) missing or invalid.
US0408 (ClaimsParty)	First Name (GeneralPartyInfo/NameInfo/GivenName) missing or invalid.
US0410 (ClaimInvestigationAddRq)	Action Indicator (SearchBasisCd) missing or invalid.
US0501 (com.iso_Update)	Original Insurance Company (com.iso_OriginalFields/ItemIdInfo/AgencyId) missing or invalid.
US0502 (com.iso_Update)	Policy Number (com.iso_OriginalFields/PolicyNumber) missing or invalid.
US0503 (com.iso_Update)	Claim Number (com.iso_OriginalFields/ItemIdInfo/InsurerId) missing or invalid.
US0504 (com.iso_Update)	Date of Loss (com.iso_OriginalFields/LossDt) missing or invalid.
US0505 (Policy)	New Insurance Company (MiscParty/ItemIdInfo/AgencyId) missing or invalid.
US0506 (Policy)	New Policy Number (PolicyNumber) missing or invalid.
US0507 (ClaimsOccurrence)	New Claim Number (ItemIdInfo/InsurerId) missing or invalid.
US0508 (ClaimsOccurrence)	New Date of Loss (LossDt) missing or invalid.

Error code	Description
UF0001	Request would create a duplicate record. The system uses date of loss, insuring company code, policy number and claim number as unique identifiers. Those identifiers can only be added to the database one time for initial claims otherwise a duplicate claim is rejected.
UF0002	Every claim must contain either a role of CI or IN.
UF0003	A coverage record required for roles: CL, CP, CI, CD. If you submit any of these roles as described in Appendix D: • there must be a UC01 (`ClaimsInjuredInfo`) for casualty • there must be a UP01 or UP02 (`PropertyLossInfo`) for property • there must be a (UV01) `AutoLossInfo` for vehicle claims referencing the Involved Party
UF0004	A UP01 (`PropertyLossInfo`) can only exist for the roles of IN, CI, TN, PT, IP and ID.
UF0005	A UV01 (`AutoLossInfo`) can only exist for the roles of CL, CI, IN, CD, CP, ID and IP.
UF0006	A UC01 (`ClaimsInjuredInfo`) can only exist for the roles of CL, CP, CD, CI, IN, ID, IP, IE and CE.
UF0007	Every claim must have at least one coverage record. A coverage record is either a UC01 (`ClaimsInjuredInfo`), UP01 or UP02 (`PropertyLossInfo`), or UV01 (`AutoLossInfo`).
UF0008	Coverage/Loss combination is invalid.
UF0009	Policy/Coverage combination is invalid.
UF0010	Initial claim not found. ISO tries to match the initiating claim as follows: • the same date of loss • the same insuring company code • the same policy number • the same claim number. If no initial claim is found, the records are rejected.
UF0011	Invalid message keys. This error applies if using the XML or MQ communication methods and indicates an overall hierarchy failure of the claim.
UF0012	UP01 (`PropertyLossInfo/ClaimsSubjectInsuranceInfo`) or UP02 (`PropertyLossInfo/ItemInfo` or `PropertyLossInfo/Watercraft`) can only exist once in a claim. Because General Property (fire, theft, other peril) and Boat or Mobile Equipment claims are first party reporting, they can only be reported once within a claim.
UF0013	Duplicate Coverage and Loss Combination Code. The claim contains the same ClaimsParty with the same coverage type with the same loss type more than once.
UF0014	At least one stolen item must be reported. On a property theft record, at least one item must be reported as stolen.
UF0015	Auto Loss Type Theft can only be reported once in a claim

IMPORTANT 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 learn about the error. Encourage them to add the new error information to the ISO documentation.

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 348.
- **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.

Conversion: 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.

chapter 20

FNOL Mapper

The FNOL mapper is a ClaimCenter integration tool that imports First Notice of Loss reports (initial claim reports) from a standard XML-based file formats, including the ACORD XML format. If you use the ACORD format or another format, you can customize the mappings by modifying built-in Gosu classes or writing new mapping classes. The mappers parse the XML and populate ClaimCenter objects and add subobjects as appropriate. Due to Gosu's native XML processing, you do not need to parse the XML explicitly as text.

This topic includes:

FNOL Mapper Overview

The FNOL mapper is a ClaimCenter integration tool that imports First Notice of Loss reports (initial claim reports) typically from an industry standard XML-based file format called *ACORD*.

The ACORD XML format is a standard format but variants of the ACORD 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 there are additional properties on an exposure, the mapping must specify how the map properties to the ClaimCenter datamodel. The names of properties or typecodes might vary between the external system and ClaimCenter. If you have custom exposure types, you might need to change the exposure type based on other fields.

To customize the ACORD mapping, modify the built-in Gosu classes or writing new ones to specially handle each object. The included classes are already modular to encapsulate exposure mapping code apart from contact-mapping code and address-mapping code, for example.

You can also import from entirely new non-ACORD XML formats. To do this, you write new Gosu classes that implement a special mapper interface that indicates that your class can map the incoming XML.

After you complete your customization, test your code by invoking the mapper from an external system. An external system import san FNOL using web services that ClaimCenter publishes. The external system passes the XML data across the network as one large `String` object.

For ACORD XML data, the external system calls the `IClaimAPI` web service interface method `importAcordClaimFromXML`. If you wrote a custom mapper class for non-ACORD XML data, the external system calls the `IClaimAPI` web service interface method `importClaimFromXML`.

The XSD of ACORD allows multiple claims to be included in one ACORD XML. However, the current version of the FNOL mapper tool requires only one claim per ACORD file.

An ACORD document can have many `<MClaimSvcRq>` subelements. Each of those can have many `<CLAIMSSVCRQMGS>` subelements below that with a `<xsd:CHOICE>` element, one of which is `<ClaimsNotificationAddRq>`. The default implementation chooses the first valid `<ClaimsNotificationAddRq>` element and uses it. If there are not valid `<ClaimsNotificationAddRq>` elements, Gosu throws an exception.

FNOL Mapper Detailed Flow

Of all the server files for the FNOL mapper, 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 to a `Claim` entity and its subobjects. This only has a single method on it.

For ACORD XML data, external systems call the `IClaimAPI` web service interface method `importAcordClaimFromXML`. For a custom mapper class for non-ACORD XML data, external systems calls the method `importClaimFromXML`. The `importClaimFromXML` method takes an extra parameter for the name of the mapper class you want to use. The code to invoke the mapper for ACORD data from Java web services client code is fairly straightforward:

```
String myClaimPublicID = claimAPI.importAcordClaimFromXML(myXMLData);
```

Note: For more information about ClaimCenter web services, see “Web Services (SOAP)” on page 25.

For both these methods, the web service implementation itself is relatively simple because most of the work is done by the mapper classes.

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 mapper class’s `populateClaim` method with the new `Claim` entity. Its 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 and sets fields on the `Claim` entity and adds 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.

Structure of FNOL Mapper Classes

To support basic ACORD files, ClaimCenter includes 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	`gw.fnolmapper.acord.IAddressMapper`	`gw.fnolmapper.acord.impl.AcordAddressMapper`
contact	`gw.fnolmapper.acord.IContactMapper`	`gw.fnolmapper.acord.impl.AcordContactMapper`
exposure	`gw.fnolmapper.acord.IExposureMapper`	`gw.fnolmapper.acord.impl.AcordExposureMapper`
incident	`gw.fnolmapper.acord.IIncidentMapper`	`gw.fnolmapper.acord.impl.AcordIncidentMapper`
policy	`gw.fnolmapper.acord.IPolicyMapper`	`gw.fnolmapper.acord.impl.AcordPolicyMapper`

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 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.

IMPORTANT For information about manipulating XML objects from Gosu, see “Gosu and XML” on page 245 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 81 in the *Gosu Reference Guide*. 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 Util

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.

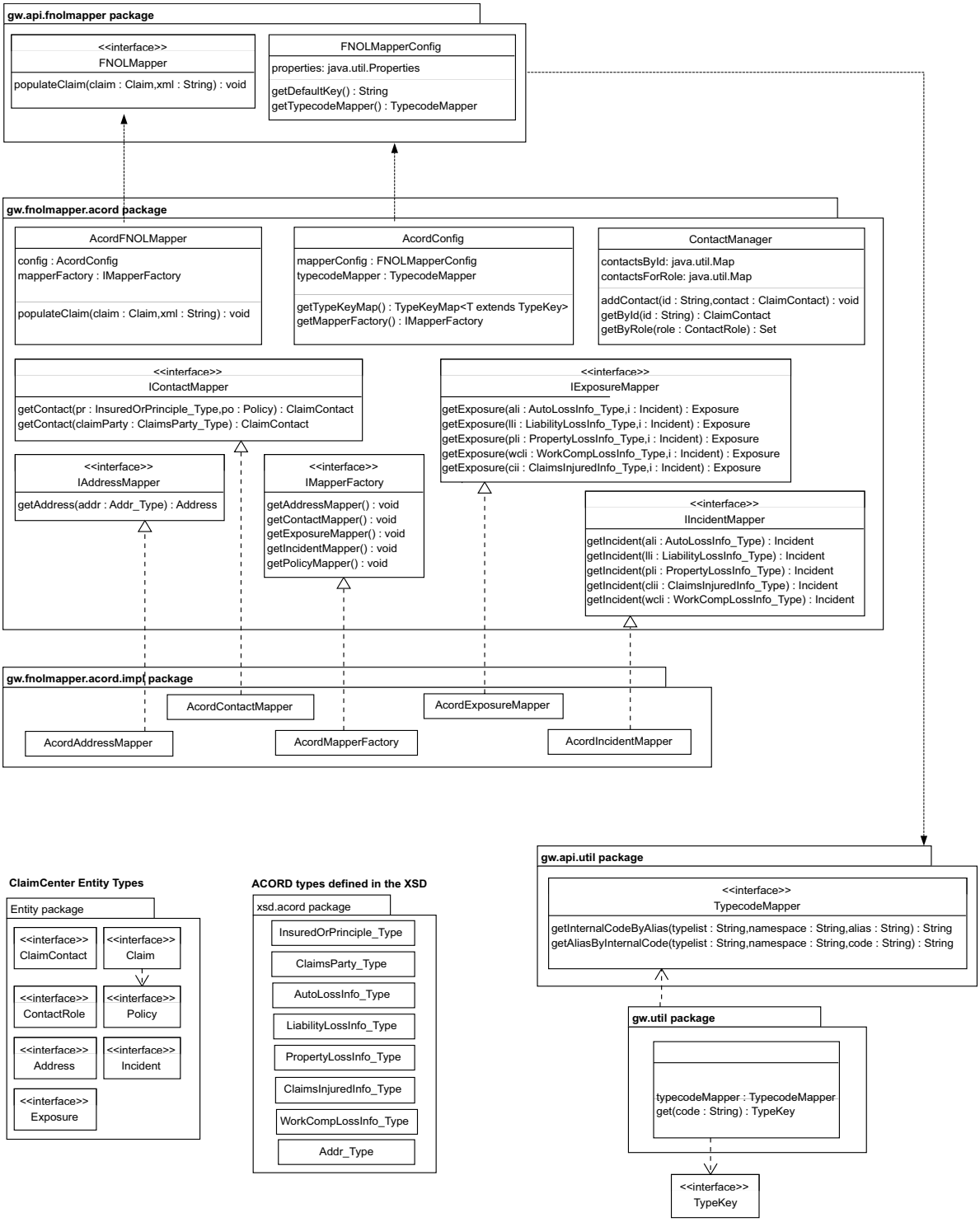
ACORD XSD Type Enhancements

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, 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.ABCMapperFactory(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: On a related note, just like the exposure mapper, the incident mapper interface also 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 387 in the *Gosu Reference Guide* 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:** On a related note, just like the exposure mapper, the incident mapper interface also 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 384.

chapter 21

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:

See also

For more information about the services that the Metropolitan Reporting Bureau provides, see the web site:

`http://www.metroreporting.com`

Overview of ClaimCenter-Metropolitan Integration

ClaimCenter integrates with Metropolitan Reporting Bureau (hereafter, Metropolitan) using their so-called *XML Gateway* for requesting police accident and incident reports. 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 (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, and ClaimCenter supports most of the current 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 in 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 attachs 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 within the ClaimCenter user interface. One way is to click the **New** button after the Loss Detail page is in edit mode. Alternatively, click on the Report Type links from the list view of claims.

The Metropolitan Reports list view looks like this:

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 “MetroReportStatus and How It Changes” on page 397.
- **Order date.** The date ClaimCenter send the report request to Metropolitan. The order date is empty if the report status is “InsufficientData”.
- **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.
- **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 pre-update rules. ClaimCenter can send a new report request if it passes pre-update rules.

Within the Metropolitan Report user interface, the user can select the report type and 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 user requests a Metropolitan report, returns to the **Loss Details** page. You see a new row in the Metropolitan reports table that indicates the report status as “`New`”.

The following table lists the current report types and their report type codes:

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 know how 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.
- **Pre-update rules.** Claim pre-update 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 meant 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 249.

- **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, but later in this topic 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 to 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 so-called *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 415.

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 display properties file, `display.properties`.

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 also defined within `config.xml`. You can make minor configuration changes (such as retry times) to these *destination* settings. See “Metropolitan Error Handling” on page 399.

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, the ClaimCenter application must poll the Metropolitan server regularly. One of the most important configuration parameters that you can make is the time interval between inquiries. This 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:

```
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 typical properties in a `Metro.properties` file:

```
Metro.CustNAIC          = 11882
Metro.CustCompanyName   = Allstate Insurance
Metro.CustAddress1      = 123 West Ave
Metro.CustAddress2      =
Metro.CustCity          = Erie
Metro.CustState         = PA
Metro.CustZip           = 41222
```

```
Metro.CustAccount       = Demo
Metro.CustBillingAccount= MYCODE
```

There are other properties in that file that typically do not require modification. Only change the other properties if Metropolitan changes the XML formats for date, time, and social security numbers. These property formats are in standard regular expression format.

ClaimCenter Display Properties for Metropolitan

Within the ClaimCenter display properties file (`display.properties`), define your own error messages and property names for claim pre-update business rules.

You can modify many of the parameters, such as the following strings:

```
Metro.Activity.InsufficientData.Message = Missing field\(s\)\:
Metro.Fields.AgentCity = City of the investigating agency
Metro.Fields.ClaimNumber = Claim Number
```

The earlier error messages and property names are used (in pre-update rules) if required properties are not defined. ClaimCenter creates an activity that indicate the missing properties so you know where to fix the report request, after which you can resubmit the request.

There are many other parameters that begin with "`Metro.`" such as the examples earlier and you can modify all of them. Do not modify any other Metropolitan properties that do not begin with "`Metro.`", since they may be implementation details critical to proper functioning and they may change in the future.

This file includes other configuration options such as the login name, password, and other parameters necessary to integrate with Metropolitan. Do not update these properties because these properties are common to *all* ClaimCenter customers and thus are correct in the built-in configuration.

Configuring Activity Patterns

The Metropolitan integration includes three *activity patterns*, which are a type of template for a user activity notification. These activity patterns can be modified directly in addition to the configurable strings discussed in "ClaimCenter Display Properties for Metropolitan" on page 393.

The Metropolitan activity patterns are:

- `metropolitan_request_failed` - For any report request that fails due to incomplete data or if initial order message fails. This activity pattern creates an `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.
- `metropolitan_report_unavailable` - For any report request that fails after sending the request because no report is available. ClaimCenter creates an `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.
- `metropolitan_report_available` - If a report request succeeds and the report is attached it to the claim, ClaimCenter creates an `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_held` - If Metropolitcan tells ClaimCenter that it needs additional information from ClaimCenter implementation before Metropolitan can process it, ClaimCenter creates this activity. ClaimCenter creates this activity this if Metropolitan replies with the response `hold`. The Metropolitan workflow uses this activity pattern.
- `metropolitan_report_deferred` - For any report requests that takes additional time to process. This is used if Metropolitan replies with the response `deferred`. The Metropolitan workflow uses this activity pattern.

- `metropolitan_report_inquiry_failed` - Used if a request to Metropolitan other than the initial order request fails. An activity is generated and then the workflow retries. The Metropolitan workflow uses this activity pattern.

For more information about configuring activity patterns, see the *ClaimCenter Configuration Guide*.

Configuring the Messaging Plugin Retries

If the messaging plugins fail to send to Metropolitan, they retries after a time delay between tries, up to a maximum number of times. If Metropolitan does not acknowledge receipt successfully, the messaging plugins retries up to the maximum at the specified interval. The default retry maximum is five times at an interval of 1000 milliseconds.

To change this, refer to the server `config.xml` file and change the values in the `metro` destination section, which is highlighted in the following configuration example in bold:

```
<destination name="metro" id="11" requestplugin="metroRequest"
  transportplugin="metroTransport"
  initialretryinterval="1000"
  maxretries="5"
  retrybackoffmultiplier="2">
  <event name="MetroReportAdded"/>
  <event name="MetroReportChanged"/>
</destination>
```

For more information about messaging and destinations in general, see “Messaging and Events”, on page 139. However, this is a built-in destination and cannot be modified other than the configuration settings documented in this topic.

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.

MetroReportTypes 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.xml` specifies the mapping between a Metropolitan report type and a ClaimCenter line of business (LOB). 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, so Gosu templates and pre-update 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 may 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 (one template per report) are stored within the directory:

```
ClaimCenter/cc/modules/configuration/config/web/templates/metroreport/...
```

For example, the `BurglaryReport.gs` template generates the Metropolitan burglary report request. You can modify the templates but do not change the header or the footer.

WARNING You can modify Metropolitan report Gosu templates but do not change the template's header or its footer.

A sample report template looks like this:

```
<mrb:Insured>
  <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 the example earlier, the templates use *translators* to format the data using an object with the symbol `translator`. This utility class allows 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 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 `.gs`. The following example include one `<report>` element that maps the Metropolitan report `A` (auto accident report) to a specific Gosu template file:

```
<Report>
  <Type>A</Type>
  <Template>MetroAutoAccidentReport.gs</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. ClaimCenter assumes that at the time the auto accident happened, the car was parked so there was no driver inside.

Worker's Compensation

In the ClaimCenter reference implementation, 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.xml`.
5. Update the ClaimCenter pre-update rules to check for all required properties for the new report type.
6. Regenerate Java API and SOAP API files, as described on "Regenerating the Integration Libraries" on page 17.
7. Rebuild and redeploy the application WAR file to make the new mappings and files available at runtime.

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

The `MetroReportType` typelist defines a type of report that adjusters can request, such as Auto Accident, Auto Theft, Coroner Reports and Title History, and others. There is a mapping between loss detail type and `MetroReportType`. For example `MetroReport` type Auto Accident is only available for the type of loss Auto. The Typelist and the mapping is specified by `MetroReportType.xml`. Example:

```
<typecode code="G" name="Burglary" desc="Burglary" priority="7">
   <category typelist="LossType" code="AUTO"/>
   <category typelist="LossType" code="PR"/>
 </typecode>
```

MetroAgencyType Entity

The investigating agency type for each report is defined by a typelist called `MetroAgencyType`. Each `MetroReport` entity has an associated `MetroAgencyType`. The user who requests the report from the user interface can specify the investigating agency type. This property can be `null` if the requester does not have the information. `MetroAgencyType` is defined in `MetroAgencyType.xml`.

The following are example mappings of agency types with a 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 a report successfully returns from Metropolitan, the built-in code requests that the *document content source* plugin (`IDocumentContentSource`) add the document to the claim.

MetroReportStatus and How It Changes

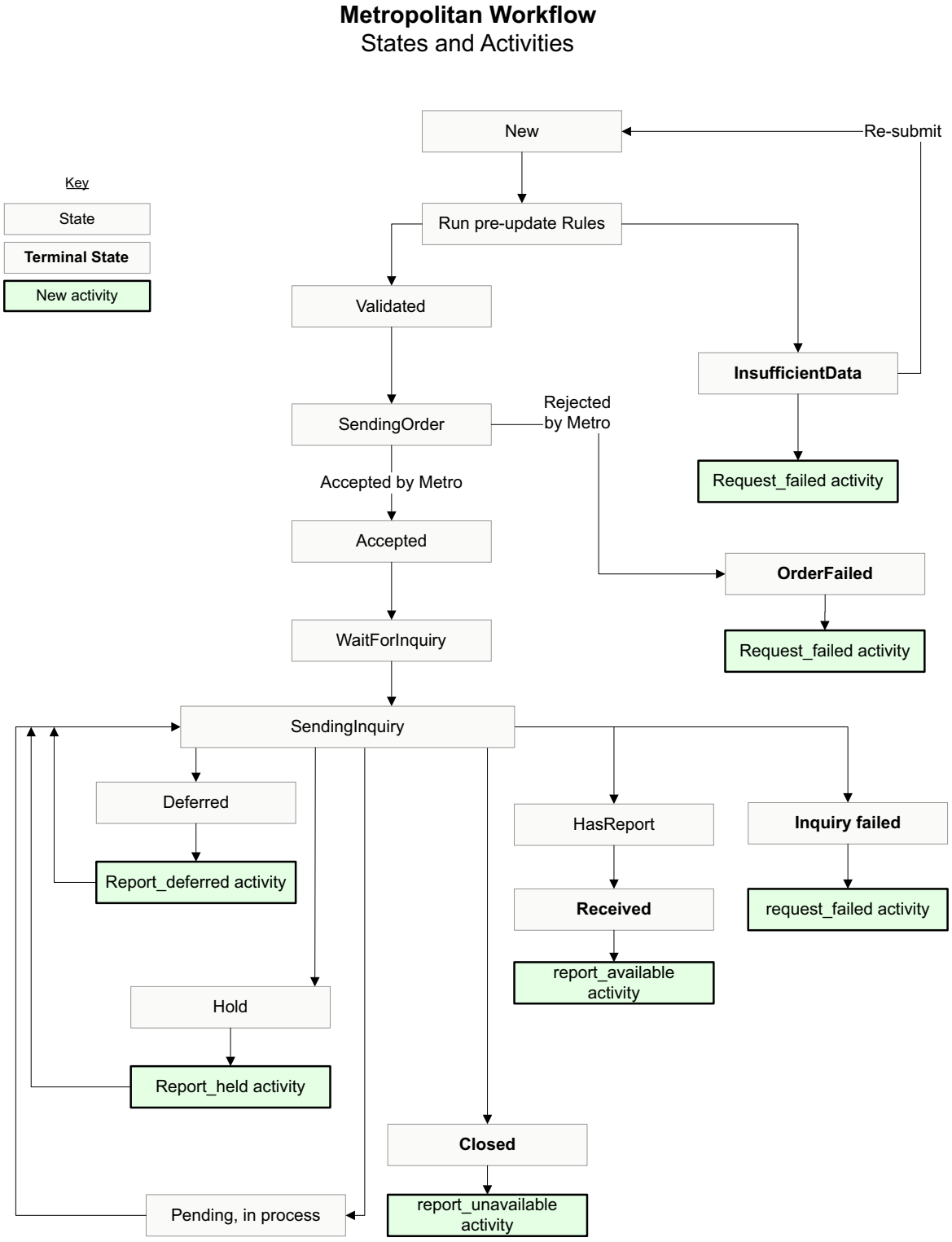
`MetroReportStatus` indicates the current status of the report request. You cannot modify nor extend this status list. This document lists the values to provide a sense of the internal state machine's flow during a typical report request. This also helps you understand the meaning of user-visible text within the application user interface .

The possible values of the `MetroReportStatus` entity are:

Metro report status	Meaning
`none`	No order was located that matched the information supplied.
`pending`	Order is in the system and currently awaiting a response from the data source.
`mailed`	A result was sent to you and the order is closed. Usually this is the case if the matched item is old and the image is no longer in the active files.
`image`	An image was supplied for this order. The order is closed.
`hold`	The order is awaiting additional information from you.
`deferred`	An image was returned with a notice that the data source takes some additional time to provide the requested information. Order is still open.
`inprocess`	The order was received in the last few hours and has not yet been acted upon. This would only be the case within the first 24 hours after receiving a report. Depending on the time of day the report is sent, it would be acted upon usually within a few hours
`closed`	The order was canceled or cannot be completed for some reason.
`error`	The inquiry sent could not be matched to any existing order in the Metropolitan system.

You **cannot** modify the state machine. Refer to the following diagram for an overview of the process. Note the relationship between report status strings in the user interface and Metropolitan activity patterns (see "Configuring Activity Patterns" on page 393). The only ways to configure the Metropolitan integration state machine is

the `InquiryInterval` parameter and the other settings documented in this topic. The following diagram defines the status changes within the internal Metropolitan state machine.



Customizing Metropolitan Timeouts

The built-in Metropolitan reports integration has two time cycles in the workflow:

- A delay after sending the request before requesting the report
- After asking for the report, if the Metropolitan server does not have the report, the workflow sleeps for a delay. Next, it repeats this step to check again if the report is ready.

Both of these two time cycles have associated time values in the workflow. The defaults are as follows:

- After submitting the original request, ClaimCenter waits 2.5 hours before inquiring initially. You can modify this value in the `metro.properties` file in the property `Metro.InquiryInterval`.
- After requesting the report, ClaimCenter waits for the response 8 hours. You can modify this timeout in the `metro.properties` file in the property `Metro.OrderTimeout`. This is exposed in Gosu as the `MetroReport` 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 1 week, then ClaimCenter stops it and create 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 timeout value.

In the `metro.properties` file, specify the time values using a number followed by a suffix for the unit. Use `d` for days, `h` for hours, and `m` for minutes. For example, `2d` means *two days*. You can use a combination of values, such as `3d2h` represents the sum of *three days* and *two hours*.

Metropolitan Error Handling

Keep in mind the following facts about error-handling with Metropolitan:

- Pre-update 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* to 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. If no acknowledgement returns from Metropolitan, the messaging plugins retries up to the maximum at the specified time interval. For more information, see “Configuring the Messaging Plugin Retries” on page 394.
- 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`

chapter 22

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.

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.

This topic includes:

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.

WARNING Carefully consider security implications of any integrations with external systems for properties that normally are encrypted. 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.

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 ContactCenter
- 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 the column in the data model files by overriding the column information (adding a *column override*).

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 property `encrypted_column` to be the type `EncryptedString`. With this approach, 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 `LastNameDenorm` is a denormalized column added to mirror the `LastName` property, and thus the `LastName` 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” comparison or “less than” comparison.

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 of `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 to 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 the last time the server ran.

For important details, see “Changing Your Encryption Algorithm Later” on page 406.

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()
  }

}
```

If You Import Data from Staging Tables

Suppose you have existing production data and you modify the data model. If you increment the data model version number, the system automatically encrypts your data the first time you launch the server during upgrade. This drops your staging tables during this process if you are using staging table import. Consequently, perform your basic data encryption data model changes and upgrade **before** beginning any staging table import work.

You can change your encryption algorithm later. See “Changing Your Encryption Algorithm Later” on page 406.

Installing Your Encryption Plugin

After you have written your implementation of the `IEncryption` plugin, you must register your plugin.

Perform the following steps to enable your encryption plugin

1. Create a new class that implements the `IEncryption` plugin interface. Be certain that your `EncryptionIdentifier` method returns a unique encryption identifier. If you ever 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 the algorithm itself. For example, `"encryptDES3"`.

2. In Studio, navigate to **Resources** → **Configuration** → **Plugins** → **gw** → **plugins** → **util** → **IEncryption**.
3. Right-click on `IEncryption` and choose **Add Implementation...**
4. Because you can have more than one registered implementation of an `IEncryption` plugin, Studio prompts you to name your new plugin. This name must be unique among all your plugins. This is called the *plugin name*.

   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"`. Do not confuse the *plugin name* with the *full class name* or the *encryption ID*. These text values are all used in different ways.

5. Edit standard plugin fields in the Plugins editor in Studio. See “Using the Plugins Editor” on page 141.
6. There is a `config.xml` configuration parameter called `CurrentEncryptionPlugin`. It specifies which encryption plugin (among potentially multiple versions) is the current encryption algorithm for the main database. Set this parameter to the *plugin name* (not the class name, not the encryption ID) for the current encryption plugin. If this configuration parameter is missing, the application assumes that the current algorithm `IEncryption` plugin with name `"IEncryption"` is the current encryption plugin. Any legacy plugins for which you did not choose a name have the default name `IEncryption`.

   Note: You can register any number of `IEncryption` plugins (just as with messaging plugins). This is to support changing your encryption algorithm later. For related information, see “Changing Your Encryption Algorithm Later” on page 406.

7. 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.

   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 Archiving and Snapshots” on page 407

If you want to change your encryption algorithm, it is critical to review the section “Changing Your Encryption Algorithm Later” on page 406.

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 Archiving and Snapshots” on page 407

Changing Your Encryption Algorithm Later

As mentioned in “Writing Your Encryption Plugin” on page 403, you can register any number of `IEncryption` plugins (just as with messaging 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 implementations than two if you have encrypted claim snapshots in archive databases, as is discussed later.)

The Current Encryption Algorithm

Regardless the number of registered encryption plugins, only one encryption plugin is the *current encryption plugin*. The `config.xml` configuration parameter `CurrentEncryptionPlugin` specifies which registered plugin is the current encryption plugin for the main database. Set the parameter to the plugin name for the current encryption plugin, not its class name nor its encryption ID.

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, it encrypts the encrypted fields with the new encryption plugin (found by plugin name as specified by the parameter `CurrentEncryptionPlugin`).

The most important things to remember for changing encryption algorithms are

- all encryption plugins must return the appropriate *encryption ID* correctly
- all encryption plugins must implement `getEncryptedLength` correctly
- 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, just ensure that every encryption plugin returns its appropriate *encryption ID* and set `CurrentEncryptionPlugin` to the correct *plugin name*.

Be careful not to confuse a plugin’s *encryption ID* with its *plugin name* or *class name*. The application 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 with each claim snapshot (with each encrypted field).

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 Archiving and Snapshots” on page 407 and “Encryption Changes with Archiving and Snapshots” on page 407.

Changing the Current Encryption Algorithm

The following procedure describes how to change your encryption algorithm. It is extremely important to follow the procedure 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.

To change the current encryption algorithm

1. Shut down your server.
2. Register a new plugin implementation of the `IEncryption` plugin for your new algorithm. (see more instructions in “Writing Your Encryption Plugin” on page 403.) 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. Start the server. The upgrader uses the old encryption plugin to decrypt your data and then reencrypt it with the new algorithm.
6. Archived claims are not upgraded immediately as part of the normal upgrader. See “Encryption Changes with Archiving and Snapshots” on page 407 and “Special Issues For Changing Your Encryption Algorithm” on page 408.

   WARNING ClaimCenter archive databases **must** have the same encryption status (encrypted or unencrypted) for each field as in the main database. It is unsafe for data integrity to attempt otherwise. See “Encryption Changes with Archiving and Snapshots” on page 407 for important archive upgrade information.
7. Claim snapshots in the main database are not upgraded immediately as part of the normal upgrader. See “Encryption Changes with Archiving and Snapshots” on page 407 and “Special Issues For Changing Your Encryption Algorithm” on page 408.
8. Continue to register your older encryption plugins so that ClaimCenter can decrypt any encrypted data in claim snapshots in archived claims. See “Encryption Changes with Archiving and Snapshots” on page 407 and “Encryption Changes with Archiving and Snapshots” on page 407.

Encryption Changes with Archiving and Snapshots

On server startup with archive mode off, the following changes trigger the upgrader to update encryption settings in the main database:

- Adding encrypted properties
- Removing encrypted properties
- Adding an encryption plugin for the first time.

- Changing 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 406

However, the upgrader **does not upgrade claim snapshots** during the normal upgrade process.

After startup, a work queue runs that eventually converts all claim snapshots to use the current encryption settings (decrypting if necessary then encrypting). For details of the Encryption Upgrade work queue, see “Batch Processes and Distributed Work Queues” on page 134 in the *System Administration Guide*.

However, claim snapshots also upgrade on demand if necessary. In other words, suppose the work queue is running but is not complete and did not yet 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 upgraded at one time, set the `SnapshotEncryptionUpgradeChunkSize` configuration parameter to the number of claims to upgrade each time the worker runs. This is typically once a day.

To force the snapshot upgrade to occur all at once, set the parameter to 0 (process all snapshots at once). Next, manually run the Encryption Upgrade work queue writer from the tools page after the usual database upgrade completes.

IMPORTANT Think carefully before `SnapshotEncryptionUpgradeChunkSize` to 0 to process all snapshots. This can take a very long time. Your servers could be busy running worker threads to decrypt the snapshots for a long time instead of one chunk every night at 1 AM when usage needs are low.

This information applies only to claim snapshots in the main database. For information about claim snapshots in the archive databases, see “Archived Claims” on page 408.

Archived Claims

On server startup with archive mode off, the upgrader does **not upgrade archived claims** during the normal upgrade process. To upgrade claims in archive databases, simply start the server in *archive mode*.

However, this process does **not upgrade claim snapshots in archived claims**. If an archived claim is restored at a later time, the server upgrades the claim snapshot to the current encryption settings as part of restoring the claim.

In contrast to the main database, if the server is in archive mode, the work queue to upgrade claim snapshots does **not** run. The upgrade batch process does **not** run automatically on the archive server. Additionally, running the batch process manually on the archive server is unsupported. Any non-upgraded claim snapshots in archive databases remain in their non-upgrade state.

If the claim is ever restored, the server upgrades the encryption settings on the claim snapshots as part of restoring the claim.

IMPORTANT Any claim snapshots in archive databases with out-of-date encryption remain in their non-upgrade state until the claim is restored. The work queue that runs in the main database to upgrade snapshots does **not** run when the server is in archive mode.

Special Issues For Changing Your Encryption Algorithm

As mentioned earlier, the work queue that runs in the main database to upgrade snapshots does **not** run for claim archives. This means that claim snapshots in archived claims may have used encryption algorithms that are no longer the current encryption algorithm.

Because of this, even after running the upgrader in archive mode, you must continue to register your encryption plugins for your old encryption algorithm. The encryption ID for the plugin that encrypted the snapshot is stored

in the snapshot. If that claim is restored, ClaimCenter needs that encryption plugin to decrypt the snapshot as part of restoring the claim.

This means you must register multiple encryption plugins. Register one encryption plugin implementation for every encryption algorithm that might be referenced (by encryption ID) in any claim snapshots. Be sure that all your plugin implementations return the correct *encryption ID* so ClaimCenter can identify which plugin to use to decrypt the snapshots.

Encrypted Properties in Staging Tables

If you have any encrypted properties, you must **load the staging tables with this data pre-encrypted**.

If you ever need to change your encryption algorithm after data is encrypted, see “Changing Your Encryption Algorithm Later” on page 406.

chapter 23

Management Integration

This topic discusses *management*, which is a special type of API that can allow 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.

IMPORTANT This topic focuses on writing a new management plugin. To enable and use the **built-in JMX management plugin**, see "Enabling JMX with ClaimCenter" on page 91 in the *System Administration Guide*. To write code for an external system to communicate with this plugin, see "Integrating With the Included JMX Management Plugin" on page 413.

For more information about plugins, see "Plugin Overview" on page 101. For the complete list of all ClaimCenter plugins, see "Summary of All ClaimCenter Plugins" on page 106.

This topic includes:

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 MBeans page** – All of these items are exposed in the user interface in the MBeans section of the ClaimCenter **Server Tools** tab. This tab is accessible only to users with the `soapadmin` permission. For more information about the Server Tools, see the *ClaimCenter 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
  ```

IMPORTANT This topic focuses on writing a new management plugin. To enable and use the **built-in JMX management plugin**, see "Enabling JMX with ClaimCenter" on page 91 in the *System Administration Guide*. To write code for an external system to communicate with this plugin, see "Integrating With the Included JMX Management Plugin" on page 413.

The Abstract Management Plugin Interface

The following tabled lists the important interfaces and classes used by the abstract management plugin interface.

Interface or class	Description
`ManagementPlugin`	The management interface for Guidewire systems. This is the interface that the included JMX plugin implements.
`GWMBean`	The interface for *managed beans* exposed by the system. The management plugin must expose these internal management beans to the outside world using JMX, SMNP, or some other management interface.
`GWMBeanInfo`	Meta-information about a Guidewire managed bean (`GWMBean`), such as its name and description.
`ManagementAuthorizationCallbackHandler`	A callback handler interface that ClaimCenter invokes if a user attempts management operations.
`NotificationSenderMarker`	A marker interface that indicates that a `GWMBean` can send notifications.
`Attribute`	The `Attribute` class encapsulates two strings that represent a system attribute name and its value.
`AttributeInfo`	Metadata about an `Attribute`, such as name, description, type, and whether the attribute is readable and/or writable.
`Notification`	A ClaimCenter notification, such as those sent if ClaimCenter locks out a user due to too many failed login attempts.
`NotificationInfo`	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`

IMPORTANT This topic focuses on writing a new management plugin. To enable and use the **built-in JMX management plugin**, see "Enabling JMX with ClaimCenter" on page 91 in the *System Administration Guide*. To write code for an external system to communicate with this plugin, see "Integrating With the Included JMX Management Plugin" on page 413.

Integrating With the Included JMX Management Plugin

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 change one setting in the plugin registry in Studio to enable it.

IMPORTANT To enable and use the **built-in JMX management plugin**, see "Enabling JMX with ClaimCenter" on page 91 in the *System Administration Guide*. This topic is about writing external code that connects to this management plugin.

In the Plugins editor, select **Resources** → **Plugins** → **gw** → **plugin** → **management** → **ManagementPlugin**, then check the box for **Enabled**. Currently, this supported JMX management plugin works only with the Apache Tomcat web server currently. The source code for this example is included as part of the ClaimCenter examples directory.

IMPORTANT Because JMX support differs between web servers, 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.

To enable the JMX plugin or to configure it for use with the JMX console MC4J which is included on the ClaimCenter CD, see the *ClaimCenter System Administration Guide*.

Once the JMX plugin is enabled, 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 JMXCconnectorServer
            JMXConnector cntor = 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();
        }
    }
}
```

chapter 24

Proxy Servers

Guidewire recommends deploying proxy servers to insulate ClaimCenter from the external Internet. Incoming requests are the most dangerous interaction with the Internet. Because Insurance Service Office (ISO) sends incoming requests directly to ClaimCenter, proxy servers are important for integrating ISO with ClaimCenter. In addition, Guidewire recommends proxy servers for outgoing requests to Metropolitan Reporting Bureau and outgoing requests to geocoding and routing services.

This topic includes:

See also

- "Insurance Services Office (ISO) Integration", on page 339.
- "Metropolitan Reporting Bureau Integration", on page 389.
- "Geographic Data Integration", on page 315.
- For configuring web services URLs to support proxy servers, see "Working with Web Services" on page 145 in the *Configuration Guide*.
- For general information about web services, see "Web Services (SOAP)", on page 25.

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 to the server simply 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	`http://en.wikipedia.org/wiki/Public_key`
Detailed description of public key encryption	`ftp://ftp.pgpi.org/pub/pgp/7.0/docs/english/IntroToCrypto.pdf`
Detailed description of SSL/TLS Encryption	`http://httpd.apache.org/docs/2.0/ssl/ssl_intro.html`
Overview of Apache's SSL module	`http://httpd.apache.org/docs/2.0/mod/mod_ssl.html`
Overview of Apache's proxy server module	`http://httpd.apache.org/docs/2.0/mod/mod_proxy.html`

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. For more information, see "Working with Web Services" on page 145 in the *Configuration Guide*. For additional information about web services, see "Web Services (SOAP)", on page 25.

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. 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.

To install Apache

1. Download Apache HTTP server. Get it from `http://httpd.apache.org`.

   Note: The Apache configuration files blocks listed in this topic were designed for Apache 2.2.3. Guidewire has observed problems with Apache HTTP versions older than version 2.2.3. Do not use older versions.

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
`$DestinationTrustedCACertFile`	File containing the certificate used to sign the destination web site.
`$ReverseProxyTrustedCertFile`	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
`$ReverseProxyTrustedProtectedPrivateKeyFile`	File containing the private key used to decrypt the messages in the source to reverse proxy communication. This file is generally signed by a passphrase script, `$ReverseProxyTrustedPassPhraseScript`

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 pass-phrase. 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 419

ISO sends Message 3 asynchronously to ClaimCenter. For these message, use this configuration:

- "Upstream (Reverse) Proxy with Encryption for Service Connections" on page 419

Metropolitan Proxy Communication

For Metropolitcan reports, use the following configuration to connect to the Metropolitan Request URL:

- "Downstream Proxy With Encryption" on page 419

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.

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 420

For more information about SSL encryption, see the *ClaimCenter 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 $ReverseProxyTrustedProtectedPrivateKeyFile

  #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>
```

For more information about SSL support, see the *ClaimCenter System Administration Guide*.

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:

port	Specifies the port number for the additional connector for access through the proxy.
proxyName	Identifies the deployment server's name.
proxyPort	Specifies the port for encrypted access through Apache.
scheme	Identifies the protocol used by the client to access the server.

After configuring the `server.xml` file, restart your application server.

For more information about SSL support, see the *ClaimCenter System Administration Guide*.

chapter 25

Importing from Database Staging Tables

ClaimCenter supports high-volume data import 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 table-based import works and how to use it as part of a larger conversion process.

This topic includes:

Introduction to Database Staging Table Import

Database import provides significantly higher performance than importing individual records with web services APIs. Database staging table import avoids intermediate data formats such as XML. It also avoids the need to parse and transform that data into internal Java objects.

Also, database import uses bulk SQL `Insert` and `Select` statements to insert entire tables at a time. It does not operate on single rows at a time to import such as if using web services APIs.

The following sections explain important ClaimCenter database staging table import concepts. Review the topics and terminology in this section before proceeding to following sections in this topic.

Staging Tables

Staging tables are database tables that are near-duplicates of your ClaimCenter database's operational tables that contain all ClaimCenter data. To prepare data for import into ClaimCenter using its 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 `cc_` prefix, there is a corresponding staging table with a `ccst_` prefix.

For example, the claim table for the `Claim` entity is `cc_claim` and it has a corresponding staging table `ccst_claim`.

Similarly, the ContactCenter application has an address table `ab_address` that has a corresponding staging table `abst_address`.

There are important differences between staging table columns and operational columns, discussed further in "Populating the Staging Tables" on page 435. However the basic data columns are the same between the corresponding 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 the `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 if both of the following are `true`:

- At least one entity is defined as `loadable`
- The entity has at least one property defined as `loadable`.

  Note: Only loadable properties appear as columns in staging tables.

Zone Import

ClaimCenter uses zone mapping functionality used by both the *assign by location* feature and the *address autofill* feature. ClaimCenter uses staging tables to import this data.

The zone mapping import is a several step process:

1. **Get your zone data files or create them.** Create zone mapping data in comma separated value (CSV) files containing the postal code, the state, the city, and the county. Each line must look like this:

   ```
   postalcode,state,city,county
   ```

   For example,

   ```
   94114,CA,San Francisco,San Francisco
   ```

   You can also use the built-in zone data files that you can find in the directory:

   ```
   ClaimCenter/modules/pl/config/geodata
   ```

2. **Run the zone import tool to add data to staging tables.** The easiest way to run the zone import tool is to use 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 reference of all options on the `zone_import` tool, see "ClaimCenter Administrative Commands" on page 169.

3. **Run the main staging table import procedure.** This topic contains detailed information about the staging table import process. Carefully follow the instructions to import the staging tables into the operational tables.

Your Conversion Tool

A critical component of this process is some sort of custom *conversion tool* that you write. Your conversion tool converts legacy data into the ClaimCenter format and place the converted data into the staging tables to await table-based 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 the 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-in to the application.

You can check if any integrity checks failed using the user interface at **Server Tools → Info Pages → Load Errors**. For more information, see “Using the Server Tools” on page 151 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 425)
- LUWIDs identify which data to exclude from future integrity checks or load requests (see “Exclusion Table” on page 426)

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 Server Tools user interface. You can view these errors in ClaimCenter within **Server Tools → Info Pages → Load Errors**. For more information about using this screen, see “Using the Server Tools” on page 151 in the *System Administration Guide*. Most errors relate to a particular staging table row and so this user interface error 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 or write these tables. To add records to load exclusion tables, 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, and 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 within **Server Tools→ Info Pages → Load History.** For more information about using this screen, see the *ClaimCenter System Administration Guide*.

Load Commands and Loadable Entities

If a ClaimCenter table is *loadable* and has at least one loadable property, the system creates a staging table for that table during the upgrade process during server startup.

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 corresponds 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 load command ID 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 load command ID property stays the same as the time it 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 load command ID. 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 and all entities with that 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 load command ID. Alternatively, you can call the SOAP API to load database tables:

```
result = ITableImport.integrityCheckStagingTableContentsAndLoadSourceTables(...);
```

The result of that method is a `TableImportResult` entity, which contains a 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 load command ID. Similarly, if you used the command line tools to trigger database table import, those tools return the load command ID.

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 the "Using the Server Tools" on page 151 in the *System Administration Guide*.

IMPORTANT You can use the `LoadCommandID` property to check whether an entity was loaded through the staging tables, and which load job was associated. Be aware that entity data may have been changed at some later time after it was loaded.

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.

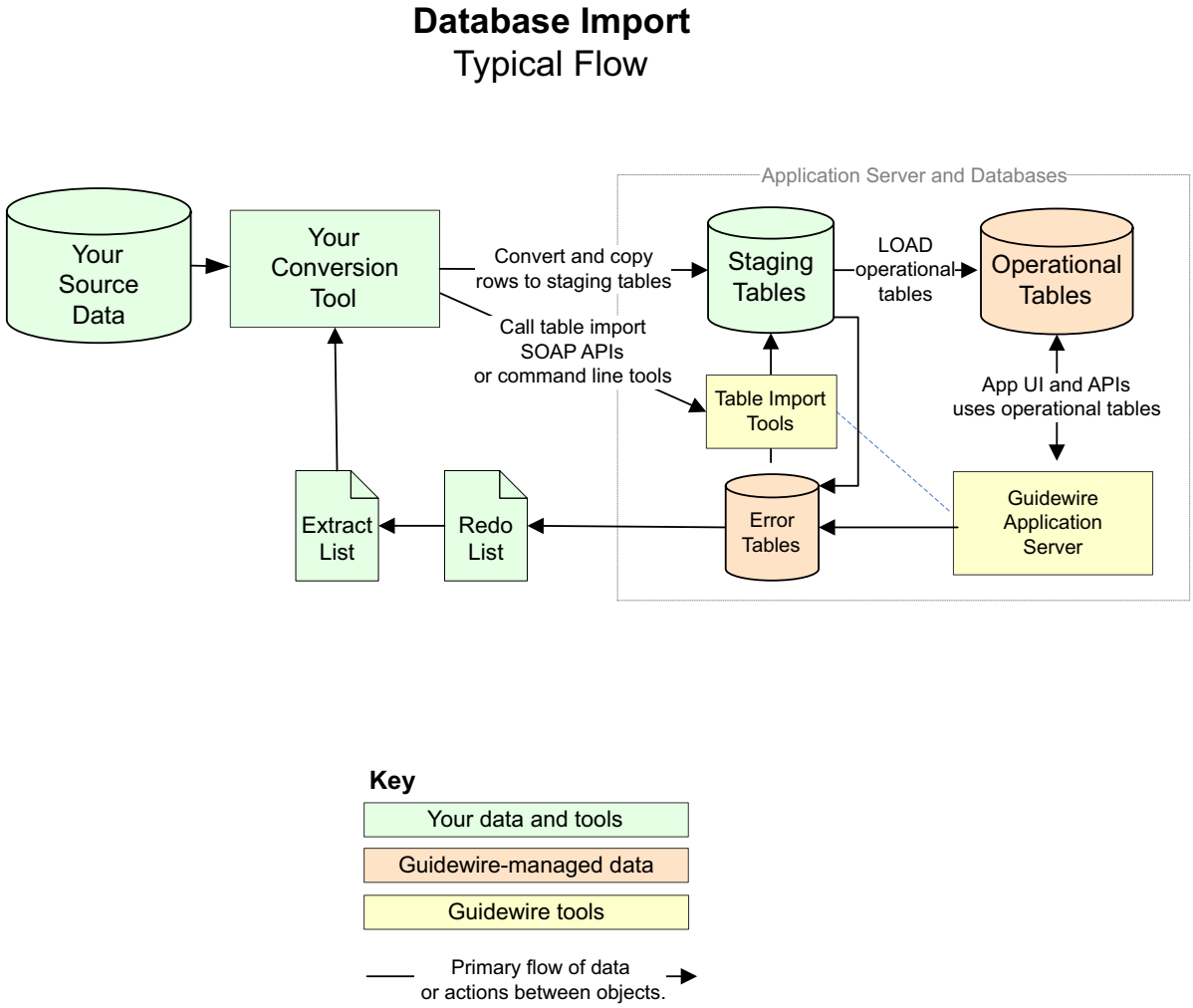
To conduct a typical database import

1. Convert your source data to ClaimCenter format in *staging tables*.
2. Run integrity checks on staging table data by using SOAP APIs or command line tools.
3. View load errors in the Server Tools user interface.
4. Fix errors and run integrity checks again, repeating some or all of the previous steps.

   Optionally, exclude data with errors by specifying the LUWIDs of bad records. This optional exclusion is not shown in the diagram that follows.
5. After you assure that all integrity checks succeed, load the data into operational tables by using SOAP APIs or command line table import tools.

   ClaimCenter always runs the integrity check during a load request to assure data integrity.
6. Check the load history in the Server Tools user interface, as discussed in “Using the Server Tools” on page 151 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 data model, run 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 have encrypted fields, you must encrypt the fields before loading the staging tables. For more information, see "Encrypted Properties in Staging Tables" on page 409.

2. **Put server in database maintenance run level.** Set each ClaimCenter server to the `dbmaintenance` run level using the `system_tools` command line tool or the web services API `ISystemToolsAPI.setRunLevel(...)` method. After the system goes into database maintenance mode, the system prevents new user connections, halts existing user sessions, and releases all database resources. This allows no access to the database while you are backing up the database.

3. **Back up your database's operational tables.** It is important to back up operational tables before import.

   WARNING As with any major database operation, backup **all** operational tables before importing.

4. **Put 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 `ISystemToolsAPI.setRunLevel(...)` method.

After the system goes into database maintenance mode, the system prevents new user connections, halts existing user sessions, and releases all database resources. This allows no access to the database while you are backing up the database.

5. **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 433.

6. **Update database statistics on tables.** Update with statistics using **one** of the following commands:

   ```
   maintenance_tools -startprocess dbstatistics
   maintenance_tools -getdbstatisticsstatements
   ```

   Run this tool this *every* time the amount of data in ClaimCenter tables changes by a factor of two. For more information about statistics, see "Database Import Performance and Statistics" on page 432.

   IMPORTANT It is important to understand that stale or missing statistics on the main operational tables can severely affect performance for the integrity check and database consistency checks.

7. **Run ClaimCenter database consistency checks.** Database consistency checks verify that the data in your operational tables does not contain any data integrity errors. Do this using the web services API `ISystemToolsAPI` method `checkDatabaseConsistency`, or using the command line:

   ```
   system_tools -password pwd -checkdbconsistency
   ```

   See the *ClaimCenter System Administration Guide* for more information about the command line tools.

   IMPORTANT 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.

8. **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.

9. **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 435.

10. **Update database statistics on tables.** There are multiple commands to update statistics. Use the following guidelines to choose which one to run:

    - If the main operational tables are empty, run the command `table_import -updatedatabasestatistics` updates database statistics on staging tables only.
    - Otherwise, run either of these commands unless you already ran them in step 6 (in which case you can skip this step):

      ```
      maintenance_tools -startprocess dbstatistics
      maintenance_tools -getdbstatisticsstatements
      ```

      Note that these commands are a superset of the `table_import -updatedatabasestatistics` command. Thus the `table_import` statistics command is unnecessary if you run the `maintenance_tools` version at this step.

    IMPORTANT It is important to understand that stale or missing statistics on the main operational tables can severely affect performance for the integrity check process (not the load process).

11. **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 433. 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) only 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.

12. **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.

13. **Check load errors using the Server Tools user interface.** ClaimCenter includes special pages for Server Tools, which includes a user interface for looking at the load errors in the internal error tables. Most importantly is the load errors screen found at **Server Tools→ Info Pages → Load History.** That shows all the load errors. For more information about using this screen, see "Using the Server Tools" on page 151 in the *System Administration Guide*.

14. **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 433. Some errors are not associated with specific LUWIDs, in which cases nothing can be move to the exclusion table.

    Note: For example, some types of financials errors are not associated with specific LUWIDs.

15. **Repeat integrity checks until they succeed.** Repeat integrity checks (see step 12) until there are no errors.

16. **Optionally, remove excluded records from staging tables.** Remove all the rows from all the staging tables for records whose LUWIDs are listed in the exclusion table. To do this, you can use the `deleteExcludedRowsFromStagingTables` tool described in "Table Import Tools" on page 433.

17. 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 "Encrypted Properties in Staging Tables" on page 409.

18. **Load staging tables into operational tables.** Eventually, integrity checks succeed with all claims except for records in the exclusion table (see step 14 for exclusion info). At this point, integration teams can *load* the data from the staging tables into the *operational tables* used by the ClaimCenter application. Do this using one of the various web service (SOAP) APIs or command line tools for loading. See "Table Import Tools" on page 433. For example, use the table import web service method `integrityCheckStagingTableContentsAndLoadSourceTables`. As ClaimCenter inserts rows into the operational tables, it also insert 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) only 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 on the tool for Oracle databases.

The server automatically removes all data from staging tables at completion. If there were errors, data remains in the staging tables.

19. **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.

20. **Review the load history.** Review the load history in the Server Tools user interface, found at **Server Tools** → **Info Pages** → **Load Errors**. Ensure that the amount of data loaded is correct. For more information, see "Using the Server Tools" on page 151 in the *System Administration Guide*

21. **Update the database statistics again.** Particularly after a large conversion, update the database statistics. You can use one of the following commands:

```
maintenance_tools -startprocess dbstatistics
maintenance_tools -getdbstatisticsstatements
```

22. **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 7 for the data consistency check options.

23. **Convert more records if necessary.** If you have more data to convert, begin again at step 7.

24. **Run batch processes.** Run the following batch processes in the following order:

    a. `financialscalculations`

    b. `claimcontactscalculations`

    c. `checkdbconsistency`

    However, this is the minimal recommendation. You may want to run other batch processes, depending on what other features you need, such as:

    - `aggregatelimitcalculations`
    - `aggregatelimitloadercalculations`
    - `dashboardstatistics`
    - `statistics`
    - `datadistribution`

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 151 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.

IMPORTANT In contrast, absolutely never modify *operational tables* with direct SQL commands.

Alternatively, you may want to correct errors in the source system. In that case, remove claims 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 as way as to isolate your code performance from ClaimCenter database import API performance:

- Oracle database users must take statistics snapshots (statspack snapshots at level 10 or `AWR` snapshots) at the beginning and end of `integritycheck` commands and `integritycheckandload` commands.
- You must 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.
- You must 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 `ITableImportAPI` interface method `updateStatisticsOnStagingTables` or use the command line tool command:

```
table_import -updatedatabasestatistics
```

Table Import Tools

ClaimCenter provides a set of table-based import functions to help you in the process described in “Overview of a Typical Database Import” on page 427.

ClaimCenter exposes the table import tools in the following ways for you:

- **Web service (SOAP) APIs.** You can use SOAP APIs defined in the `ITableImportAPI` interface. For more information about logging into and using web services, see “Web Services (SOAP)”, on page 25. All tools are provided as typical synchronous API methods that do not return until the command is complete. Some tools also have an asynchronous method that returns immediately and then performs the command as a batch process. The asynchronous versions have methods with names that end in “`AsBatchProcess`”. For example `deleteExcludedRowsFromStagingTablesAsBatchProcess`.
- **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 command line tools, see the “ClaimCenter Administrative Commands” on page 169 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 adminpwd
```

IMPORTANT All table import commands require all ClaimCenter servers in a cluster to be at the `maintenance` run level or table import commands fails.

Alternatively, set the run level using web service APIs in the `ISystemTools` interface’s `setRunLevel` method.

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

Runs the integrity check process and, if no errors, proceeds with loading the operational tables. 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 169 in the *System Administration Guide*. For more information about logging into and using web service APIs, see “Web Services

(SOAP)", on page 25

Setting Run Level

Set the run level to `maintenance` **before** performing a table import.

- API method: `ISystemToolsAPI.setRunLevel(SystemRunlevel.GW_MAINTENANCE);`
- Command line option: `system_tools -password pwd -maintenance`

Checking Operational Table Consistency

Check the operational database consistency **before** running a table import.

- API method: `ISystemToolsAPI.checkDatabaseConsistency(`*`returnAllResults`*`);`
- Command line option: `system_tools -password pwd -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 ContactCenter `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 426.

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 Use the **Conversion View** in the Data Dictionary to see the application data model from a conversion engineer perspective. It only shows properties backed by real columns that are also importable.

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 incrementing counter.

Set the retired property (if any) to the value 0, which indicates an active rather than retired row in the operational tables. To retire a row, set the `retired` property to the object's public ID. After you retire an object, you cannot revert it to active.

WARNING Guidewire recommends not loading 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 57.

Note: 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 ContactCenter, 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 as 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 172 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 `open` or `closed` for all claims being imported. If `closed`, then also set the `ClosedDate` 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.
`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** checks.
- Do not insert **transferred** checks.
- Do not insert **multipayee** 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 171 in the *Application Guide* for more information about multicurrency. See "Exchange Rate Integration" on page 236 in the *Application Guide* 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 the staging table data into the operational (real) 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.

It is important that *integrity checks* check different things from user-interface-enforced rules. For example, a property that is *nullable* in the database may be a property that users are required to 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 to you provide a non-`null` value before saving because of data model validation.

Similarly, integrity checks are different from validation rule sets discussed in the *ClaimCenter Rules Guide* or in the validation plugin discussed in "Other Plugin Interfaces", on page 333.

IMPORTANT Table import data integrity checks have entirely **different requirements** than user interface data restrictions (enforced by PCF code) or application-level validation (enforced by validation rule sets).

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 Info Pages user interface within the ClaimCenter Server Tools tab. You can view all integrity check SQL queries. For more details about this ClaimCenter feature, see “Using the Server Tools” on page 151 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 claims 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 427 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
```

IMPORTANT Always run database consistency checks on the operational database tables both before and after table imports

- 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 You cannot 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, your staging table data must have appropriately encrypted properties before importing them into ClaimCenter.

For more information about these encryption in staging tables, see “Encrypted Properties in Staging Tables” on page 409.

chapter 26

Claim and Policy Integration

This topic describes web services and plugin interfaces for communicating with policy systems, such as Guidewire PolicyCenter.

This topic includes:

See also

- "Web Services (SOAP)", on page 25
- "Plugin Overview", on page 101

Policy System Notifications

ClaimCenter includes a general architecture for notifications from ClaimCenter to policy 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 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 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 rules sets that ClaimCenter uses for large loss notifications.

Purpose	Rule Set	Description
Preupdate rules to detect large losses	Large Loss Notification	This **Preupdate** → **TransactionSetPreupdate** → **Large Loss Notification** rule fires whenever transaction sets are created or changed. The rule checks if the claim exceeds the large loss threshold for the claim's policy type. It also checks whether a message is in the queue or if PolicyCenter has been notified. If the condition passes, it adds a `ClaimExceedsLargeLoss` event to the claim. Define the thresholds within the administration user interface of ClaimCenter.
Event rules to support policy system notifications in general	Policy System Notification	This **EventMessage** → **EventFired** → **Policy System Notification** rule is a general rule for all policy system notification events: **1.** The rules determine the event name. For a large loss, the event name is `ClaimExceedsLargeLoss`. If you register additional notification handlers, the event name is different. **2.** The rules find the policy system notification handler that supports the current event name. For large loss, the handler class is `LargeLossPolicySystemNotification`. **3.** The rules call the handler's `createMessage` 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 does not actually send the message yet, which happens asynchronously in another thread. The handler's `createMessage` method delegates the actual sending behavior to the `IPolicySystemNotificationPlugin` interface. The `message.EventName` 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. At message send time, the messaging transport calls the notification handler's `send` method and does not use this rule set. If a claim resync happens (a `ClaimResync` event), then ClaimCenter drops all queued (pending/errant) messages for the destination. However, first ClaimCenter calls the **EventFired** rule set to preserve and queue messages. These event rules trigger code that checks the notification handler's `MessageResyncBehavior` property for how to handle it. If the value is `COPY_LAST`, then only one pending message corresponding to that notification will be copied (the last one, by send order).

ClaimCenter can notify a policy system if a claim reaches a critical threshold, defined by policy type. The policy 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 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 337 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 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 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 337 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” on page 340 in the *Application 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 the 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 GWAuthenticationHandler( "su", "gw" ))
```

For the full procedure, see “Enabling Large Loss Notification” on page 340 in the *Application Guide*.

Using a Policy System Other than PolicyCenter

If you use a policy system other than PolicyCenter, you must write your own implementation of the `IPolicySystemNotitificationPlugin` plugin interface. Instead of contacting PolicyCenter, you can send the notification to your policy 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 `IPolicySystemNotitificationPlugin` 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 system correctly received the message. This is called message acknowledgement. For more information about messaging acknowledgements, refer to “Messaging and Events” on page 139.

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 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 it 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" on page 340 in the *Application Guide*.

10. Test your notification system with a development version of ClaimCenter and your policy system.

Policy Search Plugin

The policy search plugin allows 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 do this, in Studio, go to the Plugins editor and change the implementation class for the `IPolicySearchAdapter` plugin to the Gosu class `gw.plugin.pcintegration.v2.PolicySearchPCPlugin`. For detailed instructions on this, see "Enabling Integration between ClaimCenter and PolicyCenter" on page 64.

If you use another policy system, write your own implementation of this plugin.

ClaimCenter includes a demo version of this plugin, which generates example policies.

To understand the flow of ClaimCenter policy search, it is important to distinguish the two concepts of search versus retrieve in the context of ClaimCenter:

- **Search means getting policy summaries, and possibly many of them.** Search refers to a call out 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 which policy 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 a 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. It retrieves 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 data in properties in a `PolicySearchCriteria` object. This is a *non-persistent object*, which means it is described in the ClaimCenter Data Dictionary but with 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 use the `PolicySummary` entity to retrieve the policy. Because of this, 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, then 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 only return 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 452.

   ---

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 one of the summaries in the user interface to retrieve.
7. To retrieve a single unique policy from the external system, ClaimCenter calls `IPolicySearchAdapter.retrievePolicyFromPolicySummary(PolicySummary)`. This method must return a `PolicyRetrievalResultSet`. This object contains a `Policy` entity. For more details about its properties, see "Policy Retrieval Additional Information" on page 452.
8. If you refresh (reload) the `Policy` in the application 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 these 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 searching for a policy in the typical case:

- 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 `searchPolicies` method.

However, the path is slightly different for commercial auto/property policies, where there is an extra page for you to 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, 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 that means that 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 then 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`), then do more searches, and then change the policy. In that case, ClaimCenter calls `searchPolicies` once per search and `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 display in the `MaxPolicySearchResults` parameter within `config.xml`. This only affects the number of rows that ClaimCenter *displays*. 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 `PolicySearchResultSet.ResultsCapped` property to `true`. If you do this, the ClaimCenter user interface displays a message that tells the user "too many results found to display".

To add search criteria for policy searches

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 input new user data that relate to those properties.

   Alternatively, programmatically set the new search criteria on factors other than new user data. For example, base the criteria on the context of the ClaimCenter user interface.
3. Alter the PCF files appropriately to set these extension properties from data entered in form fields.

Sometimes policies have a very large number of insured risks, for example many 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 passing a vehicle's Vehicle Identification Number (VIN). By adding this search criteria, 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. Although it may be appropriate to remove those search fields from the user interface in that case 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 or 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 may 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 to only 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 displays a message that says 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 simply set it to the number of real estate properties in `Policy.Properties`. Setting the total correctly allows the user interface to notice this difference between the two numbers. The application displays a message that says not all real estate properties on the policy are available in ClaimCenter.

IMPORTANT In all cases, you must set the properties `Policy.TotalProperties` and `Policy.TotalVehicles` correctly as the total values for the policy, independent of how many you return in the result.

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 then 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 displays these in a list view so you can choose a policy. When you hit **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 using a method called `isSamePolicy` on a policy summary. The wizard compares the currently picked summary with policy that is on the claim. If `isSamePolicy` returns true it assumes the claim's current policy is still the one we want. If `isSamePolicy` returns false it 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, and that 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 whether 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 directly, since PolicyCenter calls this automatically if you configure PolicyCenter to connect to ClaimCenter for claim search.

If you use a policy system other than PolicyCenter, your policy 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 system other than PolicyCenter, consider writing your own custom web services that directly address integration points between ClaimCenter and your specific policy system.

> **Note:** The `IClaimAPI` contains some similar methods. For example, the `IClaimAPI` interface provides the `findPublicIdByClaimNumber` method and the `PCPolicySearchIntegrationAPI` interface provides the `getClaimByClaimNumber` method. However, neither web service interface supports return a complex claim graph. Instead, add custom web services to search for claims or export claim summaries in the data model format your policy system needs. Only export the claim fields and subobjects that are necessary.

Search for Claims

If PolicyCenter or another policy 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 system. It includes the properties described in the following table.

Property	Type	Description
`BeginDate`	`Calendar`	The begin date.
`EndDate`	`Calendar`	The end date.
`Lob`	`String`	Line of business, specified by its name
`PolicyNumbers`	`String[]`	An array of policy numbers, as an array of `String` 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
`ClaimNumber`	`String`	The claim number
`LossDate`	`Calendar`	The loss date
`PolicyNumber`	`String`	The policy number of the best matched policy
`PolicyTypeName`	`String`	The policy type name
`Status`	`String`	The claim status, as a string
`TotalIncurred`	`BigDecimal`	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 all you want to do is get the number of results that match, PolicyCenter or another policy 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 system want 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.