



ClaimCenter Rules Guide

RELEASE 8.0.2

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

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

All other trademarks are the property of their respective owners.

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

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

Product Name: Guidewire ClaimCenter

Product Release: 8.0.2

Document Name: *ClaimCenter Rules Guide*

Document Revision: 20-May-2014

Contents

About ClaimCenter Documentation	9
Conventions in This Document	10
Support	10

Part I Gosu Business Rules

1 Rules: A Background	13
Introduction to Business Rules	13
Rule Hierarchy	13
Rule Execution	14
Rule Management	14
Sample Rules	15
Important Terminology	15
Overview of ClaimCenter Rule Sets	16
2 Rules Overview	19
Rule Design Template	19
Rule Structure	20
Rule Syntax	20
Rule Members	21
Defining the Rule Conditions	21
Defining the Rule Actions	22
Exiting a Rule	22
Gosu Annotations and ClaimCenter Business Rules	23
Invoking a Gosu Rule from Gosu Code	23
3 Using the Rules Editor	25
Working with Rules	25
Renaming or Deleting a Rule	28
Changing the Root Entity of a Rule	28
Why Change a Root Entity?	28
Making a Rule Active or Inactive	30
4 Writing Rules: Testing and Debugging	31
Generating Rule Debugging Information	31
Printing Debugging Information	31
Logging Debugging Information	31
Using Custom Logging Methods	32
5 Writing Rules: Examples	33
Accessing Fields on Subtypes	33
Looking for One or More Items Meeting Conditions	34
Preventing Repeated Actions	34
Taking Actions on More Than One Subitem	35
Checking Permissions	35
Determining the Original Attributes of a Changed Entity	35
6 Rule Set Categories	37
Rule Set Summaries	38

Approval Routing Rule Set Category	38
Approval Routing Rules	39
BulkInvoice Approval Assignment Rules.....	40
Archive Rule Set Category	40
Detecting Archive Events	41
Skipped Claims Versus Excluded Claims.....	42
Default Group Claim Archiving Rules	43
Archive Claim Purge Rules.....	44
Claim Eligible for Archive Rules	44
Assignment Rule Set Category	44
BulkInvoice Approval Rule Set Category	45
BulkInvoice Approval Rules	45
Closed Rule Set Category	46
Activity Closed Rules	46
Claim Closed Rules.....	47
Exposure Closed Rules	47
Matter Closed Rules	48
Event Message Rule Set Category	48
Event Fired Rule Example	50
Exception Rule Set Category	52
Activity Escalation Rules	52
Claim Exception Rules	53
Group Exception Rules.....	55
User Exception Rules	55
Initial Reserve Rule Set Category	55
Initial Reserve Rules	56
Loaded Rule Set Category	56
Claim Loaded Rules	56
Exposure Loaded Rules.....	57
Postsetup Rule Set Category	57
Activity Postsetup Rules.....	57
Claim Postsetup Rules	58
Exposure Postsetup Rules.....	58
Matter Postsetup Rules	58
Transaction Postsetup Rules	58
Presetup Rule Set Category	59
Claim and Exposure Setup	59
Transaction Set Presetup Rules.....	61
Workers' Compensation Presetup Example	63
Preupdate Rule Set Category	63
Triggering Preupdate Rules.....	64
Preupdate Rules and Custom Entities	65
Activity Preupdate Rules	65
Claim Preupdate Rules	65
Contact Preupdate Rules.....	67
Exposure Preupdate Rules.....	67
Group Preupdate Rules	68
Matter Preupdate Rules.....	68
Policy Preupdate Rules	68
RIAgreementPreupdate Rules.....	69
RITransactionSetPreupdate Rules.....	69
Transaction Set Preupdate Rules.....	69
User Preupdate Rules	70

Reopened Rule Set Category	70
Claim Reopened Rules	71
Exposure Reopened Rules.....	71
Matter Reopened Rules.....	71
Segmentation Rule Set Category	72
Strategy Property.....	72
Claim Segmentation Rules	73
Exposure Segmentation Rules	73
Transaction Approval Rule Set Category.....	73
Transaction Approval Rules	74
Validation Rule Set Category	74
Validation Levels	75
Validation Warnings.....	75
Validation Errors.....	76
Adding New Validation Levels.....	76
Triggering Validation Rules	76
Validation and Preupdate Rules	77
Determining the Validity of an Object	77
Correcting Validation Errors.....	78
Activity Validation Rules	79
Claim Closed Validation Rules.....	80
Claim Reopened Validation Rules	80
Claim Validation Rules.....	81
Contact Validation Rules	81
Exposure Closed Validation Rules	81
Exposure Reopened Validation Rules.....	81
Exposure Validation Rules	82
Group Validation Rules	82
Matter Closed Validation Rules	82
Matter Reopened Validation Rules.....	82
Matter Validation Rules	82
Policy Validation Rules	82
Service Request Validation Rule Set	83
Transaction Set Validation Rule Set.....	84
User Validation Rules.....	84
Workplan Rule Set Category	84
Claim Workplan Rules	85
Exposure Workplan Rules.....	85
Matter Workplan Rules.....	86
7 ClaimCenter Rule Execution	87
Generating a Rule Repository Report	87
Generating a Rule Execution Report	88
Interpreting a Rule Execution Report.....	88

Part II

Advanced Topics

8 Assignment in ClaimCenter	93
Understanding Assignment.....	93
ClaimCenter Assignment	94
Global Assignment Rules	96
Default Group Assignment Rules.....	97
Assignment Execution Session	97

Primary and Secondary Assignment	98
Primary (User-based) Assignment	98
Secondary (Role-based) Assignment	98
Assignment within the Assignment Rules	99
Assignment Success or Failure	100
Assignment Methods that Use the Assignment Engine	100
Assignment Methods that Do Not Use the Assignment Engine	100
Determining Success or Failure	100
Logging Assignment Activity	101
Assignment Cascading	101
Assignment Events	101
Assignment Method Reference	102
Assignment by Assignment Engine	102
Group Assignment (within the Assignment Rules)	102
Queue Assignment	104
Immediate Assignment	105
Claim-Based Assignment	106
Condition-Based Assignment	107
Proximity-Based Assignment	110
Round-robin Assignment	114
Dynamic Assignment	115
Manual Assignment	117
Using Assignment Methods in Assignment Pop-ups	119
Assignment by Workload Count	120
9 Validation in ClaimCenter	121
What is Validation?	121
Overview of ClaimCenter Validation	122
Entity Validation Order	122
The Validation Graph in Guidewire ClaimCenter	123
Traversing the Validation Graph	123
Top-level Entities that Trigger Validation	124
Overriding Validation Triggers	126
Validation Performance Issues	126
Administration Objects	126
Query Path Length	126
Links Between Top-level Objects	127
Graph Direction Consistency	127
Illegal Links and Arrays	127
Debugging the Validation Graph	127
10 Detecting Claim Fraud	129
Claim Fraud	129
The Special Investigation Details Screen	130
Special Investigation Trigger Rules	130
SI Rule Evaluation Sequence	130
Using the Special Investigation Rules	131
Set the Initial Life Cycle Stage	131
Advance the Life Cycle Stage	132
Evaluate SIU Triggers	132
Escalate Claim to Supervisor	133
Assign Activity to SIU Group	133
Add Assigned User in SIU Role	133

11 Sending Emails	135
Guidewire ClaimCenter and Email	135
The Email Object Model	136
Email Utility Methods	136
Email Transmission	137
Understanding Email Templates	137
Creating an Email Template	139
Localizing an Email Template	139
The IEmailTemplateSource Plugin	140
Class LocalEmailTemplateSource	140
Configuring ClaimCenter to Send Emails	140
Class EmailMessageTransport	141
Class JavaxEmailMessageTransport	142
Working with Email Attachments	143
Sending Emails from Gosu	143
Saving an Email Message as a Document	143
12 Document Creation.....	145
Synchronous and Asynchronous Document Production	145
Integrating Document Functionality with ClaimCenter	146
The IDocumentTemplateDescriptor Interface	147
The IDocumentTemplateDescriptor API	148
Template Metadata	148
Document Metadata	150
Context Objects	150
Form Fields	151
Document Locale	151
The DocumentProduction Class	151
How to Determine the Supported Document Creation Type	152
Asynchronous Document Creation Methods	152
Synchronous Document Creation Methods	153
Document Templates	154
Document Creation Examples	154
Method createAndStoreDocumentSynchronously Example 1	156
Method createAndStoreDocumentSynchronously Example 2	156
Troubleshooting	157
IDocumentContentSource.addDocument Called with Null InputStream	157
IDocumentMetadataSource.saveDocument Called Twice	158
UnsupportedOperationException Exception	158
Document Template Descriptor Upgrade Errors	158
“Automation server cannot create object” Error	158
“IDocumentProduction implementation must return document...” Error	159
Large Size Microsoft Word Documents	159
13 Transaction Set Validation.....	161
Transaction Set Validation Rules	161
TXV01000 – Total payments not greater than Exposure limit	162
TXV02000 – Check Exposure limit when increasing reserves	163
TXV03000 – Total payments not greater than Incident limit	164
TXV04000 – Check Incident limit when increasing reserves	164
TXV05000 – PIP	165
TXV17000 – Reserve Lines Compatible with Service Requests	166
TXV18000 – Currencies Compatible with Service Requests	166
sumForPIPAgg Enhancement Methods	166

14 Working with Queues	167
What Are Activity Queues?	167
Using a Queue to Handle FNOL Claims: Example	167
Using a Queue to Void/Stop Checks: Example	170
Create the Stop/Void Queue	170
Assign the Activity to the Appropriate Group	170
Add the Activity to the Queue	171

About ClaimCenter Documentation

The following table lists the documents in ClaimCenter documentation.

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

Conventions in This Document

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

Support

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

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

part I

Gosu Business Rules

Rules: A Background

This topic provides an overview of rules and discusses some basic terminology associated with rules and rule sets. It also gives a high-level view of the ClaimCenter rule set categories.

- “Introduction to Business Rules” on page 13
- “Important Terminology” on page 15
- “Overview of ClaimCenter Rule Sets” on page 16

Introduction to Business Rules

In general, Guidewire strongly recommends that you develop and document the business logic of rules before attempting to turn that logic into rules within ClaimCenter. In a large implementation, there can be a large number of rules, so it is extremely beneficial to organize the basic structure of the rules in advance. Use this guide to understand how ClaimCenter rules work. It can also help you make decisions about changing your rules as your use of ClaimCenter evolves over time.

Rule Hierarchy

A rule set can be thought of as a logical grouping of rules that are specific to a business function within ClaimCenter. You typically organize these rules sets into a hierarchy that fits your business model. Guidewire strongly recommends that you implement a rule-naming scheme as you create rules and organize these rules into a hierarchy. This can be similar to that described in “Generating Rule Debugging Information” on page 31.

Prior to implementing rules, it is important to first understand the rule hierarchy that groups the rules. The rule hierarchy is the context in which ClaimCenter groups all rules. You can implement a rule hierarchy in several formats, depending on the needs of your organization. However, it is important to outline this hierarchy up-front before creating the individualized rules to reduce potential duplicates or unnecessary rules. You can create multiple hierarchies within ClaimCenter. However, make each specific to the rule set to which it belongs.

Rule Execution

The hierarchy of rules in ClaimCenter mirrors a decision tree that you might diagram on paper. ClaimCenter considers the rules in a very specific order, starting with the first direct child of the root. (The first direct child is the first rule immediately below the line of the rule set.) ClaimCenter moves through the hierarchy according to the following algorithm. Recursively navigating the rule tree, it processes the parent and then its children, before continuing to the next peer of the parent.

- Start with the first rule
- Evaluate the rule's conditions. If true...
 - Perform the rule's actions
 - Start evaluating the rule's children, starting with the first one in the list.
- You are done with the rule if a) its conditions are false or b) its conditions are true and you processed its actions and all its child rules.
- Move to the next peer rule in the list. If there are no more peers, then the rules at the current level in the tree are complete. After running all the rules at the top level, rule execution is complete for that rule set.

To illustrate how to use a rule hierarchy, take segmentation logic as an example. You would probably expect to describe a series of questions to figure out in which segment to put a claim. For convenience, you would want to describe these questions hierarchically. For example, if you answer question #1 as No, then skip directly to question #20 because questions #2-19 only pertain if you answered #1 as Yes.

You might go through logic similar to the following:

- **If this is an auto claim,** then go through a list of more detailed segmentation rules for auto claims:
 - If this has glass damage only, then segment as “Auto Glass” [done]
 - Otherwise...If this was a collision, then consider collision segmentation rules
 - If there are injuries, then segment as “injury accident” [done]
 - Else if there are more than two cars, segment as “multi-car accident” [done]
 - Else if there are two cars, segment as “2 car accident” [done]
 - Else segment as “single car accident” [done]
 - Otherwise, consider non-collision segmentation rules
 - If there are third-party exposures, segment as “non-accident liability” [done]
 - Else, if the cause of loss is theft, then segment as “theft” [done]
 - Else, segment as “non-accident vehicle damage” [done]
- **If this is a homeowner's claim,** then go through a list of more detailed segmentation rules for homeowner's claims...
- **If all else fails [Default],** segment this as “unknown” [done]

You can see from this example that your decision tree follows a hierarchy that maps to the way ClaimCenter keeps rules in the hierarchy. If the first rule checks whether the claim is an auto claim, then ClaimCenter does not need to check any of the follow-up (child) rules. (That is, unless the claim is actually an auto claim.)

See also

- “Generating a Rule Execution Report” on page 88

Rule Management

Guidewire strongly recommends that you tightly control access to changing rules within your organization. Editing rules can be complicated. Because ClaimCenter uses rules to make automated decisions regarding many important business objects, you need to be careful to verify rule changes before moving them into production use.

Two kinds of people need to be involved in managing your business rules:

- **Business Analysts** – First, you need one or more business analysts who own decision-making for making the necessary rules. Business analysts must understand the normal business process flow and must understand the needs of your business organization and how to support these needs through rules in ClaimCenter.
- **Technical Rule Writers** – Second, you need one or more rule writers. Generally, these are more technical people. Possibly, this can be someone on the business side with a good technical aptitude. Or possibly, this can be someone within IT with a good understanding of the business entities that are important to your business. Rule writers are responsible for encoding rules and editing the existing set of rules to implement the logic described by business analysts. The rule writers work with the business analysts to create feasible business rules. These are rules that you can actually implement with the information available to ClaimCenter.

Sample Rules

Guidewire provides a set of sample rules as examples and for use in testing. These are sample rules only, and Guidewire provides these rules merely as a starting point for designing your own rules and rule sets. You access sample rules (and other Studio resources) through the Studio interface.

See also

- “Overview of ClaimCenter Rule Sets” on page 16
- “Generating a Rule Repository Report” on page 87
- “Generating a Rule Execution Report” on page 88

Important Terminology

This guide and other ClaimCenter documents use the following terminology throughout the discussion of business rules:

Term	Definition
assignment engine	The assignment engine handles assignment of claims, exposures, and activities to users by repeatedly asking ClaimCenter to evaluate assignment rules for the item. Guidewire structures the assignment rule sets so that the assignment engine first asks to which claim handling office the assignment goes. It then asks to which group, and finally to which person (with each of these decisions governed by a different rule set). See “Assignment in ClaimCenter” on page 93”.
entity	An entity is a type of business data configured in the data model configuration files, for example <code>Claim</code> . You can use the Data Dictionary to review the entity types and their properties. For an entity type, this documentation refers to an instance as an <i>entity instance</i> or, for brevity, <i>object</i> .
Guidewire Studio	Guidewire Studio is the Guidewire administration tool for managing ClaimCenter resources, such as PCF pages, business rules, and Gosu classes.
library	A library is a collection of functions (methods) that you can call from within your Gosu programs. Guidewire provides a number of standard library functions (in the <code>gw.api.*</code> packages).
object	An object refers to any of the following: <ul style="list-style-type: none">• an instance of an <i>entity type</i>, such as <code>Claim</code> or <code>Activity</code>. For an entity type, an object is also called an <i>entity instance</i>.• an instance of an <i>Gosu class</i>• an instance of an <i>Java class</i>, such as <code>java.util.ArrayList</code>.

Term	Definition
rule	<p>A rule is a single decision in the following form:</p> <pre>If {some conditions} Then {take some action}</pre> <p>For example, a validation rule might mean: <i>If the claim has no loss date or a loss date in the future, then mark the loss date field as invalid.</i> This individual rule is just one of many validation tests that you can conduct on a new claim, for example.</p>
rule set	<p>A rule set combines many individual rules into a useful set to consider as a group.</p> <p>For example, the Claim Segmentation Rules rules contain all of the individual decisions (rules) that determine how to categorize a claim. ClaimCenter uses rules by taking some item (for example, a claim) and running a rule set (for example, the Claim Segmentation rules) against it to make one or more decisions. ClaimCenter—at various points in the claim lifecycle—runs different rule sets (to make different decisions) against the same claim.</p>
workflow	<p>A workflow is a Guidewire mechanism to run custom business processes asynchronously, optionally with multiple states that transition over time. Studio stores individual workflows as XML files. However, you manage workflow directly through Guidewire Studio user interface.</p>

Overview of ClaimCenter Rule Sets

A rule set can be thought of as a logical grouping of rules that are specific to a business function within ClaimCenter. Guidewire provides the following ClaimCenter sample rule sets:

Rule set category	Contains rules to
Approval Routing Rule Set Category	Route approval activities for a transaction that requires approval to the appropriate user. See “Approval Routing Rule Set Category” on page 38.
Archive Rule Set Category	Manage the claim archive process. See “Archive Rule Set Category” on page 40.
Assignment Rule Set Category	Determine the responsible party for an activity, claim, exposure, or matter. See “Assignment Rule Set Category” on page 44.
BulkInvoice Approval Rule Set Category	Define the approval requirements for Bulk Invoices. See “BulkInvoice Approval Rule Set Category” on page 45.
Closed Rule Set Category	Perform an action if you close an activity, claim, exposure, or matter—for example, sending a notification. See “Closed Rule Set Category” on page 46.
Event Message Rule Set Category	Handle communication with integrated external applications. See the “Messaging and Events” on page 299 in the <i>Integration Guide</i> for information about events and event messaging. See “Event Message Rule Set Category” on page 48.
Exception Rule Set Category	Specify an action to take under certain circumstances. For example, if no one touches a claim for a certain number of days, then create a review activity. See “Exception Rule Set Category” on page 52.
Initial Reserve Rule Set Category	Determine the initial financial reserves to create for a new exposure. See “Initial Reserve Rule Set Category” on page 55.
Loaded Rule Set Category	Trigger automatic actions if the FNOL import interface loads a new claim. See “Loaded Rule Set Category” on page 56.
Postsetup Rule Set Category	Trigger automatic actions any time that a new claim is added to ClaimCenter. These rules run after the Assignment and Workplan rules complete their execution. See “Postsetup Rule Set Category” on page 57.
Presetup Rule Set Category	Trigger automatic actions any time that a new claim, exposure, matter or transaction is added to ClaimCenter. These rules run prior to the execution of the Segmentation Rule Set, Assignment Rule Set, and Workplan Rule Set rules. See “Presetup Rule Set Category” on page 59.
Preupdate Rule Set Category	Trigger automatic actions any time that a claim or other high-level entity changes. See “Preupdate Rule Set Category” on page 63.

Rule set category	Contains rules to
Reopened Rule Set Category	Trigger automated actions on reopening a claim or exposure. See “Reopened Rule Set Category” on page 70.
Segmentation Rule Set Category	Categorize a new claim or an exposure based on complexity, severity of damage, or other attributes. See “Segmentation Rule Set Category” on page 72.
Transaction Approval Rules Set Category	Determine if a user has the required authorization to submit a financial transaction. See “Transaction Approval Rule Set Category” on page 73.
Validation Rule Set Category	Check for missing information or invalid data on Address Book, Claim, Exposure, Matter, Policy and Transaction objects. See “Validation Rule Set Category” on page 74.
Workplan Rule Set Category	Add initial activities to a claim, exposure or matter as a checklist of work that various people need to do on the claim. See “Workplan Rule Set Category” on page 84.

Rules Overview

This topic describes the basic structure of a Gosu rule and how to create a rule in Studio. It also describes how to exit a rule and how to call a Gosu rule from Gosu code.

This topic includes:

- “Rule Design Template” on page 19
- “Rule Structure” on page 20
- “Exiting a Rule” on page 22
- “Gosu Annotations and ClaimCenter Business Rules” on page 23
- “Invoking a Gosu Rule from Gosu Code” on page 23

Rule Design Template

Prior to entering rules into Guidewire Studio, it is important to identify and document the requirements for each rule. The following template follows the Studio format and you can use it as the handshake between the business and technical teams. It can also be the record for all error messages. This assists you in checking messages to insure consistency in wording and format, so that consistent messages are shown within ClaimCenter.

Guidewire does not intend that this suggested template house pseudo-code or rule syntax. Instead, it stores the actual requirements that you can later use to translate into the rule programming language.

Hierarchy	Rule condition	IF action	Error message	ClaimCenter
All	If Index Only Claim = Y	Require the following fields: <ul style="list-style-type: none">• Insured Name or Policy Number• Loss Type/Claim Line• Claimant Name• Policy Date• HCO• Date of Loss• Date Reported• Claim Representative	Complete the required fields	Basics

Rule Structure

The basic structure of every rule has the following syntax:

```
IF {some conditions}
THEN {do some actions}
```

The following table summarizes each of the parts of the rule.

Pane	Requirements
Rule Conditions	<p>A Boolean expression (that is, a series of questions connected by AND and OR logic that evaluates to TRUE or FALSE). For example:</p> <p>(This is an auto claim) AND (the policy includes collision coverage)</p> <p>It is also possible to insert a statement list, instead of a simple expression. However, the statement list must contain a return statement. For example:</p> <pre>uses java.util.HashSet uses gw.lang.reflect.IType var o = new HashSet<IType>() {A, B, C, ...} return o.contains(typeof(...))</pre>
Rule Actions	<p>A list of actions to take. For example:</p> <p>{Mark the claim as flagged}</p> <p>{Add an activity for the adjuster to follow up}</p>

The best way to add rules to the rule set structure is to right-click in the Studio rule pane. After you do this, Studio opens a window containing a tree of options from which to select. As you use the right-click menu, ClaimCenter gives you access only to conditions and actions that are appropriate for the type of your current rule.

Rule Syntax

Guidewire Studio uses a programming language called Gosu, which is a high-level language tailored for expressing business rule logic. The syntax supports rule creation with business terms, while also using common programming methods for ease of use. Gosu syntax can be thought of in terms of both statements and expressions. Before you begin to write rules, Guidewire strongly recommends that you make yourself completely acquainted with the Gosu programming language.

Statements are merely phrases that perform tasks within Studio. Examples of statements include the following:

- Assignment statements
- If statements
- Iteration statements

All expressions follow the dot notation to reference fields and subobjects. For example, to reference the license plate data from a vehicle associated with a claim, the field path is:

```
claim.Vehicles[0].LicensePlate
```

A statement can consist of one or many expressions.

IMPORTANT Use only Gosu expressions and statements to create ClaimCenter business rules. For example, do not attempt to define a Gosu function in a business rule. Instead, define a Gosu function in a Gosu class or enhancement, then call that function from a Gosu rule. Guidewire expressly does not support the definition of functions—and especially nested functions—in Gosu rules.

Rule Members

As described previously, a rule consists of a set of conditions and actions to perform if all the conditions evaluate to true. It typically references the important business entities and objects (claims, for example).

- **Rule Conditions** – A collection of expressions that provide true/false analysis of an entity. If the condition evaluates to true, then Studio runs the activities in the Rule Actions pane.

For example, you can use the following condition to test whether the claim passes a certain level of validation tests:

```
claim.isValid( "loadsolve", true)
```

- **Rule Actions** – A collection of action expressions that perform business activities such as making an assignment or marking a field as invalid. These actions occur only if the corresponding condition is true.

For example, you can use the following action to assign a claim to a local group based on which group handles the loss location's region:

```
claim.CurrentAssignment.assignGroupByLocation("independent", claim.LossLocation, true, local)
```

- **APIs** – A collection of Gosu methods accessed through gw.api.*. They include many standard math, date, string, and financial methods.

For example, you can use the following API method to test whether it has been more than 30 days since the insured reported the claim:

```
gw.api.util.DateUtil.daysSince( claim.ReportedDate ) > 30
```

- **Entities** – The collection of business object entities supported by ClaimCenter. Studio objects use the familiar “dot” notation to reference fields and objects.

For example, you can use the following object conditions to verify if the loss is an Auto claim involving a vehicle collision without the weather field entered:

```
Claim.LossType == "auto" and Claim.LossCause == "vehcollision" and Claim.Weather == null
```

Defining the Rule Conditions

The simplest kind of condition looks at a single field on the object or business entity. For example:

```
Activity.ActivityPattern.Code == "AuthorityLimitActivity"
```

This example demonstrates some important basics:

1. To refer to an object (for example, an Activity object, or, in this case, an ActivityPattern object) and its attributes, you begin the reference with a *root object*. While running rules on an activity, you refer to the activity in question as *Activity*. (Other root objects, depending on your Guidewire application, are *Account*, *PolicyPeriod*, *Producer*, *Invoice*, *TroubleTicket*, and *Charge*.)
2. ClaimCenter uses *dot* notation to access fields (for example, *Activity.ActivityPattern*) or objects (*Activity.ActivityPattern.Category*, for example), starting from the root object.

Combining Rule Conditions

For more complicated conditions, you can combine simple conditions like the previous one with standard Boolean logic operators (and, or, not). For example:

```
Activity.ActivityPattern.Code == "AuthorityLimitActivity" and not Activity.Approved
```

(See the “Gosu Operators and Expressions” on page 63 in the *Gosu Reference Guide* for details on operator precedence and other syntactical information.)

IMPORTANT The rule condition statement must evaluate to either Boolean true or false. If you create a condition statement that evaluates to null, ClaimCenter interprets this false. This can happen inadvertently, especially if you create a condition statement with multiple conditions to evaluate. If your condition evaluates to null (false), then ClaimCenter never executes the associated rule actions.

Defining a Statement List as Rule Conditions

It is also possible to use a statement list, instead of a simple expression, in the **Rule Actions** pane. However, the statement list must contain a `return` statement. For example:

```
uses java.util.HashSet
uses gw.lang.reflect.IType

var o = new HashSet<IType>() {A, B, C, ...}
return o.contains(typeof(...))
```

Defining the Rule Actions

Within the **Rule Actions** pane, you create the outcome for the criteria identified in the **Conditions** section. Actions can be single statements or they can be strung together to fulfill multiple criteria. You can add any number of actions to a rule. Studio executes all of these actions in order if the condition evaluates to `true`.

For example, suppose that you want an exposure to fail validation if a certain coverage exists but it is not selected on the claim. The following syntax creates this action:

```
for( Exposure in claim.Exposures ) {

    if( Exposure.OtherCoverage==true and (Exposure.OtherCoverageInfo==null or
        Exposure.OtherCoverageInfo=="") ) {

        // Mark each exposure in which this occurs as invalid
        Exposure.rejectSubField( Exposure, "OtherCoverageInfo", "loadsSave",
            "Please provide information about claimant's other coverage.", null, null )

    }
}
```

(See “Correcting Validation Errors” on page 78 for a discussion of the various `reject` methods.)

Exiting a Rule

At some point in the decision tree, ClaimCenter makes the decision that you want. At this point, it is important that ClaimCenter not continue. Indeed, if rule checking did continue, if ClaimCenter processed the *default* rule, it might overwrite the decision that came earlier. Therefore, you need to be able to instruct ClaimCenter at what point to stop considering any further rules.

Guidewire Studio provides several options for this flow control, with the simplest version simply meaning:

Stop everything! I am done with this rule set.

The following list describe the methods that you can add as an action for a rule to tell ClaimCenter what to do next.

Flow control action	Description
<code>actions.exit</code>	This is the simplest version. ClaimCenter stops processing the rule set as soon as it encounters this action.
<code>actions.exitAfter</code>	This action causes ClaimCenter to stop processing the rule set after processing any child rules.
<code>actions.exitToNext</code>	This action causes ClaimCenter to stop processing the current rule and immediately go to the next peer rule. The next peer rule is one that is at the same level in the hierarchy. This is likely to be used only rarely within very complicated action sections.
<code>actions.exitToNextParent</code>	This action causes ClaimCenter to stop processing the current rule and immediately go to the next rule at the level of the parent rule. It thus skips any child rules of the current rule.
<code>actions.exitToNextRoot</code>	This action causes ClaimCenter to stop processing the current rule and immediately go to the next rule at the top level. It thus skips any other rules in the entire top-level branch containing this rule.

Usually, you list an exit action as the last action in a rule so that it occurs only after all the other actions have been taken.

exitToNextRoot

It is useful to employ the `exitToNextRoot` method if you set up your rule set to make two separate decisions. For example, you might set up your segmentation rules to first determine the severity rating of an injury, and then decide on the segmentation using the results of this determination. For example, you can structure the rule set as follows:

- Always process my child rules to evaluate severity...
 - If (conditions) Then (Set severity) [Done but go to next top-level rule]
 - Otherwise...
- Always process my child rules to evaluate categorization...
 - If (conditions) Then (Set segment) [Done; no need to go further]
 - Otherwise...

After making the first decision (setting the severity), ClaimCenter is finished with that part of the decision logic. However, it needs to move on to the next major block of decision-making (that is, deciding on the categorization). After setting the segment, the rule set is finally complete, so it can just use the simple exit method.

Gosu Annotations and ClaimCenter Business Rules

Guidewire ClaimCenter uses the annotation syntax to add metadata to a Gosu class, constructor, method, or property. For example, you can add an annotation to indicate what a method returns or to indicate what kinds of exceptions the method might throw.

Guidewire marks certain Gosu code in the base application with a `@scriptable-ui` annotation. This restricts its usage (or *visibility*) to non-rules code. (The converse of this is the `@scriptable-all` annotation that indicates that the class, method, or property is visible to Gosu code everywhere.)

Within the business rules, the Gosu compiler ignores a class, a property, or a method marked as `@scriptable-ui`. For example, suppose that you attempt to access a property in Studio that has a `@scriptable-ui` annotation on it. Studio tells you that it cannot find a property descriptor for that property. The compiler does recognize the code, however, in other places such as in classes or enhancements. *Thus, you find that some Gosu code appears valid in some situations and not others. This is something of which you need to be aware as it can appear that you are writing incorrect Gosu code. It is the location of the code that is an issue, not the code itself.*

To see a list of the annotations that are valid for a particular piece of Gosu code (a class, constructor, method, or property), do the following:

- Place your cursor at the beginning of the line directly above the affected code.
- Type an @ sign.

Studio displays a list of validation annotations.

See also

- For information on working with annotation, including creating your own annotations, see “Annotating a Class, Method, Type, Class Variable, or Argument” on page 219 in the *Gosu Reference Guide*.

Invoking a Gosu Rule from Gosu Code

It is possible to invoke the Gosu business rules in a rule set from Gosu code. To do so, use the following syntax:

```
rules.[rule set category].[rule set name].invoke([root entity])
```

It is important to understand that you use the `invoke` method on a rule set to trigger all of the rules in that rule set. You cannot invoke individual rules within a rule set using the `invoke` method. ClaimCenter executes all of the rules in the invoked rule set in sequential order. If a given rule's condition expression evaluates to `true`, then ClaimCenter executes the Gosu code in **Rule Actions** pane for that rule.

See also

- “Using the Rules Editor” on page 25
- “Working with Rules” on page 25

Using the Rules Editor

This topic describes the Studio Rules editor and how you use it to work with Gosu business rules.

This topic includes:

- “Working with Rules” on page 25
- “Changing the Root Entity of a Rule” on page 28
- “Making a Rule Active or Inactive” on page 30
- “Making a Rule Active or Inactive” on page 30

Working with Rules

ClaimCenter organizes and displays rules as a hierarchy in the center pane of Guidewire Studio, with the rule set appearing at the root, the top level, of the hierarchy tree. Studio displays the **Rule** menu only if you first select a rule set category in the **Project** window. Otherwise, it is unavailable.

There are a number of operations involving rules that you can perform in the Studio Rules (Gosu) editor. See the following for details:

- To view or edit a rule
- To change rule order
- To create a new rule set category
- To create a new rule set
- To create a new rule
- To access a rule set from Gosu code
- To change the root entity of a rule

To view or edit a rule

1. In the Studio Project window, navigate to **configuration** → **config** → **Rule Sets**.
2. Expand the **Rule Sets** folder, and then expand the rule set category.

3. Select the rule set you want to view or edit. All editing and saving in the tool occurs at the level of a rule set.

To change rule order

If you want to change the order of your rules, you can drag and drop rules within the hierarchy.

1. Click the rule that you want to move, and then hold down the mouse button and move the pointer to the new location for the rule. Studio then displays a line at the insertion point of the rule. Release the mouse button to paste the rule at that location.
2. To make a rule a child of another rule, select the rule you want to be the child, and then choose **Edit → Cut**. Click on the rule that you want to be the parent, and then choose **Edit → Paste**.
3. To move a rule up a level, drag the rule next to another rule at the desired level in the hierarchy (the reference rule). Notice how far left the insertion line extends.
 - If the line ends before the beginning of the reference rule's name, Studio inserts the rule as a child of the reference rule.
 - If the line extends all the way to the left, Studio inserts the rule as a peer of the reference rule.

By moving the cursor slightly up or down, you can indicate whether you want to insert the rule as a child or a peer.

If you move rules using drag and drop, ClaimCenter moves the chosen rule and all its children as a group. This behavior makes it easy to reposition entire branches of the hierarchy.

ClaimCenter also supports standard cut, copy, and paste commands, which can be used to move rules within the hierarchy. If you paste a cut or copied rule, Studio inserts the rule as if you added a new rule. It becomes the last child of the currently selected rule.

To create a new rule set category

Guidewire divides the sample rule sets into categories, or large groupings of rules that center around a certain business process, for example, assignment or validation. In the rules hierarchy, rule set categories consist of rule sets, which, in turn, further subdivide into individual rules.

- Rule sets are logical groupings of rules specific to a business function with Guidewire ClaimCenter.
- Rules contain the Gosu code (condition and action) that perform the actual business logic.

It is possible to create new rule set categories through Guidewire Studio.

1. In the Studio Project window, navigate to **configuration → config → Rule Sets**.
2. Right-click **Rule Sets**, and then click **New → Rule Set Category**.
3. Enter a name for the rule set category.

Studio inserts the rule set category in the category list in alphabetic order.

To create a new rule set

In the base configuration, Guidewire provides a number of business rules, divided into rule sets, which organize the business rules by function. It is possible to create new rule sets through Guidewire Studio.

1. In the Studio Project window, navigate to **configuration → config → Rule Sets**. Although the label is **Rule Sets**, the children of this folder are actually rule set categories.
2. Select a rule set category or create a new rule set category.
3. Right-click the category, and then click **New → Rule Set**.

4. Enter the following information:

Field	Description
Name	Studio displays the rule name in the middle pane. For the Guidewire recommendations on rule set names, see "Generating Rule Debugging Information" on page 31. In general, however, if you create a rule set for a custom entity named Entity_Ext, you must name your rule set Entity_Ext<RuleSet>. Thus, if you want the custom entity to invoke the Preupdate rules, then name your rule set Entity_ExtPreupdate. There are some variations in how to name a rule set. See the existing rule sets in that category to determine the exact string to append and follow that same pattern with new rule sets in that category.
Description	Studio displays the description in a tab at the right of the Studio if you select the rule set name in the middle pane. Guidewire recommends that you make the description meaningful, especially if you have multiple people working on rule development. In any case, a meaningful rule description is particularly useful as time passes and memories fade.
Entity Type	ClaimCenter uses the entity type as the basis on which to trigger the rules in this rule set. For example, suppose that you select a rule set, then a rule within the set. Right-click and select Complete Code from the menu. Studio displays the entity type around which you base the rule actions and conditions.

To create a new rule

1. Select the rule set that will contain the new rule in the Studio Resources pane.
2. Do one of the following:
 - If the new rule is to be a top-level rule, select the rule set name in the middle pane.
 - If the new rule is to be a child rule, expand the rule set hierarchy in the middle pane and select the parent rule.
3. Select New Rule from the Rule menu, or right-click and select New Rule.
4. Enter a name for the new rule in the New Rule dialog box. Studio creates the new rule as the last child rule of the currently selected rule (or rule set).

To access a rule set from Gosu code

You can access a rule set within a rule set category (and thus, all the rules within the rule set) by using the following Gosu invoke method. You can use this method to invoke a rule set in any place that you use Gosu code.

```
rules.RuleSetCategory.RuleSet.invoke(entity)
```

You can only invoke a rule set through the Gosu invoke method, not individual rules. Invoking the rule set triggers evaluation of every rule in that rule set, in sequential order. If the conditions for a rule evaluate to true, then ClaimCenter executes the actions for that rule.

Renaming or Deleting a Rule

Use the following menu commands to rename a rule or to delete it entirely from the rule set. You can also access these commands by right-clicking a rule and selecting the command from the drop-down list.

Command	Description	Actions you take
Rule → Rename Rule	Renames the currently selected rule	Select the rule to rename in the center pane of Studio, and then select Rename Rule from the Rule menu, or right-click and select Rename Rule . Enter the new name for the rule in the Input dialog box. You must save the rule for the change to become permanent.
Edit → Delete	Deletes the currently selected rule	Select the rule to delete in the center pane of Studio, then select Delete from the Edit menu, or right-click and select Delete . Note that there is no secondary dialog window that asks you to confirm your selection. You can use the Delete command only to delete rules.

Renaming a Rule

At a structural level, Guidewire ClaimCenter stores each rule as a separate Gosu class, with a .gr extension. The name of the Gosu class corresponds to the name of the rule that you see in the Studio Project window.

ClaimCenter stores the rule definition classes in the following location:

ClaimCenter/modules/configuration/config/rules/...

If you rename a rule set, ClaimCenter renames the class definition file in the directory structure and any internal class names. It also renames the directory name if the rule has children. Thus, ClaimCenter ensures that the rule class names and the file names are always synchronized.

Changing the Root Entity of a Rule

ClaimCenter bases each Gosu rule on a specific business entity. In general, the rule set name reflects this entity. For example, in the Preupdate rule set category, you have Activity Preupdate rules and Contact Preupdate rules. These rule set names indicate that the root entity for each rule set is—respectively—the **Activity** object and the **Contact** object. You can also determine the root entity for a specific rule by using the Studio code completion functionality:

1. Create a sample line of code similar to the following:

```
var test =
```

2. Right-click after the = sign and select **Complete Code** from the menu. The Studio code completion functionality provides the root entity. For example:

```
activity : Activity
contact : Contact
```

You can also use CTRL-Space to access the Studio code completion functionality.

ClaimCenter provides the ability to change the root entity of a rule through the use of the right-click **Change Root Entity** command on a rule set.

Why Change a Root Entity?

The intent of this command is to enable you to edit a rule that you otherwise cannot open in Studio because the declarations failed to parse.

Do not use this command in other circumstances.

For example, suppose that you have the following sequence of events:

1. You create a new entity in ClaimCenter, for example, `TestEntity`. Studio creates a `TestEntity.eti` file and places it in the following location:

```
modules/configuration/config/extensions
```

2. You create a new rule set category called `TestEntityRuleSetCategory` in Rule Sets, setting `TestEntity` as the root entity. Studio creates a new folder named `TestEntityRuleSetCategory` and places it in the following location:

```
modules/configuration/config/rules/rules
```

3. You create a new rule set under `TestEntityRuleSetCategory` named `TestEntityRuleSet`. Folder `TestEntityRuleSetCategory` now contains the rule set definition file named `TestEntityRuleSet.grs`. This file contains the following (simplified) Gosu code:

```
@gw.rules.RuleName("TestEntityRuleSet")
class TestEntityRuleSet extends gw.rules.RuleSetBase {
    static function invoke(bean : entity.TestEntity) : gw.rules.ExecutionSession {
        return invoke( new gw.rules.ExecutionSession(), bean )
    }
    ...
}
```

Notice that the rule set definition explicitly invokes the root entity object: `TestEntity`.

4. You create one or more rules in this rule set that use `TestEntity` object, for example, `TestEntityRule`. Studio creates a `TestEntityRule.gr` file that contains the following (simplified) Gosu code:

```
internal class TestEntityRule {
    static function doCondition(testEntity : entity.TestEntity) : boolean {
        return /*start00rule*/true/*end00rule*/
    }
    ...
}
```

Notice that this definition file also references the `TestEntity` object.

5. Because of upgrade or other reasons, you rename your `TestEntity` object to `TestEntityNew` by changing the file name to `TestEntityNew.eti` and updating the entity name in the XML entity definition:

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
    entity="TestEntityNew" ... >
</entity>
```

This action effectively removes the `TestEntity` object from the data model. This action, however, does not remove references to the entity that currently exist in the rules files.

6. You update the database by stopping and restarting the application server.

7. You stop and restart Studio.

As Studio reopens, it presents you with an error message dialog. The message states that Studio could not parse the listed rule set files. It is at this point that you can use the **Change Root Entity** command to shift the root entity in the rule files to the new root entity. After you do so, Studio recognizes the new root entity for these rule files.

To change the root entity of a rule

1. Select a rule set.
2. Right-click and select **Change Root Entity** from the drop-down menu. Studio prompts you for an entity name.
3. Enter the name of the new root entity.

After you supply a name:

- Studio performs a check to ensure that the name you supplied is a valid entity name.
- Studio replaces occurrences of the old entity in the function declarations of all the files in the rule set with the new entity. This replacement only works, however, if the old root type was in fact an entity.
- Studio changes the name of the entity instance passed into the condition and action of each rule.

- Studio does not propagate the root entity change to the body of any affected rule. You must correct any code that references the old entity manually.

The intent of this command is to enable you to edit a rule that you otherwise cannot open in Studio because the declarations failed to parse. Do not use this command in other circumstances.

Making a Rule Active or Inactive

ClaimCenter skips any inactive rule, acting as if its conditions are false. (This causes it to skip the child rules of an inactive rule, also.) You can use this mechanism to temporarily disable a rule that is causing incorrect behavior (or that is no longer needed) without deleting it. Sometimes, it is helpful to keep the unused rule around in case you need that rule or something similar to it in the future.

To make a rule active, set the check box next to it. To make a rule inactive, clear the check box next to it.

Writing Rules: Testing and Debugging

This topic describes ways to test and debug your rules.

This topic includes:

- “Generating Rule Debugging Information” on page 31
- “Using Custom Logging Methods” on page 32

Generating Rule Debugging Information

It is a very useful practice to add printing and logging statements to your rules to identify the currently executing rule, and to provide information useful for debugging purposes. Guidewire recommends that you use all of the following means of providing information extensively:

- Use the `print` statement to provide instant feedback in the server console window.
- Use the `gw.api.util.Logger.*` methods to print out helpful error messages to the application log files.
- Use comments embedded within rules to provide useful information while troubleshooting rules.

Printing Debugging Information

The `print(String)` statement prints the *String* text to the server console window. This provides immediate feedback. You can use this method to print out the name of the currently executing rule and any relevant variables or parameters. For example, the following code prints the name of the currently executing rule set to the server console window:

```
print("This is an ANNOUNCEMENT MESSAGE -- I am running the "
+ actions.getRuleSet().DisplayName + " Rule Set")
```

Logging Debugging Information

Use the `gw.api.util.Logger.*` methods to send information to application log files or the console window for debugging purposes. See the following for details:

- “Logging” on page 613 in the *Integration Guide*

- “Configuring Logging” on page 25 in the *System Administration Guide*

Using Custom Logging Methods

You can also write a custom Gosu method that logs information. You then call this method from the rule to perform the logging operation. The following example uses Gosu class method `logRule` to output information about the rule and rule set currently executing. Notice that you must create a new instance of the class before you can use it.

- See the for information about creating and using Gosu class methods.
- See “Script Parameters” on page 105 in the *Configuration Guide* for information about creating and using script parameters. (Notice that using a script parameter in this fashion enables you to easily turn rule logging on and off.)

Rule Action

```
var log = new util.CustomLogging.logRule( claim, actions.getRuleSet().DisplayName,  
actions.getRule(), "Custom text..." );
```

CustomLogging Class

```
package util;  
  
class CustomLogging {  
    public function logRule( claim:Claim, ruleSetName: String, ruleName:String,  
    logMessage:String ) : Boolean {  
  
        var ruleInfo:String = "Rule Set: " + ruleSetName + "\n Rule Name: " + ruleName  
        var message:String = (logMessage != null) ? (" \n Message: " + logMessage) : ("")  
  
        if(ScriptParameters.DO_LOGGING) {  
            gw.api.util.Logger.logDebug( "\n### CLAIMCENTER RULE EXECUTION LOGGER (Policy #: "  
                + claim.ClaimNumber + ") ###" + "\n" + ruleInfo + message + "\n" + "Timestamp: "  
                + gw.api.util.StringUtil.formatTime(gw.api.util.DateUtil.currentDate(), "full") + "\n" )  
        }  
        return true  
    }  
}
```

Result

```
### CLAIMCENTER RULE EXECUTION LOGGER (Claim #: 123456) ###  
Rule Set: Segmentation  
Rule Name: Exposure Segmentation Rules  
Message: Custom Text...  
Timestamp: 4:37:24 PM PDT
```

Writing Rules: Examples

This topic describes ways to perform more complex or sophisticated actions in rules.

This topic includes:

- “Accessing Fields on Subtypes” on page 33
- “Looking for One or More Items Meeting Conditions” on page 34
- “Preventing Repeated Actions” on page 34
- “Taking Actions on More Than One Subitem” on page 35
- “Checking Permissions” on page 35
- “Determining the Original Attributes of a Changed Entity” on page 35

Accessing Fields on Subtypes

Various entities in ClaimCenter have subtypes, and a subtype may have fields that apply only to it, and not to other subtypes. For example, a Contact object has a Person subtype, and that subtype contains a DateOfBirth field. However, DateOfBirth does not exist on a Company subtype. Similarly, only the Company subtype has the Name (company name) field.

Because these fields apply only to particular subtypes, you cannot reference them in rules by using the primary root object. For example, the following illustrates an invalid way to refer to the lienholder of a vehicle:

```
Exposure.lienholder[0].FirstName == "Joe"    // invalid
```

To access a field that belongs to a subtype, you must “cast” (or convert) the primary object to the subtype by using the as operator. For example, you would cast a contact to the Person subtype using the following syntax:

```
(Exposure.lienholder[0] as Person).FirstName == "Joe"    // valid
```

As another example, consider transaction validation rules. A transaction set has several subtypes, such as a ReserveSet, CheckSet, and others. To validate that primary checks being sent by mail have a mailing address that is not null, you can use the following in the rule condition:

```
TransactionSet.Subtype == "checkset" &&
  (transactionSet as CheckSet).PrimaryCheck.DeliveryMethod == "send" &&
  (transactionSet as CheckSet).PrimaryCheck.MailToAddress == null
```

It is important to cast the type of an object correctly, otherwise, a Runtime Error can occur.

Looking for One or More Items Meeting Conditions

If you want to check the value of a single field, such as whether a particular activity is overdue, you use a construction similar to the following:

```
Activity.TargetDate < gw.api.util.DateUtil.currentDate()
```

However, if you want to know whether the current claim has any unfinished activities that are overdue, you need to look at all activities on the claim. Clearly, you need something more complicated than `Claim.*` to express this condition. ClaimCenter provides an `exists(...in...where...)` syntax to describe these tests.

For example:

```
exists ( Activity in Claim.Activities where Activity.Status == "open" and  
Activity.TargetDate < gw.api.util.DateUtil.currentDate() )
```

This condition evaluates to `true` if there are one or more activities connected to the claim that meet the conditions described. It is also common to use the `exists` method to look for a certain kind of exposure within a claim. For example, “does this Auto claim have any BI exposures?”

See “Existence Testing Expressions” on page 70 in the *Gosu Reference Guide* for more information on Gosu `exists(...in...where)` expressions.

Preventing Repeated Actions

Many times, during exception rule execution, you want to take action on an exception the first time you discover it only. For example, suppose that you want to remind the adjuster to set the fault rating on an claim that is open for more than 30 days. If this action has not been completed within 30 days, then you might want to add an activity on day 31 to remind the adjuster. On day 32, if the adjuster still has not set the rating, you do not want to add another activity, even though ClaimCenter finds the same exception again.

One way to handle this is to use ClaimCenter methods for noting a custom event in the Claim History. The first time ClaimCenter finds an exception, you need to instruct ClaimCenter to note it in the claim’s history. Then, in the rule, include a check to verify if this event appeared before in the rule’s exceptions. If the event already exists, then do not execute the actions again.

The syntax for this is similar to the following:

Rule Conditions

```
gw.api.util.DateUtil.daysSince( Claim.ReportedDate ) > 30 and  
not exists (hist in Claim.History where hist.CustomType == "a_n_f_r")
```

Rule Actions

```
claim.createCustomHistoryEvent( "a_n_f_r", "description" )  
actions.claim.createActivity( parameterList )
```

In this example, ClaimCenter determines if whether the event occurred previously. You can also determine if the event occurred within a certain time period. In this way, you can repeat the rule’s actions if enough time has occurred since ClaimCenter first noted the exception.

Taking Actions on More Than One Subitem

A common situation to handle is raising the priority on every activity on a claim that is overdue. ClaimCenter provides a `for(... in ...)` syntax and an `if(... { then do something })` syntax that you can use to construct fairly complicated actions:

```
for (act in Claim.Activities) {
    if (act.Status == "open" and act.TargetDate < gw.api.util.DateUtil.currentDate() ) {
        act.Priority = "urgent"
    }
}
```

Notice that the “{” and the “}” curly braces mark the beginning and end of a block of commands.

- The outer set brackets one or more commands to do “for” each activity on the claim.
- The inner set brackets the commands to do “if” the activity is overdue.

You can use the `Length` method on a subobject to determine how many subobjects exist. For example, if there are five activities on a claim, then the following expression returns the value 5:

```
Claim.Activities.Length
```

See “Gosu Conditional Execution and Looping” on page 92 in the *Gosu Reference Guide* for more information on the `for(... in ...)` Gosu syntax.

Checking Permissions

ClaimCenter provides a Gosu mechanism for checking user permission on an object by accessing properties and methods off the object in the `perm` namespace.

- ClaimCenter exposes static permissions that are non-object-based (like the permission to create a user) as Boolean properties.
- ClaimCenter exposes permissions that take an object (like the permission to edit a claim) as methods that take an entity as their single parameter.
- ClaimCenter exposes application interface permissions as typecodes on the `perm.System` object.

All the properties and methods return Boolean values indicating whether or not the user has permission to perform the task. ClaimCenter always evaluates permissions relative to the current user unless specifically instructed to do otherwise. You can use permissions anywhere that you can use Gosu (in PCF files, rules, and classes) and there is a current user.

For example:

```
print(perm.Account.view(Account))
print(perm.User.create)
print(perm.System.accountcontacts)
```

You can also check that any given user has a specific permission, using the following Gosu code:

```
var u : User = User( "SomeUser" /* Valid user name*/ )
var hasPermission = exists (role in u.Roles
    where exists (perm in role.Role.Privileges where perm.Permission=="SomeValidPermission"))
```

If using this code in a development environment, you must connect Studio to a running development application server before Studio recognizes users and permissions.

Determining the Original Attributes of a Changed Entity

Note: For information on how to determine if an entity changed, see “Determining What Data Changed in a Bundle” on page 338 in the *Gosu Reference Guide*.

The following example illustrates how to determine if the original values of fields on an entity have been changed, and, if so, what the original values were. For example, suppose that the Loss Location on the claim changes for some reason. You might want to retrieve the original values to compare with the current values to determine how the change in the Loss Location state.

It is not sufficient to determine the original values simply by knowing that there was a change to the Loss Location. The following expression does not provide the correct value, as it does not point back to the original Loss Location.

```
Claim.LossLocation.getOriginalValue("State")
```

Instead, to determine the original attributes of a changed object, you must first re-instantiate the original object.

In this example, the rule executes if the `isFieldChanged("LossLocation")` becomes `true`. Then, if the new Loss Location state is different from the original state, the rule creates a custom history event identifying the change.

Rule Condition

```
Claim.isFieldChanged("LossLocation")
```

Rule Actions

```
var add : Address = Address( claim.getOriginalValue("LossLocation") )
if (add.State <> Claim.LossLocation.State) {
    Claim.createCustomHistoryEvent( "DataChange", "The loss location state changed from " +
        add.State + " to " + Claim.LossLocation.State)
}
```

Additional “Changed Entity” Methods

Guidewire provides a number of methods that you can use to determine if an entity changes. (For information on each, place your cursor on the method signature and press F1.)

```
Entity.getOriginalValue(fieldname : String)
Entity.getAddedArrayElements(arrayFieldName : String)
Entity.getChangedArrayElements( arrayFieldName : String)
Entity.getRemovedArrayElements( arrayFieldName : String)
Entity.isArrayElementChanged(arrayFieldName : String)
Entity.isArrayElementAddedOrRemoved(arrayFieldName : String)
Entity.isFieldChanged(fieldname : String)
```

You can also use the read-only `Changed` property on the entity to determine if a change has occurred. The `Changed` property becomes `true` only after a modification to the entity.

```
Entity.Changed
```

These methods are available in all Gosu code, not just Gosu rules.

Rule Set Categories

This topic reviews the sample rules that Guidewire provides as part of the base configuration of ClaimCenter. You can access the Gosu business rules by opening Guidewire Studio and navigating in the **Projects** window to **configuration → config → Rule Sets**.

This topic includes:

- “Rule Set Summaries” on page 38
- “Approval Routing Rule Set Category” on page 38
- “Archive Rule Set Category” on page 40
- “Assignment Rule Set Category” on page 44
- “BulkInvoice Approval Rule Set Category” on page 45
- “Closed Rule Set Category” on page 46
- “Event Message Rule Set Category” on page 48
- “Exception Rule Set Category” on page 52
- “Initial Reserve Rule Set Category” on page 55
- “Loaded Rule Set Category” on page 56
- “Postsetup Rule Set Category” on page 57
- “Presetup Rule Set Category” on page 59
- “Preupdate Rule Set Category” on page 63
- “Reopened Rule Set Category” on page 70
- “Segmentation Rule Set Category” on page 72
- “Transaction Approval Rule Set Category” on page 73
- “Validation Rule Set Category” on page 74
- “Workplan Rule Set Category” on page 84

Rule Set Summaries

ClaimCenter Studio contains the following sample rule sets:

Rule set category	Contains rules to
Approval Routing	Route approval activities for a transaction that requires approval to the appropriate user. See “Approval Routing Rule Set Category” on page 38.
Archive	Manage the claim archive process. See “Archive Rule Set Category” on page 40.
Assignment	Determine the responsible party for an activity, claim, exposure, matter, or service request. See “Assignment Rule Set Category” on page 44.
BulkInvoice Approval	Define the approval requirements for Bulk Invoices. See “BulkInvoice Approval Rule Set Category” on page 45.
Closed	Perform an action any time that you close an activity, claim, exposure, or matter—for example, sending a notification. See “Closed Rule Set Category” on page 46.
Event Message	Handle communication with integrated external applications. See “Messaging and Events” on page 299 in the <i>Integration Guide</i> for information events and event messaging. See “Event Message Rule Set Category” on page 48.
Exception	Specify an action to take under certain circumstances. For example, if no one touches a claim for a certain number of days, then create a review activity. See “Exception Rule Set Category” on page 52.
Initial Reserve	Determine the initial financial reserves that to create for a new exposure. See “Initial Reserve Rule Set Category” on page 55.
Loaded	Trigger automatic actions any time that you load a new claim using the FNOL import interface. See “Loaded Rule Set Category” on page 56.
Postsetup	Trigger automatic actions any time that you add a new claim within ClaimCenter. These rules run after the Assignment and Workplan rules complete their execution. See “Postsetup Rule Set Category” on page 57.
Presetup	Trigger automatic actions any time that you add a new claim, exposure, matter or transaction within ClaimCenter. These rules run prior to the execution of rules in the Segmentation, Assignment, and Workplan rule set categories. See “Presetup Rule Set Category” on page 59.
Preupdate	Trigger automatic actions any time that a claim or other high-level entity changes. See “Preupdate Rule Set Category” on page 63.
Reopened	Trigger automated actions on reopening a claim or exposure. See “Reopened Rule Set Category” on page 70.
Segmentation	Categorize a new claim or an exposure based on complexity, severity of damage, or other attributes. See “Segmentation Rule Set Category” on page 72.
Transaction Approval	Verify that a user has the appropriate authorization to submit a financial transaction. See “Transaction Approval Rule Set Category” on page 73.
Validation	Check for missing information or invalid data on Address Book, Claim, Exposure, Matter, ServiceRequest, Policy and Transaction objects. See “Validation Rule Set Category” on page 74.
Workplan	Add initial activities to a claim, exposure or matter as a checklist of work that various people need to do on the claim. See “Workplan Rule Set Category” on page 84.

Approval Routing Rule Set Category

Approval routing rules route an approval activity for a transaction that requires approval. ClaimCenter uses approval routing rules to determine which person to assign the approval activity to. Typically, a supervisor is the approving authority. The transaction set and its transactions remain in a *pending approval* state until someone approves the activity.

Approval Routing Rules

The Approval Routing rule set and the rule sets in the Transaction Approval rule set category form a pair. ClaimCenter calls these rule sets sequentially.

- ClaimCenter calls rule sets in the Transaction Approval rule set category any time that you submit any kind of financial transaction. See “Transaction Approval Rule Set Category” on page 73.
- ClaimCenter calls the Approval Routing rule set either if the outcome of transaction approval is Yes or if the transaction authority limit is exceeded, causing the transaction to need approval. See “Approval Routing Rules” on page 39.

IMPORTANT Be careful not to create approval rules that generate an infinite loop. For example, suppose a rule first assigns the transaction to a supervisor for approval, who subsequently approves it. Then, the rule assigns the transaction approval to someone who does not have the authority to approve the transaction. ClaimCenter runs the rule set again, and again reassigns the approval to the supervisor, generating a never-ending loop.

The following list describes the main methods used to set the person who can approve the activity. These methods are available on the `TransactionSet` object.

Method	Description
<code>approveByGroupSupervisor</code>	<p>Use to assign an approval active to a group supervisor based on the exposure or exposures associated with the transaction:</p> <ul style="list-style-type: none"> • If the transaction set has transactions associated with <i>only one</i> exposure—the method assigns the approval activity to the supervisor of the group assigned to the exposure. • If the transaction set has transactions associated with <i>more than one</i> exposure—the method assigns the approval activity to the supervisor of the group assigned to the associated claim. <p>In either case, if the supervisor of the group is the same as the user requesting approval, then the activity remains unassigned. <i>If you use this method, you must explicitly handle this case.</i></p>
<code>approveByUserSupervisor</code>	<p>Use to assign an approval activity to a group supervisor based on the user who submitted the transaction:</p> <ul style="list-style-type: none"> • If the submitting user is in the owning group and is not the supervisor, then approval goes to the group supervisor. • If the submitting user is the group supervisor, then ClaimCenter escalates the approval to the group supervisor of the next group up the group hierarchy. • If the submitting user is a group supervisor someplace higher in the hierarchy, then the approval escalates to the supervisor of the next group up the hierarchy. • If the submitting user is in <i>one and only one</i> group, then the approval goes to the supervisor of that group. Otherwise, approval goes to the supervisor of the owning group. <p>Finally, if ClaimCenter worked its way through the previous conditions to the supervisor of the root group without making an assignment, then ClaimCenter assigns the approval to user <code>defaultowner</code>.</p>
<code>setApprovingUser</code>	<p>Use to assign an approval activity to a specified user. You must provide a user and group as parameters. The method checks whether the user is retired or inactive (<code>user.credential.Active != true</code>) and returns <code>false</code> if either of these conditions exist. It also returns <code>false</code> if the user or group is missing. It returns <code>true</code> otherwise.</p> <p>IMPORTANT Only call this method from within the Approval Routing rule set.</p>

In the following example, the rule assigns the approval to the supervisor, unless the supervisor is the one who entered the transaction.

```
Condition
return true
```

Action

```

var exp = transactionSet.AllTransactions*.Exposure
// Verify that the supervisor has not already approved this TransactionSet

if (not exists( act in transactionSet.Claim.Activities where (act.TransactionSet == transactionSet
    and (act.ActivityPattern.Code == "approve_reserve_change"
    or act.ActivityPattern.Code == "approve_payment" )
    and act.Approved == true and act.AssignedUser == exp[0].AssignedGroup.Supervisor ))) {
    if ( transactionSet.approveByGroupSupervisor() ) {
        gw.api.util.Logger.logDebug( " Assign to supervisor" )
        gw.api.util.Logger.logDebug( transactionSet.AllTransactions*.Exposure[0].AssignedGroup.Supervisor.
            Contact.DisplayName )
        actions.exit()
    }
}

```

BulkInvoice Approval Assignment Rules

The BulkInvoice Approval Assignment Rules and the BulkInvoice Approval Rules rule sets form a pair. These rules work together similarly to the rule sets in the Approval Routing rule set category and the Transaction Approval rule set category. ClaimCenter calls these rule sets sequentially.

ClaimCenter runs the BulkInvoice Approval Assignment rule set for each bulk invoice to determine to whom to assign it for approval. See “[BulkInvoice Approval Rules](#)” on page 45.

In the following example, the rule assigns approval for bulk invoice payments to user Carla Levitt in the Headquarters group.

Condition

```
return true
```

Action

```

bulkInvoice.setApprovingUser(User( "demo_sample:39" /* Carla Levitt */ ),
    util.RuleGroups.instance.getHeadquarters())

```

See also

- “[Approval Routing Rule Set Category](#)” on page 38
- “[Transaction Approval Rule Set Category](#)” on page 73

Archive Rule Set Category

IMPORTANT Guidewire provides a sample implementation of an archive plugin in the base configuration. You must provide an implementation of this plugin to meet your business needs. See “[Archiving Integration](#)” on page 575 in the *Integration Guide* for details.

Archiving is the process of moving a closed claim and associated data from the active ClaimCenter database to a document storage area. You can still search for and retrieve archived claims. But, while archived, these claims occupy less space in the active database.

Claims archiving in Guidewire ClaimCenter has the following characteristics:

- ClaimCenter itself automatically identifies claims that need to be archived and archives them.
- ClaimCenter enables you to search for archived claims in a similar fashion to searching for non-archived claims in the main database. In ClaimCenter, you can search for a claim without knowing or caring whether a claim has undergone the archiving process.
- ClaimCenter presents the same claim search results that contains the same claim summary information whether the claim is active or archived.

- ClaimCenter retrieves an archived claim after you click the **Retrieve from Archive** button if you want to work with an archived claim found through a search.

See also

- “Archiving” on page 123 in the *Application Guide* – Information on archiving claims, searching for archived claims, and retrieving archived claims.
- “Archive Parameters” on page 36 in the *Configuration Guide* – Information on the configuration parameters used in archiving.
- “Configuring Claim Archiving” on page 573 in the *Configuration Guide* – Information on configuring archiving, selecting entities for archiving, and archiving and the object (domain) graph.
- “Archiving Integration” on page 575 in the *Integration Guide* – Information on the archiving integration flow, storage and retrieval integration, and the plugin interfaces.
- “Archive Rule Set Category” on page 40 – Information on base configuration archive rules and their use in detecting archive events and managing the claims archive and retrieval process.
- “Logging Successfully Archived Claims” on page 27 in the *System Administration Guide* – Describes the logging file and how to configure logging for archived claims.
- “Purging Unwanted Claims” on page 50 in the *System Administration Guide* – Information on how to purge claims from the database and the archive.
- “Archive Info” on page 169 in the *System Administration Guide* – Information on using the **Archive Info** screen to view information about the archive. This screen includes an overview, information about the archiving plugin, a summary of archiving information by data model version, and details of each Archive work queue run.
- “Upgrading Archived Entities” on page 256 in the *Upgrade Guide* – Information on upgrading the XML for archived entities if you have made custom data model changes.

Detecting Archive Events

Every time that ClaimCenter creates a claim, it also creates a `ClaimInfo` entity. Every time a `ClaimInfo` entity changes (including during claim archiving, retrieval, claim exclusion, or archive failure), ClaimCenter generates a `ClaimInfoChanged` event. To determine the archive state of a claim, use `ClaimInfo.ArchiveState`. This field, from the `ArchiveState` typelist, can have the following possible values in the base configuration:

Archive state	The claim is...
archived	Finished archiving
retrieving	Marked for retrieval

Thus:

- The act of archiving a claim generates a `ClaimInfoChanged` event and the `ClaimInfo.ArchiveState` is **archived**.
- The act of retrieving an archived claim generates a `ClaimInfoChanged` event and the `ClaimInfo.ArchiveState` is **retrieving**.

A null `ArchiveState` indicates that the claim is not archived. This could indicate either that it has never been archived or that ClaimCenter successfully retrieved the claim from the archive.

In addition to the `ArchiveState` events, a claim’s `History` array receives a new entry each time ClaimCenter archives or retrieves the claim. Archive rules that reference these history entries or a `ClaimInfoChanged` event can distinguish archived and retrieved claims from other claims.

Skipped Claims Versus Excluded Claims

Through the Archive rules, you can do the following:

Skip a claim	<p>Skipping a claim during archiving makes that claim temporarily unavailable for archiving during this particular archiving pass. To skip a claim, call the following method on the claim and provide a reason:</p> <pre>claim.skipFromArchiving(String)</pre> <p>IMPORTANT Calling this method on a claim terminates rule set execution.</p>
Exclude a claim	<p>Excluding a claim from archiving makes that claim unavailable for archiving during this and future archiving passes. The claim becomes not archivable on a semi-permanent basis. To exclude a claim from archiving, call the following method on the claim and provide a reason for the exclusion:</p> <pre>claim.reportArchiveProblem(String)</pre> <p>Calling this method on a claim does not terminate rule set execution.</p>

Detecting Excluded Claims

There are several cases in which ClaimCenter excludes claims from archiving. They are described in the following topics:

- “Archiving Rule Conditions” on page 42
- “Archiving Process Failure” on page 42

Archiving Rule Conditions

ClaimCenter can exclude a claim from the set of claims to archive due to some condition related to the archiving rules. For example, a rule condition determines that the claim did not meet a certain business-related condition. In this case, ClaimCenter populates the following `ClaimInfo` properties, which you can use to view more details of the failure.

<code>ClaimInfo.ExcludeFromArchive</code>	If true, ClaimCenter excluded this claim from the archiving process.
<code>ClaimInfo.ExcludeReason</code>	Provides details on why ClaimCenter excluded this claim from the archiving.

Archiving Process Failure

ClaimCenter can exclude a claim from the set of claims to archive due to the failure of the archive process itself. For example, there was a foreign key violation. In this case, ClaimCenter populates the following `ClaimInfo` properties, which you can use to view more details of the failure.

<code>ClaimInfo.ArchiveFailure</code>	A foreign key to an <code>ArchiveFailure</code> object that provides a one-line summary of why the claim failed the archive process
<code>ClaimInfo.ArchiveFailureDetails</code>	A foreign key to an <code>ArchiveFailureDetails</code> object that provides the full stack trace related to the failure.

Managing Skipped or Excluded Claims

You can view information about skipped and excluded claims in ClaimCenter by clicking ALT+SHIFT+T and on the Server Tools tab navigating to the Info Pages → Archive Info screen.

Note: To see this screen, you must enable archiving. See “Enabling Archiving or Disabling Archiving Work Queue” on page 101 in the *Installation Guide*. Additionally, you might need special permissions, as described at “Using the Server Tools” on page 159 in the *System Administration Guide*.

The Archive Info screen is divided into three sections. The top section is an Overview section. Below that is a section with archive source information showing availability of the store, of retrieval, and of delete functionality. The bottom section summarizes archives by data model version.

The **Overview** section lists:

- Total number of claims skipped by the archive process
- Number of claims excluded because of rules
- Number of claims excluded because of failure

For skipped and excluded claims, you can investigate each individual item. You can also reset any excluded claim so that the archive process attempts to archive that claim the next time the archive process runs.

See also

- “Info Pages” on page 169 in the *System Administration Guide*

Default Group Claim Archiving Rules

You use the rules in this rule set to prevent ClaimCenter from archiving an otherwise eligible claim. ClaimCenter invokes this rule set immediately before archiving the claim, after the claim meets certain internal archivability criteria.

In the base configuration, Guidewire provides the following sample rules in this rule set. Guidewire expects you to customize the archiving rules to meet your business needs.

Default Group Claim Archiving Rule	Description
ARC01000 - Claim State Rule	If the claim is not closed, ClaimCenter skips this claim.
ARC03000 - Bulk Invoice Item State Rule	If the claim is linked to a bulk invoice item with a status of draft, notvalid, approved, checkpendingapproval, or awaitingsubmission, ClaimCenter skips this claim. The intent is to not force a user to retrieve the claim if there is an item that needs escalation. An item with status In Review or Rejected does not prevent ClaimCenter from archiving the claim. The claim can retain one of these statuses long after a bulk invoice item is escalated and cleared.
ARC04000 - Open Activities Rule	If the claim has open activities, ClaimCenter skips the claim. Note The Work Queue writer does not typically mark a claim with open activities for archiving. However, an activity can open on the claim between the time the worker queued the claim for archive and the time that the archive batch process actually processes the claim.
ARC05000 - Incomplete Review Rule	If a claim has a vendor review that is incomplete, ClaimCenter skips the claim.
ARC06000 - Unsynced Review Rule	If a claim has a vendor review that is not synchronized with ContactManager, ClaimCenter skips the claim.
ARC07000 - Transaction State Rule	If a claim has transactions that have yet to be escalated or acknowledged, ClaimCenter skips the claim.

You can add your own archive-related rules to this rule set or modify the existing sample rules to create rules that reflect your unique business conditions. For example, you can write rules to do the following:

- If a claim is sensitive or had restricted access control, do not archive it.
- If a claim meets a certain condition, such as having medical payments over some amount, do not archive it.
- If a claim has a claimant who has other open claims pending, do not archive it.

Archive Claim Purge Rules

Guidewire provides several rules that affect the purging of archived claims. They are:

Rule set	Rule	Description
Claim Closed	CCL03000 - Sample rule to set purge date	Sets <code>claim.PurgeDate</code> to seven years after <code>Claim.CloseDate</code> . Guidewire disables this rule in the base configuration.
Claim Reopened	CRO03000 - Sample rule to remove purge date	Clears (sets to null) the <code>claim.PurgeDate</code> field. Guidewire disables this rule in the base configuration.

If you close a claim, the first rule sets a purge date on the claim to seven years from the closing of the claim file. If you reopen a claim, the second rule clears the purge date.

These are sample rules only and the rules are not active in the base configuration. The two rule work as a pair. You must activate or deactivate the two rules in conjunction with each other.

Claim Eligible for Archive Rules

Guidewire provides several rules that affect the eligibility of a claim to be archived. They are:

Rule set	Rule	Description
Claim Closed	CCL04000 - Set archive eligibility date	Sets <code>claim.DateEligibleForArchive</code> to a date created by adding the value of configuration parameter <code>DaysClosedBeforeArchive</code> to the current system date.
Claim Reopened	CRO04000 - Clear archive eligibility date	Sets <code>claim.DateEligibleForArchive</code> to null.

If you close a claim, the first rule sets the date on which a claim is eligible for archiving. If you reopen a claim, the second rule clears that eligibility date. The two rule work as a pair. You must activate or deactivate the two rules in conjunction with each other.

Assignment Rule Set Category

Guidewire refers to certain business entities as *assignable* entities. Thus, it is possible to determine—either through the use of assignment business rules or through the use of Gosu assignment methods—the party responsible for that entity. In the ClaimCenter base configuration, Guidewire defines the following entities as assignable:

- Activities
- Claims
- Exposures
- Matters
- Service Requests

It is not possible to make an entity that is not assignable in the base configuration to be assignable. You can, however, extend an entity, and make the new entity assignable.

You use Gosu assignment methods to set an assigned group and user for an assignable entity. These assignment methods run either in the context of the Assignment engine (within the Assignment rule set) or simply within Gosu code (within a class or enhancement, for example).

For information on assignment and assignable entities in Guidewire ClaimCenter, see “Assignment in ClaimCenter” on page 93.

BulkInvoice Approval Rule Set Category

In Guidewire ClaimCenter, the approval process for a bulk invoice is much the same as the approval process for transaction sets. However, there is one critical difference. There are no authority limits for bulk invoices. The Bulk Invoice Approval rules alone control whether a bulk invoice requires approval.

The BulkInvoice Approval Rules and the BulkInvoice Approval Assignment Rules rule sets form a pair. These rules work together similarly to the rule sets in the Approval Routing and Transaction Approval rule set categories. ClaimCenter calls these rule sets sequentially.

See also

- “BulkInvoice Approval Rules” on page 45
- “BulkInvoice Approval Assignment Rules” on page 40
- “Approval Routing Rule Set Category” on page 38
- “Transaction Approval Rule Set Category” on page 73
- For information on bulk invoice web service APIs, validating bulk invoices, and bulk invoice processing performance, see “Bulk Invoice Integration” on page 391 in the *Integration Guide*.

BulkInvoice Approval Rules

ClaimCenter runs the BulkInvoice Approval rules for each bulk invoice to determine whether the bulk invoice requires approval. In the base configuration, ClaimCenter provides a sample approval rule for bulk invoices that always requires approval. Guidewire ships this rule disabled. Writing an approval rule for a bulk invoice is much like writing an approval rule for a transaction set. Some important things to keep in mind, however, are the following:

- For a typical `TransactionSet` approval rule, the rule condition requires verification of the subtype of the transaction set, such as `TransactionSet.subtype == "checkset"`. This verification is required because ClaimCenter passes all created transaction sets, regardless of subtype, to the same approval rules. In contrast, the bulk invoice approval rules have no need for subtype verification in their rule conditions. The only implicit entity that ClaimCenter ever passes to the bulk invoice approval rules is a bulk invoice.
- Bulk invoice approval rules have an implicit reference to the bulk invoice submitted for approval. You can access this reference in a rule condition or action block through the `BulkInvoice` entity. This behavior is different from the rules in the Transaction Approval rule set category, which have an implicit reference to the transaction set being approved. See “Transaction Approval Rule Set Category” on page 73.

In the following example, the rule requires that a human manually approve each bulk invoice. Rules in the Bulk Invoice Approval Assignment Rules determine who can actually approve bulk invoices. As noted, Guidewire disables these rule in the base configuration. See “BulkInvoice Approval Assignment Rules” on page 40.

Condition

```
return true
```

Action

```
bulkInvoice.requireApproval(displaykey.Rules.BulkInvoice.ForceApproval)
```

Automatic Approval of Bulk Invoices

If none of the approval rules determine that the bulk invoice requires approval, then ClaimCenter automatically and immediately approves the bulk invoice. ClaimCenter then does the following:

- Marks the bulk invoice as “Approved” and sets the approval date to the current day.
- Transitions the bulk invoice status to “Pending” upon bulk invoice item validation.
- Transitions all the bulk invoice items that have a status of “Draft” to “Approved” status.

Closed Rule Set Category

These rule sets fire any time that you close an activity, claim, exposure, or matter. They fire as immediately on closing one, not before. You can use these rules to send a notification or to generate additional activities, for example.

The following example rules illustrate how to use these rule sets.

- The first example rule closes the claim file from within the Exposure Closed Rules rule set. See “Exposure Closed Rules” on page 47.
- The second example rule closes a claim automatically if all its Exposures are in a status of Closed.

Condition

```
return !exists (exp in claim.Exposures where exp.State != "closed")
```

Action

```
exposure.Claim.close("completed", "Claim number " + Exposure.Claim.ClaimNumber  
+ " closed by ClaimCenter because last open exposure was closed.")
```

The second example, in the Claim Closed Rules rule set, works in conjunction with the previous example. The rule closes any open activities that are not available to closed claims. Therefore, if ClaimCenter closes a claim, the rule closes all open activities associated with the claim as well. See “Claim Closed Rules” on page 47.

Condition

```
return true
```

Action

```
for( act in claim.Activities) {  
    if (act.Status == "open" and act.ActivityPattern != null  
        and act.ActivityPattern.ClosedClaimAvlble == false) {  
        act.complete()  
        if (act.Description == null) { act.Description = "Closed by ClaimCenter." }  
        else { act.Description = act.Description + "Closed by ClaimCenter." }  
    }  
}
```

Note: In actual practice, Guidewire recommends that you make all String values into display keys.

Activity Closed Rules

ClaimCenter executes these rules immediately after an activity closes. Use these rules to create follow-up actions and activities. In the following example, the rule first verifies that an activity to salvage a vehicle is complete. If so, it then sets the vehicle recovery date to the current date.

Condition

```
return activity.ActivityPattern.Code == "recover_vehicle" and activity.Status == "complete"
```

Action

```
if (activity.Exposure <> null) {  
    activity.Exposure.VehicleIncident.DateVehicleRecovered = gw.api.util.DateUtil.currentTimeMillis()
```

Claim Closed Rules

These rules execute immediately after a claim closes. Use these rules to create follow-up actions and activities. Several of the rules in this rule set have special functionality:

Rule	Description
CCL01000 - Notify external systems	Notifies external systems that the claim is closed.
CCL02000 - Suspend AutoSync for Related Contacts	Helps facilitate the synchronization of contact information between ClaimCenter and ContactManager. See "Linking and Synchronizing Contacts" on page 191 in the <i>Contact Management Guide</i> for more information.
CCL03000 - Sample rule to set purge date	Sets a purge date on the claim that you close to seven years from the closing of the claim file. This rule is inactive by default. If you activate it, you must also activate the following rule. Claim Reopened Rules: CRO03000 - Sample rule to remove purge date. See "Archive Claim Purge Rules" on page 44.
CCL04000 - Set archive eligibility date	Sets <code>claim.DateEligibleForArchive</code> to a date created by adding the value of configuration parameter <code>DaysClosedBeforeArchive</code> to the current system date. This rule works in conjunction with Claim Reopened Rules: CRO04000 - Clear archive eligibility date. See "Claim Eligible for Archive Rules" on page 44.

In the following example, the rule closes all open activities associated with a claim if the claim closes.

Condition

```
return true
```

Action

```
for( act in claim.Activities) {
    if (act.Status=="open" and act.ActivityPattern != null
        and act.ActivityPattern.ClosedClaimAvlble == false) {
        act.complete()
        if (act.Description == null) {
            act.Description = "Closed by ClaimCenter."
        }
        else {
            act.Description = act.Description + " Closed by ClaimCenter."
        }
    }
}
```

Note: In actual practice, Guidewire recommends that you make all `String` values into display keys.

Exposure Closed Rules

These rules execute immediately after the rules in the Exposure Closed Validation Rules rule set run. Use these rules to take automated actions on the closure of the exposure. See "Exposure Closed Validation Rules" on page 81.

In the following example, the rule closes a claim any time that a user closes the last open exposure associated with the claim.

Condition

```
return not exists (exp in exposure.Claim.Exposures where exp.State != "closed" )
```

Action

```
exposure.Claim.close( "completed", "Claim closed by ClaimCenter as last open exposure closed." )
```

Matter Closed Rules

A matter is the set of data organized around a single lawsuit or potential lawsuit. This data includes information on the attorneys involved, the trial details, and the lawsuit details. These rules execute immediately after a matter closes. Use these rules to take automated actions on the closure of a matter.

Event Message Rule Set Category

IMPORTANT ClaimCenter runs the Event Message rules as part of the database bundle commit process. Use these rules only to create integration messages.

In the base configuration, there is a single rule set, Event Fired, in the Event Message rule category. The rules in this rule set:

- Perform event processing.
- Generate messages about events that have occurred.

ClaimCenter calls the Event Fired rules if an entity involved in a bundle commit triggers an event for which a message destination has registered interest. As part of the event processing:

- ClaimCenter runs the rules in the Event Fired rule set once for every event for which a message destination has registered interest.
- ClaimCenter runs the Event Fired rule set once for each destination that is listening for that particular event. Thus, if there are multiple events listening for an event, ClaimCenter can run the Event Fired rule set multiple times for each event.

Messaging Events

ClaimCenter automatically generates *standard* events for most top-level objects. In general, event generation occurs for any addition, modification, removal, or retirement of a top-level entity. ClaimCenter automatically generates the following events on the Activity object:

- `ActivityAdded`
- `ActivityChanged`
- `ActivityRemoved`

It is also possible to create a custom event on an entity by using the `addEvent` method. This method takes a single parameter, `eventName`, which is a `String` value that sets the name of the event.

```
entity.addEvent(eventName)
```

See also

- Base configuration events, see “List of Messaging Events in ClaimCenter” on page 320 in the *Integration Guide*.
- Custom events, see “Triggering Custom Event Names” on page 325 in the *Integration Guide*.

Message Destinations and Message Events

You use the Studio Messaging editor to link an event to a message destinations:

- A *message destination* is an external system to which it is possible to send messages.
- A *message event* is an abstract notification of a change in ClaimCenter that is of possible interest to an external system. For example, this change can be adding, changing, or removing a Guidewire entity.

Using the Studio Messaging editor, it is possible to associate (register) one or more events with a particular message definition. For example, in the base configuration, ClaimCenter associates the following events with the `metro` message destination (ID=67):

- MetroReportAdded
- MetroReportChanged

If you modify or add a MetroReport object, ClaimCenter generates the relevant event. Then, during a bundle commit of the MetroReport object, ClaimCenter notifies any Event Message rule that has registered an interest in one of these events that the event has occurred. You can then use this information to generate a message to send to an external system or to the system console for logging purposes, for example.

Message Destinations and Message Plugins

Each message destination encapsulates all the necessary behavior for an external system, but uses three different plugin interfaces to implement the destination. Each plugin handles different parts of what a destination does, as follows:

- The message request plugin handles message pre-processing.
- The message transport plugin handles message transport.
- The message reply plugin handles message replies.

You register new messaging plugins in Studio first in the Plugins editor. After you create a new implementation, Studio prompts you for a plugin interface name, 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, first in the Plugins editor, then in the Messaging editor.

IMPORTANT After the ClaimCenter application server starts, ClaimCenter initializes all message destinations. ClaimCenter saves a list of events for which each destination requested notifications. Because this process happens at system start-up, you must restart the ClaimCenter application if you change the list of events or destinations.

Generating Messages

Use method `createMessage` to create the message text, which can be either a simple text message or a more involved constructed message. The following code is an example of a simple text message that uses the Gosu in-line dynamic template functionality to construct the message. Gosu in-line dynamic templates combine static text with values from variables or other calculations that Gosu evaluates at run time. For example, the following `createMessage` method call creates a message that lists the event name and the entity that triggered this rule set.

```
messageContext.createMessage("${messageContext.EventName} - ${messageContext.Root}")
```

The following is an example of a constructed message that provides more detailed information if a claim changes.

```
var session = messageContext.SessionMarker
var claim = messageContext.Root as Claim

// Create a message for this claim.
var msg = messageContext.createMessage("Message for ClaimChanged")
msg.putEntityByName( "assigneduser", claim.AssignedUser )

// Add assigned user to session map, so it's available in the session.
session.addToTempMap( "assigneduser", claim.AssignedUser )

// Create a message for each exposure.
for (exposure in claim.Exposures) {
    msg.messageRoot = exposure
    var user : User = session.getFromTempMap( "assigneduser" ) as User
    msg.putEntityByName( "assigneduser", user )
    msg.putEntityByName( "claim", claim )
}
```

Warning about Modifying Data

Be extremely careful about modifying entity data in Event Fired rules and messaging plugin implementations. Use these rule to perform only the minimal data changes necessary for integration code. Entity changes in these

code locations do not cause the application to run or re-run validation or preupdate rules. Therefore, do not change fields that might require those rules to re-run. Change only the fields that are not modifiable from the user interface. For example, set custom data model extension flags used only by messaging code.

Guidewire does not support the following:

- Guidewire does not support (and, it is dangerous) to add or delete business entities from Event Fired rules or messaging plugins (even indirectly through other APIs).
- Guidewire does not support—even within the Event Message rule set—calling any message acknowledgment or skipping methods such as `reportAck`, `reportError`, or `skip`. Use those methods only within messaging plugins.
- Guidewire does not support creating messages outside the Event Message rule set.

See Also

- “Messaging Overview” on page 300 in the *Integration Guide*
- “List of Messaging Events in ClaimCenter” on page 320 in the *Integration Guide*
- “Using the Messaging Editor” on page 137 in the *Configuration Guide*
- “ContactManager Messaging Events by Entity” on page 280 in the *Contact Management Guide*

Detecting Object Changes

You might want to listen for a change to an object that does not automatically trigger an event if you update it. For example, suppose that you want to listen for a change to the `User` object, such as an update of the address. The `User` object itself does not contain an address. Instead, ClaimCenter stores addresses as an array on `UserContact`, which is an extension of `Person`, which is a subtype of `Contact`, which points to `User`. Therefore, updating an address does not directly touch the `User` object.

However, in an Event Message rule, you can listen for the `ContactChanged` event that ClaimCenter generates every time that the address changes. The following example illustrates this concept. It prints out a message to the system console any time that the address changes. In actual practice, you would use a different message destination.

Condition

```
//DestID 68 is the Console Message Logger  
return MessageContext.DestID == 68 and MessageContext.EventName == "ContactChanged"
```

Action

```
uses gw.api.util.ArrayUtil  
  
var message = messageContext.createMessage( "Event: " + messageContext.EventName )  
var contact = message.MessageRoot as Contact  
var fields = contact.PrimaryAddress.ChangedFields  
print( ArrayUtil.toString( fields ) )
```

Event Fired Rule Example

In the following example, the rule creates a `CashReceiptDocument` any time that a user creates a new Recovery transaction in ClaimCenter. See “Document Creation” on page 145 for information on how to construct a document in Guidewire ClaimCenter.

Condition

```
return messageContext.Root typeis Recovery and (messageContext.Root).Status == "submitting"
```

Rule Actions

```
uses java.util.HashMap  
  
//This code uses custom fields  
var rec = messageContext.Root as Recovery  
var values = new HashMap()
```

```

var template : gw.plugin.document.IDocumentTemplateDescriptor

values["InsuredName"] = rec.Claim.Insured
values["ReceivedFrom"] = rec.Payer
values["DateReceived"] = rec.CreateTime
values["AmtReceived"] = rec.Amount
values["PmntNo"] = null
values["CheckDate"] = rec.TransactionDate
values["CompletedBy"] = rec.UpdateUser
values["ClaimNo"] = rec.Claim.ClaimNumber
values["PolicyNo"] = rec.Claim.Policy.PolicyNumber
values["CostCat"] = rec.CostCategory
values["RecoveryCat"] = rec.RecoveryCategory
values["ReceiptNo"] = rec.CashReceiptNumberExt //Custom field

var doc : Document = new Document(rec.Claim)
doc.MimeType = "application/pdf"
doc.Claim = rec.Claim
doc.Name = "CashReceivedTicket"
doc.Exposure = rec.Exposure
doc.TypeExt = "Recovery" //Custom field
doc.SubTypeExt = "Other" //Custom field
doc.PrivilegedExt = "No" //Custom field
doc.ProcessMethodExt = "Sent_to_File" //Custom field
doc.Claimant = rec.Claim.claimant
doc.Status = "approved"
doc.DocUID = "ID-" + java.util.Calendar.getInstance().getTimeInMillis()

gw.document.DocumentProduction.createDocumentSynchronously(template, values, doc)

```

MetroReport Events

Note: For information on Metropolitan Police Reports and the MetroReport object, see “Metropolitan Reporting Bureau Integration” on page 487 in the *Integration Guide*. See also “Metropolitan Police Report Request Rules” on page 67.

ClaimCenter uses event rules to check the status of a Metropolitan Police Report request.

- If the status is `InsufficientData`, ClaimCenter does not send the report request to the Metropolitan Reporting Bureau. Instead, it creates an activity as described in “Metropolitan Police Report Request Rules” on page 67.
- If the status is `Sending Order`, ClaimCenter fires an event and sends a message to the Metropolitan Reporting Bureau requesting a report.

ClaimCenter defines two types of event messages for Metropolitan Police Report integration:

Send Order Message	ClaimCenter sends out the initial report request to Metropolitan Reporting Bureau with all the claim information and the type of the report it requests.
Send Inquiry Message	ClaimCenter sends an inquiry on a scheduled basis to Metropolitan Reporting Bureau for each outstanding report requested. The interval for sending the inquiry is set to hourly in the base configuration.

Vendor Notification Events

ClaimCenter uses event rules to track the state transitions of a service request. When an operation is performed on a `ServiceRequest`, events can be triggered and processed by this rule set, which then sends a corresponding message to the `ServiceRequestNotificationAPI` web service on the vendor portal.

The following entities generate messages, which are handled by `Vendor Notification` event rules:

- `ServiceRequest`
- `ServiceRequestInvoice`
- `ServiceRequestDocumentLink`
- `ServiceRequestMessage`

The `Vendor Notification` rule set processes each event and creates a corresponding `wsi.remote.gw.webservice.vendor.dto.Notification` to build a message. The message is processed by the

vendorNotificationTransport message transport plugin. This plugin parses the message by **Notification** type and sends it to the **ServiceRequestNotificationAPI** web service, which, in turn, notifies the vendor.

ClaimCenter defines three types of event notification types for **Vendor Notifications**:

NewMessageNotification	A new service request message has been created.
NewDocumentNotification	The service request has a new document link or attachment.
OperationNotification	The service request has undergone a status change. For example, the vendor has accepted the specified work (ServiceRequestOperation_specialistacceptedwork).

See also

- “List of Messaging Events in ClaimCenter” on page 320 in the *Integration Guide*.

Exception Rule Set Category

Use the Exception rule sets to specify an action to take if an activity or claim is overdue and is in the escalation state. Additionally, there are rule sets to handle user exceptions and group exceptions, neither of which has sample rules. You can use rule sets in the Exception category to send email, create an activity, or set a flag, among other actions.

ClaimCenter runs the exception rule sets at regularly scheduled intervals. For information on exception batch processes, see “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide*.

This topic includes:

- “Activity Escalation Rules” on page 52
- “Claim Exception Rules” on page 53
- “Group Exception Rules” on page 55
- “User Exception Rules” on page 55

Activity Escalation Rules

Note: ClaimCenter does not run Activity Escalation rules on activities or users who are external to ClaimCenter. For example, ClaimCenter ignores any activity assigned to a third-party claims adjuster who is external to ClaimCenter and flagged as such in ClaimCenter.

An activity has two dates associated with it:

Due date	The target date to complete this activity. If the activity is still open after this date, it becomes overdue.
Escalation date	The date at which an open and overdue activity becomes escalated and needs urgent attention.

In the base configuration, ClaimCenter runs the Activity Escalation scheduled process **activityesc** every 30 minutes. This batch process runs against all open activities in ClaimCenter to find activities that need to be escalated. It uses the following criteria:

- The activity has an escalation date that is non-null.
- The escalation date has passed.
- The activity has not already been escalated.

ClaimCenter marks each activity that meets the criteria as escalated. After the batch process runs, ClaimCenter runs the escalation rules for all the activities that have hit their escalation date. Use this rule set to specify what

actions to take if the activity becomes escalated. For example, you can use this rule set to reassign escalated activities.

The simplest approach to escalated activities is to add an activity on the claim for the appropriate supervisor to determine why the work is not yet complete. You can mark the claim as a flagged claim as well. You can also decide that there is no follow-up action needed for certain kinds of activities. For information on setting a claim flag, see “Claim Exception Rules” on page 53.

In the following example, the rule adds a description to an activity associated with a high-priority, overdue claim.

Condition

```
return Activity.Priority=="high"
```

Action

```
activity.Claim.setFlag(displaykey.Rules.Exception.ActivityEscalation.Priority.High(activity.Subject))
```

Claim Exception Rules

Note: For a discussion of Claim Exception rules that relate to fraud and special investigations, see “Detecting Claim Fraud” on page 129.

ClaimCenter monitors claims for unusual or potentially concerning conditions so that adjusters or supervisors can be alerted. Guidewire breaks this process into two pieces:

- Determining the list of claims to evaluate
- Consulting claim exception rules for each claim to find exceptions and take appropriate actions

For ClaimCenter to evaluate a claim, the claim must meet one of the following criteria:

- **Significant change** — The claim must have changed in some significant way since the last time ClaimCenter ran the exception checking process. Significant changes include editing the claim basics, adding or editing an exposure, and adding or changing a financial transaction. ClaimCenter does not consider adding notes or documents, for example, a significant change as the exception rules do not typically evaluate the content of a new note or document.
- **Idle for a length of time** — The claim is idle, aging without any significant change, for a certain length of time. The `IdleClaimThresholdDays` configuration parameter in the `config.xml` file defines the number of days after which ClaimCenter considers a non-modified claim as idle. In the base configuration, this parameter is set to seven days.

Checking for Recently Modified Claims

Guidewire defines *recently modified* as any claim for which the update time is newer than the last time the exception rules ran on the claim. The check for recently modified claims examines the following:

- The update time on the claim
- The update time on any exposure on the claim
- The update time on any transaction on the claim
- The update time on any claim contact or claim contact role on the claim
- The update time on any matter on the claim

Configuring Exception Schedules

The following table lists the batch processes that work with changed and idle claims. See “Batch Processes and Work Queues” on page 123 in the *System Administration Guide* for a discussion of ClaimCenter batch processes.

Batch process	Description	Default time
claimexception	<p>The Claim Exception batch process runs the claim exception rules on claims that have been modified significantly since the last time that ClaimCenter ran this process. The claims processed are:</p> <ul style="list-style-type: none"> • Claims that have been updated since the last time the batch process processed it • Claims with any of the following updates: <ul style="list-style-type: none"> • exposure • transaction • claim contact • claim contact role • matter • Claims that have never been processed by the batch process <p>For all these cases, the Claim Exception batch process creates a <code>ClaimException</code> row in the database for the claim if none currently exists. If the row exists, the batch process updates the <code>ExCheckTime</code> field of the <code>ClaimException</code> to the current time.</p>	Every day at 2:00 a.m.
idleclaimexception	<p>The Idle Claim Exception batch process runs the claim exception rules on claims that are idle. A claim is idle if the exception rules have not been run on it in the number of days specified by the <code>IdleClaimThresholdDays</code> configuration parameter. More specifically, the Idle Claim Exception batch process evaluates all claims that have a foreign key to an existing <code>ClaimException</code> row in the database. If <code>ExCheckTime</code> field of <code>ClaimException</code> row is older than Today minus <code>IdleClaimThresholdDays</code>, the batch process runs the claim exception rules on that claim.</p>	Every Sunday at 12:00 noon.

To run separate schedules for changed claims and idle aging claims, set the configuration parameter `SeparateIdleClaimExceptionMonitor` to `true` in the `config.xml` file. The following table lists the configuration parameters that specifically affect changed and idle claims. See “Application Configuration Parameters” on page 33 in the *Configuration Guide* for a discussion of ClaimCenter configuration parameters.

Configuration parameter	Description	Default value
<code>IdleClaimThresholdDays</code>	Number of days without significant change after which ClaimCenter considers a claim idle.	7
<code>SeparateIdleClaimExceptionMonitor</code>	If <code>true</code> , ClaimCenter runs separately scheduled processes for determining changed claims (<code>claimexception</code>) and idle aging claims (<code>idleclaimexception</code>). If <code>false</code> , ClaimCenter runs the <code>idleclaimexception</code> process also as it runs the <code>claimexception</code> process.	<code>true</code>

There are many kinds of exceptions for which you can write rules:

- Work that has not been completed within a certain number of days, but for which there is no activity-based escalation
 - For example, you can look for auto claims that are more than 30 days since being reported but still do not have a fault rating set.
- Unusual reserve changes or values
 - For example, you can look for claims with three or more reserve increases.
- Items of interest

You can decide to flag claims that have important new information. For example, if a claim notes the existence of a plaintiff attorney, you can flag the claim for potential litigation automatically.

One action to take in regard to escalated claims is to mark—*flag*—a claim as an exception. The following list describes `Claim` fields and one method that you can use with flagged claims. ClaimCenter displays the three fields in the New Loss screen for the claim.

Property	Description
<code>Flagged</code>	Sets the flagged status of a claim to a typecode defined in the <code>FlaggedType</code> typelist: <ul style="list-style-type: none"> • <code>isflagged</code> • <code>neverflagged</code> • <code>wasflagged</code> Used to determine a status of a claim.
<code>FlaggedDate</code>	Indicates the date on which a flag was set.
<code>FlaggedReason</code>	A text field that provides a reason the flag was set. Helps the adjuster understand the nature of the exception noticed.
<code>clearFlag</code>	A method that toggles the <code>Flagged</code> field and clears the <code>FlaggedReason</code> field. Use this method after the claim is no longer considered to have any exceptions.

As shown in “Activity Escalation Rules” on page 52, you can use the `setFlag` method to set the value of the flagged reason. For example:

```
Claim.setFlag( "Claim reported within 30 days of opening policy. Claim referred to SIU  
for investigation." )
```

As noted in “Taking Actions on More Than One Subitem” on page 35, avoid writing exception rules that cause ClaimCenter to report the same exceptions repeatedly.

Group Exception Rules

ClaimCenter runs the Group Exception rules on a scheduled basis to look for certain conditions on groups that might require further attention. You can also use these rules to define follow-up actions for each exception found. ClaimCenter identifies groups that have been changed or that have not been inspected for a certain period of time, and runs these rules on each group chosen.

This rule set is empty in the base configuration.

Note: In the base configuration, the ClaimCenter batch process Group Exception, code `groupexception`, runs the Group Exception rules every day at 4:00 a.m. on all groups in ClaimCenter.

User Exception Rules

ClaimCenter runs the User Exception rules on a scheduled basis to look for certain conditions on users that might require further attention. You can use these rules to define the follow-up actions for each exception found. ClaimCenter identifies users that have been changed or which have not been inspected for a certain period of time, and runs these rules on each user chosen.

This rule set is empty in the base configuration.

Note: In the base configuration, the ClaimCenter batch process User Exception, code `userexception`, runs the user exception rule sets on all users in within ClaimCenter every day at 3:00 a.m.

Initial Reserve Rule Set Category

ClaimCenter uses Initial Reserve rules to create initial reserves on new exposures. The Initial Reserve rule category executes immediately after the Postsetup rules and has a specific purpose: to automate the creation of

reserves for new exposures. If your enterprise does not automate reserve creation, you probably would not use this rule category.

Initial Reserve Rules

ClaimCenter uses these rules to determine the initial financial reserves to create for a new exposure. Each segment type has specific reserve requirements. For example, the amount of a reserve you establish for a moderate vehicle damage exposure is most likely less than the reserve you create for a serious bodily injury exposure. Whatever the amount of the reserve, the rule uses the `createInitialReserve` method to automate the creation of the reserve.

ClaimCenter does not associate a `TransactionSet` with any reserves added through these rules until it processes the new reserves. The most common approach for creating initial reserves is to write a rule or rule set for each exposure segment type.

In the following example, the rule automatically creates a reserve of \$300,000 if the exposure coverage is of type PIP Death. The rule calls an enhancement method on `Exposure`.

Note: You can modify this rule to create a reserve of any amount for any kind of exposure.

Condition

```
return exposure.CoverageSubType.DisplayName == "PIP Death"
```

Action

```
print(actions.getRuleSet() + ": " + actions.getRule())
exposure.createInitialReserve("claimcost", "death", 30000)
```

Loaded Rule Set Category

Loaded rules execute only in concert with the FNOL import interface. Only rules that relate to imported claims go into the Loaded category. Put rules that apply to the following in the Presetup rule set category:

- Claims that ClaimCenter creates
- Claims that must execute before Segmentation

Claim Loaded Rules

Claim Loaded rules execute any time that an FNOL import loads a new claim. ClaimCenter executes Loaded rules after it finishes importing, or *loading*, a claim from an external FNOL source. The purpose of this rule category is to make any changes necessary to convert initial claim data from an external format into a suitable structure for ClaimCenter. You can also create Loaded rules to record the import of claims. You can use them also, for example, to assign review of the new FNOL to a user prior to running the full automated setup process.

You can create whatever rules are appropriate to handle the import of new claims:

- Sanity checks
- Data aggregations and transformations
- Data and naming reconciliations
- Claim import logging

The most common application of the Loaded rules is to ensure that ClaimCenter makes a record of claims that it imports from an external FNOL application. Making a record typically consists of logging the transaction. The following rule illustrates this application:

Condition

```
return Claim.State <> null and Claim.State <> "draft"
```

Action

```
gw.api.util.Logger.logDebug("In Claim No. :" + claim.ClaimNumber + " - In "
    + actions.getRuleSet() + " - In Rule " + actions.getRule() )
actions.exit()
```

This example shows the basic format of a rule that causes ClaimCenter to log the import of a claim from an external application. You can define the value of the string in `gw.api.util.Logger.logDebug` as necessary.

Exposure Loaded Rules

ClaimCenter uses these rules only if ClaimCenter loads a new exposure separately from its parent claim and then adds it to that previously loaded parent claim. In this case, you might require a different set of actions for processing the new exposure.

IMPORTANT Guidewire intends that the Exposure Loaded rule set trigger after an FNOL import into ClaimCenter for the `Claim` entity only. The application does not execute the Exposure Loaded rule set. Nor does ClaimCenter execute any rule that you add to this rule set. You must manually trigger this rule set. Guidewire intends to remove this rule set in a future release.

In the following example, the rule logs a message to the system console if the import loads an exposure linked to an existing claim. As with the other Exposure Loaded rules, you must manually trigger this rule.

Condition

```
return (Exposure.State==null or Exposure.State=="draft") and Exposure.Claim.State=="open"
```

Action

```
print ("This is an A N N O U N C E M E N T   M E S S A G E -- I am running the "
    + actions.getRuleSet() + " Rule set.")

gw.api.util.Logger.logDebug( " " )
gw.api.util.Logger.logDebug( "Loaded Exposure Loaded default rules " + actions.getRule() )
```

Postsetup Rule Set Category

Postsetup rules run automatically any time that ClaimCenter adds a new claim. ClaimCenter runs these rules after the Assignment and Workplan rules complete their execution. The postsetup rules run immediately before the rule sets in the Initial Reserve rule set category or the Preupdate rule set category.

Postsetup rules can perform a wide variety of actions, depending on your business requirements. Postsetup rules often depend on the results of the rules executed during setup or other tasks. For example, you can design these rules to trigger based on specific conditions that the setup rules create.

Note: ClaimCenter runs the Postsetup rules as part of the “save and setup” operation on new claims and exposures. See “Claim and Exposure Setup” on page 59 for more information.

See also

- “Initial Reserve Rule Set Category” on page 55
- “Preupdate Rule Set Category” on page 63

Activity Postsetup Rules

The Activity Postsetup rules fire just prior to completing the setup process and saving any changes to activities. They run after the assignment rules have run.

In the following example, the rule sets the claim status to indicate that ClaimCenter referred the claim to a Special Investigation Unit for possible fraud.

Condition

```
return activity.ActivityPattern.Code == "fraud_review"
```

Action

```
activity.Claim.SIUStatus = "Under_Investigation"
```

Note: See “Detecting Claim Fraud” on page 129 for a discussion of SIU (Special Investigation Unit) rules.

Claim Postsetup Rules

The Claim Postsetup rules fire just prior to completing the set up process and saving any changes to claims. In the base configuration, there are no rules active in this rule set.

In the following example, the rule creates a new activity for FNOL.

Condition

```
return claim.isValid( "newloss", true )
```

Action

```
var strClaimNumber = claim.DuplicateClaimNumbers

if ( strClaimNumber != null ) {
    /* Create new claim activity if there are possibly duplicate claims listed. */
    var description = "Check that claim is not a duplicate of " + strClaimNumber
    claim.createActivity( null, ActivityPattern.finder.getActivityPatternByCode( "fnol" ),
        "Check that duplicate claim does not exist", description, null, null, null, null )
} else {
    /* Create new claim activity */
    claim.createActivityFromPattern( null,
        ActivityPattern.finder.getActivityPatternByCode( "fnol" ) )
}
```

Exposure Postsetup Rules

The Exposure Postsetup rules fire just prior to completing the setup process and saving any changes to exposures. Guidewire does not provide any sample rules in the base ClaimCenter configuration.

Matter Postsetup Rules

The Matter Postsetup rules fire just prior to completing the setup process and saving any changes to matters. In the following example, the rule sets up an activity for the claim owner if the matter has a bodily injury exposure in its array of exposures. The activity is to initiate a legal review of a bodily injury claim 10 days before a scheduled mediation meeting with the insured.

Condition

```
return true
```

Action

```
var exp = matter.Exposures*.Exposure.firstWhere(
    \ elt -> elt.ExposureType == ExposureType.TC_BODILYINJURYDAMAGE)
if (exp !=null) {
    var act = matter.Claim.newActivity(
        ActivityPattern.finder.getActivityPatternByCode( "legal_review" ), exp)
    act.TargetDate = gw.api.util.DateUtil.addDays( act.Matter.MediationDate, -10 )
    act.assignToClaimOwner()
}
```

Transaction Postsetup Rules

The Transaction Postsetup rules fire under the following conditions:

- Immediately after the Exposure Closed rule set, after closing an exposure that has a transaction associated with it. See “Exposure Closed Rules” on page 47.

- Immediately after the Transaction Approval Rules rule set except if you are in the process of creating a reserve. See “Transaction Approval Rules” on page 74.

These rules also fire under the following conditions:

- ClaimCenter approves a TransactionSet
- ClaimCenter voids, stops, or escalates a check
- ClaimCenter records a payment
- ClaimCenter voids a recovery
- ClaimCenter closes an exposure

Presetup Rule Set Category

ClaimCenter runs the Presetup rules automatically any time that ClaimCenter adds a new claim, exposure, matter, or transaction. Presetup rules do not apply to modifications made to existing claims. You use Presetup rules to perform actions on new claims before claim or exposure setup. *Setup* typically encompasses running rules in the Segmentation, Assignment, and Workplan rule set categories. If you need to modify a new claim before beginning setup, the Presetup rule category provides a place for the rules that perform the actions.

See also

- “Segmentation Rule Set Category” on page 72
- “Assignment Rule Set Category” on page 44
- “Workplan Rule Set Category” on page 84

Claim and Exposure Setup

Guidewire uses the term *setup* as a generic term for invoking multiple rules (Presetup, Segmentation, Assignment, Workplan, and Postsetup, for example). The term also encompasses ClaimCenter performing some initialization work as it first creates a Claim or Exposure as an open item. ClaimCenter typically performs this kind of setup under the following circumstances:

- At the end of the **New Claim** wizard, ClaimCenter sets up the new claim and all its exposures.
- After creating a new exposure through ClaimCenter, ClaimCenter sets up the new exposure after you click **Update**.
- Any time that you add a new claim through the `addFNOL` SOAP call, ClaimCenter performs a setup before it commits data to the database.

Setup Operations Performed for the `addFNOL` Web Service Method Call

ClaimCenter performs the following setup tasks any time that you add a new claim through the `addFNOL` web service method call. This method is defined in the class `gw.webservice.cc.claim.ICollectionAPI`:

1. Adds the FNOL claim.
2. Invokes the Loaded rule set category rules on the FNOL claim. See “Loaded Rule Set Category” on page 56.
3. Creates an `import` history event.
4. Performs the sequence of actions classified as *save-and-setup*, setting the claim status to `open` at the end of the process. See the next topic, “Save and Setup Process” on page 60.
5. Commits the claim. This commit includes all exposures, activities, and other linked objects created during the *save-and-setup* process.
6. At commit time, runs the rules in the Preupdate, Validation, and Event Message rule set categories, validating the claim at the `loadsolve` level.

See also

- “Adding First Notice of Loss from External Systems” on page 152 in the *Integration Guide*
- “Preupdate Rule Set Category” on page 63
- “Validation Rule Set Category” on page 74
- “Event Message Rule Set Category” on page 48

Save and Setup Process

Guidewire ClaimCenter performs the following steps for a save-and-setup operation on an FNOL claim. ClaimCenter:

1. Creates a new claim number (if one does not exist).
2. Creates a claim snapshot.
3. Runs the rules in the Presetup rule set category on the claim and each exposure. See “Presetup Rule Set Category” on page 59.
4. Runs the rules in the Segmentation rule set category on each exposure, then the claim. See “Segmentation Rule Set Category” on page 72.
5. Runs the rules in the Assignment rule set category on the claim, then each exposure. See “Assignment Rule Set Category” on page 44.
6. Runs the rules in the Workplan rule set category on the claim, then each exposure. This process generates the workplan. See “Workplan Rule Set Category” on page 84.
7. Runs the rules in the Postsetup rule set category on the claim and each exposure. See “Postsetup Rule Set Category” on page 57.
8. Updates history timestamps so that client-added history events can be correctly stamped.
9. Sets the status of the claim and each exposure to “open”.
10. Ensures that each exposure has a valid claim order field set.
11. Sets initial reserves.

At the end of the setup process for an FNOL claim, ClaimCenter returns the public ID of the newly created claim.

Setup Operations Performed for Exposures

ClaimCenter performs the following *save-and-setup* pre-processing operations on an exposure after the `Exposure.saveAndSetup` method creates it:

1. Sets the claim order on the exposure.
2. Runs rules in the Presetup rule set category on the exposure. See “Presetup Rule Set Category” on page 59.
3. Runs the segmentation engine on the exposure.
4. Runs the strategy engine on the exposure.
5. Runs the assignment engine on the exposure.
6. Runs the workplan engine on the exposure.
7. Runs rules in the Postsetup rule set category on the exposure. See “Postsetup Rule Set Category” on page 57.
8. Sets the exposure's status to open.
9. Create initial reserves.

10. Commits the exposures bundle, which contains the exposure, activities, and other objects creating during the call.

Transaction Set Presetup Rules

The Transaction Set Presetup rules execute prior to transaction setup. *Setup* for transactions encompasses submitting a `TransactionSet` for approval and updating the T-accounts. In the base configuration, ClaimCenter does not provide any rules for this rule set.

The Transaction Set Presetup rule set is useful primarily for a limited purpose. Use Transaction Set Presetup rules to make the same modifications to transactions created in the ClaimCenter user interface and to transactions created in rules or outside rules in Gosu. When you use the rules for this purpose, you can modify transaction values that affect financial calculations. These values include the transaction `Amount`, the `Eroding` state, the `Exposure`, `CostType`, or `CostCategory`, or the `ScheduledSendDate` on a payment's check.

If you create or modify transactions as described in the following cases, you do not need to use presetup rules on them:

- If you need to modify transactions created only in the ClaimCenter user interface, rather than using rules, you can modify them in the `beforeCommit` attribute of the PCF page. Use the financial APIs for transactions in your Gosu code.
- If you need to modify transactions created only in rules or Gosu, use the financial APIs to create the transactions correctly in the first place instead of modifying them later.

See “Financial APIs for Transactions” on page 61.

Financial APIs for Transactions

There are classes and methods that provide extensive control over payments and transaction line items as you set up checks, and these APIs also update the T-accounts when used correctly.

If you want to create, modify, or remove transactions programmatically, Guidewire recommends that you use the Gosu methods described in the following topics.

- “Creating Checks and Payments by Using CheckCreator” on page 637 in the *Configuration Guide*
- “Creating Reserve Transactions Directly” on page 637 in the *Configuration Guide*
- “Creating Recovery Transactions” on page 639 in the *Configuration Guide*
- “Creating Recovery Reserve Transactions” on page 640 in the *Configuration Guide*

IMPORTANT Guidewire does not provide a mechanism for editing previously saved checks or recoveries exclusively with rules or Gosu. The editing of a check must be initiated in the user interface.

When ClaimCenter Executes Transaction Set Presetup Rules

ClaimCenter executes the Transaction Set Presetup rule set under the following circumstances:

- If you click the `Finish` at the end of the `Check` wizard.
- If you click `Save` or `Update` on the `New Reserves`, `New Recovery Reserves`, or `New Recoveries` screen.
- If you create a claim in the `New Claim` wizard in `Auto - First and Final` mode, which creates a check.

ClaimCenter also runs the Transaction Set Presetup rules:

- During `CheckCreator.prepareForCommit`.
- During `BulkInvoiceItem` processing of its placeholder check.

ClaimCenter does not execute these presetup rules if you use the following methods on `Claim`, `Exposure`, or `ReserveLine` to create a transaction:

- `createRecovery`

- `setAvailableReserves`
- `setOpenRecoveryReserves`

Using Transaction Set Presetup Rules

By using rules in the Transaction Set Presetup rule set, you can modify transactions and checks before they go through the setup process. You can:

- Add to or remove a payment from a check.
- Add to or remove a line item from a transaction.
- Modify the amount of a line item.
- Add or remove a Recovery, a Reserve, or RecoveryReserve from a TransactionSet of the appropriate subtype.

Note: Reserves and recovery reserves only have one line item.

The Transaction Set Presetup rules provide support for modifying the amounts of single and recurring checks from rules just before the checks are set up. They also provide support for adding new payment line items through the user interface just before setting up the transaction. For example, after a payment is submitted, but before it is set up and then saved, you might want to add a new payment to a check. You might want to add the payment both from the user interface and programmatically to the check while it is in Awaiting Submission status. You might want to add the payment, for example:

- To adjust a future payment
- To add a line item for a certain type of tax computed in rules

Performing your business logic in the Transaction Set Presetup rule set does cause ClaimCenter to update the T-accounts properly.

Guidewire provides the following methods that you can use in this rule set:

```
Transaction.addNewLineItem(Amount, Comments, LineCategory)
RecoveryReserveSet.newRecoveryReserve(Exposure, CostType, CostCategory)
RecoverySet.newRecovery()
ReserveSet.newReserve(Exposure, CostType, CostCategory)
Check.addNewPayment(Exposure, CostType, CostCategory)
```

If you want to change a line item, use the standard `getLineItems` method to retrieve the line item. Then, in this rule set, call any of the standard setters on the line item—for example, `lineitem.setAmount`—so that ClaimCenter correctly computes the T-accounts.

IMPORTANT

- The methods described previously are in the financial APIs described at “Financial APIs for Transactions” on page 61. If you use financial API methods in a presetup rule, do not call `prepareForCommit` in the rule. The transactions will be prepared for commit as part of the setup process.
- Do not call the constructors of financial transaction entities, such as `new Payment` or `new RecoveryReserve`, in presetup rules. Instead, use the financial APIs to create these entities outside rules.

Check Scheduled Send Date Only Modifiable in Special Situations

The check property for the schedule send date, `ScheduledSendDate`, is modifiable in the Transaction Presetup rules or from Gosu called from the application user interface (PCF) code. The date determines the place to include the check amount, such as:

- In Future Payments (tomorrow or later)
- In Awaiting Submission (today), which is included in Total Payments and similar financial calculations.

If the date is inappropriately updated, ClaimCenter throws an exception with the following message:

Check contains a payment whose LifeCycleState is inconsistent with the Check's Scheduled Send Date.

Workers' Compensation Presetup Example

One common application of presetup rules is to create exposures for a new Workers' Compensation claim. Workers' Compensation claims typically have only three possible exposure assignments:

- An employee is injured on the job and loses some work time. The employee is entitled to reimbursement for both medical costs and lost wages, and the employer's workers' compensation insurance covers the loss.
This type of claim has two exposures, one for Medical Expenses and another for Lost Wages.
- An employee is injured on the job, but misses no work time. The employee is entitled to reimbursement only for medical costs, and the employer's workers' compensation insurance covers the loss.
This type of claim has only a single Medical Expenses exposure.
- An employee is injured and the employer's insurance does not cover the loss. The claimant is therefore obliged to seek compensation in court.
This type of claim has only a single Employer Liability exposure.

There is an existing Claim Presetup rule that creates the standard workers' compensation exposures that you see when setting up a Workers' Compensation claim in ClaimCenter. This rule is CPR01000 - WCDefaultExposures. Its condition and action are:

Condition

```
return claim.LossType == "WC"
```

Action

```
claim.createRelevantWorkCompExposure()
```

The condition determines that the rule applies to any Workers' Compensation claim. The action calls a library function, `createRelevantWorkCompExposure`, which is located in an enhancement to the claim entity in FNOL.gsx. The function contains the instructions for evaluating and assigning exposures:

```
function createRelevantWorkCompExposures() {  
    //open WC Medical Exp if this is not an Incident Only report OR  
    //if Medical Report is specifically specified  
    //  
    if (this.MedicalReport || !this.IncidentReport){  
        createAndSetUpExposure("WCWorkersCompCov", "WCWorkersCompMED")  
    }  
    //open WC Time Loss Exp if Indemnity/TimeLossReport field is checked  
    if (this.TimeLossReport || this.DeathReport){  
        createAndSetUpExposure("WCWorkersCompCov", "WCWorkersCompWAGES")  
    }  
  
    //open WC Employer's Liability Exp  
    if (this.EmployerLiability) {  
        createAndSetUpExposure("WCEmpLiabCov", "WCEmpLiabCov")  
    }  
  
    linkWCExposuresToCoverages()  
    ensureNoGapsInExposureNumbers()  
}
```

See also

- “Workers’ Compensation Line of Business” on page 183 in the *Application Guide*

Preupdate Rule Set Category

Use the Preupdate rule sets to perform domain logic or validation that must be performed before the entity in question is committed. Only a change to an entity marked as validatable in its definition file can trigger a preupdate rule.

Typically, these rules execute before the rules in the Validation rule set category. See “Validation in ClaimCenter” on page 121 for a discussion of how the Validation rule set category and Preupdate rules work together in performing object validation.

In a similar fashion to the rules in the Postsetup rule set category, the Preupdate rules can perform a wide variety of actions as needed. One application of the preupdate rules is to specify an activity based on a condition. For example, if a vehicle covered in a claim has been declared a total loss, a rule can create an activity to inspect the car for its salvage potential.

See also

- “Validation Rule Set Category” on page 74
- “Postsetup Rule Set Category” on page 57

Triggering Preupdate Rules

The following entities all trigger preupdate rules in the ClaimCenter base configuration:

- | | | |
|------------|----------|------------------|
| • Activity | • Group | • TransactionSet |
| • Claim | • Matter | • User |
| • Contact | • Policy | |
| • Exposure | • Region | |

For an entity to trigger the preupdate rules, it must implement the `Validatable` delegate. This requirement applies both to an entity in the base ClaimCenter configuration and to a custom entity that you create.

All entities that implement the `Validatable` delegate trigger the validation rules as well. The set of entities that trigger preupdate rules is the same as the set of entities that trigger the validation rules.

ClaimCenter runs the preupdate and validation rules in response to the following events:

- An instance of a validatable entity is created, modified, or retired.
- A subobject of a validatable object is created, modified, or retired and the validatable object is connected to the subobject through a `triggersValidation="true"` link.

IMPORTANT In running the Preupdate rule set, ClaimCenter first computes the set of objects on which to run the preupdate rules. It then runs the rules on this set of objects. If a preupdate rule then modifies an entity, the preupdate rules for that entity do not fire unless the schedule for preupdate rules already includes this entity. In other words, changing an object in a preupdate rule does not then cause the preupdate rules to run on that object as well.

The same behavior also applies to entities newly created in a preupdate rule. For example, if you create a new activity in a preupdate rule, ClaimCenter does not then run the Activity preupdate rules on that activity.

See Also

- For information on how to create an entity that implements the `Validatable` delegate, see “Delegate Data Objects” on page 167 in the *Configuration Guide*.
- For information on implementing preupdate handling in a plugin, see “Preupdate Handler Plugin” on page 267 in the *Integration Guide*.
- For information on custom entities and validation rules, see “Triggering Validation Rules” on page 76.

Preupdate Rules and Custom Entities

It is possible to create preupdate rules for custom entities, those entities that you create yourself and that are not part of the base configuration of ClaimCenter.

For an entity that you create to trigger a preupdate rule:

- The entity must implement the `Validatable` delegate. For an extension entity to trigger a preupdate rule, the entity must implement the following code:

```
<implementsEntity name="Validatable"/>
```

- The preupdate and validation rule sets that you want the custom entity to trigger must conform to the following naming conventions:

- Put your validation rules in the Validation rule set category and name the rule set `entitynameValidationRules`.

- Put your preupdate rules in the Preupdate rule set category and name the rule set `entitynamePreupdate`.

The preupdate rule must exist in a preupdate rule set that reflects the extension entities name and ends with `Preupdate`. In other words, you must create a rule set category with the same name as the extension entity and add the word `Preupdate` to it as a suffix. You then put your rules in this rule set.

For example, if you create an extension entity named `NewEntityExt`, then you need to create a rule set to hold your preupdate rules and name it `NewEntityExtPreupdate`.

For information creating new rule sets, see “Working with Rules” on page 25. Specifically, see the topic “To create a new rule set” on page 26.

Activity Preupdate Rules

Use the Activity Preupdate rules to modify activities and related entities. In the base configuration, any time that you reassign a claim, ClaimCenter reassigns the activities associated with the claim to the new claim owner. However, it is possible that you might want a different behavior.

In the following example, the rule prevents administrative activities attached to a claim from being reassigned if the claim is reassigned. Administrative activities are those performed by an underwriting technician.

Condition

```
return activity.Status == "open" and gw.api.util.StringUtil.startsWith(
    activity.ActivityPattern.Code, "TECH")
```

Action

```
var userKey = activity.getOriginalValue("AssignedUserId") as Key
if( userKey != null &&
    (activity.Claim.isFieldChanged("AssignedUser"))) {
    var originalUser = activity.Bundle.loadBean(userKey) as User
    activity.AssignedUser = originalUser
}
```

The rule behaves as follows:

- If `userkey != null` evaluates to `true`, then this activity is already assigned, meaning that it is not a new activity, so continue.
- If `activity.Claim.isFieldChanged("AssignedUser")` evaluates to `true`, then the claim owner has changed, so continue.
- If both conditions evaluate to `true`, then assign this administrative activity back to the original owner.

Claim Preupdate Rules

Use the Claim Preupdate rules to modify claims and related entities. The Claim Preupdate rules contain a number of rules useful in working with claims fraud. In the following example, the rule creates a review activity if the claim's loss date changes.

Uses

```
uses gw.api.database.Query
uses gw.api.database.Relop
```

Condition

```
return claim.State=="open"
```

Action

```
var dbClaim = claim.New
? null
: Query.make(Claim).compare(
    Claim#ID, Relop.Equals, claim.ID).select().AtMostOneRow

if (dbClaim.LossDate != null and dbClaim.LossDate <> claim.LossDate ) {

    // Create activity that will be assigned to team leader.
    claim.createActivity( null,
        ActivityPattern.finder.getActivityPatternByCode("loss_date_change"),
        "The loss date was changed on this claim",
        "Please see claim note documenting that the loss date was changed",
        null, null, null, null )

    // Create variables needed for claim note.
    var changedby = claim.UpdateUser.Contact.DisplayName
    changedby = (changedby == null)
    ? "Problem - call support"
    : changedby + ". "
    claim.addNote( null, "general",
        "The loss date was changed on this claim by " + changedby
        + "It was changed from " + dbClaim.LossDate + " to " + claim.LossDate )
}
```

This topic includes

- “Catastrophe-Related Rules” on page 66
- “Metropolitan Police Report Request Rules” on page 67
- “Special Investigation Unit (SIU) Rules” on page 67

Catastrophe-Related Rules

In the base configuration, Guidewire provides several rules related to catastrophe categorization in the Claim Preupdate rule set. These rules include:

Rule	Description
CPU09000 - Related to Catastrophe	Checks whether a claim matches a named catastrophe that is already in the system. If there is a match, the rule creates an activity for the claim owner to look into it.
CPU13000 - Catastrophe History	Determines if there is a change to the claim Catastrophe field. If there is, the rule updates History and marks any open <i>Review for Catastrophe</i> activity complete.
CPU19000 - Geocode Catastrophe Claims	Determines if the claim Catastrophe field is set (not null) and if the claim's loss location has never been geocoded. If so, the rule schedules the claim's loss location for geocoding. This rule takes effect only if geocoding is enabled. See “Using the Geocoding Feature” on page 19 in the <i>System Administration Guide</i> .

ClaimCenter uses these rules in conjunction with the catastrophe profile that an administrative user creates through the **Administration** → **Catastrophes** screen.

See Also

- “Catastrophes and Disasters” on page 155 in the *Application Guide*
- “Working with Catastrophe Bulk Associations” on page 535 in the *Configuration Guide*

Metropolitan Police Report Request Rules

The Metropolitan Reporting Bureau at www.metroreporting.com is a nationwide service in the United States for obtaining accident and incident reports. The Claim Preupdate rule set has one rule that works with this service, CPU08000 - Metro Report Request. This rule starts an internal process to determine the type of each report requested by a claim, one at a time. The rule verifies that the claim is in draft state and then calls `libraries.Metro.validateAndStartReportFlows(claim)`. This method verifies that the claim contains the data required to request each of the requested reports.

- If there are any missing fields, ClaimCenter creates an activity called Metropolitan Report Request Failed and assigns it to the user that created the MetroReport request. The `Description` text area of the activity contains information on the type of report that was requested. It also contains any data that must be supplied in order to successfully make the request. For each `MetroReport` request for which the required data is not defined on the claim, ClaimCenter marks the status as `insufficientData`.
- If the required data is specified and the claim update completes successfully, ClaimCenter changes the `MetroReport` status to `validated`. After the report status becomes valid, ClaimCenter changes the Metropolitan report status to `Sending Order`.

IMPORTANT ClaimCenter integrates with Metropolitan Reporting Bureau in the base configuration. However, you must have an account with Metropolitan Reporting Bureau for report requests to work. For information on Metropolitan Police Reports and the `MetroReport` object, see “Metropolitan Reporting Bureau Integration” on page 487 in the *Integration Guide*.

Special Investigation Unit (SIU) Rules

The Claim Preupdate SIU rules and the Claim Exception SIU rules work together to assist in determining claim fraud. These rules identify certain characteristics of a claim that increase the suspicion that fraud has occurred and assign points to each of these characteristics. For a discussion on how these two rule sets work in conjunction with each other, see “Detecting Claim Fraud” on page 129. See also “Claim Exception Rules” on page 53.

Contact Preupdate Rules

Use the Contact Preupdate rules to modify contacts and related entities. For example, the rule COP03000 - Add Default Tags adds the minimum number of required tags to each contact.

Condition

```
return true
```

Action

```
contact.ensureMinimumTags()
```

Note: For a discussion of contact tags, see “Contact Tag Overview” on page 177 in the *Contact Management Guide*.

Exposure Preupdate Rules

Use the Exposure Preupdate rules to modify exposures and related entities. If a rule encounters an exception, the bounding database transaction is rolled back, which prevents the update of the exposure.

The following example shows the code for the rule EPU02000 - Salvage. This rule verifies that the exposure is a vehicle incident with a total loss and that the exposure does not already have an activity for salvage review. It then creates two activities and sets the date the salvage activity was assigned if it is not already set. The activities it creates are to review the claim exposure for potential salvage value and for possible vehicle recovery.

Condition

```
return exposure.VehicleIncident.TotalLoss == true and
not exists (
    act in exposure.Claim.Activities where(
        act.Exposure != null
        and exposure.VehicleIncident == act.Exposure.VehicleIncident
        and act.ActivityPattern ==
            ActivityPattern.finder.getActivityPatternByCode("salvage_vehicle")))
```

Action

```
exposure.Claim.createActivityFromPattern(
    exposure, ActivityPattern.finder.getActivityPatternByCode("salvage_vehicle"))

exposure.Claim.createActivityFromPattern(
    exposure, ActivityPattern.finder.getActivityPatternByCode("recover_vehicle"))

// Set date
if (exposure.VehicleIncident.DateSalvageAssigned == null) {
    exposure.VehicleIncident.DateSalvageAssigned = gw.api.util.DateUtil.currentDate()
}
```

Group Preupdate Rules

Use Group Preupdate rules to modify groups and related entities. In the base configuration, there are no rules in this rule set.

ClaimCenter runs the Group Preupdate rules upon a save of the Group entity. Any exceptions found cause the bounding database transaction to roll back, thus effectively vetoing the update.

After running these rules, Claim Center runs the Group Validation rules. See “Group Validation Rules” on page 82.

Matter Preupdate Rules

Use the Matter Preupdate rules to modify matters and related entities. This rule set is empty in the base configuration.

In the following example, the rule creates a new activity to review the claim file 30 days before the date set to litigate the claim in court.

Condition

```
return not exists( Activity in matter.Claim.Activities
    where (Activity.ActivityPattern !=
        ActivityPattern.finder.getActivityPatternByCode( "trial_review" ) ) )
```

Action

```
var act = matter.Claim.newActivity(
    ActivityPattern.finder.getActivityPatternByCode( "trial_review" ), null )
act.TargetDate = gw.api.util.DateUtil.addDays( act.Matter.TrialDate, -30 )
act.assignToClaimOwner()
```

Policy Preupdate Rules

Use the Policy Preupdate rules to modify policies and related entities. In the base configuration, this rule set has one rule that is turned off by default, PPU01000 - Assign Default Coverage. The rule is disabled because its function, linking workers’ compensation exposures to policy coverages, is now handled in the New Claim wizard. See the comments for this rule for more information.

In the following example, the rule standardizes the policy number using standard Gosu library functions. The nested functions trim any spaces in the number and turn all alpha characters to lower case.

Condition

```
return true
```

Action

```
policy.PolicyNumber =  
    gw.api.util.StringUtil.toLowerCase(  
        gw.api.util.StringUtil.trim( policy.PolicyNumber ) )
```

RIAgreementPreupdate Rules

ClaimCenter runs the RIAgreement Preupdate rule set any time that a user adds or edits a reinsurance agreement. Use this rule set to modify RIAgreement and related entities. Exceptions cause the bounding database transaction to roll back, effectively vetoing the update.

IMPORTANT Modifying financial values on an RIAgreement object that are used by the IReinsuranceTransactionPlugin to set up RITransaction objects does not affect the calculations in the current commit. The reason is that the IReinsuranceTransactionPlugin has already run and created the RITransaction objects. Instead, modify RIAgreement objects between the invocation of the IReinsurancePlugin and the invocation of the IReinsuranceTransactionPlugin. IReinsurancePlugin retrieves the agreements from an external system. IReinsuranceTransactionPlugin creates RITransaction objects based on the regular financials values and the RIAgreement objects.

This rule set is empty by default in the base configuration.

Note: See “Reinsurance Management Concepts” on page 595 in the *Application Guide* for more information on reinsurance.

RITransactionSetPreupdate Rules

ClaimCenter runs the RITransactionSet Preupdate rule set any time that a user adds or edits a reinsurance transaction. Exceptions cause the bounding database transaction to roll back, effectively vetoing the update.

You can use rules in this rule set to update extension fields. However, see the note that follows for the limitations on using this rule set.

IMPORTANT The ClaimAmount, ReportingAmount, ReserveLine, and RIAgreement fields of an RITransaction entity must not be modified in the RITransactionSet Preupdate rules. Changes made in this rule set to RITransaction entities do not affect automatic reinsurance calculations based on reinsurance agreements or T-accounts for RITransaction entities. Reinsurance calculations are performed by the IReinsurancePlugin and the IReinsuranceTransactionPlugin, which are run in the IPreUpdateHandler plugin. These calculations are complete before Preupdate rules run. To perform preupdate work on RITransactionSet entities, add your code to the class `gw.plugin.preupdate.impl.CCPreupdateHandlerImpl`. This class is registered in the plugin registry `IPreupdateHandler.gwp`. See “Preupdate Handler Plugin” on page 267 in the *Integration Guide*.

This rule set is empty by default in the base configuration.

Note: See “Reinsurance Management Concepts” on page 595 in the *Application Guide* for more information on reinsurance.

Transaction Set Preupdate Rules

ClaimCenter runs the Transaction Set preupdate rules any time that a user adds or edits a transaction.

IMPORTANT

- Do not modify fields that affect financial calculations, such as the transaction Amount, the Eroding state, the Exposure, CostType, or CostCategory, or the ScheduledSendDate on a payment's check

- You can modify other fields that do not affect financials calculations, including extension fields
- Guidewire recommends that you not create new transactions in preupdate rules. Preupdate rules do not run on transactions created this way. Instead, create them in a class that implements the `IPreUpdateHandler` plugin interface, which runs immediately before preupdate rules. In the base implementation, ClaimCenter provides a class that you can use for this purpose, `CCPreupdateHandlerImpl`, which is registered in the plugin registry `IPreupdateHandler.gwp`. See “[Preupdate Handler Plugin](#)” on page 267 in the *Integration Guide*.

An example of a rule in this rule set is the transaction set preupdate rule TPU08000 - Clean Up Payment Method Related Check Fields. This rule is useful for cleaning up data if the payment method changes. It nulls out unwanted fields on a check if the user changes the payment method from check to EFT (Electronic Check Transfer).

This rule calls the `gw.entity.GWCheckEnhancement.removeUnusedPaymentMethodRelatedFields` method on the `Check` object. In the base configuration, this method nulls out fields related to a specific payment method.

Payment method	Method nulls out...
Check	<code>BankAccountNumber</code> <code>BankName</code> <code>BankAccountType</code> <code>BankRoutingNumber</code>
Electronic funds transfer	<code>MailToAddress</code> <code>MailTo</code> <code>DeliveryMethod</code>

User Preupdate Rules

You can create User Preupdate rules that modify users and related entities prior to user validation. ClaimCenter runs the User Preupdate rules on a save of the `User` entity. Any exceptions found in a rule in this rule set cause the bounding database transaction to roll back, thus effectively vetoing the update.

ClaimCenter runs the User Validation rule set after this one. See “[User Validation Rules](#)” on page 84.

In the base configuration, there is one rule in this rule set, UPU00005 - User Workload Resync On Status Change. The rule first verifies that weighted workload is enabled and that the user’s vacation status or active status has changed. It then logs the changes that were made. See the rule in Guidewire Studio for more information.

See also

- “[Weighted Workload](#)” on page 207 in the *Application Guide*
- “[Configuring Weighted Workload Assignment](#)” on page 517 in the *Configuration Guide*

Reopened Rule Set Category

The Reopened rule sets provide a means of taking automatic action any time that a `Claim`, `Exposure`, or `Matter` is reopened.

Claim Reopened Rules

The Claim Reopened rule set executes immediately after a Claim has been reopened in order to take automated actions on reopening. Several of the rules in this rule set have special functionality:

Rule	Description
<i>CCCA0001 - Allow AutoSync for Related Contacts</i>	Helps facilitate the synchronization of contact information between ClaimCenter and ContactManager. See “Linking and Synchronizing Contacts” on page 191 in the <i>Contact Management Guide</i> for more information.
<i>CCCA0002 - Sample rule to remove purge date</i>	Removes a purge date on the claim that you reopen. This rule is inactive by default. If you activate it, you must also activate the following rule: <ul style="list-style-type: none"> • Claim Closed → CCL03000 - Sample rule to set purge date See “Claim Closed Rules” on page 47.

In the following example, the rule reopens any exposure that involves some kind of medical treatment if the reopened claim loss type is Workers’ Compensation. It also uses the `try... catch...` exception handling construction to deal with potential exceptions.

Condition

```
return Claim.LossType == "WC"
```

Action

```
try { for (exp in Claim.Exposures index i)
    if (Claim.Exposures[i].MedicalTreatment != null) {
      Claim.Exposures[i].reopen("newinfo", "Reopened Exposure - Claim Reopened")
    }
} catch (e) {
  print("Caught exception: " + e)
}
```

Exposure Reopened Rules

The Exposure Reopened rule set executes immediately after an Exposure has been reopened in order to take automated actions on reopening. In the following example, the rule sets the reserve amount (using a script parameter) for the logged-in user if a Workers’ Compensation exposure is reopened. It also uses the `try... catch...` exception handling construction to deal with potential exceptions. (For details on creating and using script parameters, see “Script Parameters” on page 105 in the *Configuration Guide*.)

Condition

```
return Exposure.Type == "WCIInjuryDamage"
```

Action

```
try {
  exposure.setAvailableReserves( "claimcost", "medical", ScriptParameters.InitialReserve_ReopenClaim,
    User.util.CurrentUser )
} catch (e) {
  print("Caught exception: " + e)
}
```

Matter Reopened Rules

The Matter Reopened rule set executes immediately after a Matter has been reopened in order to take automated actions on the reopening. In the following example, the rule creates an event that can be used to notify an external application that a matter has been reopened.

Condition

```
true
```

Action

```
matter.Claim.addEvent( "matter_reopened" )
```

Segmentation Rule Set Category

The goal of the segmentation rules is to set the Segment and Strategy properties on new claims and exposures. ClaimCenter runs segmentation rules just prior to running assignment rules any time that you select automated assignment for a new claim or exposure. Typically, values set for the segmentation and strategy for a (new) claim or exposure are then used in determining the assignment of the claim or exposure. You set the segment value using the following property:

```
Claim.Segment = segment_value
```

To arrive at a decision on the segment of an exposure, ClaimCenter examines the following:

- The fields on the exposure (for example, for an injury: severity, body part injured, nature of injury, first as opposed to third-party claimant)
- And possibly, the fields on the claim (for example, the cause of loss, the time between loss date and the time that the claim was reported).

It is easier to make decisions about the segmentation of the claim as a whole once each exposure has been categorized. For example, an auto claim could be categorized as complex if there are any third-party injury exposures. For this reason, the default ClaimCenter workflow first segments each exposure and then segments the claim.

In the following example, ClaimCenter segments the claim first by the claim line and then by the segment type on the exposure. The claim is then categorized into the segment *auto_high*. Segment values are defined within the **ClaimSegment** typelist and are available for selection from within Studio.

Condition

```
exists( Exposure in claim.Exposures where Exposure.Segment == "injury_high" )
```

Action

```
claim.Segment = "auto_high"
```

Strategy Property

You set the strategy value after setting the segment by using the following property:

```
Exposure.Strategy = strategy_value
```

There is no reason to repeat all the logic performed during segmentation while setting the strategy value. In the simplest form, these rules simply look at the claim's segment and map a segment to a strategy. For example:

Condition

```
segment is A or B
```

Action

```
set the strategy to "Fast Track"
```

There are several reasons for defining the handling strategy for a claim (or exposure) separately from the segment:

- There can be many different categories of claims for which it is interesting to report statistics separately. However, there is usually a much smaller number of handling approaches (for example, fast track as opposed

to field adjusted). The segment can capture the more detailed understanding of categorization, while the strategy can govern how the work is assigned and how much effort is put into adjusting the claim.

- You can experiment with different handling strategies for the same segment of claim. For example, suppose that you decided to employ a new approach for a certain segment of claims within one office. You would want to be able to analyze the results later to determine if average outcomes were better following the new strategy.

Claim Segmentation Rules

You can use the Claim Segmentation rules to categorize each claim as a whole based on complexity, severity of damage, and other attributes. The results set the segment field of a Claim. You can also set the strategy in this rule set. In the following example, the rule becomes active if the claim loss type is “AUTO” and the exposure segment is set to “high”. If triggered, the rule sets the claim segment to “high” and the claim strategy to “normal”.

Condition

```
return claim.LossType=="AUTO" and exists(  
    Exposure in claim.Exposures where Exposure.Segment == "auto_high")
```

Action

```
claim.Segment="auto_high"  
claim.Strategy ="auto_normal"
```

Exposure Segmentation Rules

You can use the Exposure Segmentation rules to categorize each exposure based on complexity, severity of damage and other attributes. The results set the segment field of an exposure. You can also set the strategy in this rule set. In the following example, the rule sets the Exposure segment and strategy if the Exposure type is *VehicleDamage* and the segment has not already been set.

Condition

```
return exposure.ExposureType == "VehicleDamage" and exposure.Segment == null
```

Action

```
Exposure.Segment = "auto_low"  
Exposure.Strategy = "auto_fast"
```

Transaction Approval Rule Set Category

Transaction approval rules ensure that a user has authorization to submit certain financial transactions. A transaction set contains one or more transactions that are submitted as a group for approval. If a user attempts to save a transaction set, ClaimCenter can ensure that the transaction be marked as requiring approval.

ClaimCenter calls this rule set to handle transaction approvals of all kinds, not just those that involve authority limits (meaning, not just reserves or payments). Guidewire recommends that you create rules in this rule set that are used only to determine whether a transaction requires approval or not.

The Approval Routing and Transaction Approval rule set categories form a pair. ClaimCenter calls these rule sets sequentially.

- ClaimCenter calls rules in the Transaction Approval rule set category any time that you submit any kind of financial transaction.
- ClaimCenter calls rules in the Approval Routing rule set category if the transaction approval rules mark the transaction as requiring approval. It also calls this rule set if the transaction authority limit fails such that the transaction needs approval. See “Approval Routing Rule Set Category” on page 38.

You must ensure that if you approve the activity that you approve the transaction, also. If a transaction requires approval, ClaimCenter uses the Approval Routing rules to determine who must give the approval.

- Use the `requireApproval` method to require approval.
- Use `TransactionSet.requestingUser` to determine the person on whose behalf ClaimCenter runs the rules. For example, suppose that Andy Applegate initiates the reserve, and ClaimCenter routes the transaction to Sue Smith for approval. After Sue approves the transaction, she becomes the `requestingUser` for the next person in the approval chain.

IMPORTANT Often, the Transaction Approval rules iterate through all the transactions on the `TransactionSet` entity. Do not use these rules to set a claim to a new value within the course of working with a transaction.

Transaction Approval Rules

The Transaction Approval rules determine the approval requirements for financial transactions. ClaimCenter runs these rules for each transaction to determine if the transaction requires approval. In a typical `TransactionSet` approval rule, the rule condition requires a check for the subtype of the transaction set, such as `TransactionSet.subtype == "checkset"`. This subtype check is required because all created transaction sets—regardless of subtype—are passed to the same approval rules.

In the following example, the rule requires special approval from finance if the payee has an IRS lien, as has previously been entered into ClaimCenter.

Condition

```
return transactionSet.Subtype=="checkset"
```

Action

```
for(pye in (transactionSet as CheckSet).PrimaryCheck.Payees) {
    var cntr : Number = 0
    while( cntr < IRSlienIDs.length ) {
        if( pye.ClaimContact.Contact.TaxID == IRSlienIDs[cntr] ) {
            transactionSet.requireApproval("Reason Code:00001 : Payee " +
                pye.ClaimContact.Contact.DisplayName + " is in the list of Payees with IRS Liens. Please
                resolve the lien, then approve this payment.")
        }
        cntr = cntr + 1
    }
}
```

Validation Rule Set Category

Note: ClaimCenter runs the validation rules at many different points in the ClaimCenter workflow. They are also run any time that ClaimCenter commits a claim (or other validatable business entity) to the database.

ClaimCenter uses validation levels to ensure that certain conditions are met before critical stages can be reached. This could be, for example, as you create a new claim or make a payment. You can view these levels as maturity levels—as a claim gains data, it achieves new levels of maturity. A ClaimCenter object can always achieve a new, higher, level of maturity, but it can never go backwards to a lower validation level.

One way to think of this is to think of validation rules as hurdles on a running track. A change to a claim object sends the object down the track. As long as the object can get past the hurdle, the maturity level is achieved. What allows an object to pass the hurdle is your rule. As you write a validation rule, you set the maturity level to which the rule applies. ClaimCenter then enforces rule errors only if set at or below the level the object has achieved.

Validation Levels

Guidewire delivers ClaimCenter with five levels of validation (listed in increasing order of maturity). Certain of these validation levels are internal to ClaimCenter and you cannot change them. Others are editable, and you can access them through Guidewire Studio using the `ValidationLevel` typelist.

ClaimCenter uses the `priority` attribute of the typekeys to order the validation levels. As in golf, low priority numbers are better. This is sometimes confusing. Thus, in the base configuration `newloss` with a priority of 300 is more restrictive than `loadsave` with a priority of 400. The following list describes the base configuration validation levels.

Level	Name	Priority
<code>loadsave</code>	<code>Load and Save</code>	400
<code>newloss</code>	New Loss	300
<code>ISO</code>	Solid for ISO	250
<code>external</code>	Sent to External System	200
<code>payment</code>	Ability to Pay	100

In the base configuration, ClaimCenter defines three immutable validation levels. You cannot delete these validation levels as the internal ClaimCenter system uses them for its purposes. (In the `ValidationLevel` typelist, you see these three validation levels in shaded gray fields, indicating that they are not editable.)

Level	Priority	Description
<code>loadsave</code>	400	The loosest possible validation level against which ClaimCenter validates claims. <i>All data must pass “loadsave” to be saved to the database.</i>
<code>newloss</code>	300	The lowest level at which a claim can exit the New Claim wizard.
<code>payment</code>	100	The lowest level at which it is possible to pay a claim or an exposure. For example, in the base configuration, ClaimCenter does not permit you to enter the New Check wizard if the claim is not in Ability to Pay.

ClaimCenter provides other validation levels in the base configuration. However, as Guidewire defines these levels in the `ValidationLevel` typelist, it is possible to modify, remove, or even add to them using the Studio typelist editor.

Level	Priority	Description
<code>iso</code>	250	The level at which the claim passes ISO validation.
<code>external</code>	200	The level at which a claim must pass before it can be submitted to an external system. Validation occurs any time that an external submission or decline is triggered by a workflow button.

Note: It is possible to add additional validation levels. See “Adding New Validation Levels” on page 76 for details.

Validation Warnings

ClaimCenter can generate validation *warnings* that serve to provide information only. In this case, the issue does not affect the claim or exposure, but perhaps affects other business practices. For example, if an automobile claim is missing weather information, a warning can remind the user to enter the information.

For a discussion of the `rejectField` method, see “Correcting Validation Errors” on page 78.

Condition

```
return Claim.LossType=="auto" and Claim.Weather==null
```

Action

```
Claim.rejectField("Weather", null, null, "newloss", "Remember to enter weather information  
for an auto claim!")
```

Validation Errors

ClaimCenter can generate validation *errors* that prevent the claim or exposure from being saved in ClaimCenter. For example, the user cannot save a claim in which the loss date is in the future.

For a discussion of the `rejectField` method, see “Correcting Validation Errors” on page 78.

Condition

```
return Claim.LossDate > gw.api.util.DateUtil.currentDate()
```

Action

```
Claim.rejectField("LossDate", "newloss", "Please provide a loss date that is not a future date.",  
null, null)
```

In this example, ClaimCenter marks the `Claim.LossDate` field as invalid and prompts the user to fix it.

Adding New Validation Levels

It is possible to add additional validation levels to meet your business needs.

System Validation Levels

You can override the *name* and the *priority* of the three system validations levels (`loadsolve`, `newloss`, and `payment`). For example, you can change the name of the `payment` validation level from *Ability to pay* to *Ability to pay expenses*. You can also modify the priority so this level is easier to reach. You *cannot*, however, delete the three system validation levels.

Non-system Validation Levels

You can modify or delete any non-system validation level at will.

Custom Validation Levels

You can add new validation levels. If you do so, then you must provide a linkage to the actions you would like to control through the new validation level and to the validation level itself. You do this by calling the following method on a `Claim` or `Exposure` object (or, indeed, any `Validatable` object):

```
isValid(validationLevel, includeExposures)
```

For example:

```
Claim.isValid("newLoss", false)
```

This method determines if the claim or exposure is valid for the action you are attempting to perform. This is required for checking actions triggered in preupdate rules or from within ClaimCenter. It is important to understand that ClaimCenter runs the Validation rules after the preupdate rules, so the `Claim.ValidationLevel` property reflects the previous validation level. However, this is not true for rules in the Event Message rule set category, which ClaimCenter runs after validation. See “Event Message Rule Set Category” on page 48.

Triggering Validation Rules

For an entity to trigger the validation rules, it must implement the `Validatable` delegate. This is true for an entity in the base ClaimCenter configuration or for a custom entity that you create.

ClaimCenter runs the validation (and preupdate) rules:

- Whenever an instance of a validatable entity is created, modified, or retired.
- Whenever a *subentity* of a validatable entity is created, modified, or retired *and* the validatable entity is connected to the subentity through a `triggersValidation="true"` link.

However, for any entity that you create to trigger the validation rules, you must also create a validation rule set and name it using the name of your new entity. Thus, to create a rule set that ClaimCenter recognizes as triggering validation, do the following:

- Put your validation rules in the *Validation* rule set category.
- Name the rule set `<entity name>ValidationRules`.

For example, if you create an entity named `MyEntityExt`, put validation rules for that entity in the following rule set that you create in the *Validation* rule set category:

```
MyEntityExtValidationRules
```

See Also

- For information on preupdate rules and custom entities, see “Preupdate Rules and Custom Entities” on page 65.
- For information on how to create an entity that implements the `Validatable` delegate, see “Delegate Data Objects” on page 167 in the *Configuration Guide*.

Validation and Preupdate Rules

ClaimCenter runs preupdate and validation rules every time that it commits data to the database. (This is known as a database bundle commit.) Validation rules execute after ClaimCenter runs all preupdate callbacks and preupdate rules. Execution of the validation rules forms the last step before ClaimCenter actually writes data to the database.

It is possible for the preupdate rules to modify objects during execution, requiring changes to the database. However, ClaimCenter does not permit you to modify objects during validation rule execution that require changes to the database. To allow this would make it impossible to ever completely validate data. (Data validation would be a ever-shifting target.)

See Also

- For more information on how the preupdate and validation rules work together, see “Validation in ClaimCenter” on page 121.
- For information on how to create custom entities that trigger validation (and preupdate) rules, see “Preupdate Rules and Custom Entities” on page 65.

Determining the Validity of an Object

ClaimCenter provides a `validate` method that you can call on a number of different business objects to determine if that object is valid. For example, the `validate` method on `Claim` executes the Claim Validation Rules on the supplied claim. See “Claim Validation Rules” on page 81. It then returns a validation object containing any errors. If the validation was successful, the method returns an empty validation object. Also, if `validateExposures` is `true`, then the method validates the claim and all the exposures on the claim. Otherwise, it validates the claim only.

```
Claim.validate(validateExposures)
```

It is important to understand that this works on a single object at a time, and that it returns a validation object that you must then handle. For this reason, Guidewire does not recommend that you use this for bulk operations. Instead, for example, if you wish to process a large number of converted claims by escalating them to whatever level they can achieve, use the following:

```
IClaimAPI.bulkValidate(loadCommandID)
```

The `bulkValidate` method does not perform any validation. Instead, it starts a batch process that creates a separate work item to validate each claim. The batch process runs unattended and does not return a result. The method `loadCommandID` parameter is an integer value that identifies the conversion batch process that imported the claims. The `loadCommandID` value is available through the `TableImportResult` object returned from a table import operation. For example:

```
ITableImportAPI.integrityCheckStagingTableContentsAndLoadSourceTables()
```

Again, to determine if a claim (and its exposures) are valid, call the following method: on a `Claim` or `Exposure` object (or, indeed, any validatable object):

```
isValid(validationLevel, includeExposures)
```

Note: As always, for more information on this and other ClaimCenter methods, see the *Gosu API Reference* available from the Studio Help menu.

Correcting Validation Errors

Use the `reject` method—in one of its many forms—to prevent continued processing of the object, and to inform the user of the problem and how to correct it. The following table describes the various forms of the `reject` method. These methods are available on `ABContact`, `Claim`, `Exposure`, `Matter`, `Policy`, and `TransactionSet` business entities. (To determine which objects you can use within a rule, right-click within the rule and select Complete Code to view a list of valid objects.)

IMPORTANT Guidewire ClaimCenter currently supports the following `reject` methods only, even though additional methods can show in Studio. Specifically, ClaimCenter does not support `reject` methods that take a `flowtStepId` argument. ClaimCenter ignores this argument if you attempt to set it.

Method form	Description
<code>reject</code>	Indicates a problem and provides an error message, but does not point the user to a specific field.
<code>rejectField</code>	Indicates a problem with a particular field (<code>relatedObject</code>), provides an error message, and directs the user to the correct field to fix.
<code>rejectSubField</code>	Similar to <code>rejectField</code> , but also indicates that the problem is with a field on a subobject (for example, a form).

The method signature can take one of the following forms:

```
reject( errorLevel, strErrorReason, warningLevel, strWarningReason)
rejectField( strRelativeFieldPath, errorLevel, strErrorReason, warningLevel, strWarningReason )
rejectSubField( relatedObject, strRelativeFieldPath, errorLevel, strErrorReason,
               warningLevel, strWarningReason )
```

The following table lists the `reject` method parameters and describes their use. Identified issues or problems are either errors (blocking) or warnings (non-blocking). Note that you can indicate failure as both an error and a warning simultaneously. However, if the failure is both an error and a warning, use different error and warning levels for the failure.

Method parameter	Description
<code>errorLevel</code>	Corresponding level effected by the validation error.
<code>relatedObject</code>	Object related to the root entity that is causing the error.
<code>strErrorReason</code>	Message to show to the user indicating the reason for the error.
<code>strRelativeFieldPath</code>	Relative path from the root entity to the field that failed validation. Using a relative path (as opposed to a full path) sets a link to the problem field from within the message shown to the user. It also identifies the invalid field on-screen.
<code>strWarningReason</code>	Message to show to the user indicating the reason for the warning.
<code>warningLevel</code>	Corresponding level effected by the validation warning.

Validation Rules and Entity Types

As you define a validation rule, you do so in a rule set that is declared for a specific entity type. For example, this would be for `Claim`, `User`, or one of the other business entities. Although it is possible to access other business entities within that rule (other than the declared type), *only* call the `reject` method on the declared entity in the rule set. Studio does not detect violations of this guideline. During validation, it is possible that some traversal of the objects to be validated can reset that entity's previous result. If this is before the entity's own rules are executed, then ClaimCenter would lose this particular reject.

Error Text Strings

You can enter the error text string (`strErrorReason`) directly into `rejectField`, or you can use one of the following methods to insert the error text.

getMessage

The `getMessage` method returns an instance of `UserMessage`, which represents all the information needed to render a user message. Guidewire recommends that you also call the `localize` method to handle any localization issues. In the following example, the Gosu code checks roles at the exposure level and generates an error message for each invalid role.

```
/* Check for invalid role configurations at the exposure level and report each individually
   on the Contacts array - if they exist
*/
var failures = exposure.validateRoles()

for ( failure in failures ) {
    exposure.rejectField( "Contacts", "newloss", failure.getMessage().localize(), null, null )
}
```

displaykey Class

You can also use the `displaykey` class to return the value of the display key, which you can then use in the error text string. For example:

```
displaykey.Java.Admin.User.DuplicateRoleError
returns
User has duplicate roles
```

This also works with display keys that require a parameter or parameters. For example, `display.properties` defines the following display key with placeholder `{0}`:

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

For example, if you use the following displaykey code (`GroupName` has been retrieved already):

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

then, it returns the following:

```
Cannot delete user because they are supervisor of the following groups: AutoClaim1
```

Activity Validation Rules

Use to provide validation for Activity objects. For example, in the following rule:

- If you have an activity of type Claim Acceptance with a status of Complete
- And, the associated workers' comp claim has a Compensable status of either `null` or Pending

Then, do the following:

- Do not close the activity
- Display the following message:

Prior to completing the activity, {0}, please change the Compensability status to either Accepted or Denied

Condition

```
return activity.ActivityPattern.Code == "claim_acceptance"
and (activity.Claim.ClaimWorkComp.Compensable == null
     or activity.Claim.ClaimWorkComp.Compensable == "pending")
and activity.Status == "complete"
```

Action

```
var myMessage = displaykey.Rules.Validation.Activity.Compensable.Status(activity.Subject)
activity.reject("external", myMessage, null, null)
```

Claim Closed Validation Rules

The Claim Closed Validation rules execute any time that a claim is closed and after claim validation and preupdate rules execute, but before ClaimCenter commits data to the database:

- The Claim Closed Validation rules execute before ClaimCenter inserts a new or updated entity into the database.
- The Claim Closed Validation rules execute before ClaimCenter executes the Claim Closed rules.
- The Claim Closed Validation rules do not execute if validation rules deem the entity invalid.

Use these rules to raise validation errors or warnings if ClaimCenter could not close the claim for some reason. In the following example, the rule prevents claim closure if the claim has open activities associated with it.

Condition

```
return exists( activity in claim.Activities where activity.Status == "open" and
              activity.ActivityPattern != null )
```

Action

```
claim.reject("newloss", displaykey.Rules.Validation.CloseExposure.CloseAllOpenActivities, null, null)
```

In the base configuration, the display key contains the following string:

This claim has open activities. To close the claim, you must first complete or skip all open activities. Please complete or skip these activities before closing the claim.

Claim Reopened Validation Rules

The Claim Reopened Validation rules execute any time that a claim is reopened and after claim validation rules execute, but before ClaimCenter commits the data commits to the database:

- The Claim Reopened Validation rules execute before ClaimCenter inserts a new entity or updated entity into the database.
- The Claim Reopened Validation rules execute before ClaimCenter executes the Claim Reopened rules.
- The Claim Reopened Validation rules do not execute if validation rules deem the entity invalid.

Use to raise validation errors or warnings if ClaimCenter cannot reopen a claim for some reason. This rule set is empty in the base ClaimCenter configuration.

In the following example, the rule logs the fact that the claim has been reopened.

Condition

```
return true
```

Action

```
gw.api.util.Logger.logInfo( "Log Rule Info - " + actions.getRuleSet() + " Rule Set - "
                           + actions.getRule() + " Rule")
```

Claim Validation Rules

Use the claim validation rules to ensure that claims data stored in ClaimCenter is sufficient and valid for downstream processing. In the following example, the rule verifies that the chosen policy type is valid for the kind of loss being reported

Condition

```
return ( claim.LossType=="WC" ) and ( claim.Policy.PolicyType != "CommercialPackage" )
```

Action

```
claim.reject( "newloss", "Invalid policy type for this type of loss", null, null )
```

Contact Validation Rules

Use the contact validation rules to ensure that contacts data stored is sufficient and valid for downstream processing. In the base configuration, this rule set is empty.

Exposure Closed Validation Rules

The Exposure Closed Validation rules execute any time that an exposure is closed and after exposure validation rules execute, but before ClaimCenter commits the data to the database:

- The Exposure Closed Validation rules execute before ClaimCenter inserts a new entity or updated entity into the database.
- The Exposure Closed Validation rules execute before ClaimCenter executes the Exposure Closed rules.
- The Exposure Closed Validation rules do not execute if validation rules deem the entity invalid.

Use to raise validation errors or warnings if ClaimCenter must not close an exposure for some reason. In the following example, the rule prevents closure of an exposure if the exposure has open activities.

Condition

```
return exists(activity in exposure.Claim.Activities where (activity.Exposure == exposure  
and activity.Status == "open"  
and activity.ActivityPattern != null  
and not activity.ActivityPattern.ClosedClaimAvlble)
```

Action

```
exposure.reject("newloss", displaykey.Rules.Validation.Exposure.CompleteOpenActivities, null, null)
```

Exposure Reopened Validation Rules

The Exposure Reopened Validation rules execute any time that an exposure is reopened and after exposure validation rules execute, but before ClaimCenter commits the data to the database:

- The Exposure Reopened Validation rules execute before ClaimCenter inserts a new entity or updated entity into the database.
- The Exposure Reopened Validation rules execute before ClaimCenter executes the Exposure Reopened rules.
- The Exposure Reopened Validation rules do not execute if validation rules deem the entity invalid.

Use to raise validation errors or warnings if ClaimCenter cannot reopen an exposure for some reason. In the following example, the rule does not permit reopening an exposure if the associated claim is closed.

Condition

```
return exposure.Claim.State<>"open"
```

Action

```
exposure.reject("newloss", displaykey.Rules.Validation.ReopenExposure.OpenClaimRequired, null, null)
```

In the base configuration, the display key contains the following string:

You can reopen an exposure only for an open claim.

Exposure Validation Rules

Use the Exposure Validation rules to ensure that exposure data stored in ClaimCenter is sufficient and valid for downstream processing.

Group Validation Rules

ClaimCenter runs the Group Validation (and Group Preupdate) rules upon a save of the Group entity. Any exceptions found during validation cause the bounding database transaction to roll back, thus effectively vetoing the update. In the base configuration, this rule set is empty.

Guidewire recommends that you use the *LoadSave* validation level for rules pertaining to validation for Users and Groups.

Matter Closed Validation Rules

The Matter Closed Validation rules execute any time that a matter is closed and after matter validation rules execute, but before ClaimCenter commits the data to the database:

- The Matter Closed Validation rules execute before ClaimCenter inserts a new or updated entity into the database.
- The Matter Closed Validation rules execute before ClaimCenter executes the Matter Closed rules.
- The Matter Closed Validation rules do not execute if validation rules deem the entity invalid.

Use to raise validation errors or warnings if ClaimCenter cannot close a matter for some reason.

Matter Reopened Validation Rules

The Matter Reopened Validation rules execute any time that a matter is reopened and after matter validation rules execute, but before ClaimCenter commits the data to the database:

- The Matter Reopened Validation rules execute before ClaimCenter inserts a new or updated entity into the database.
- The Matter Reopened Validation rules execute before ClaimCenter executes the Matter Reopened rules.
- The Matter Reopened Validation rules do not execute if validation rules deem the entity invalid.

Use to raise validation errors or warnings if ClaimCenter cannot reopen a matter for some reason.

Matter Validation Rules

Use to ensure that matter data stored in ClaimCenter is sufficient and valid for downstream processing.

ClaimCenter does not enforce validation levels on Matter objects, meaning the Matter Validation rule set will always fire, regardless of the validation level.

In the base configuration, this rule set is empty.

Policy Validation Rules

Use to ensure that policy data stored in ClaimCenter is sufficient and valid for downstream processing.

ClaimCenter does not enforce validation levels on Policy objects, meaning the Policy Validation Rule Set always fire, regardless of the validation level.

In the following example, the rule raises a warning if the endorsement effective date is earlier than the policy effective date. It also raises a warning if the endorsement expiration date is later than the policy expiration date.

Condition

```
return exists(end in policy.Endorsements where (end.EffectiveDate < policy.EffectiveDate) or
          (end.ExpirationDate > policy.ExpirationDate) )
```

Action

```
policy.reject( null, null, "loadsav", "You are adding an endorsement whose date range is not
entirely within the policy date range." )
```

Note: As always, Guidewire recommends that you use display keys, rather than fixed strings for error or warning messages.

TBD:

Service Request Validation Rule Set

Use to ensure that service request data is valid for downstream processing. ClaimCenter does not enforce validation levels on ServiceRequest objects. Errors will cause validation to fail, regardless of the validation level.

The following two validations rules are included in the base configuration. You can expand or edit these rules, as needed.

Check Invoice Currency

This rule raises a warning if the currency of the service request invoice does not match the currency of the associated check.

Condition

```
return true
```

Action

```
for(sri in serviceRequest.Invoices) {
    if(!sri.HasConsistentCheckCurrency) {
        serviceRequest.rejectField("Currency", "loadsav",
            displaykey.Rules.Validation.ServiceRequest.InvalidInvoice(sri.ReferenceNumber ?: "",
                serviceRequest.ServiceRequestNumber), null, null)
    }
}
```

In the base configuration, this currency compatibility check is also done for checks in the transaction validation rule set. The constraint is enforced on both sides, by service requests and checks, in case only one of the other validation rules is running during a commit.

Check Service Request History

The following rule raises an error if the data in the service request history is not consistent.

Condition

```
return true
```

Action

```
var changedFields = ServiceRequestChangePropertySet.getAll()
var currentHistoryValueByProperty = new HashMap<IEntityPropertyInfo, Object>()
for(srChange in serviceRequest.OrderedHistory index i) {
    for(changeFieldSet in changedFields) {
        var fieldChanged = srChange.getFieldValue(changeFieldSet.valueChangedChangeProp) as boolean
        // The change indicates that this field was modified, we will validate that it actually was.
        if(fieldChanged) {
            // We don't validate the first change, since it records all initial values as changes.
            if(i != 0) {
                // Check to make sure the field actually changed.
                if(currentHistoryValueByProperty[changeFieldSet.newValueChangeProp] == srChange.
                    getFieldValue(changeFieldSet.newValueChangeProp)) {
                    serviceRequest.rejectSubField(srChange, changeFieldSet.newValueChangeProp.Name, "loadsav",
                        displaykey.Rules.Validation.ServiceRequest.FalseChange(changeFieldSet.serviceRequestProp.
                        DisplayName), null, null)
                }
            }
        }
    }
}
```

```

        }
        //Store what the value changed to.
        currentHistoryValueByProperty[changeFieldSet.newValueChangeProp] =
            srChange.getFieldValue(changeFieldSet.newValueChangeProp)
    }
}

for(changeFieldSet in changedFields) {
    if(currentHistoryValueByProperty[changeFieldSet.newValueChangeProp] != serviceRequest.getFieldValue(changeFieldSet.serviceRequestProp)) {
        //The latest change value does not match the ServiceRequest's current value.
        serviceRequest.rejectField(changeFieldSet.serviceRequestProp as String, "loadsav",
            displaykey.Rules.Validation.ServiceRequest.MissingChange(changeFieldSet.serviceRequestProp,
            Display Name), null, null)
    }
}

```

Note: As always, Guidewire recommends that you use display keys, rather than fixed strings for error or warning messages.

Transaction Set Validation Rule Set

Note: See “Transaction Set Validation” on page 161 for a discussion of the transaction-related validation rules.

ClaimCenter runs the Transaction Set Validation rules any time that a `TransactionSet` is saved. ClaimCenter does not enforce validation levels on `Transaction` objects, so the Transaction Set Validation rule set always fires, regardless of the validation level.

In the following example, the rule rejects attempts to make payments on closed claims.

Condition

```
return transactionSet.Claim.isClosed() and transactionSet.Subtype=="checkset"
```

Action

```
transactionSet.reject( "payment", "You cannot make payments for closed claims. To proceed,
cancel this payment and reopen the claim.", null, null )
```

Note: As always, Guidewire recommends that you use display keys, rather than fixed strings for error or warning messages.

User Validation Rules

ClaimCenter runs the User Validation and User Preupdate rules on a save of the User entity. Any exceptions found during validation cause the bounding database transaction to roll back, thus effectively vetoing the update.

Guidewire recommends that you use the LoadSave validation level for rules pertaining to validation for users and groups.

In the base configuration, this rule set is empty.

Workplan Rule Set Category

A *workplan* adds initial activities to a Claim, Exposure or Matter object as a checklist of work that various people need to perform. Guidewire broadly divides these activities into the following categories:

- Activities that are performed a single time for the entire claim

For example, these activities include contacting the insured, investigating the accident scene, and getting a police report. Guidewire recommends that you add these types of activities as part of the claim’s workplan rules and associate them with the claim as a whole.

- Activities that are performed for each exposure or matter

For example, these activities include contacting the individual claimants, and getting a damage estimate for each vehicle. Guidewire recommends that you add these types of activities as part of the exposure or matter workplan rules and associate them with the exposure or matter.

In the following example, the rule creates an activity for each claim to contact the insured. Workplan values are defined within the activity pattern file.

Condition

```
return true
```

Action

```
claim.createActivityFromPattern( null,
    ActivityPattern.finder.getActivityPatternByCode("contact_insured") )
```

The purpose of this rule set is to add the most important (or most commonly overlooked) activities to the workplan for the claim. The following list describes the main methods to use within workplan rules. These rules are available on **Claim**, **Exposure**, and **Matter** business entities. (To determine which objects you can use within a rule, right-click within the rule and select **Complete Code** to view a list of valid objects.)

Method	Description
createActivity	Adds an activity based on an activity pattern, but allows you to override some of the defaults.
createActivityFromPattern	Adds an activity based on an activity pattern, using all of the pattern's defaults.
hasDuplicateClaimNumbers	Returns <code>true</code> if there are other claims that are possible duplicates of the claim being checked. The <code>Claim.DuplicateClaimNumbers</code> property contains a comma-delimited list of claim numbers that might be duplicates of the current claim, or <code>null</code> if no duplicates were found.

See also

- “Defining Activity Patterns” on page 425 in the *Configuration Guide*
- “Use Activity Pattern Codes Instead of Public IDs in Comparisons” on page 38 in the *Best Practices Guide*

Claim Workplan Rules

The Claim Workplan rules add initial activities to the claim as a checklist of work that various people need to perform on the claim. In the following example, the rule creates an activity to remind a user to obtain the police report of the incident if the claim is a property claim involving fire.

Condition

```
return claim.LossType == "PR" and claim.LossCause == "fire"
```

Action

```
claim.createActivityFromPattern( null,
    ActivityPattern.finder.getActivityPatternByCode("police_report") )
```

Exposure Workplan Rules

The Exposure Workplan rules add initial activities to the exposure as a checklist of work that various people need to perform on the exposure. If you are adding activities for each exposure, then it is important to avoid adding redundant activities. For example, if the same person is a claimant on multiple exposures, then you would not want to add a separate activity to contact the claimant for each exposure.

In the following example, the rule creates an activity to remind a team leader to perform a 10-day review of the exposure.

Condition

```
return not exists( activity in ( exposure.Claim.Activities ) where ( activity.ActivityPattern ==  
ActivityPattern.finder.getActivityPatternByCode( "10_day_review" ) ) )
```

Action

```
exposure.Claim.createActivityFromPattern( exposure,  
ActivityPattern.finder.getActivityPatternByCode( "10_day_review" ) )  
gw.api.util.Logger.logDebug( "10 Day Review activity has been created." )
```

Matter Workplan Rules

The Matter Workplan rules add initial activities to the matter as a checklist of work that various people need to perform on the matter. In the following example, the rule creates an activity to remind a user to review and document the legal details of a matter.

Condition

```
return true
```

Action

```
matter.Claim.createActivityFromPattern( null,  
ActivityPattern.finder.getActivityPatternByCode( "review_legal_expl" ) )
```

ClaimCenter Rule Execution

This topic provides information on how to generate reports that provide information on the ClaimCenter business rules.

This topic includes:

- “Generating a Rule Repository Report” on page 87
- “Generating a Rule Execution Report” on page 88

Generating a Rule Repository Report

To facilitate working with the Gosu business rules, ClaimCenter provides a command line tool to generate a report describing all the existing business rules. This tool generates the following:

- An XML file that contains the report information
- An XSLT file that provides a style sheet for the generated XML file

To create a rule repository report

1. Navigate to the `ClaimCenter/bin` directory.
2. At a command prompt, type:

```
gwcc regen-rulereport
```

This command generates the following files:

```
build/rules/RuleRepositoryReport.xml  
build/rules/RuleRepositoryReport.xslt
```

After you generate these files, it is possible to import the XML file into Microsoft Excel, for example. You can also provide a new style sheet to format the report to meet your business needs.

Generating a Rule Execution Report

The Guidewire Profiler provides information about the runtime performance of specific application code. It can also generate a report listing the business rules that individual user actions trigger within Guidewire ClaimCenter. The Profiler is part of the Guidewire-provided Server Tools. To access the Server Tools, Guidewire Profiler, and the rule execution reports, you must have administrative privileges.

To generate a rule execution report

1. Log into Guidewire ClaimCenter using a user account with access to the Server Tools.
2. Access the Server Tools and click **Guidewire Profiler** on the menu at the left-hand side of the screen.
3. On the Profiler Configuration page, click **Enable Web Profiling for this Session**. This action enables profiling for the current session. Profiling provides information about the runtime performance of the application, including information on any rules that the application executes.
Guidewire also provides a way to enable web profiling directly from within the ClaimCenter interface. To use the alternative method, press ALT+SHIFT+p within ClaimCenter to open a popup in which you can enable web profiling for the current session. If you use this shortcut, you do not need to access the Profiler directly to initiate web profiling. You still need to access the Profiler, however, to view the rule execution report.
4. Navigate back to the ClaimCenter application screens.
5. Perform a task for which you want to view rule execution.
6. Upon completion of the task, return to Server Tools and reopen ClaimCenter Profiler.
7. On the Profiler Configuration page, click **Profiler Analysis**. This action opens the default **Stack Queries** analysis page.
8. Under **View Type**, select **Rule Execution**.

See also

- “Guidewire Profiler” on page 163 in the *System Administration Guide*

Interpreting a Rule Execution Report

After you enable the profiler and perform activity within ClaimCenter, the profiler **Profiler Analysis** screen displays zero, one, or multiple stack traces.

Stack trace	Profiler displays...
None	A message stating that the Profiler did not find any stack traces. This happens if the actions in the ClaimCenter interface did not trigger any business rules.
One	A single expanded stack trace. This stack trace lists, among other information, each rule set and rule that ClaimCenter executed as a result of user actions within the ClaimCenter interface. The Profiler identifies each stack trace with the user action that created the stack trace. For example, if you create a new user within ClaimCenter, you see <code>NewUser -> UserDetailPage</code> for its stack name.
Multiple	A single expanded stack trace and multiple stack trace expansion buttons. There are multiple stack trace buttons if you perform multiple tasks in the interface. Click a stack trace button to access that particular stack trace and expand its details.

Each stack trace lists the following information about the profiled code:

- Time
- Frame (ms)
- Properties and Counters
- Name
- Elapsed (ms)

The profiler lists the rule sets and rules that ClaimCenter executed in the **Properties and Counters** column in the order in which they occurred.

part II

Advanced Topics

Assignment in ClaimCenter

This topic describes how Guidewire ClaimCenter assigns a business entity or object.

This topic includes:

- “Understanding Assignment” on page 93
- “ClaimCenter Assignment” on page 94
- “Assignment Execution Session” on page 97
- “Primary and Secondary Assignment” on page 98
- “Assignment Success or Failure” on page 100
- “Assignment Cascading” on page 101
- “Assignment Events” on page 101
- “Assignment Method Reference” on page 102
- “Using Assignment Methods in Assignment Pop-ups” on page 119
- “Assignment by Workload Count” on page 120

Understanding Assignment

Guidewire refers to certain business entities as *assignable* entities. This means, generally, that it is possible to designate a specific user as the party responsible for that entity. Guidewire defines the following entities as assignable in the ClaimCenter base configuration:

- Activity
- Claim
- Exposure
- Matter
- ServiceRequest
- UserRoleAssignment

You can create additional assignable entities through the following methods:

- Modify the base configuration data model by extending an existing entity to make it assignable

- Modify the base configuration data model by creating a new entity and making it assignable

To be assignable, an entity must implement certain required delegates and interfaces, not the least of which is the `Assignable` delegate class.

You use Gosu assignment methods to set an assigned group and user for an assignable entity. You can use these assignment methods either within the Assignment rule sets or within any other Gosu code (a class or an enhancement, for example).

Assignment Persistence

ClaimCenter persists the assignment any time that you persist the entity being assigned. Therefore, if you invoke the assignment methods on some entity from arbitrary Gosu code, you need to persist that entity. This can be, for example, either as part of a page commit in the ClaimCenter interface or through some other mechanism.

Assignment Queues

Each Group in Guidewire ClaimCenter has a set of `AssignableQueue` entities associated with it. (For example, the base configuration assigns each group a queue named FNOL.) It is possible to modify the set of associated queues.

Currently, Guidewire only supports assigning `Activity` entities to queues. If you assign an activity to a queue, however, you cannot simultaneously assign it to a user as well.

With the release of ClaimCenter 5.0, Guidewire officially deprecated the `assignToQueue` methods on `Assignable` in favor of the `assignActivityToQueue` method. This makes the restriction more clear. See “Queue Assignment” on page 104 for more information.

See also

- “ClaimCenter Assignment” on page 94
- “Primary and Secondary Assignment” on page 98
- “Assignment Success or Failure” on page 100
- “Assignment Method Reference” on page 102

ClaimCenter Assignment

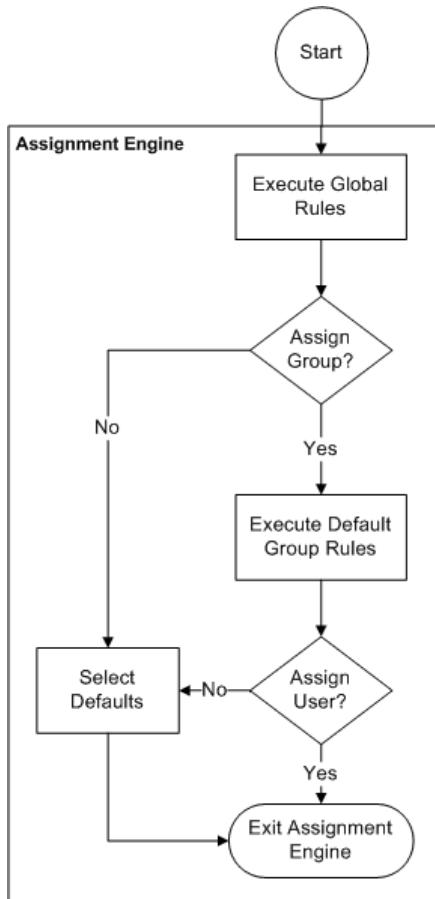
Assignment that uses the Assignment engine works within the context of the Assignment rules, which ClaimCenter groups into the categories described in the following topics:

- “Global Assignment Rules” on page 96
- “Default Group Assignment Rules” on page 97

In general, ClaimCenter uses the global assignment rules to determine the group to which to assign the entity. It then runs the default group assignment rules to assign the entity to a user in the chosen group. The Assignment engine runs the assignment rules until it either completes the assignment or it runs out of rules to run.

Guidewire designs the assignment rules to *trickle down* through the group structure. For example, if a user assigns a claim using automated assignment, the Assignment engine runs a global claim assignment rule first. Typically, the global rule sets a group. Then, the Assignment engine calls the assignment rule set for that group. This rule set can either perform the final assignment or assign the claim to another group. If assigned to another group, ClaimCenter calls the rule set for that group. This process continues until the Assignment engine completes the final assignment. (For simplicity, these examples use claim throughout, although the same logic applies to all assignable entities.)

The following graphic illustrates this process.



To support this process, the assignment engine contains the sense of *execution session* and *current group*. You cannot explicitly set these items.

- The *execution session* is one complete cycle through the assignment rules. In the Assignment engine graphic, this means all the processes that occur between the *Start* point and the *Exit Assignment Engine* termination point.
- The *current group* is the group to which the Assignment engine assigns the assignable entity after it executes an assignment rule set.

IMPORTANT The Assignment engine sets the current group as it exits the rule set. To alleviate unintended assignments, Guidewire recommends that you exit the rule set immediately after making a group assignment.

Several of the assignment methods (notably `assignByUserAttributes` and `assignUser`) require a current group to be set. If the current group is not set, these methods return `false` immediately. The combination of the current group and these assignment actions can give unpredictable results if you do not implement them carefully.

Consider the following rule:

```

claim.CurrentAssignment.assignGroup(someGroup)

if ( claim.CurrentAssignment.assignByUserAttributes(someAttributes, false,
        claim.CurrentAssignment.AssignedGroup) ) {
    actions.exit()
}
  
```

At first, it would seem that this rule assigns the claim to someone in `someGroup` who has the appropriate attributes. However, it does the following:

- If the rule is in a global rule set, it does nothing at all.
- If the rule is in the rule set of another group—say group X—it assigns the claim to a user in group X.

The reason for this is that the logic for `assignByUserAttributes` only cares about the current group in the assignment context. (The current group is `null` in the global rule set and group X in group X's rule set). The `assignByUserAttributes` method does not care about what group the claim is currently assigned.

In actuality, as the rules trickle down through the organization, the Assignment engine sets the current group as it exits the rule set. To restate, Guidewire strongly recommends that you exit the rule set immediately after making a group assignment.

Therefore, for the previous example, you need to immediately exit the rule upon assigning the claim to a group (`someGroup`). Then, you would use a separate rule to assign the claim to a user who is a member of `someGroup` as determined by that user's attributes.

Rule 1

```
if ( claim.CurrentAssignment.assignGroup(someGroup) ) { actions.exit() }
```

Rule 2

```
if ( claim.CurrentAssignment.assignByUserAttributes(someAttributes, false,
    claim.CurrentAssignment.AssignedGroup) ) {
    actions.exit()
}
```

See also

- “Assignment Execution Session” on page 97

Global Assignment Rules

Initially, the Assignment engine invokes the Global assignment rules for the entity type in question. There is a rule set for each entity type, meaning there is a rule set each to handle activities, claims, exposures, matters, and service requests.

The Assignment engine runs the global rules one time only. There are three valid outcomes for a Global Assignment rule:

- One of the rules assigns both a group and a user. In this case, the assignment process completes and the Assignment engine exits.
- One of the rules assigns a group. In this case the assignment process continues with the Default assignment rules.
- None of the rules perform any assignment. In this case, the Assignment engine assigns the entity to a default user and group and the Assignment engine exits.

Global assignment rules trigger anytime that you use auto-assignment to assign an entity. ClaimCenter runs rule in the Segmentation rule set category just prior to running assignment rules any time that you select automated assignment for a new claim or exposure. See “Segmentation Rule Set Category” on page 72 for details.

There is one global rule set for each type of assignable entity. These rules make the initial assignment decision. This, in turn, triggers further assignment rules for a particular group to narrow the selection criteria. (These rules can define geographical or business criteria to use in the selection process, for example.) Then, the assignment rules distribute the work to a specific user or queue within the chosen group.

Assignment logic is often *trickle-down* or hierarchical. For example, ClaimCenter might use the Assignment rules to determine the ultimate assignment owner by first determining each of the following:

- The regional group
- The state group

- The local group
- The loss type, and finally,
- The individual

Guidewire recommends that you do not make user assignments directly using the Global assignment rules. Instead, use these rules to trigger further group selection rules. For example, suppose that you make an assignment directly to a user who does not have the requisite job role, or requisite authority. In this case, there is no other rule that governs the situation, and the Assignment engine makes no assignment.

It is possible for you to implement your own assignment class and call it directly from within ClaimCenter, if you choose. Guidewire does not restrict you to using the base configuration Assignment rules only.

Default Group Assignment Rules

The Default Group assignment rules again contain one rule set for each assignable entity type. These rule sets contain rules that execute anytime that you manually assign an item from within ClaimCenter. (These rules govern assignment selection also if you do not specify any other rule set for a group.)

After the Assignment engine invokes the rules, there are again three possibilities:

- One of the rules assigns a User. In this case, the assignment process is done and the Assignment engine exits.
- None of the rules assigns a User. However, one of the rules assigns a different Group. In this case, the Assignment engine runs the Default Assignment rules again.
- None of the rules perform any assignment. In this case, the assignment fails and the Assignment engine exits.

Assignment Execution Session

IMPORTANT Guidewire recommends that you do not use the `getCurrentGroupFromES` method. Guidewire intends to deprecate this method in a future release.

Guidewire provides a `gw.api.assignment.AssignmentEngineUtil` class that provides several useful helper assignment methods. Do not use these helper methods as you create new assignment logic. Instead, use this class to provide a bridge between any existing deprecated assignment methods without a group parameter and the current assignment methods that do take a group parameter.

The following helper method, in particular is useful in that it returns the default group stored in the `ExecutionSession`:

```
getDefaultValueFromExecutionSession
```

Formerly, a number of (now deprecated) assignment methods did not use the `GroupBase` parameter, and instead relied on the implicit Assignment engine state. As you can invoke these methods outside of the Assignment engine, you can not rely on this implicit state during assignment. Therefore, Guidewire requires that all assignment methods take a `GroupBase` parameter. You can use the `getDefaultValueFromExecutionSession` method to retrieve the current `GroupBase` value in `ExecutionSession`.

The returned `GroupBase` value from the `getDefaultValueFromExecutionSession` method is the result of the last run of the rule set (global or default), *instead of the current value*. If a rule changes the assigned group but does not exit (by calling `actions.exit`), the assigned group is different from all the following rules in the current rule set.

However, if you want the currently assigned group on an object, a `Claim` for example, then use the following:

```
Claim.CurrentAssignment.AssignedGroup
```

This method returns the currently assigned group on the current assignment. Unlike `getDefaultValueFromExecutionSession`, if a rule sets the `AssignedGroup` value and does *not* exit (does not call `actions.exit`), all the following rules use this value for the current group.

As the return value of `getDefaultValueFromExecutionSession` can be `null`, you need to check for this condition.

Primary and Secondary Assignment

At its core, the concept of assignment in Guidewire ClaimCenter is basically equivalent to ownership. The user to whom you assign an activity is the user who owns that activity, and who, therefore, has primary responsibility for it. Ownership of a `Claim` entity, for example, works in a similar fashion. Guidewire defines an entity that someone can own in this way as *assignable*.

Assignment in Guidewire ClaimCenter is always made to a user and a group, as a pair (group, user). However, the assignment of the user is not dependent on the group. In other words, you can assign a user that is not a member of the specified group.

IMPORTANT In the base configuration, ClaimCenter provides assignment methods that take only a user and that assign the entity to the default group for that user. Guidewire has now deprecated these types of assignment methods. Do not use them. Guidewire recommends that you rework any existing assignment methods that only take a user into assignment methods that take both a user and a group.

Guidewire ClaimCenter distinguishes between two different types of assignment:

Primary	Also known as user-based assignment, primary assignment assigns an object that implements the <code>Assignable</code> delegate to a particular user. See "Primary (User-based) Assignment" on page 98.
Secondary	Also known as role-based assignment, secondary assignment assigns an object that implements the <code>RoleAssignments</code> array to a particular user role. Each role is held by a single user at a time, even though the user who holds that role can change over time. See "Secondary (Role-based) Assignment" on page 98.

Primary (User-based) Assignment

ClaimCenter uses *primary* assignment in the context of ownership. For example, only a single user (and group) can own an activity. Therefore, an `Activity` object is primarily assignable. *Primary* assignment takes place any time that ClaimCenter assigns an item to a *single* user.

Primary assignment objects *must* implement the `Assignable` and `CCAssignable` delegates. In the ClaimCenter base configuration, the following objects implement the required delegates:

- `Activity`
- `Claim`
- `Exposure`
- `Matter`
- `ServiceRequest`
- `UserRoleAssignment`

It is common for ClaimCenter to implement the `Assignable` delegate in the main entity definition file and the `CCAssignable` delegate in an entity extension file.

Secondary (Role-based) Assignment

ClaimCenter uses *secondary* assignment in the context of user roles assigned to an entity that does not have a single owner. For example, an entity can have multiple roles associated with it as it moves through ClaimCenter, with each role assigned to a specific person. Since each of the roles can be held by only a single user,

ClaimCenter represents the relationship by an array of `UserRoleAssignment` entities. These `UserRoleAssignment` entities are primarily assignable and implement the `Assignable` delegate.

Thus, for an entity to be secondarily assignable, it must have an associated array of role assignments, the `RoleAssignments` array. (This array contains the set of `UserRoleAssignment` objects.) In the ClaimCenter base configuration, the following objects all contain a `RoleAssignments` array:

- `Claim`
- `Exposure`

It is possible for an entity to be both primarily and secondarily assignable.

It is important to understand that `UserRoleAssignment` entities share the rule set of their primary entities. Thus, ClaimCenter assigns the `UserRoleAssignment` entities for an activity using the Activity Assignment rule sets, and so on. It is for this reason that assignment statements in the rules always use the `CurrentAssignment` formulation, such as the following:

```
Activity.CurrentAssignment.assignUserAndDefaultGroup( user )
```

It is important for rules to use this syntax, because it allows ClaimCenter to use the rules (in this case) for both activities and activity-related `UserRoleAssignment` entities.

Secondary Assignment and Round-robin Assignment

Secondary assignment uses the assignment owner to retrieve the round-robin state. What this means is that different secondary assignments on the same assignment owner can end up using the same round-robin state, and they can affect each other.

In general, if you use different search criteria for different secondary assignments, you do not encounter this problem as the search criteria is most likely different. However, if you want to want to make absolutely sure that different secondary assignments follow different round-robin states, then you need to extend the search criteria. In this case, add a flag column and set it to a different value for each different kind of secondary assignment. See “Round-robin Assignment” on page 114 for more information on this type of assignment.

Assignment within the Assignment Rules

Guidewire provides sample assignment rules in the `Assignment` folder in the Studio Resources tree. These rule include both primary and secondary assignment. For assignment that use the Assignment engine (which is all rules in the `Assignment` folder), use the following assignment properties:

Assignment type	Use...
Primary	<code>entity.currentAssignment</code> This property returns the current assignment, regardless of whether you attempt to perform primary or secondary assignment. If you attempt to use it for secondary assignment, then the property returns the same object as the <code>CurrentRoleAssignment</code> property.
Secondary	<code>entity.currentRoleAssignment</code> This property returns <code>null</code> if you attempt to use it for primary assignment.

If you assign an entity outside of the Assignment engine (meaning outside of the Assignment rules), then you do not need to use `currentAssignment`. Instead, you can use any of the methods that are available to entities that implement the `Assignable` delegate directly, for example:

```
activity.assign(group, user)
```

ClaimCenter does not require that `user` be a member of `group`, although this is the most usual practice. In other words, you can assign a user that is not a member of the specified group.

Assignment Success or Failure

All assignment methods, except those that invoke the Assignment engine, return a Boolean value that is `true` if an assignment was successfully found and `false` otherwise. There are two general types of assignment methods, methods that use the assignment engine and methods that do not.

This topic includes:

- “Assignment Methods that Use the Assignment Engine” on page 100
- “Assignment Methods that Do Not Use the Assignment Engine” on page 100
- “Determining Success or Failure” on page 100
- “Logging Assignment Activity” on page 101

Assignment Methods that Use the Assignment Engine

You can invoke assignment by calling the `autoAssign` method from outside the assignment rules, which invokes assignment automatically.

If you invoke assignment and a user cannot be assigned, the item is assigned to the group supervisor. If it is not possible to assign the item to the group supervisor, then the item is assigned to the ClaimCenter default owner (`defaultowner`).

Note: In some cases, it might not be possible to make an assignment to a group supervisor. For example, this can be if the assignment rules failed to assign to a Group even, or, perhaps, if the assigned Group has no supervisor.

ClaimCenter provides user `defaultowner` as the assignee of last resort. This user’s first name is Default and the last name is Owner, with a user name of `defaultowner`. Guidewire recommends, as a business practice, that you have someone in the organization periodically search for outstanding work assigned to this user. If the user finds one of these assignments, the user must reassign these items to a proper owner. Guidewire also recommends that the Rule Administrator investigate why ClaimCenter did not assign an item of that type, so that you can correct any errors in the rules.

IMPORTANT Do not attempt to make direct assignments to user `defaultowner`.

Assignment Methods that Do Not Use the Assignment Engine

If you call one of the assignment methods directly from outside the Assignment rules, the method simply returns `true` if it has found a valid assignment or `false` otherwise. If the method returns `false`, it is your responsibility to determine what to do. It is possible to assign the item to the group supervisor or to invoke another assignment method, for example.

Determining Success or Failure

Guidewire recommends that you always determine if assignment actually succeeded.

- In general, if you call an assignment method directly and it was successful, then you need do nothing further. However, you need to take care if you call an assignment method that simply assigns the item to a group, such as one of the `assignGroup` methods. In this case, it can be necessary to call another assignment method to assign the item to an actual user.
- If you call assignment methods in an Assignment rule, you can exit the rule (not consider any further rules), or perform other actions before exiting.

Logging Assignment Activity

Guidewire recommends that you log the action anytime that you use one of the assignment methods. The following example, called in an Assignment rule, shows how to log assignment activity.

```
if (Activity.CurrentAssignment.assignUserByRoundRobin( false,  
    Activity.CurrentAssignment.AssignedGroup ) ) {  
    Activity.CurrentAssignment.confirmManually( User( 2 /* Super User */ ) )  
    gw.api.util.Logger.logDebug( "Assigned to: " + Activity.AssignedUser.DisplayName )  
    actions.exit()  
}
```

Putting the exit action inside the `if` block forces ClaimCenter to continue to the next rule if the current rule is not successful. A rule is not successful if it cannot choose a group or person to which to assign the item. It is important that you plan an effective exit from a rule. Otherwise, ClaimCenter can make further unanticipated and unwanted assignments. If you are performing assignment from an Assignment rules you need only to use the `actions.exit` method to exit the rule.

Assignment Cascading

For certain top-level assignable entities, ClaimCenter needs to re-assign some subobjects any time that it assigns a top-level entity. For example, if you reassigns a claim, ClaimCenter does the following:

- It reassigns all open activities connected to that claim. (These includes the activities connected to the claim only, and not those connected to any specific exposure.) This includes all activities currently assigned to the same group and user as the claim.
- It reassigns all non-closed exposures for the claim currently assigned to the same group and user as the claim.

This cascading behavior is Guidewire application-specific. You cannot configure it.

- If you call the assignment methods through the Assignment engine, the Assignment engine cascades the assignments at the end of the process, as the engine exits with a completed assignment.
- If you call the assignment methods directly, then the assignment methods perform cascading of the assignment immediately as each method exits. For this reason, if you perform assignment outside of the Assignment rules, Guidewire strongly recommends that your Gosu code first determine the appropriate assignee (or set of assignees). It can then call a single assignment method on the assignee (or set of assignees). Do not rely on the recursive structure assumed by the assignment rules to perform assignment outside of the assignment rules.

Assignment Events

Anytime that the assignment status changes on an assignment, ClaimCenter fires an assignment event. The following events can trigger an assignment change event:

- `AssignmentAdded`
- `AssignmentChanged`
- `AssignmentRemoved`

The following list describes these events.

Old status	New status	Event fired	Code
Unassigned	Unassigned	None	None
Unassigned	Assigned	<code>AssignmentAdded</code>	<code>Assignable.ASSIGNMENTADDED_EVENT</code>
Assigned	Assigned	<code>AssignmentChanged</code> —if a field changes, for example, the assigned user, group, or date	<code>Assignable.ASSIGNMENTCHANGED_EVENT</code>
Assigned	Unassigned	<code>AssignmentRemoved</code>	<code>Assignable.ASSIGNMENTREMOVED_EVENT</code>

Assignment Method Reference

IMPORTANT Guidewire deprecates assignment method signatures that do not have a `Group` parameter. Studio indicates this status by marking through the method signature in Gosu code and Studio code completion does not display deprecated methods. You can, however, see them in listed in the full list of assignment methods (with Studio again indicating their deprecated status). Do not use a deprecated assignment method—one without the `Group` parameter—outside of the Assignment rules. In general, Guidewire recommends that you do not use deprecated methods at all. If your existing Gosu code contains deprecated methods, Guidewire recommends that you rework your code to use non-deprecated methods.

Guidewire divides the assignment methods into the general categories described in the following topics:

- “Assignment by Assignment Engine” on page 102
- “Group Assignment (within the Assignment Rules)” on page 102
- “Queue Assignment” on page 104
- “Immediate Assignment” on page 105
- “Claim-Based Assignment” on page 106
- “Condition-Based Assignment” on page 107
- “Proximity-Based Assignment” on page 110
- “Round-robin Assignment” on page 114
- “Dynamic Assignment” on page 115
- “Manual Assignment” on page 117

Note: For the latest information and description on assignment methods, consult the Studio API reference material available through [Help → Gosu API Reference](#). You can also place the Studio cursor within a method signature and select the F1 keyboard key.

Assignment by Assignment Engine

This type of assignment invokes the Assignment engine. Do not call the following from within the Assignment rules themselves.

```
autoAssign  
public boolean autoAssign()
```

This method invokes the Assignment engine to assign the entity that called it. Call this method outside of the Assignment rules if you want to invoke the Assignment engine to carry out assignment. For example, you can call this method as part of entity creation to perform the initial assignment.

```
var actv = Claim.newActivity( null, null )  
actv.autoAssign()
```

IMPORTANT Do not call this assignment method in the context of the Assignment rules. The method invokes the Assignment engine. Calling it within the context of the Assignment rules can potentially create an infinite loop. If you attempt to do so, ClaimCenter ignores the method call and outputs an error message.

Group Assignment (within the Assignment Rules)

Methods that only assign a group are most useful when called in rules in the Global Assignment rule set, which are responsible only for assigning a group. In all other contexts, you need to assign both a user and a group.

Therefore, it makes more sense to use one of the other types of assignment methods that perform both assignments. These methods are described in the following topics:

- “assignGroup” on page 103
- “assignGroupByLocation” on page 103
- “assignGroupByRoundRobin” on page 104

Note: It is important to understand that these methods do not assign the assignable item to a user. They simply pick a group to use during the rest of the assignment process. To complete the assignment, you must use one of the other assignment methods to assign the item to a user.

assignGroup

```
boolean assignGroup(GroupBase group)
```

Use this method to assign the indicated group. You use this method in situations in which you want to assign a known group.

The following example assigns a specific group based on a couple of possibilities.

```
var CurrentSIUserGroup : Group
var currentSIUser : User
var theActivity : Activity

for (theRelatedContact in Activity.Claim.RoleAssignments) {
    if (therelatedcontact.role=="SIUInvestigator") {
        currentSIUser = therelatedcontact.User
        currentSIUserGroup = therelatedcontact.Group
    }
}

if (currentSIUser != null) {
    if (Activity.CurrentAssignment.assignGroup( currentSIUserGroup ) ) {
        actions.exit()
    } else {
        if (activity.CurrentAssignment.assignGroup( Group( "demo_sample:46" /* Western SIU */ ) ) ) {
            actions.exit()
        }
    }
}
```

assignGroupByLocation

```
boolean assignGroupByLocation(groupType, address, directChildrenOnly, group)
```

This method uses the location-based assigner to assign an assignable entity based on a given address. You use this method, for example, to assign an activity based on geographic closeness to a particular processing office, to match up with the specific time zone.

The method filters the groups by the `groupType` parameter (which can be `null`). It matches a group first by zip, then by county, then by state. The first match wins. If two or more groups match at a particular level, the method returns the first one found. Matching is case insensitive.

The search begins with the children of the specified group. ClaimCenter does not include the specified group itself in the search. If there is no group specified, ClaimCenter uses the root instead.

If you specify the `directChildrenOnly` parameter as `true`, the method only searches for children directly underneath the current group. The search stops without searching the children of the children.

IMPORTANT If you pass a `null` value for the `group` parameter, the method starts with the top group in the hierarchy. This can cause severe performance issues.

The following example first filters the group type by claim loss type and segment to assign the claim by location.

```
var losstype = claim.LossType
var segment = claim.Segment
var result = libraries.ClaimAssignment.getGroupTypebasedonClaimSegment(losstype, segment)
var primarygroupype = result[0]
var secondarygroupype = result[1]
```

```

if (claim.LossLocation <> null) {
    claim.CurrentAssignment.assignGroupByLocation( primarygroupType, Claim.LossLocation, false,
        claim.AssignedGroup)
}

```

assignGroupByRoundRobin

```
boolean assignGroupByRoundRobin(groupType, includeSubGroups, group)
```

This method assigns an entity to a group (or one of its subgroups) using round-robin assignment. The method restricts the set of available groups to those matching the specified group type. Use this method to distribute work (assignable items) among the different groups equitably.

Note: The `assignGroupByRoundRobin` method ignores any group for which the `Group.LoadFactor` value is zero (0).

For example, suppose you have several groups of people who specialize in handling complex issues. For some high-complexity items, you might want to rotate among the groups on an equitable basis. The passed-in `groupType` (if non-null) ensures that the method only considers for round-robin assignment those groups that have the matching value for their `groupType` fields. Thus, if there are three groups, all with the same group type, this method performs round-robin assignment between the three groups.

The following example assigns a claim to one of the “complex claim” groups within the currently assigned group.

```
claim.assignGroupByRoundRobin( "autocomplex", false, claim.AssignedGroup)
```

Queue Assignment

Each group in Guidewire ClaimCenter has an associated queue to which you can assign items. This is a way of putting assignable entities in a placeholder location without having to assign them to a specific person. Currently, Guidewire only supports assigning Activity entities to a Queue.

Within ClaimCenter, an administrator can define and manage queues through the ClaimCenter **Administration** screen.

assignActivityToQueue

```
boolean assignActivityToQueue(queue, currentGroup)
```

Use this method to assign this activity to the specified queue. The activity entity then remains in the queue until someone or something reassigns it to a specific user. It is possible to assign an activity in the queue manually (by the group supervisor, for example) or for an individual to choose the activity from the queue.

To use this method, you need to define an `AssignableQueue` object. You can do this in several different ways. For example:

Defining an AssignableQueue Object by Using a Finder Method

You can use one of the following finder methods to return an `AssignableQueue` object:

```
AssignableQueue.findVisibleQueuesForUser( User )
AssignableQueue.findVisibleQueuesInUserAndAncestorGroups( User )
```

Consult the Gosu API Reference for details of how to use these finder methods.

Defining an AssignableQueue Object by Using the Group Name

You can use the name of the group to retrieve a queue attached to that group.

```
Activity.AssignedGroup.getQueue(queueName)      //Returns an AssignableQueue object
Activity.AssignedGroup.AssignableQueues          //Returns an array of AssignableQueue objects
```

In the first case, you need to know the name of the queue. You cannot do this directly, as there is no real unique identifier for a Queue outside of its group.

Note: In the ClaimCenter base configuration, each Group has a “FNOL” queue defined for it by default.

If you have multiple queues attached to a group, you can do something similar to the following to retrieve one of the queues. For example, use the first method if you do not know the name of the queue. Use the second method if you know the name of the queue.

```
var queue = Activity.AssignedGroup.AssignableQueues[0]
var queue = Activity.AssignedGroup.getQueue( "QueueName" )
```

You can then use the returned `AssignableQueue` object to assign an activity to that queue.

```
Activity.CurrentAssignment.assignActivityToQueue( queue, group )
```

Immediate Assignment

The methods described in the following topics perform immediate, direct assignment to the specified user or group:

- “`assign`” on page 105
- “`assignUserAndDefaultGroup`” on page 105
- “`assignToIssueOwner`” on page 106
- “`assignToCreator`” on page 106
- “`assignToPreviousOwner`” on page 106

assign

```
boolean assign(group, user)
```

This method assigns the assignable entity to the specific group and user. This is perhaps the simplest of possible assignment operations. Use this method if you want a specific known user and group to own the entity in question.

In the following example, the assignment method assigns another exposure to the owner of an already assigned exposure.

```
for (exp in Exposure.Claim.Exposures) {
    if (exp <> Exposure
        and exp.AssignedGroup == Exposure.CurrentAssignment.AssignedGroup
        and exp.AssignedUser <> null) {
        if (Exposure.CurrentAssignment.assign( exp.AssignedGroup, exp.AssignedUser )) {
            gw.api.util.Logger.logDebug( "#### This is the Default Group Exposure rule "
                + gw.api.util.StringUtil.substring(actions.getRule().displayName, 0, 8))
            gw.api.util.Logger.logDebug( "Assigned User is " + exposure.AssignedUser)
            actions.exit()
        }
    }
}
```

assignUserAndDefaultGroup

```
boolean assignUserAndDefaultGroup(user)
```

This method assigns the assignable entity to the specified user, selecting a default group. The default group is generally the first group in the set of groups to which the user belongs. In general, use this method if a user belong to a single group only, or if the assigned group really does not matter.

It is possible that the assigned group can affect visibility and permissions. Therefore, Guidewire recommends that use this method advisedly. For example, you might want to use this method only under the following circumstances:

- The users belong to only a single group.
- The assigned group has no security implications.

The following example assigns an Activity to the current user and does not need to specify a group.

```
Activity.CurrentAssignment.AssignUserAndDefaultGroup(User.util.CurrentUser)
```

assignToIssueOwner

```
boolean assignToIssueOwner()
```

This method assigns the assignable entity to the “Issue Owner”. This concept only really applies to Guidewire ClaimCenter, in which the issue owner is the owner of the associated Claim. Use this method to assign a subsidiary entity (for example, an Exposure or a Matter) to the owner of the primary entity, such as the Claim.

The following example assigns an exposure to the issue owner (the owner of the associated claim).

```
Exposure.CurrentAssignment.assignToIssueOwner()
```

assignToCreator

```
boolean assignToCreator(sourceEntity)
```

This method assigns the assignable entity to the user who created the supplied `sourceEntity` parameter. The following example assigns an exposure to creator of the claim associated with the exposure.

```
exposure.CurrentAssignment.assignToCreator( exposure.Claim )
```

assignToPreviousOwner

```
boolean assignToPreviousOwner()
```

Assigns the entity to the previously assigned user. The method tracks the current group. If there is no current group defined, the method does nothing and logs a warning. Although not officially deprecated, Guidewire recommends that you not use it.

Claim-Based Assignment

Guidewire provides several assignment methods that assign an activity based on the claim owner. The following topics describe two of these methods:

- “`assignToClaimOwner`” on page 106
- “`assignToClaimUserWithRole`” on page 106

assignToClaimOwner

```
void assignToClaimOwner()
```

This method assigns the assignable entity to the user and group of the associated claim. *It is your responsibility to verify that the entity has a link to an assigned claim.* If the resulting assignment is invalid for any reason, the method throws `IllegalAssignmentException`. Currently, Guidewire only supports this method for use with an activity, exposure, or matter.

The following example assigns an exposure to the claim owner.

```
(Exposure.CurrentAssignment as CCAssignable).assignToClaimOwner()
```

Guidewire recommends, before attempting to assign the assignable, that you test if it is possible to assign the assignable using this method. Use the following property on the assignable to determine the owning claim for the assignable.

```
entity.OwningClaim
```

If the return value is `null`, then it is not possible to assign the assignable using the `assignToClaimOwner` method. Otherwise, it returns the owning claim object.

assignToClaimUserWithRole

```
boolean assignToClaimUserWithRole(userRole)
```

This method assigns the assignable entity based on the role associated with that entity. The method performs the following steps in searching for a user with a matching role:

1. If assigning an activity associated with an exposure, the method looks at roles associated with Exposures first.
2. It next attempts to match the supplied `userRole` with a `Claim UserRole`.
3. If not assigning an activity associated with an exposure, the method then searches for a match with a `User Role` associated with an Exposure.

During the search:

- If the search finds a match at any step, it stops.
- If the match is unique, the method performs the assignment and returns `true`.
- If the match is not unique, the method returns `false`.
- If the search goes through all steps without finding a match, the method returns `false`.

In the base configuration, you can assign the entity to one of the following roles:

- | | | |
|-------------------------|--------------------|---------------------|
| • Attorney | • Related User | • Subrogation Owner |
| • Doctor | • Repair Shop | • Supervisor |
| • Independent Appraiser | • Reporter | • Vendor |
| • Nurse Case Manager | • Salvage Owner | • Vehicle Inspector |
| • Police | • SIU | |
| • Property Inspector | • SIU Investigator | |

Note: Guidewire defines the roles that a user can have on an assignable object in typelist `User Role`.

The following example assigns an exposure to a claim owner with a user role of `siuInvestigator`, a claim fraud specialist that investigates potential claim fraud.

```
(Exposure.CurrentAssignment as CCAssignable).assignToClaimUserWithRole("siuInvestigator")
```

Condition-Based Assignment

Condition-based assignment methods follow the same general pattern:

- They use a set of criteria to find a set of qualifying users, which can span multiple groups.
- They perform round-robin assignment among the resulting set of users.

Two things are important to note:

- ClaimCenter ties the round-robin sequence to the set of criteria, not to the set of users. Thus, using the same set of restrictions to find a set of users re-uses the same round-robin sequence. However, two different sets of restrictions can result in *distinct* round-robin sequences, even if the set of resulting users is the same.
- ClaimCenter does not use this kind of round-robin assignment with the load factors maintained in the ClaimCenter **Administration** screen. Those load factors are meaningful only within a single group, and this kind of assignment can span groups.

Condition-based assignment methods include those described in the following topics:

- “`assignByUserAttributes`” on page 108
- “`assignUserByLocation`” on page 109
- “`assignUserByLocationAndAttributes`” on page 109

assignByUserAttributes

```
boolean assignByUserAttributes(attributeBasedAssignmentCriteria, includeSubGroups, currentGroup)
```

This method assigns an assignable item to the user who best matches the set of user attribute constraints defined in the `attributeBasedAssignmentCriteria` parameter. At times, it is important that you assign a particular claim to a specific user, such as one who speaks French or one who has some sort of additional qualification. It is possible that these users exist across an organization, rather than concentrated into a single group.

You use the `currentGroup` and `includeSubGroups` parameters to further restrict the set of users under consideration to certain groups or subgroups. The `currentGroup` parameter can be `null`. If it is non-null, the assignment method uses the parameter for two purposes:

1. The assignment method maintains separate round-robin states for the search criteria within each group. This is so that ClaimCenter can use the method for group-specific assignment rotations.
2. After the method selects a user, it uses the group to determine the best group for the assignment. This means that the method assigns the entity to this group or one of its subgroups—if the method can find an appropriate membership for the user. Otherwise, the method uses the user's first group.

The `AttributeBasedSearchCriteria` object has two fields:

<code>GroupID</code>	If set, restricts the search to the indicated group. This can be <code>null</code> .
<code>AttributeCriteria</code>	An array of <code>AttributeCriteriaElement</code> entities.

The `AttributeCriteriaElement` entities represent the conditions to be met. If more than one `AttributeCriteriaElement` entity is present, the method attempts to assign the assignable entity to those users who satisfy all of them. In other words, the method ANDs the restrictions together.

The `AttributeCriteriaElement` entity has a number of fields, which are all optional. These fields can interact in very dependant ways, depending on the value of `UserField`.

Field	Description
<code>UserField</code>	The the <code>AttributeCriteriaElement</code> behaves differently depending on whether the <code>UserField</code> property contains an actual value: <ul style="list-style-type: none"> • If set, then <code>UserField</code> must be the name of a property on the <code>User</code> entity. The method imposes a search restriction using the <code>Operator</code> and <code>Value</code> fields to find users based on their value for this field. • If <code>null</code>, then the method imposes a search restriction based on attributes of the user. The exact restriction imposed can be more or less strict based on the other fields set:
<code>AttributeField</code>	If set, this is the name of a property on the <code>Attribute</code> entity. The method imposes a search restriction using the <code>AttributeValue</code> field to find users based on the user having the appropriate value for the named field for some attribute.
<code>AttributeType</code>	If set, then the method tightens the <code>AttributeField</code> -based restriction to Attributes only of the indicated type.
<code>AttributeValue</code>	If set, then the method restricts the search to users that have the specified <code>AttributeValue</code> only.
<code>State</code>	If set, then the method restricts the search to users that have an <code>Attribute</code> with the indicated value for <code>State</code> .
<code>Value</code>	If set, then the method restricts the search to users that have the specified <code>Value</code> for an <code>Attribute</code> that satisfies the other criteria.

The following example searches for all of `User` entities who have an `AttributeType` of *language* and `Attribute` value of *French*.

```
var attributeBasedAssignmentCriteria = new AttributeBasedAssignmentCriteria()
var frenchSpeaker= new AttributeCriteriaElement()
frenchSpeaker.AttributeType = UserAttributeType.TC_LANGUAGE
frenchSpeaker.AttributeField = "Name"
frenchSpeaker.AttributeValue = "French"
attributeBasedAssignmentCriteria.addToAttributeCriteria( frenchSpeaker )
```

```
activity.CurrentAssignment.assignByUserAttributes(attributeBasedAssignmentCriteria , false,  
activity.CurrentAssignment.AssignedGroup )
```

assignUserByLocation

```
boolean assignUserByLocation(address, includeSubGroups, currentGroup)
```

This method uses the location-based assigner to assign an assignable entity based on a given address. This is useful, for example, in the assignment of adjusters and accident appraisers, which is often done based on geographical territory ownership.

The method first matches user by zip, then by county, then by state. The first match wins. If one or more users match at a particular location level, then assignment performs round-robin assignment through that set, ignoring any matches at a lower level. For example, suppose the method finds no users that match by zip, but a few that match by county. In this case, the method performs round-robin assignment through the users that match by county and it ignores any others that match by state.

Note: ClaimCenter bases persistence in the round-robin assignment state on the specified location information. For this reason, it is preferable to use a partially completed location, such as one that includes only the zip code, rather than a specific house.

The following example searches on the claimant's primary address location.

```
Claim.assignUserByLocation(Claim.Claimant.PrimaryAddress, true, Claim.AssignedGroup)
```

assignUserByLocationAndAttributes

```
boolean assignUserByLocationAndAttributes(address, attributeBasedAssignmentCriteria, includeSubGroups,  
currentGroup)
```

This methods is a combination of the `assignUserByLocation` and `assignByUserAttributes` methods. You can use it apply both kinds of restrictions simultaneously. In a similar fashion to the `assignUserByLocation` method, you can use this method in situations in which the assignment needs to take a location into account. It then allows imposition of additional restrictions, such as the ability to handle large dollar amounts or foreign languages, for example.

The following example searches for a French speaker that is closest to a given address.

```
var attributeBasedAssignmentCriteria = new AttributeBasedAssignmentCriteria()  
var frenchSpeaker = new AttributeCriteriaElement()  
frenchSpeaker.AttributeType = UserAttributeType.TC_LANGUAGE  
frenchSpeaker.AttributeValue = "french"  
attributeBasedAssignmentCriteria.addToAttributeCriteria( frenchSpeaker )  
  
activity.CurrentAssignment.assignUserByLocationAndAttributes(Activity.Account.PrimaryLocation.Address,  
attributeBasedAssignmentCriteria , false, activity.CurrentAssignment.AssignedGroup)
```

Proximity-Based Assignment

Guidewire provides a number of proximity-based assignment methods. Each method provides a slightly different functionality. The following table lists these assignment methods and describes their differences.

Method	Description
assignUserByLocationUsingProximitySearch	Does not work with attributes. Searches the Group tree. See “assignUserByLocationUsingProximitySearch” on page 110.
assignUserByLocationUsingProximityAndAttributes	Works with multiple attributes. Searches the Group tree. See “assignUserByLocationUsingProximityAndAttributes” on page 111.
assignUserByProximityWithSearchCriteria	Works with a single attribute only. Uses proximity search. See “assignUserByProximityWithSearchCriteria” on page 111.
assignUserByProximityWithAssignmentSearchCriteria	Works with a single attribute only. Uses proximity search. See “assignUserByProximityWithAssignmentSearchCriteria” on page 113.

Note: All the proximity-based assignment methods are functions defined on the `CCAssignable` delegate interface.

IMPORTANT To use the proximity assignment methods, you must integrate ClaimCenter with a geocoding server and geocode all users. For more information on proximity searches and geocoding, see “Geocoding and Proximity Searches for Vendor Contacts” on page 99 in the *Contact Management Guide*.

Choose the proximity-based assignment method that suits your particular business need. Guidewire recommends that you review the Gosu API Reference material associated with each method if you choose to implement it. To access the Gosu API Reference, select it from the Studio Help menu. You can use the search functionality to find a specific assignment method. For example, entering *proximity* into the search field returns a number of proximity-related items.

assignUserByLocationUsingProximitySearch

```
boolean assignUserByLocationUsingProximitySearch(Address, includeSubGroups, currentGroup)
```

This method assigns the entity the user with the closest address to the specified address, as measured by a great-circle distance along the surface of the earth. To use this method, you must first geocode all user addresses in the database. You must also geocode the supplied address that serves as the center of the search, either in advance or at the time of the search. If the geocoding attempt fails for the supplied location, the method logs an error and returns `false`. (It also returns `false` if it unable to make an assignment.)

The `assignUserByLocationUsingProximitySearch` method performs the following search algorithm:

1. For each user in the specified group, the method computes the distance from each user location to the supplied location. (It uses the primary address of the Contact entity for each user.) The search begins with the users of the current group (`currentGroup`). It is an error if there is no current group.
2. If the `includeSubGroups` parameter is `true`, the method repeats this process with all of the descendant groups of the specified group.
3. The method then returns the user who is closest to the specified location. It does not perform round-robin assignment among the set of users.

This method takes the following parameters:

Parameter	Description
address	An address to use as the center of the search.
includeSubGroups	if true, then include users in any subgroups of the supplied group, and the users in the current group.
currentGroup	The group whose members to consider for assignment. This cannot be null.

The following example assigns a claim to someone that is closest (in distance) to the claimant's primary address.

```
Claim.assignUserByLocationUsingProximitySearch(Claim.claimant.PrimaryAddress, true,
                                              Claim.AssignedGroup)
```

assignUserByLocationUsingProximityAndAttributes

```
boolean assignUserByLocationUsingProximityAndAttributes(address, attributeBasedAssignmentCriteria,
                                                       includeSubGroups, currentGroup)
```

This method is similar to the `assignUserByLocationAndAttributes` method, except that it also takes user attributes into account in the search algorithm. See “`assignUserByLocationAndAttributes`” on page 109.

The method uses a distance calculation and one or more attributes to select the closest user. It does not perform round-robin assignment among the set of users. You use this method, therefore, if you want to find the closest user with certain attributes.

This method takes the following parameters:

Parameter	Description
address	The location to use as the center of the search.
attributeBasedAssignmentCriteria	The user attributes to match against.
includeSubGroups	if true, then include users in any subgroups of the supplied group, and the users in the current group.
currentGroup	The group that includes the users to consider for the assignment

The following example searches for a French speaker that is closest to a given address. The example code first creates a new `AttributeCriteriaElement` object and assigns it a value of French. It then adds the new object to the `attributeBasedAssignmentCriteria` object and uses that object as part of the search criteria. The assignment method then tries to assign the activity to a French speaker that closest, in straight-line distance, to the primary address of the claimant on a claim.

```
var attributeBasedAssignmentCriteria = new AttributeBasedAssignmentCriteria()
var frenchSpeaker = new AttributeCriteriaElement()
frenchSpeaker.AttributeType = UserAttributeType.TC_LANGUAGE
frenchSpeaker.AttributeValue = "french"
attributeBasedAssignmentCriteria.addToAttributeCriteria( frenchSpeaker )

Activity.CurrentAssignment.
    assignUserByLocationUsingProximityAndAttributes(Activity.Claimant.Person.PrimaryAddress,
                                                   attributeBasedAssignmentCriteria,
                                                   false,
                                                   Activity.CurrentAssignment.AssignedGroup )
```

assignUserByProximityWithSearchCriteria

```
public boolean assignUserByProximityWithSearchCriteria(usc, cap, includeSubGroups, currentGroup)
```

This method assigns the entity to a user based on a *user search by proximity*. It uses the following search algorithm:

1. The method first geocodes the location that serves as the center of the search (if it is not already geocoded). You can then access this location through the following code:

```
usc.getContact().getProximitySearchCriteria()
```

2. The method compiles a list of users who satisfy the user search criteria. Use the specified cap to limit the number of users in this list. If you include proximity restrictions on the search, the method discards the users farthest from the search center.
3. The method uses the round-robin algorithm to pick one of the found users. This means that repeated, identical calls to this method rotate through the resulting set of users to find the user to return. The method bases the round-robin rotation on the exact UserSearchCriteria that you supply. Guidewire recommends that you use as general a location as possible (for example, a city, state or postal code, rather than a specific street address). This maximizes the benefit of the round-robin processing and reduces the load on the system.

This method takes the following parameters:

Parameter	Description
usc	An object of type UserSearchCriteria that sets the starting location for the proximity search. This cannot be null. The UserSearchCriteria object has a foreign key to a ContactSearchCriteria entity, which, in turn, has a foreign key to Address.
cap	Integer value that determines the number of users to return. <ul style="list-style-type: none"> • If greater than zero, it sets the maximum number of users to include in the round-robin assignment algorithm from the search results. • If zero or less, the method includes all users from the search results in the round-robin assignment algorithm. To always assign the entity to the closest user, set cap to 1.
includeSubGroups	if true, then include users in any subgroups of the supplied group, and the users in the current group. The method ignores this parameter if the currentGroup parameter is null.
currentGroup	The group whose members to consider for assignment. This can be null. If it is non-null, then the method uses it as part of the search.

IMPORTANT This assignment method can be very slow compared to other assignment methods. Guidewire recommends that you consider its performance implications before using it.

Setting Up the Proximity Search

To avoid errors in setting up the proximity search, Guidewire recommends that you use the following method:

```
gw.api.geocode.GeocodeScriptHelper.setupUserProximitySearch
```

This method sets up a number of specific fields on the search criteria. The GeocodeScriptHelper method returns a UserSearchCriteria object that other proximity assignment methods can use.

The GeocodeScriptHelper method has the following signature and parameters:

```
setupUserProximitySearch(searchCenter, isDistanceBased, number, unitOfDistance )
```

Parameter	Description
searchCenter	The location to use as the center of the search. This is an Address object.
isDistanceBased	<ul style="list-style-type: none"> • Distance-based searches – Set to true if you want to perform a distance-based search. This type of search finds users within n miles (kilometers) of the searchCenter location. • Ordinal-based searches – Set to false if you want to perform an ordinal-based search. This type of search finds the nearest n users to the searchCenter location (computed as the straight-line distance).

Parameter	Description
number	<ul style="list-style-type: none"> Distance-based searches – The search radius in miles or kilometers for distance-based searches Ordinal-based searches – The maximum number of results to return for an ordinal search.
unitOfDistance	<ul style="list-style-type: none"> Distance-based searches – Indicates the unit (miles or kilometers) to use for distance-based searches. Ordinal-based searches – Determines how ClaimCenter displays the results, ignored otherwise. <p>This parameter takes its value from UnitOfDistance typelist.</p> <ul style="list-style-type: none"> UnitOfDistance.TC_KILOMETER UnitOfDistance.TC_MILE

Distance-based Proximity Assignment Example

The following example sets up the proximity information to use in the search using the `Claim.LossLocation` as the supplied address, starting with a search radius of 10 miles. This is a distance-based search as the `isDistanceBased` parameter is set to `true`. Notice that this code also logs whether the assignment succeeded or failed.

```
uses gw.api.geocode.GeocodeScriptHelper

// Set up a distance-based proximity search for claim assignment
if (claim.LossLocation <> null) {

    var usc = GeocodeScriptHelper.setupUserProximitySearch(claim.LossLocation, true, 10,
        UnitOfDistance.TC_MILE)

    if (claim.CurrentAssignment.assignUserByProximityWithSearchCriteria(usc, -1, true,
        claim.AssignedGroup)) {
        actions.exit() //Needed if using this method within the context of the assignment rules
    } else {
        print ("#####: Assigning by proximity to Loss Location = " + claim.LossLocation + ": FAILED")
    }
} else {
    print ("#####: claim.LossLocation is NULL ...")
}
```

Ordinal-based Proximity Assignment Example

The following example sets up the proximity information to use in the search using a supplied address. This is a ordinal-based search as the `isDistanceBased` parameter is set to `false`. It returns the closest three users to the supplied address.

```
uses gw.api.geocode.GeocodeScriptHelper

// Set up a ordinal-based proximity search for claim assignment
var usc = GeocodeScriptHelper.setupUserProximitySearch(claim.LossLocation, false, 3,
    UnitOfDistance.TC_MILE)

//Set user city and postal code of search center location

usc.Contact.Address.City = claim.LossLocation.City
usc.Contact.Address.State = claim.LossLocation.State
usc.Contact.Address.PostalCode = claim.LossLocation.PostalCode

if (claim.CurrentAssignment.assignUserByProximityWithSearchCriteria(usc, -1, true,
    claim.AssignedGroup)) {
    actions.exit() //Needed if using this method within the context of the assignment rules
}
```

[assignUserByProximityWithAssignmentSearchCriteria](#)

```
boolean assignUserByProximityWithAssignmentSearchCriteria(asc, cap, includeSubGroups, currentGroup)
```

This method is nearly identical to the `assignUserByProximityWithSearchCriteria` method, except that it uses an `AssignmentSearchCriteria` object rather than a `UserSearchCriteria` object. See that method for more information. See “`assignUserByProximityWithSearchCriteria`” on page 111.

This method takes the following parameters:

Parameter	Description
asc	An object of type gw.api.assignment.AssignmentSearchCriteria that sets the starting location for the proximity search and the user attribute to match.
cap	Integer value that determines the number of users to return. <ul style="list-style-type: none"> If greater than zero, it sets the maximum number of users to include in the round-robin assignment algorithm from the search results. If 0 or less, the method includes all users from the search results in the round-robin assignment algorithm.
includeSubGroups	if true, then include users in any subgroups of the supplied group, and the users in the current group. The method ignores this parameter if the currentGroup parameter is null.
currentGroup	The group whose members to consider for assignment. This can be null. If it is non-null, then the method uses it as part of the search.

IMPORTANT This assignment method can be very slow compared to other assignment methods. Guidewire recommends that you consider its performance implications before using it.

The following example attempts to assign a claim to a French speaker within 10 mile radius of the claim loss location. Notice that the code first uses the CCGeocodeScriptHelper.setupAssignmentProximitySearch method to set up the parameters for the proximity search. It then sets up the French-speaking attribute on the CCUserSearchCriteria object.

```
uses gw.api.geocode.CCGeocodeScriptHelper

// Set up an distance-based proximity search for claim assignment
if (claim.LossLocation <> null) {

    var asc = CCGeocodeScriptHelper.setupAssignmentProximitySearch(Claim.LossLocation, true, 10,
        UnitOfDistance.TC_MILE )

    // Set the custom attribute match.
    asc.CCUUserSearchCriteria.AttributeName = "French"
    asc.CCUUserSearchCriteria.AttributeValue = 1

    if (claim.assignUserByProximityWithAssignmentSearchCriteria(asc, -1, true,
        claim.CurrentAssignment.AssignedGroup )) {
        actions.exit() //Needed if using this method within the context of the assignment rules
    } else {
        print ("#####: Assigning by proximity to Claimant Address = " + Claim.LossLocation + ": FAILED")
    }
} else {
    print ("#####: Claim.LossLocation is NULL ...")
}
```

Round-robin Assignment

The round-robin algorithm rotates through a set of users, assigning work to each in sequence. It is important to understand that round-robin assignment is distinct from workload-based assignment. You can use load factors (in some circumstances) to affect the frequency of assignment to one user or another. However, round-robin assignment does not take the current number of entities assigned to any of the users into account.

Note: See “Secondary (Role-based) Assignment” on page 98 for a discussion on secondary assignment and round-robin states.

assignUserByRoundRobin

```
boolean assignUserByRoundRobin(includeSubGroups, currentGroup)
```

This method uses the round-robin user selector to choose the next user from the current group or group tree to receive the assignable. If the includeSubGroups parameter is true, the selector then performs round-robin assignment not only through the current group, but also through all its subgroups. To give a concrete example,

suppose that you have a set of claims that you want to assign to a particular group. If you want all of the users within the group to share the workload, then you set the `includeSubGroups` parameter to `true`.

The following example assigns an activity to the next user in a set of users in a group.

```
Activity.CurrentAssignment.assignUserByRoundRobin( false, Activity.AssignedGroup )
```

Dynamic Assignment

Dynamic assignment provides a generic hook for you to implement your own assignment logic, which you can use to perform automated assignment under more complex conditions. For example, you can use dynamic assignment to implement your own version of “load balancing” assignment.

There are two dynamic methods available, one for users and the other for groups. Both the user- and group-assignment methods are exactly parallel, with the only difference being in the names of the various methods and interfaces.

```
public boolean assignGroupDynamically(dynamicGroupAssignmentStrategy)
public boolean assignUserDynamically(dynamicUserAssignmentStrategy)
```

These methods take a single argument. Make this argument a class that implements one of the following interfaces:

```
DynamicUserAssignmentStrategy
DynamicGroupAssignmentStrategy
```

Interface Methods and Assignment Flow

The `DynamicUserAssignmentStrategy` interface defines the following methods. (The Group version is equivalent.)

```
public Set<Object> getCandidateUsers(assignable, group, includeSubGroups)
public Set<Object> getLocksForAssignable(assignable, candidateUsers)
public GroupUser findUserToAssign(assignable, candidateGroups, locks)
boolean rollbackAssignment(assignable, assignedEntity)
Object getAssignmentToken(assignable)
```

The first three methods are the major methods on the interface. Your implementation of these interface methods must have the following assignment flow:

1. Call `DynamicUserAssignmentStrategy.getCandidateUsers`, which returns a set of assignable candidates.
2. Call `DynamicUserAssignmentStrategy.getLocksForAssignable`, passing in the set of candidates. It returns a set of entities for which rows in the database you must lock.
3. Open a new database transaction.
4. For each entity in the set of locks, lock that row in the transaction.
5. Call `DynamicUserAssignmentStrategy.findUserToAssign`, passing in the two sets generated in step 1 and step 2 previously. That method returns a `GroupUser` entity representing the user and group that you need to assign.
6. Commit the transaction, which results in the lock entities being updated and unlocked.

Dynamic assignment is not complete after these steps. Often, such as during FNOL intake or creating a new claim in a wizard, ClaimCenter performs assignment and updates workload information well before it saves the claim. If ClaimCenter cannot save the claim, the database still shows the increase in the user’s workload. The interface methods allow for the failure of the commit operation by adding one last final step.

7. If the commit fails, roll back all changes made to the user information, if possible. If this is not possible, save the user name and reassign that user to the assignable item later, during a future save operation.

Implementing the Interface Methods

Any class that implements the `DynamicUserAssignmentStrategy` interface (or the Group version) must provide implementations of the following methods:

- `getCandidateUsers`
- `getLocksForAssignable`
- `findUserToAssign`
- `rollbackAssignment`
- `getAssignmentToken`

getCandidateUsers

Your implementation of the `getCandidateUsers` method must return the set of users to consider for assignment. (As elsewhere, the `Group` parameter establishes the root group to use to find the users under consideration. The Boolean `includeSubGroups` parameter indicates whether to include users belonging to descendant groups, or only those that are members of the parent group.)

getLocksForAssignable

The `getLocksForAssignable` method takes the set of users returned by `getCandidateUsers` and returns a set of entities that you must lock. By locked, Guidewire means that current machine obtains the database rows corresponding to those entities (which must be persistent entities). Any other machine that needs to access these rows must wait until the assignment process finishes. Round-robin and dynamic assignment require this sort of locking to mandate that multiple machines do not perform simultaneous assignments. This ensures that multiple machines do not perform simultaneous assignments and assign multiple activities (for example) to the same person, instead of progressing through the set of candidates.

findUserToAssign

Your implementation of the `findUserToAssign` method must perform the actual assignment work, using the two sets of entities returned by the previous two methods. (That is, it takes a set of users and the set of entities for which you need to lock the database rows and performs that actual assignment.) This method must do the following:

- It makes any necessary state modifications (such as updating counters, and similar operations).
- It returns the `GroupUser` entity representing the selected User and Group.

Make any modifications to items such as load count, for example, to entities in the bundle of the assignable. This ensures that ClaimCenter commits the modifications at the same time as it commits the assignment change.

rollbackAssignment

Guidewire provides the final two API methods to deal with situations in which, after the assignment flow, some problem in the bundle commit blocks the assignment. This might happen, for example, if a validation rule caused a database rollback. However, the locked objects have already been updated and committed to the database (as in step 6 in the assignment flow).

If the bundle commit does not succeed, ClaimCenter calls the `rollbackAssignment` method automatically. Construct your implementation of this method to return `true` if it succeeds in rolling back the state numbers, and `false` otherwise.

In the event that the assignment does not get saved, you have the opportunity in your implementation of this method to re-adjust the load numbers.

getAssignmentToken

If the `rollbackAssignment` method returns `false`, then ClaimCenter calls the `getAssignmentToken` method. Your implementation of this method must return some object that you can use to preserve the results of the

assignment operation. The basic idea is that in the event that it is not possible to commit an assignment, your logic does one of the following:

- ClaimCenter rolls back any database changes that have already been made.
- ClaimCenter preserves the assignment in the event that you invoke the assignment logic again.

Sample DynamicUserAssignmentStrategy Implementation

As a very simple implementation of this API, Guidewire includes the following in the base configuration:

```
gw.api.LeastRecentlyModifiedAssignmentStrategy
```

The following code shows the implementation of the `LeastRecentlyModifiedAssignmentStrategy` class. This is a very simple application of the necessary concepts needed to create a working implementation. The class performs a very simple user selection, simply looking for the user that has gone the longest without modification.

Since the selection algorithm needs to inspect the user data to do the assignment, the class returns the candidate users themselves as the set of entities to lock. This ensures that the assignment code can work without interference from other machines.

```
package gw.api.assignment.examples

uses gw.api.assignment.DynamicUserAssignmentStrategy
uses java.util.Set
uses java.util.HashSet

@Export
class LeastRecentlyModifiedAssignmentStrategy implements DynamicUserAssignmentStrategy {

    construct() { }

    override function getCandidateUsers(assignable:Assignable, group:Group, includeSubGroups:boolean) :
        Set {
        var users = (group.Users as Set<GroupUser>).map( \ groupUser -> groupUser.User )
        var result = new HashSet()
        result.addAll( users )
        return result
    }
    override function findUserToAssign(assignable:Assignable, candidates:Set, locks:Set) : GroupUser {
        var users : Set<User> = candidates as Set<User>
        var oldestModifiedUser : User = users.iterator().next()
        for (nextUser in users) {
            if (nextUser.UpdateTime < oldestModifiedUser.UpdateTime) {
                oldestModifiedUser = nextUser
            }
        }
        return oldestModifiedUser.GroupUsers[0]
    }
    override function getLocksForAssignable(assignable:Assignable, candidates:Set) : Set {
        return candidates
    }

    //Must return a unique token
    override function getAssignmentToken(assignable:Assignable) : Object {
        return "LeastRecentlyModifiedAssignmentStrategy_" + assignable
    }

    override function rollbackAssignment(assignable:Assignable, assignedEntity:Object) : boolean {
        return false
    }
}
```

Manual Assignment

Manual assignment is slightly different from other assignment scenarios. With manual assignment, you need to flag the assignable entity for manual assignment by someone else. As such, a call to one of these methods does

not assign the assignable itself. Instead, it creates a new assignment activity for the specified user to perform (such as to review or to assign the item).

Note: To be eligible to receive a manual assignment activity, you must assign a role to the group member that has the Review Assignment (actreviewassign) permission. In the ClaimCenter base configuration, the Claim Supervisor, New Loss Processing Supervisor, Manager, and Superuser roles all have this permission.

Guidewire recommends, before attempting to assign the assignable entity, that you test if it is possible to perform the assignment manually. Use the following property on the assignable entity to determine the owning claim for the entity.

```
entity.OwningClaim
```

If the return value is `null`, it is not possible to manually assign this entity because it is not possible to create an assignment review activity for it. Otherwise, the method returns the owning claim object.

Guidewire provides the manual-based assignment methods described in the following topics:

- “`assignManually`” on page 118
- “`confirmManually`” on page 118
- “`assignManuallyByRoundRobin`” on page 118
- “`confirmManuallyByRoundRobin`” on page 119

assignManually

```
boolean assignManually(responsibleUser)
```

This method designate a user to look at the assignable and manually assign it to someone. Use this method to flag the assignable for manual review and assignment. If you call this method, ClaimCenter does not actually assign the assignable entity *until* the designated user takes action to do so.

The following example assigns the claim to the group supervisor to assign manually.

```
Claim.CurrentAssignment.assignManually( Claim.CurrentAssignment.AssignedGroup.Supervisor )
```

confirmManually

```
boolean confirmManually(responsibleUser)
```

This method requests manual confirmation of the assignable entity's current assignment. It is roughly equivalent to `assignManually`, except that it assumes that a provisional assignment has already been made. ClaimCenter does not consider the assignment to be final until someone reviews and approves it.

Note: Items that you mark for manual confirmation appear in the **Pending Assignment** list of the supervisor of the assigned group.

The following example first assigns a claim provisionally to a user using round-robin assignment, then creates an activity for that user's supervisor to review the assignment and approve it.

```
Claim.CurrentAssignment.assignUserByRoundRobin( false, Claim.CurrentAssignment.AssignedGroup )
Claim.CurrentAssignment.confirmManually( Claim.CurrentAssignment.AssignedGroup.Supervisor )
```

assignManuallyByRoundRobin

```
boolean assignManuallyByRoundRobin(group)
```

This method is similar to `assignManually`, except that it shifts the responsibility for manual assignment to various members of the specified group using round-robin assignment. Use this method, for example, if the workload of manual assignment is too large for a single user. You would use this method to distribute the workload among members of a group.

For example, you might have a team of supervisors responsible for assigning certain sensitive tasks to a set of Claims adjusters. You want manual assignment because you want the supervisors to select someone manually. But, you also want to rotate the responsibility for making assignment through the group of supervisors. This is

different from the standard round-robin assignment, which rotates the assignment itself. This method rotates the designation of the person responsible for completing the manual assignment.

The following example moves the assignment responsibility among members of a particular group.

```
Claim.CurrentAssignment.assignManuallybyRoundRobin( Claim.CurrentAssignment.AssignedGroup)
```

confirmManuallyByRoundRobin

```
boolean confirmManuallyByRoundRobin(group)
```

This method is similar to `confirmManually`, except that it determines the responsibility for assignment confirmation through round-robin assignment among users in the specified group. Again, you would use a method of this type if the workload of assignment confirmation is too large for a single user.

The following example first assigns a claim to a user using round-robin assignment through members of a group. It then assigns responsibility for assignment confirmation to a member of that group.

```
Claim.CurrentAssignment.assignUserByRoundRobin( false, Claim.CurrentAssignment.AssignedGroup)
Claim.CurrentAssignment.confirmManuallybyRoundRobin( Claim.CurrentAssignment.AssignedGroup)
```

Using Assignment Methods in Assignment Pop-ups

In Guidewire ClaimCenter, you typically reassign an existing entity through an **Assignment** popup screen. This screen is usually two-part:

- You use the upper part to select from a pre-populated list of likely assignees, including (for example) the Claim owner. It also includes an option to perform rule-based assignment.
- You use the lower part of the popup to search for a specific assignee.

You can run assignment methods directly, from outside of the Assignment engine. Therefore, it is possible to modify the upper part of this popup to call a specific assignment method without invoking the Assignment engine. This requires both new Gosu code and some PCF configuration.

One possibility is the following:

1. Create a new Gosu class, implementing the `gw.api.assignment.Assignee` interface. Use this class to perform your business logic to assign the passed-in `Assignable` object. The following is an example of this.

```
package gw.api

class ExtensionAssignee implements gw.api.assignment.Assignee {

    construct()

    override function assignToThis(assignableBean : com.guidewire.pl.domain.assignment.Assignable) {
        var users = gw.api.database.Query.make(User).compare("PublicID", Equals, userID)
        var usr = users.select().AtMostOneRow
        assignableBean.autoAssign(usr.GroupUsers[0].Group, usr )
    }

    function toString() : String {
        return "Bakeriffic Assignment"
    }
}
```

2. Modify the PCF files to include this new `Assignee` object, removing others as necessary. One option, for example, would be to modify the assignment popup to add a new method in the Code section, such as the following. You can then reference it in the `valueRange` section of the assignment widget in place of the call to `SuggestedAssignees`.

```
function getSuggestedAssignees() : gw.api.assignment.Assignee[] {
    var assignees = Claim.SuggestedAssignees
    assignees[0] = new gw.api.ExtensionAssignee()
    return assignees
}
```

Assignment by Workload Count

In many cases, a ClaimCenter user can be a member of multiple work teams or groups. This means that you can assign a user to a claim review activity as a member of one group. You can then also assign that user a claim as a member of a different group. Therefore, in determining workload assignments, it is important to take the total (global) number of items assigned to an individual into account.

To this end, ClaimCenter provides a count of the total number of activities, claims, exposures, and matters assigned to an individual. You can access these total counts both through the ClaimCenter interface and through fields on the User object in Gosu. (To access the counts in the Team tab, you must be the supervisor of a group.)

Access through the Team Tab

The Team tab within ClaimCenter displays item counts for each user who is part of that team, organized into columns by each item type (claim, activity, and so on). However, these numbers are simply the item counts for that user in that group and do not include items assigned to that user as a member of another group. To identify the total number of items assigned to a user, ClaimCenter displays a global total for an item type in parentheses next to the subtotal number. You must be an administrator or a group supervisor to access the Team tab.

Access through Gosu

The User and Group objects have fields that you can access through Gosu to determine the total number of items of a given type assigned to an individual or group. The fields are:

- OpenActivityCount
- OpenClaimCount
- OpenExposureCount
- OpenMatterCount

For example, the following Gosu code returns the total number of activities across all groups for the current owner of a claim:

```
var total = Claim.AssignedUser.OpenActivityCount
```

Note: ClaimCenter updates these global numbers hourly while running the statistics batch process. For details of batch processes, see “Batch Processes and Work Queues” on page 123 in the *System Administration Guide*.

Validation in ClaimCenter

This topic describes rule-based validation in Guidewire ClaimCenter. It also describes the validation graph.

This topic includes:

- “What is Validation?” on page 121
- “Overview of ClaimCenter Validation” on page 122
- “The Validation Graph in Guidewire ClaimCenter” on page 123
- “Validation Performance Issues” on page 126
- “Debugging the Validation Graph” on page 127

What is Validation?

Note: The terms *object* and *entity*, while not identical, are used interchangeably throughout this section. See “Important Terminology” on page 15 for a description of each.

ClaimCenter runs preupdate and validation rules every time that it commits data to the database. A commit to the database is called a *database bundle commit*. The validation rules execute after ClaimCenter runs all preupdate callbacks and preupdate rules. ClaimCenter runs the validation rules as the last step before ClaimCenter actually writes data to the database.

Before applying rules during the bundle commit operation, ClaimCenter builds a validation object graph according to configuration settings. The graph contains possible targets for rule execution and is used by both the preupdate and validation rules.

Preupdate rules are different from validation rules, in that they can modify additional objects during execution. If a preupdate rule does modify additional objects, these objects might need to be constructed after the preupdate rules run and before the validation rules run.

If the preupdate rules modify an entity that is not in the commit bundle...

ClaimCenter can possibly run additional validation rules if the entity being modified also triggers validation.

If the preupdate rules do not modify any entities other than those that are already in the commit bundle...

ClaimCenter runs the validation rules for the same set of entities on which the preupdate rules ran.

If, however, the preupdate rules modify something that would simply trigger validation for one of the top-level entities...

There is no noticeable difference from those validation rules, as they were already going to be run anyway.

Therefore, the set of entities that are validated can be a superset of the entities for which preupdate rules are run. The preupdate rules are non-recursive, however, meaning that they do not run a second time on objects modified during execution of the preupdate rules. ClaimCenter simply adds any objects modified by the preupdate rules to the list of objects needing validation as described by the validation graph.

Overview of ClaimCenter Validation

For an entity to have preupdate or validation rules associated with it, the entity must be *validatable*. To be validatable, the entity must implement the `Validatable` delegate using `<implementsEntity name="Validatable"/>` in the entity definition. In the base configuration, Guidewire ClaimCenter provides a number of high-level entities that trigger validation, meaning that they are validatable. The following list describes the validatable entities, with the ClaimCenter entities listed first and platform entities following:

- `Claim`
- `Exposure`
- `Matter`
- `Policy`
- `RIAgreement`
- `RITransaction`
- `RITransactionSet`
- `TransactionSet` (and all its subclasses)
- `Activity`
- `Contact`
- `Group`
- `Region`
- `User`

See also

- “`<implementsEntity>`” on page 200 in the *Configuration Guide*

Entity Validation Order

ClaimCenter evaluates entities in ClaimCenter in the order that is set through configuration parameter `EntityValidationOrder`. You use this parameter to set a comma-separated list of the validatable entities in the order that you want ClaimCenter to perform validation. If you do not explicitly change this parameter, ClaimCenter performs entity validation using the default order, which is (in the base configuration):

- `Policy`
- `Claim`
- `Exposure`
- `Matter`
- `TransactionSet`

ClaimCenter validates all other validatable entities not specified in the list *after* all listed entities in configuration parameter `EntityValidationOrder`, in no particular order. Guidewire does not, however, guarantee that ClaimCenter validates entities of a given type (such as the exposures on a claim) in any deterministic order with respect to one another.

The Validation Graph in Guidewire ClaimCenter

During database commit, ClaimCenter performs validation on the following:

- Any validatable entity that is itself either updated or inserted.
- Any validatable entity that refers to a entity that is updated, inserted, or removed.

ClaimCenter gathers the entities that reference a changed entity into a virtual graph. This graph maps all the paths from each type of entity to the top-level validatable entities like `Claim` and `Exposure`. ClaimCenter queries these paths in the database or in memory to determine which validatable entities (if any) reference the entity that was inserted, updated, or removed.

ClaimCenter determines the validation graph by traversing the set of foreign keys and arrays that trigger validation. For example, suppose the data model marks the `Exposures` array on `Claim` as triggering validation. Therefore, any changes made to an exposure causes ClaimCenter to validate the claim as well.

ClaimCenter follows foreign keys and arrays that triggers validation through any links and arrays on the referenced entities down the object tree. For example, you might end up with a path like `Claim` → `ClaimContact` → `ContactAddress` → `Address`. (To actually trigger validation, each link in the chain—`Address`, `ContactAddress`, and `ClaimContact`—must be marked as triggering validation, and `Claim` must be marked as validatable.)

ClaimCenter stores this path in reverse in the validation graph. Thus, if an address changes, ClaimCenter does the following to find any claims that references a changed address. ClaimCenter transverses the tree from address to contact address. It then moves to the claim contact, and finally to claim (`Address` → `ContactAddress` → `ClaimContact` → `Claim`).

Traversing the Validation Graph

If a entity is validatable, ClaimCenter applies the preupdate and validation rules any time that you modify the entity contents directly. Suppose, as described previously, you update an object linked to an object. For example, a foreign key links each of the following objects to the `User` object:

- `Contact`
- `Credential`
- `UserSettings`

If you update a user's credential, you might expect the preupdate and validation rules to execute before ClaimCenter commits the updates to the database. However, updating a user's credentials does not normally trigger rules to the container object, `User` in this case. The reason is that implementation at the metadata level of a user's credential is a pointer from a `User` entity to a `Credential` entity.

The standard way to trigger preupdate and validation rules on linked objects is by setting the `triggersValidation` attribute of the foreign key to the object in the metadata XML file. For example, in `User.eti`, you see the following:

```
<entity xmlns="http://guidewire.com/datamodel" ... entity="User">
  ...
  <implementsEntity name="Validatable"/>
  ...
  <foreignkey columnName="ContactID"
    desc="Contact entry related to the user."
    fkentity="UserContact"
    name="Contact"
    ...
    triggersValidation="true"/>
  <foreignkey columnName="CredentialID"
    desc="Security credential for the user."
    fkentity="Credential"
    name="Credential"
    ...
    triggersValidation="true"/>
  ...
</entity>
```

Setting the `triggersValidation` attribute to `true` ensures that ClaimCenter runs the preupdate and validation rules on the linked object any time that you modify it. (This is only true if the container object implements the `Validatable` delegate.)

The `triggersValidation` attribute works in the opposite direction from what you might assume. If `triggersValidation="true"` on a link (foreign key or array) from entity A to entity B, then a change to B triggers validation in A. For instance, imagine that `triggersValidation` is set to `true` on the array of `ClaimContact` objects on the `Policy` object (which it is by default). That means a change to a `ClaimContact` also causes the ClaimCenter to validate the `Policy` object.

In the base configuration, Guidewire sets the validation triggers so that modifying a validatable entity in a bundle causes ClaimCenter to validate that entity and all its revisioning parents.

The `triggersValidation` attribute is also available on foreign keys and arrays that have been added to custom subtypes, custom entities, or to core entities using extensions. ClaimCenter defaults this attribute to `false` if you do not set it specifically. You must specifically change the attribute to `true` if adding a new foreign key or array that you want to trigger validation to a validatable object.

Owned Arrays

ClaimCenter treats owned arrays (arrays attached to a validatable entity) and marked as `triggersValidation` in a special fashion. ClaimCenter performs validation based on the validation graph. If an entity changes, ClaimCenter follows all the arrays and foreign keys that reference that entity and for which `triggersValidation="true"`. It then walks up the graph to find the validatable entities that ultimately references the changed entity. In the case of owned arrays, changes to the array cause ClaimCenter to consider the parent entity to be changed.

Consider, for example, the following data model:

- Entity A has a foreign key to B
- Entity B has an owned array of C
- Array C has a foreign key to D

Suppose that both A and B are validatable. Essentially, these relationships looks like:

A → B → C[] → D

Suppose that you also mark both $A \rightarrow B$ and $C \rightarrow D$ as triggering validation. What happens with the $B \rightarrow C$ link?

- If the $B \rightarrow C$ array is marked as `triggersValidation="false"`, changes to C cause ClaimCenter to validate B and A. This is because ClaimCenter treats the change as a change to B directly. A change to D, however, does not cause ClaimCenter to validate anything—the graph stops at C. As C is not actually changed, ClaimCenter does not consider B to have changed and performs no validation.
- If $B \rightarrow C$ is marked as `triggersValidation="true"`, changes to either C or D cause ClaimCenter to validate both B and A.

Top-level Entities that Trigger Validation

To be validatable, an entity (or subtype, or delegate, or base object) must implement the `Validatable` delegate by adding the following to the entity definition:

```
<implementsEntity name="Validatable"/>
```

The following table lists the entities that trigger validation in the base ClaimCenter configuration, meaning that all these entities implement the `Validatable` delegate. If a table cell is empty, there are no additional entities associated with that top-level entity that trigger validation in the base configuration.

Validatable entity	Foreign key entity	Array name	One-to-one entity
Activity			
Claim	Address ClaimWorkComp EmploymentData Policy PolicyLocation	Activities ConcurrentEmpl Contacts ContribFactors Documents Evaluations Exposures Incidents Matters MetroReports Negotiations Notes Officials OtherBenefits RIAgreementGroups RoleAssignments SIAnswerSet Text	ClaimMetricRecalculationTime PropertyFireDamage PropertyWaterDamage SubrogationSummary
Contact			
Exposure	Address Benefits Coverage EmploymentData IMEPerformed Incident RIAgreementGroup StatCode	ContactAddresses BenefitPeriods Documents IMEPerformed MedicalActions Notes OtherCoverageDet Roles RoleAssignments Settlements Text	
Group			
Matter		Roles StatusTypeLines	
Policy		ClassCodes Contacts Covgeries Endorsements PolicyLocations RiskUnits Roles StatCodes	
Region			
RIAgreement			
RITransaction	RITransactionSet		
RITransactionSet		RITransactions	
TransactionSet			
User	Credential UserContact UserSettings	Attributes	

In many cases, an entity several levels down triggers validation on one of the top-level entities. For example, Claim preupdate and validation rules run after you update the address for a contact. The `Contact` entity definition contains a `triggersValidation` foreign key attribute to the `Address` entity. The `Claim` entity also has a `triggersValidation` foreign key relationship with the `Address` entity. In this way, ClaimCenter follows the chain of triggering validations during a database commit.

Overriding Validation Triggers

In the base configuration, Guidewire sets validation to trigger on many top-level entities and on many of the arrays and foreign keys associated with them. You cannot modify the base metadata XML files in which these configurations are set. However, there can be occasions in which you want to override the default validation behavior. For example:

- You want to trigger validation upon modification of an entity in the ClaimCenter base data model that does not currently trigger validation.
- You do not want to trigger validation on entities on which ClaimCenter performs validation in the base configuration.

You can override validation only on certain objects associated with a base configuration entity. It is not necessary to override your own extension entities as you simply set `triggersValidation` to the desired value as you define the entity.

The following tables lists the data objects for which you can override validation, the XML override element to use, and the location of additional information.

Data field	Override element	See
<code><array></code>	<code><array-override></code>	" <code><array></code> " on page 185 in the <i>Configuration Guide</i>
<code><foreignkey></code>	<code><foreignkey-override></code>	" <code><foreignkey></code> " on page 197 in the <i>Configuration Guide</i>
<code><onetoono></code>	<code><onetoono-override></code>	" <code><onetoono></code> " on page 203 in the <i>Configuration Guide</i>

See “Working with Attribute Overrides” on page 213 for details on working with these override elements.

Validation Performance Issues

There are three ways in which validation can cause performance problems:

- The rules themselves are too performance intensive.
- There are too many objects being validated.
- The queries used to determine which objects to validate take too long.

The following sections discuss the various performance issues.

Administration Objects

Guidewire recommends that you never mark foreign keys and arrays that point to administration objects—such as `User`, `Group`, and `Catastrophe` objects—as triggering validation. An *administration object* is basically any object that can be referenced by a large number of claims. For example, suppose that you set the `catastrophe` field on `Claim` as triggering validation. Then, it would be possible that editing one catastrophe can bring the entire application to a crawl as ClaimCenter validates hundreds or even thousands of claims. At the very least, it would make editing those objects nearly impossible.

Query Path Length

In some cases, an entity can have a large number of foreign keys pointing at it. Triggering validation on the entity can cause performance problems, as ClaimCenter must follow each of those chains of relationships during validation. The longer the paths through the tables, the more expensive the queries that ClaimCenter executes anytime that an object changes. Having a consistent direction for graph references helps to avoid this.

Triggering validation on parent-pointing foreign keys on entities that have many possible owners, like `ClaimContact`, can result in much longer and unintended paths. For example, a number of entities point to

`ClaimContact`, including `Claim`, `Check`, and `MedicalTreatment`, to name a few. Each of these entities then contains links to other entities. Validating the entire web of relationships can have a serious negative performance impact.

Guidewire designs the ClaimCenter default configuration to minimize this issue. However, heavy modification of the default configuration using validation trigger overrides can introduce unintended performance issues. To debug performance issues, see “Debugging the Validation Graph” on page 127.

Links Between Top-level Objects

As previously described, it is legal to have top-level entities trigger validation on each other. This, however, increases the number of paths and the number of objects that ClaimCenter must validate on any particular commit.

Graph Direction Consistency

In general, Guidewire strongly recommends that you consistently order the validation graph to avoid the previously described problems.

- Arrays and foreign keys that represent some sort of containment are candidates for triggering validation. For instance, you can logically consider contacts and vehicles on a claim as part of the claim.
- Foreign keys that you reference as arrays or that are merely there to indicate association are not considered candidates for triggering validation. For instance, the `matter` on a `ClaimContact` merely indicates association rather than indicating that the matter is logically part of the contact.

Illegal Links and Arrays

Virtual foreign keys, since they are not actually in the database, cannot be set to trigger validation. Attempting to do so causes an error to be reported at application startup. Similarly, any array or link property defined at the application level that is not in the database, such as the `Driver` link on `Claim`, cannot be set as triggering validation.

An array is considered to be in the database if the foreign key that forms the array is in the database.

Debugging the Validation Graph

To see a text version of the validation graph, open the file `logging.properties` by navigating in Studio to the **Other Resources → logging** folder, and then add the following entry:

```
log4j.category.com.guidewire.pl.system.bundle.validation.ValidationTriggerGraphImpl=DEBUG
```

This entry causes the validation graph to print as text in the system console, but only after application deployment, and after you have opened any claim in that deployment.

See also

- For more information on application logging, see “Configuring Logging” on page 25 in the *System Administration Guide*.

Detecting Claim Fraud

This topic discusses how you can use Guidewire ClaimCenter to detect claim fraud and how to create Gosu business rules to trigger a special investigation of a suspicious claim.

This topic includes:

- “Claim Fraud” on page 129
- “The Special Investigation Details Screen” on page 130
- “Special Investigation Trigger Rules” on page 130
- “SI Rule Evaluation Sequence” on page 130
- “Using the Special Investigation Rules” on page 131

Claim Fraud

Guidewire provides a number of sample rules to assist in determining claim fraud, which you can use to trigger a special investigation of a suspicious claim. These rules identify certain characteristics of a claim that increase the suspicion that fraud has occurred and assign points to each of these characteristics. Depending on your business logic, you can:

- Assign different point values to these fraud characteristics based on their “severity”
- Set fraud triggers based on the presence of data, the absence of data, or the value of certain data

Using these rules, you can identify a point threshold after which the claim merits human review. A human reviewer (typically, the claim owner’s supervisor) can then determine whether to assign the claim to a Fraud or Special Investigation team for further investigation.

Using business rules to trigger claim fraud investigations provides a number of benefits, including:

- Providing a standardization of the process
- Enforcing business processes across the organization
- Assigning standardized weight to every characteristic evaluated
- Providing transparency of process
- Providing a consistent evaluation of all claims, rather than using “gut feel” or individual prejudices

- Keeping the trail of why and how claims were reviewed

Many of these benefits can be important from a legal perspective.

The Special Investigation Details Screen

Within Guidewire ClaimCenter, you use the **Special Investigation Details** screen to view and track details of suspicious claims. This screen contains:

- The list of conditions (Special Investigation triggers) which this claim violates (as a read-only list)
- A SI score field, which is a “counter” that tracks the SI points accumulated on this claim
- Fields to track escalation of the claim to the fraud detection or Special Investigation (SI) team

You control what a specific user can access and view through ClaimCenter permission settings.

Special Investigation Trigger Rules

SI trigger rules flag suspicious information (or lack of information) on claims. For example, the following all trigger SIU rules:

- All the telephone fields for the contact on the exposure are null.
- No police report or on-scene report was recorded in the claim file.
- There was an unreasonable delay in reporting the loss (for example, if more than 10 days elapsed between the loss date and the claim report date).

During evaluation of the SI trigger rules, if a trigger rule evaluates to `true` and the rule has not yet been logged on the claim, then ClaimCenter:

- Adds the rule to the SI Triggers array
- Increments the SI score by the value specified in the rule actions

When the SI score reaches a defined threshold, ClaimCenter creates an activity for the claim handler’s supervisor to review this particular claim. You can control this threshold value (the default is 5) through a script parameter called `SpecialInvestigation_CreateActivityForSupervisorThreshold`.

Upon receiving the activity, a claim supervisor reviews the contents of the **SI Details** subtab and details of the claim. The supervisor can choose to escalate the claim to the SI team. If the supervisor does escalate the claim to the SI team, ClaimCenter does the following:

- It routes the activity to a member of the SI group by round-robin assignment
- It adds that person automatically to the claim in the role of SIU investigator

If the claim already has an associated investigator, ClaimCenter sends the activity to that individual.

SI Rule Evaluation Sequence

The following sequence of events occurs anytime that you modify a claim or exposure.

1. The user modifies a claim or exposure, causing ClaimCenter to run the Claim Preupdate Rules rules. See “Claim Preupdate Rules” on page 65.
2. ClaimCenter evaluates the SI trigger rules and then determines whether the rule conditions have been met and whether a particular trigger has already been logged on the claim. If not, it writes the trigger to the SI Triggers array and increments the SI score field on the claim in ClaimCenter.

3. ClaimCenter determines if the current SI score exceeds the threshold value set by script parameter `SpecialInvestigation_CreateActivityForSupervisorThreshold`. If so, it creates a Special Investigations review activity for the supervisor of the claim owner. If not, it starts the cycle again.
4. Upon examination, the claim supervisor decides whether to escalate the claim (within ClaimCenter) to the Special Investigation Unit.
5. Once ClaimCenter creates an SI escalation activity for the SI team, it assigns the activity to the SI investigator assigned to the claim if there is one. If a SI investigator is not currently assigned to this claim, then ClaimCenter does the following:
 - It calls the SI assignment rules to assign the SI escalation activity to the SI group.
 - It then uses round-robin assignment to assign the activity to a SI team member.
6. Finally, ClaimCenter adds the SI escalation activity owner to the claim (in the role of SIU Investigator).

Using the Special Investigation Rules

It is necessary to evaluate the SI trigger rules at the correct stage of the claim's life cycle. For example, it is not possible to truly evaluate at the first report of the claim whether the claim is missing the police report or whether there are no witnesses. It is possible that this information is only available after a certain number of days have passed.

To handle these situations, the SI rules break the claim life cycle into stages. Different ClaimCenter rule sets handle different stages in the claim life cycle. The following list describes the order in which the rule sets run.

Rule set	Rules	SIU Action
Claim Preupdate Rules	Set SIU Life Cycle	See "Set the Initial Life Cycle Stage" on page 131
Claim Exception Rules	Setting SIU Life Cycle State	See "Advance the Life Cycle Stage" on page 132
Claim Preupdate Rules Exposure Preupdate Rules	SIU Life Cycle Stage 1 SIU Life Cycle Stage 2 SI Triggers	See "Evaluate SIU Triggers" on page 132
Claim Preupdate Rules Default Group Assignment Rules (Activity)	Create Supervisor Review Activity Assign claim review activity to supervisor	See "Escalate Claim to Supervisor" on page 133
Global Assignment Rules (Activity) Default Group Assignment Rules (Activity)	Assign SIU escalation activity to SIU group Assign Escalation Activity to named SIU user Default SI Escalation activity routing	See "Assign Activity to SIU Group" on page 133
Activity Postsetup Rules	Add user in SIU Role to claim if necessary	See "Add Assigned User in SIU Role" on page 133

Set the Initial Life Cycle Stage

ClaimCenter uses claim property `SIULifeCycleState` to track a claim's stage. The Claim Preupdate rule `Set SIU Life Cycle` sets the initial claim stage to 1 (step1) the first time that ClaimCenter runs the preupdate rules against a claim. ClaimCenter uses the value of a claim's state to determine which SIU rules to run against the claim. See "Claim Preupdate Rules" on page 65.

Rule Conditions

```
Claim.SimpleClaimTimer == null and Claim.State != "draft"
```

Rule Actions

```
Claim.SIULifeCycleState="step1"
```

Advance the Life Cycle Stage

A rule in the Exception rule set category advances the claim stage based on the number of days which have passed since claim inception. See “Exception Rule Set Category” on page 52.

ClaimCenter automatically runs the claim exception rules on a daily basis, in the base configuration, at 2:00 a.m. server time. Claim exception rule Setting SIU Life Cycle State calls ClaimCenter library function `SettinSIULifeCycleState`, which actually sets the state. The following table lists the default configuration for the various stages. You can customize this process by setting new values in the library function.

Stage	Number of days since creation
1	Less than 5
2	More than 5 and less than 20
3	More than 20

Note: ClaimCenter evaluates claims not touched today on the standard rule schedule in the Exception rule set category.

Evaluate SIU Triggers

Each time that ClaimCenter runs the Claim Preupdate rules for a claim, ClaimCenter evaluates the rules tied to the relevant stage. See “Claim Preupdate Rules” on page 65.

For example, suppose that ClaimCenter pushes a claim to stage 2 during the evening’s Exception process. Then, the next time that ClaimCenter evaluates the Claim Preupdate rules, it will also evaluate the SIU rules associated with stage 2.

Guidewire ships several sample stage 1 SIU rules. For example, these rules add SIU points under the following conditions:

- The claimant filed the claim more than 10 days after the reported loss date.
- The claimant filed the claim within days of the policy issuance.

Guidewire ships several sample stage 2 SIU rules. For example, these rules add SIU points under the following conditions:

- The police report is not available for this claim.
- The driver was a minor who is not listed on the policy.

In addition, ClaimCenter runs several sample Exposure Preupdate SIU rules anytime that an exposure is updated or validated. These rules add SIU points under the following conditions:

- The exposure contains no telephone number for the claimant.
- The exposure contains a P.O. Box address as the primary address for the claimant.

Each preupdate SIU rule calls an enhancement method, one each for claims and exposures, that determines whether ClaimCenter has already evaluated this condition. If not, ClaimCenter adds the specified number of points to the total SIU score.

Note: In the base configuration, all ClaimCenter SIU rules add one (1) point each to the overall SI score. However, you can modify this behavior to meet your business needs.

Escalate Claim to Supervisor

In the base configuration, ClaimCenter provides an SIU script parameter named `SpecialInvestigation_CreateActivityForSupervisorThreshold`, which is set to a value of 5. ClaimCenter continually compares the overall SIU value (score) as set by the various SIU trigger rules with the value of `SpecialInvestigation_CreateActivityForSupervisorThreshold`. If the SIU score goes above this threshold value, a Claim Preupdate rule, SI - Create Supervisor Review Activity, creates an activity for a supervisor to review the suspicious claim. This, in turn, triggers execution of special SIU activity assignment rules:

- A Global activity assignment rule, Assign Claim Review to Claim Owner's Group, assigns the claim review activity to the claim owner's group.
- A Default Group activity rule, SI - Assign Claim Review Activity to Supervisor, assigns the review activity to the supervisor of the group.

The group supervisor then reviews the claim and determines whether it requires further action.

Note: You set the initial value for a script parameter in Guidewire ClaimCenter. Thereafter, you modify this value through the administration interface in ClaimCenter. See “Script Parameters” on page 105 in the *Configuration Guide*.

See also

- “Claim Preupdate Rules” on page 65

Assign Activity to SIU Group

If desired, the reviewing supervisor can escalate the claim (within ClaimCenter) to a Special Investigation group to investigate the claim further. ClaimCenter provides additional assignment rules to assist in this process, including the following:

Assignment Rule	Task
Assign Escalation Activity to named SIU user	If there is a Special Investigator associated with the claim, assign the SI escalation activity to the person with that role.
Assign SIU escalation activity to SIU group	If there is no Special Investigator associated with the claim, assign the SI escalation activity to the SI group.
Default SI Escalation activity routing	Use round-robin assignment to assign the SI escalation activity to a member of the SI group.

Add Assigned User in SIU Role

After the ClaimCenter assignment rules assign an escalation activity to a member of the Special Investigation team, the Rule engine runs an additional Activity Postsetup rule. This rule associates the user with the suspect claim.

Assignment Rule	Task
Add user in SIU Role to claim if necessary	Determine if the assigned user is associated with the claim in the role of SIU Investigator. If not, add this user to the claim in that role.

See “Activity Postsetup Rules” on page 57.

Sending Emails

This topic describes how to send email messages from Guidewire ClaimCenter.

This topic includes:

- “Guidewire ClaimCenter and Email” on page 135
- “The Email Object Model” on page 136
- “Email Utility Methods” on page 136
- “Email Transmission” on page 137
- “Understanding Email Templates” on page 137
- “Creating an Email Template” on page 139
- “Localizing an Email Template” on page 139
- “The IEmailTemplateSource Plugin” on page 140
- “Configuring ClaimCenter to Send Emails” on page 140
- “Sending Emails from Gosu” on page 143
- “Saving an Email Message as a Document” on page 143

Guidewire ClaimCenter and Email

The Guidewire platform includes support for sending emails from ClaimCenter. You can access this capability from any Gosu code. For example, you can access email functionality from Gosu rules or in Gosu embedded in the ClaimCenter PCF screens.

ClaimCenter provides the following email functionality:

- Support for various types of email recipients (To, CC, and BCC)
- Support for templates that can be used to populate the subject and body of the email
- Support for attaching documents stored in the configured DMS (Document Management System) to the email message

Because email messages are sent using the same integration infrastructure as event-based messages, you can use the same administrative tools for monitoring the status of messages. You can view unsent messages in the ClaimCenter Administration interface.

The Email Object Model

Guidewire ClaimCenter uses the following two classes to define email messages:

- `gw.api.email.Email`
- `gw.api.email.EmailContact`

Both these classes are simple data containers with almost no logic.

`gw.api.email.Email`

The `Email` class contains the following fields, most of which are self-explanatory:

Field	Description
<code>Subject</code>	Subject of the email
<code>Body</code>	Body of the email
<code>Sender</code>	<code>EmailContact</code>
<code>ReplyTo</code>	<code>EmailContact</code> (It is possible for this to be different from the <code>Sender</code> .)
<code>ToRecipients</code>	List of <code>EmailContacts</code>
<code>CcRecipients</code>	List of <code>EmailContacts</code>
<code>BccRecipients</code>	List of <code>EmailContacts</code>
<code>Documents</code>	List of <code>DocumentBase</code> entities to be attached to the email

`gw.api.email.EmailContact`

The `EmailContact` class contains three fields:

Field	Description
<code>Name</code>	Name of contact
<code>EmailAddress</code>	Email address of contact
<code>Contact</code>	Contact entity, which can be <code>null</code> . If this is set, it sets the <code>Name</code> and <code>EmailAddress</code> fields to the appropriate values from the specific Contact entity.

Email Utility Methods

Besides the `Email` and `EmailContact` classes, Guidewire also provides a set of static utility methods in the `gw.api.email.EmailUtil` class for generating and sending emails from Gosu:

```
gw.api.email.EmailUtil.sendEmailWithBody( KeyableBean entity, Email email )
gw.api.email.EmailUtil.sendEmailWithBody( KeyableBean entity,
                                         Contact to,
                                         Contact from,
                                         String subject,
                                         String body )
gw.api.email.EmailUtil.sendEmailWithBody( KeyableBean entity,
                                         String toEmailAddress,
                                         String toName,
                                         String fromEmailAddress,
                                         String fromName,
                                         String subject,
                                         String body )
```

All three methods take an entity as the first parameter. This parameter can be `null`. However, if specified, use the application entity to which this email is related, such as a specific claim, exposure, or activity. ClaimCenter uses this parameter only while processing the email for transmission. See “Email Transmission” on page 137.

Emails that Use an Email Object

This variation of the `sendEmailWithBody` method requires that you create a `gw.api.email.Email` entity, and then define its properties to build the `Email` entity. For example:

```
...
var testEmail : gw.api.email.Email
testEmail.Body = "This is a test."
testEmail.Subject = "Test"
...
gw.api.email.EmailUtil.sendEmailWithBody( thisClaim, testEmail)
```

Emails that Use Contact Objects

The second variation of the `sendEmailWithBody` method uses `Contact` objects for the `to` and `from` parameters. In Gosu, you can obtain `Contact` objects from various places. For example, in a claim rule, to send an email from an insurance company employee to the insured, do the following:

- Set the `to` parameter to `Claim.insured`.
- Set `from` to `Claim.AssignedUser.Contact`.

The following Gosu example generates an email from the current assigned user to that user’s supervisor:

```
gw.api.email.EmailUtil.sendEmailWithBody(thisClaim, thisClaim.AssignedGroup.Supervisor.Contact,
                                         thisClaim.AssignedUser.Contact, "A claim got a ClaimValid event", "This is the text." )
```

Emails that Use an Email Address

Use the following variation of the `sendMailWithBody` method if you do not have a full `Contact` object for a recipient or sender. The contact might have been generated dynamically through some other application. The `sendMailWithBody` method uses a name and email address instead of entire `Contact` records and does not require that you have access to a `Contact` record. In the following example, all arguments are `String` objects:

```
gw.api.email.EmailUtil.sendEmailWithBody( Entity, toName, toEmail, fromName, fromEmail, subject, body)
```

Email Transmission

Guidewire ClaimCenter, from the user’s perspective, sends emails asynchronously by using the ClaimCenter Messaging subsystem. If there is a method call for one of the `EmailUtil.sendEmail` methods, ClaimCenter creates a `Message` entity with the contents and other information from the `Email` object.

- If the `entity` parameter is non-null, ClaimCenter adds the `Message` entity to the `entity` bundle. ClaimCenter persists the `Message` entity any time that it creates the bundle.
- If the `entity` parameter is `null`, ClaimCenter persists the `Message` entity immediately.

You must configure a `MessageTransport` class to consume the email Messages and do the actual sending. ClaimCenter processes messages one at a time, and sends out the emails associated with that message. For more information, see “Configuring ClaimCenter to Send Emails” on page 140.

Understanding Email Templates

You use email templates to create the body of an email message. Unlike Document templates (but like Note templates), ClaimCenter performs no Gosu interpretation on the Email templates themselves. Thus, email templates are generally only suitable for boilerplate text that does not require modification, or for presenting in

the application interface as a starting point for further modification. You cannot use Email templates for mail-merge-style operations.

WARNING Do not add, modify, or delete files from any directory other than the `modules/configuration` application directory. Otherwise, you can cause damage to ClaimCenter.

Email Template Files

An email template consists of two separate files:

- A descriptor file, whose name must end in `.gosu.descriptor`, which contains some metadata about the template.
- A template file, whose name must end in `.gosu`, which contains the desired contents of the email body.

IMPORTANT The names of the descriptor and template files must match.

An email descriptor file contains the following fields:

Field	Description
Name	The name of the template
Topic	The topic of the template (a String value)
Keywords	A list of keywords (which can be used to search the templates)
Subject	The subject of the emails created using this template
Body	The body of the emails created using this template

For example, `EmailReceived.gosu.descriptor` defines an *Email Received* descriptor file:

```
<?xml version="1.0" encoding="UTF-8"?>
<serialization>
  <emailtemplate-descriptor
    name="Email Received"
    keywords="email"
    topic="reply"
    subject="Email Received"
    body="EmailReceived.gosu"
  />
</serialization>
```

The `EmailReceived.gosu.descriptor` file pairs with the actual template file (`EmailReceived.gosu`):

Thank you for your correspondence. It has been received and someone will contact you shortly to follow up on your feedback.

Sincerely,

By default, email templates live in the following location in Studio:

[Other Resources](#) → [emailtemplates](#)

See also

For general information on templates, how to create them, and how to use them, see:

- “Gosu Templates” on page 347 in the *Gosu Reference Guide*
- “Data Extraction Integration” on page 607 in the *Integration Guide*

Creating an Email Template

ClaimCenter stores all email template files (both descriptor and template files) in the following location:

Other Resources → emailtemplates

To create a new email template

1. Navigate to Other Resources → emailtemplates, right-click, and select Other file from the New menu.
2. For the template file, do the following:
 - a. Enter the name of the email template and add the .gosu extension.
 - b. Enter the body of the email template in the view tab that opens. This file defines the text of the email message to send. For example:

Greetings:
Please contact <%= activity.AssignedUser %> at <%= activity.AssignedUser.Contact.WorkPhone %>
in regards to <%= activity.Subject %>
Thank you for your patience while we resolve this issue.
Sincerely,
3. For the descriptor file, do the following:
 - a. Enter the name of the template descriptor file and add the .gosu.descriptor extension.
 - b. Enter the body of the descriptor file in the view tab that opens. This file defines metadata about the template. For example, the following partial template descriptor file defines the email subject line and specifies the file to use for the email body text. For example:

```
<emailtemplate-descriptor  
    name="Request for XYZ"  
    keywords="activity, email"  
    topic="request"  
    subject="We require XYZ for &lt;%= activity.Subject %&gt;"  
    body="NeedXYZ.gosu"  
    requiredsymbols="Activity"  
/>
```

See also

- “Understanding Email Templates” on page 137
- “Creating an Email Template” on page 139
- “Localizing an Email Template” on page 139
- “Gosu Templates” on page 347 in the *Gosu Reference Guide*
- “Data Extraction Integration” on page 607 in the *Integration Guide*

Localizing an Email Template

Localizing an email template is generally a straight-forward process. To localize an email template:

- Create a locale folder for the template files in following location:

Other Resources → emailtemplates → *locale-folder*
- Within the locale folder, place localized versions of the email template file and its associated template descriptor file.

To use a localized email template in ClaimCenter, you must perform a search for the localized template as you create a new email.

See also

- “Localizing Templates” on page 57 in the *Globalization Guide*
- “Creating Localized Documents, Emails, and Notes” on page 58 in the *Globalization Guide*

The `IEmailTemplateSource` Plugin

In the base configuration, Guidewire defines an `IEmailTemplateSource` plugin implementation that provides a mechanism for Guidewire ClaimCenter to retrieve one or more Email templates. You can then use these templates to pre-populate email content. (The method of the `EmailTemplateSource` plugin is similar to that of `INoteTemplateSource` plugin.) You configure the `IEmailTemplateSource` plugin as you would any other plugin.

Class `LocalEmailTemplateSource`

In the base configuration, the `IEmailTemplateSource` plugin implements the following class:

```
gw.plugin.email.impl.LocalEmailTemplateSource
```

This default plugin implementation constructs an email template from files on the local file system. Therefore, it is not suitable for use in a clustered environment.

Class `LocalEmailTemplateSource` provides the following locale-aware method for searching for an email template. It returns an array of zero, one, or many `IEmailTemplateDescriptor` objects that match the locale and supplied values.

```
getEmailTemplates(locale, valuesToMatch)
```

These parameters have the following meanings:

<code>Locale</code>	Locale on which to search.
<code>valuesToMatch</code>	Values to test. You can include multiple values to match against, including: <ul style="list-style-type: none">topicnamekey wordsavailable symbols

Configuring ClaimCenter to Send Emails

To configure Guidewire ClaimCenter to send emails, you must first define an email server *destination* to process the emails. In the base configuration, ClaimCenter defines an *email* destination with ID 65. In ClaimCenter, a destination ID is a unique ID for each *external system* defined in the **Messaging** editor in Studio. Exactly one messaging destination uses the built-in email transport plugin, that destination has one ID associated with it, and that ID is 65.

To view the definition of the message destination in Studio, navigate to the following location in Studio:

Messaging → **email**

You cannot change the ID for this message destination. You can, however, set other messaging parameters as desired. For information on the **Messaging** screen, see “Using the Messaging Editor” on page 137 in the *Configuration Guide*.

You also need to configure the base configuration messaging transport that actually sends the email messages. The ClaimCenter Gosu email APIs send emails with the built-in email transport. In the Studio **Plugins** editor, this plugin has the name `emailMessageTransport`.

To view or modify the `emailMessageTransport` plugin, navigate to the following location:

Plugins → **gw** → **plugin** → **messaging** → **MessageTransport** → `emailMessageTransport`

This plugin can implement one of the following classes or you can create your own message transport class:

- `EmailMessageTransport`
- `JavaxEmailMessageTransport`

This plugin has several default parameters that you need to modify for your particular configuration.

Configuration Parameter	Description
<code>SMTPHost</code>	Name of the SMTP email application. ClaimCenter attempts to send mail messages to this server. For example, this might be <code>EmailHost1.acmeinsurance.com</code> .
<code>SMTPPort</code>	SMTP email port. Email servers normally listen for SMTP traffic on port 25 (the default), but you can change this if necessary.
<code>defaultSenderName</code>	Provides a default name for outbound email, such as <code>XYZ Company</code> .
<code>defaultSenderAddress</code>	Provides a default <i>From</i> address for out-bound email. This indicates the email address to which replies are sent.

ClaimCenter uses the default name and address listed as the sender if you do not provide a *From Contact* in the Gosu that generates the email message. Otherwise, Studio uses the name and primary email address of the From Contact.

See also

- For information on events, message acknowledgements, the types of messaging plugins, and similar topics, see “Messaging and Events” on page 299 in the *Integration Guide*.

Class `EmailMessageTransport`

In the base configuration, ClaimCenter defines a `gw.plugin.email.impl.EmailMessageTransport` Gosu class that provides the following useful methods (among others):

```
createHtmlEmailAndSend(wkSmtpHost, wkSmtpPort, email)
createEmail(wkSmtpHost, wkSmtpPort, email) : HtmlEmail
```

These parameters have the following meanings:

<code>wkSmtpHost</code>	SMTP host name to use for sending mail
<code>wkSmtpPort</code>	SMTP host port number
<code>email</code>	Email object

Method `createHtmlEmailAndSend` simply calls method `createEmail` with the required parameters. Method `createEmail` returns then an HTML email object that the calling method can then send. Method `createEmail` constructs the actual email documents for the email.

Error Handling

In the base configuration, the `emailMessageTransport` implementation has the following behavior in the face of transmission problems:

- If the mail server configuration is incorrectly set within ClaimCenter itself, then ClaimCenter does not send any messages until you resolve the problem. This ensures that ClaimCenter does not lose any emails.
- If some of the email addresses are bad, then ClaimCenter skips the emails with the bad addresses.
- If all of the email address are bad, then ClaimCenter marks the message with an error message and skips that particular message. ClaimCenter reflects this skip in the message history table.

Class JavaxEmailMessageTransport

In the base configuration, ClaimCenter defines a `gw.plugin.email.impl.JavaxEmailMessageTransport` Gosu class that provides the following useful methods (among others):

```
createHtmlEmailAndSend(wkSmtpHost, wkSmtpPort, email)
populateEmail(out, email)
```

These parameters have the following meanings:

<code>wkSmtpHost</code>	SMTP host name to use for sending mail
<code>wkSmtpPort</code>	SMTP host port number
<code>email</code>	Email object
<code>out</code>	MimeMessage object

In the base configuration, the `createHtmlEmailAndSend` method on `JavaxEmailMessageTransport` does the following:

- It sets the SMTP host name and port.
- It retrieves the default `Session` object. (If one does not exist, it creates one.) In the base configuration, this method does not restrict access to the `Session` object. In other words, there is no authentication set on this session.
- It creates a new `MimeMessage` object (using the default session) and passes it to the `populateEmail` method.
- It throws a messaging exception if any exception occurs during the operation.

The `populateEmail` method takes the `MimeMessage` object and creates a container for it that is capable of holding multiple email bodies. Use standard `javax.mail.Multipart` methods to retrieve and set the subparts.

Note: There are many reasons why there can be different versions of an email from the same information. (Locale is not one of those reasons as ClaimCenter localizes email information before writing it to the message queue.) For example, you can split an email that exceeds some maximum size into multiple emails. Or, you can generate one email body for internal users and another, different, email body for external users.

The method then adds the following to the mime object:

- Sender
- Headers
- Recipients
- Subject
- Documents (if any)
- Email body

The `populateEmail` method returns an `Email` object that you can then send.

Authentication Handling

In the base configuration, the `JavaxEmailMessageTransport.createHtmlEmailAndSend` method does not provide authentication on the `Session` object. If you want to provide user name and password authentication for email messages, then you must do one of the following:

- Modify the `JavaxEmailMessageTransport.createHtmlEmailAndSend` method to provide authentication. In the base configuration, ClaimCenter sets the `Authenticator` parameter in the following session creation method to `null` (meaning none).

```
Session.getDefaultInstance(props, null)
```

If you want to add authentication, then you must create and add your own `Authenticator` object to this method call.

- Modify the `JavaxEmailMessageTransport.createHtmlEmailAndSend` method to use the standard `javax.mail.internet.MimeMessage` encryption and signature methods.

Working with Email Attachments

By default, the `emailMessageTransport` plugin interacts with an external document management system as the *system user* during retrieval of a document attached to an email. It is possible, however, to retrieve the document on behalf of the user who generated the email message instead. To do this, you need to set the `UseMessageCreatorAsUser` property in the `emailMessageTransport` plugin.

To set the `UseMessageCreatorAsUser` property

1. In Studio, navigate to the following location:

`Configuration → Plugins → gw → plugin → messaging → MessageTransport → emailMessageTransport`

2. In the **Parameters** area, click **Add** to create a new parameter entry.

3. Enter the following:

Name	<code>UseMessageCreatorAsUser</code>
Value	<code>true</code>

Sending Emails from Gosu

Note: Creating an email message and storing it in the Send Queue occurs as part of the same database transaction in which the rules run. This is the same as regular message creations triggered through business rules.

To send an email from Gosu, you need to first create the email, typically through the use of email templates (described in “Understanding Email Templates” on page 137). You then need to send the email using one of the `sendEmailWithBody` methods described in “Email Utility Methods” on page 136.

Saving an Email Message as a Document

ClaimCenter does not store email objects created through the `Email` class in the database. However, it is possible to save the contents, recipient information, and other information of an email as a document in the configured DMS application. Because the `sendEmailXXX` methods all take all the sending information explicitly, you can save the email information by using the following process in Gosu code:

1. Create the email subject and body, and determine recipients.
2. Send the email by calling the appropriate `EmailUtil` method.
3. If ClaimCenter does not encounter an exception, create a document recording the details of the sent email.

Create Document from Email Example

See “Document Creation” on page 145 for details of the `DocumentProduction` class methods.

```
// First, construct and send the email
var toEmailAddress : String = "Recipient email address"
var toName = "Recipient's Name"
var fromEmailAddress : String = "Sender email address"
var fromName : String = "Sender's name"
var subject : String = "Email Subject"
var body : String = "Email Body"
gw.api.email.EmailUtil.sendEmailWithBody( null, toEmailAddress, toName, fromEmailAddress, fromName,
                                         subject, body )

// Next, create the document recording the email
var contextObjects = new java.util.HashMap()
```

```
contextObjects.put("To", toEmailAddress)
contextObjects.put("From", fromEmailAddress)
contextObjects.put("Subject", subject)
contextObjects.put("Body", body)
...
var document : Document = new Document()
document.Name = "EmailSent"

// Set other properties as needed
var template : gw.plugin.document.IDocumentTemplateDescriptor
var templateName = "EmailSent.gosu"
...

// Call this method to create and store the document for later retrieval
gw.document.DocumentProduction.createDocumentSynchronously(template, contextObjects, document)
```

Document Creation

This topic describes synchronous and asynchronous document creation in Guidewire ClaimCenter. It briefly describes the integration points between a document management system and Guidewire ClaimCenter. (For detailed integration information, see “Document Management” on page 199 in the *Integration Guide*.) It also covers some of the more important document management APIs and document production classes.

This topic includes:

- “Synchronous and Asynchronous Document Production” on page 145
- “Integrating Document Functionality with ClaimCenter” on page 146
- “The `IDocumentTemplateDescriptor` Interface” on page 147
- “The `IDocumentTemplateDescriptor` API” on page 148
- “The `DocumentProduction` Class” on page 151
- “Document Templates” on page 154
- “Document Creation Examples” on page 154
- “Troubleshooting” on page 157

Synchronous and Asynchronous Document Production

Guidewire ClaimCenter supports several different types of document creation:

- *Synchronous* document creation completes *immediately* after you initiate it.
- *Asynchronous* document creation completes at a *future* time after you initiate it.

ClaimCenter uses an `IDocumentProduction` plugin to manage document creation. Guidewire also provides a Gosu helper class with a number of public `createDocumentXX` methods to facilitate working with document creation.

In the context of `IDocumentProduction` plugin, *synchronous* versus *asynchronous* refers to the perspective of ClaimCenter. In other words, after a `createDocumentXX` method call returns, did the integrated document production application create the document already (synchronous creation)? Or, is the `IDocumentProduction` implementation responsible for future creation and storage of the document (asynchronous creation)?

The following table restates the differences between synchronous and asynchronous document creation:

Type	Document contents
Synchronous	Generated immediately and returned to the calling method for further processing. In this scenario, the caller assumes responsibility for persisting the document to the Document Management system (if desired).
Asynchronous	Possibly not generated for some time, or possibly require extra workflow processing or manual intervention. The document creation system does not return the contents of the document, although it can return a URL or other information allowing for the checking of state. The <code>IDocumentProduction</code> implementation is responsible for adding the document to the Document Management system and notifying ClaimCenter upon successful creation of the document.

See also

For additional information regarding ClaimCenter document creation and management, see the following sections in the *ClaimCenter Integration Guide*:

- “Summary of All ClaimCenter Plugins” on page 181 in the *Integration Guide*
- “Document Management” on page 199 in the *Integration Guide*

Integrating Document Functionality with ClaimCenter

Note: For information on how to integrate document-related functionality with Guidewire ClaimCenter, see the information on plugins in “Document Management” on page 199 in the *Integration Guide*.

Implementing document-related functionality in a Guidewire application often requires integration with two types of external applications:

- Document Management Systems (DMS), which store document contents and metadata
- Document Production Systems (DPS), which create new documents

This integration involves the following main plugin interfaces, along with several minor ones. The following table summarizes information about the main plugin interfaces.

Interface	Used for...
<code>IDocumentContentSource</code>	Storage and retrieval of document contents. This is document <i>contents</i> only, with no metadata.
<code>IDocumentMetadataSource</code>	Storage and retrieval of document metadata. This is document <i>metadata</i> only, with no contents. Although many (if not most) DMS applications store both document contents and metadata about the documents, Guidewire provides two separate plugin interfaces. This separation is due to the different performance characteristics of the two kinds of data: <ul style="list-style-type: none"> • Document metadata is generally a collection of relatively short strings. Thus, you can usually implement remote transmission using SOAP interfaces. • Document contents are generally a large (up to many megabytes) chunk of data, often binary data, which cannot be easily or cheaply moved around between applications. However, Guidewire designs the interfaces such that (in some cases), you can use a single API call to work with both document contents and metadata. This occurs, for example, in the creation of a new document in which DMS stores the metadata and contents <i>atomically</i> so that either both exist or neither exists.
<code>IDocumentProduction</code>	Document creation. See “The <code>DocumentProduction</code> Class” on page 151.

Interface	Used for...
IDocumentTemplateDescriptor	Describe the templates used to create documents. See "The IDocumentTemplateDescriptor Interface" on page 147 for details of this interface.
IDocumentTemplateSource	<p>Retrieval of document templates, which are a part of document creation. In the base configuration, ClaimCenter provides a default implementation of the IDocumentTemplateSource plugin that retrieves the document templates from XML files stored on the server file system. This default implementation reads files from the application configuration module:</p> <p style="text-align: center;"><i>ClaimCenter/modules/configuration/config/resources/doctemplates</i></p> <p>If you desire to use a different path, then you need to supply the following parameters to specify an alternate location:</p> <ul style="list-style-type: none"> • <code>templates.path</code> • <code>descriptors.path</code> <p>If you choose to do this, Guidewire strongly recommends that you use absolute path names. Do not use relative path names as using a path relative to the document plugin directory can cause checksum problems at server start.</p> <p>The default implementation also checks the files system for template files if a user performs a search or retrieval operation.</p> <p>In general practice, however, Guidewire expects you to implement a storage solution that meets your business needs, which can include integration with a Document Management System.</p>

The following table summarizes information about the minor plugin interfaces (those used less frequently).

Interface	Used for...
IDocumentTemplateSerializer	<p>Customizing the reading and writing of IDocumentTemplateDescriptor objects. Use this sparingly.</p> <p>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 provide the ability to define the templates within simple XML files. This XML format is suitable for typical implementations.</p>
IPDFMergeHandler	Creation of PDF documents. You use this mainly to set parameters on the default implementation.

See also

- “Summary of All ClaimCenter Plugins” on page 181 in the *Integration Guide*
- “Document Management” on page 199 in the *Integration Guide*

The IDocumentTemplateDescriptor Interface

A *document template descriptor* works in conjunction with a document template, meaning a Microsoft Word MailMerge template, a PDF form, or a similar item. The descriptor file tells ClaimCenter how to populate the fields on the template. Within ClaimCenter, the IDocumentTemplateDescriptor interface defines the API that any object that represents a document template descriptor must implement.

A document template descriptor contains the following different kinds of information:

Category	Contains
Template Metadata	Metadata about the template itself (for example, the template ID, name, and similar items)
Document Metadata	Metadata defaults to apply to any documents created from the template (for example, the document status)

Category	Contains
Context Objects	Set of values that are referenced by the values to be inserted into the document template, known as <i>Context Objects</i> . This includes both default values and a set of legal alternative values for use in the document creation user interface.
Form Fields	Set of field names and values to insert into the document template, including some formatting information.
Document Locale	Locale in which to generate the document.

ClaimCenter stores all classes used by document plugins in the `Classes → gw → document` package. In the base configuration, this consists of the `DocumentProduction` class. This class relies heavily on `IDocumentTemplateDescriptor` objects in the creation of document objects.

The `IDocumentTemplateDescriptor` API

The `IDocumentTemplateDescriptor` API consists entirely of getter methods, with the addition of a single setter method (for `DateModified`). As a result, the following sections list the getter names and the return type information, broken into the following categories:

- Template Metadata
- Document Metadata
- Context Objects
- Form Fields
- Document Locale

Note: Many—but not all—of the `IDocumentTemplateDescriptor` getter methods exist as properties on `IDocumentTemplateDescriptor` object as well.

Template Metadata

The following list describes the getter methods associated with *template* metadata that the `IDocumentTemplateDescriptor` API manages.

getXXX method	Return type	Returns
<code>getDateEffective</code> <code>getDateExpiration</code>	Date	Effective and expiration dates for the template. If a user searches for a template, ClaimCenter displays only those templates for which the specified date falls between the effective and expiration dates. However, it is possible to use Gosu rules to create an <i>expired</i> template as Gosu-based document creation uses templates only. Do not attempt to use this as a mechanism for establishing different versions of templates with the same ID. All template IDs must be unique.
<code>getDateModified</code> <code> setDateModified</code>	Date	Date on which the template was last modified. In the base configuration, ClaimCenter sets this date from the information on the XML descriptor file itself. However, as both getter and setter methods exist for this property, it is possible to set this date through the <code>IDocumentTemplateSource</code> implementation.
<code>getDescription</code>	String	Human-readable description of the template or the document it creates.

getXXX method	Return type	Returns
getDocumentProductionType	String	If present, you can use this property to control which IDocumentProduction implementation to use to create a new document from the template. See “Document Management” on page 199 in the <i>Integration Guide</i> for information on implementing and configuring the IDocumentProduction plugin.
getIdentifier	String	Additional human-readable identifier for the template. This often corresponds to a well-known domain-specific document code. It can indicate, for example, to which state-mandated form this template corresponds.
getKeywords	String	Set of keywords that you can use in a search for the template.
getMetadataPropertyNames getMetadataPropertyValues	String[]	Method getMetadataPropertyNames returns the set of extra metadata properties that exist in the document template definition. You can use these properties in conjunction with the getMetadataPropertyValues method as a flexible extension mechanism. For example, you can add arbitrary new fields to document template descriptors. ClaimCenter then passes these fields onto the internal entities that it uses to display document templates in the interface. If the extra property names correspond to properties on the Document entity, ClaimCenter passes the values along to documents that it creates from the template.
getMimeType	String	Type of document that this document template creates. In the base configuration. You can also use this to determine which IDocumentProduction implementation to use to create documents from this template. For information on implementing and configuring the IDocumentProduction plugin, see “Document Management” on page 199 in the <i>Integration Guide</i> .
getName	String	A human-readable name for the template.
getPassword	String	If present, this property holds the password that ClaimCenter requires for the user to be able to create a document from this template. Not all document formats support this requirement. For example, Gosu templates do not support this functionality.
getRequiredPermission	String	Value corresponding to a type code from the SystemPermissionType typelist that ClaimCenter requires for a user to use this template. ClaimCenter does not display templates for which the user does not have the appropriate permission.
getScope	String	Contexts in which you can use this template. Possible values are: <ul style="list-style-type: none">• gosu – Indicates that you can only create this document template from Gosu code. These templates do not appear in the ClaimCenter interface• ui – Indicates you must only use this template from the ClaimCenter interface as it usually requires some kind of human interaction• all – Indicates that you can use this document template from any context
getTemplateId	String	Unique ID of the template. For most template types, this must be the same as the file name of the document template file itself (for example, ReservationRights.doc). One exception is InetSoft-based report templates, for which the templateId is the same as the name of the report to use to generate the document.

Document Metadata

The following list describes the getter methods associated with *document* metadata that the `IDocumentTemplateDescriptor` API manages.

getXXX method	Return type	Returns
<code>getTemplateType</code>	<code>String</code>	Value corresponding to a type code from the <code>DocumentType</code> typelist. If you create a document using this template, ClaimCenter sets the <code>type</code> field to this value. The XML descriptor file lists this property (and only this property) by a different name. The XML file lists this as <code>type</code> rather than as <code>templateType</code> .
<code>getDefaultSecurityType</code>	<code>String</code>	Value corresponding to a type code from the <code>DocumentSecurityType</code> typelist. If you create a document using this template, ClaimCenter sets the <code>securityType</code> field to this value.

Context Objects

The following list describes the getter methods associated with the *context objects* that the `IDocumentTemplateDescriptor` API manages.

getXXX method	Return type	Returns
<code>getContextObjectNames</code>	<code>String[]</code>	Set of context object names that the document template defines. See “The DocumentProduction Class” on page 151 for an example of how template descriptor files display context objects.
<code>getContextObjectType</code>	<code>String</code>	Type of the specified context object. Possible values include the following: <ul style="list-style-type: none">• <code>String</code>• <code>Text</code>• Bean – this means any entity type• Name of any entity type, for example <code>Claim</code>.
<code>getContextObjectAllowsNull</code>	<code>Boolean</code>	<code>true</code> if <code>null</code> is a legal value, <code>false</code> otherwise.
<code>getContextObjectDisplayName</code>	<code>String</code>	Human-readable name for the given context object. ClaimCenter displays this name in the document creation interface.
<code>getContextObjectDefaultValueExpression</code>	<code>String</code>	Gosu expression that evaluates to the desired default value for the context object. ClaimCenter uses this expression to set the default context object for the document creation interface, or as the context object value if it creates a document automatically.
<code>getContextObjectPossibleValuesExpression</code>	<code>String</code>	Gosu expression that evaluates to the desired set of legal values for the given context object. ClaimCenter uses these values to display a list of options for the user in the document creation interface.

Form Fields

The following list describes the getter methods associated with the *form fields* that the `IDocumentTemplateDescriptor` API manages.

getXXX method	Return type	Returns
<code>getFormFieldDisplayValue</code>	<code>String</code>	Value to insert into the completed document, given the field name and value. (The value, for example, can be the result of evaluating the expression returned from <code>getFormFieldValueExpression</code>). If desired, you can then implement some processing on the returned string, for example, substituting NA (not applicable) for null or formatting the dates correctly.
<code>FormFieldNames</code>	<code>String[]</code>	Set of form fields that the document template defines. See “The DocumentProduction Class” on page 151 for an example of how template descriptor files show form fields.
<code>getFormFieldValueExpression</code>	<code>String</code>	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. However, you can use any legal Gosu expression.

Document Locale

The following list describes the getter methods associated with the document *locale* that the `IDocumentTemplateDescriptor` API manages.

getXXX method	Return type	Returns
<code>getLocale</code>	<code>String</code>	Locale in which to create the document from this template descriptor. A return value of null indicates an unknown language. In most cases, the method returns the default language for the application.

The DocumentProduction Class

Guidewire Studio provides a helper `DocumentProduction` Gosu class ([Classes → gw → document → DocumentProduction](#)) with the following public methods to facilitate working with document creation:

Asynchronous methods	Synchronous methods
• <code>asynchronousDocumentCreationSupported</code>	• <code>synchronousDocumentCreationSupported</code>
• <code>createDocumentAsynchronously</code>	• <code>createAndStoreDocumentSynchronously</code>
	• <code>createDocumentSynchronously</code>

For the most part, these methods require the same passed-in parameters.

Parameter	Type	Description
template	IDocumentTemplateDescriptor	Template descriptor to use in creating the document. The file name must be the same as the file name of the document template file itself. IMPORTANT If you do not supply a locale in the IDocumentTemplateDescriptor, then ClaimCenter uses the default locale.
parameters	Map	Set of objects—keyed by name—to supply to the template generation process to create the document. The parameters Map must not be null and must contain, at a minimum, a name and value pair for each context object required by the given document template. It can also contain other information, which the document production system can use to perform additional operations.
document	Document	Document entity corresponding to the newly generated content. The passed-in Document entity can contain some fields that have already been set, and the document production system can set other fields in the course of performing the document creation.
fieldValues	Map	Set of values—keyed by field name—to set on the Document entity created at the end of the asynchronous creation process.

How to Determine the Supported Document Creation Type

ClaimCenter provides the following methods that you can use to determine if the IDocumentTemplate plugin supports the required document creation mode:

- `asynchronousDocumentCreationSupported`
- `synchronousDocumentCreationSupported`

The `asynchronousDocumentCreationSupported` Method

Call the `asynchronousDocumentCreationSupported` method to determine whether the IDocumentProduction implementation supports asynchronous creation for the specified template. This method returns `true` if it is possible, and `false` otherwise. It also returns `false` if a template cannot be found with the specified name or if you pass in `null` for the template name.

This method has the following signature:

```
public static function asynchronousDocumentCreationSupported(template : IDocumentTemplateDescriptor) : boolean
```

The `synchronousDocumentCreationSupported` Method

Call the `synchronousDocumentCreationSupported` method to determine whether the IDocumentProduction implementation supports synchronous creation for the specified template. This method returns `true` if it is possible, and `false` otherwise. It also returns `false` if ClaimCenter cannot find a template with the specified name or if you pass in `null` for the template name.

This method has the following signature:

```
public static function synchronousDocumentCreationSupported(template : IDocumentTemplateDescriptor) : boolean
```

Asynchronous Document Creation Methods

Guidewire provides the following methods for asynchronous document creation:

- `createDocumentAsynchronously`

The `createDocumentAsynchronously` Method

Call the `createDocumentAsynchronously` method to create a new document asynchronously, based on the named template and the supplied parameters. (The method returns immediately, but the actual document creation takes place over an extended period of time.) It is the responsibility of the external document production system to create a Document entity (if desired) after document creation is complete.

The last method parameter (`fieldValues`) provides a set of field name to value mappings to set in the metadata of the new Document after you create it.

This method has the following signature:

```
public static function createDocumentAsynchronously(template : IDocumentTemplateDescriptor,  
parameters : Map, fieldValues : Map) : String
```

Note: The `IDocumentProduction` implementation is responsible for persisting both the document contents and creating a new Document entity if necessary.

Synchronous Document Creation Methods

Guidewire provides the following methods for synchronous document creation:

- `createAndStoreDocumentSynchronously`
- `createDocumentSynchronously`

The `createAndStoreDocumentSynchronously` Method

Call the `createAndStoreDocumentSynchronously` method to create and store a document without any further user interaction. This method creates a document synchronously and passes it to the `IDocumentContentSource` plugin for persistence.

This method has the following signature:

```
public static function createAndStoreDocumentSynchronously(template : IDocumentTemplateDescriptor,  
parameters : Map, document : Document)
```

The `createDocumentSynchronously` Method

Call the `createDocumentSynchronously` method to create a new document synchronously, based on the named template and the supplied parameters. ClaimCenter adds the generated document to the configured `IDocumentContentSource` implementation for storage in the Document Management System (DMS).

Use this method any time that you want the generated content to be visible in the ClaimCenter interface and you do not necessarily want to persist the newly generated content. For example, you can use this method to generate content simply for viewing within ClaimCenter or for printing.

This method returns a `DocumentContentsInfo` object related to the template type, which contains the results of the document creation. The following are some examples of possible return objects:

- For Gosu-based templates, the returned object consists of the actual contents of the generated document.
- For Microsoft Mail Merge documents (chiefly MS Word and Excel documents), the returned object consists of a JavaScript file that the client machines runs to produce the document content.
- For a server-generated PDF document, the returned object consists of the actual contents of the generated document.

This method has the following signatures:

```
public static function createDocumentSynchronously(template : IDocumentTemplateDescriptor,  
parameters : Map) : DocumentContentsInfo
```

```
public static function createDocumentSynchronously(template : IDocumentTemplateDescriptor,  
parameters : Map, document : Document) : DocumentContentsInfo
```

See the *Gosu API Reference* (accessible from the Studio Help menu) for details of the difference between these two method signatures.

Document Templates

See the following topics for additional information on document templates.

Document Templates

For general information on document templates, how to create them, and how to use them, see:

- “Gosu Templates” on page 347 in the *Gosu Reference Guide*
- “Document Management” on page 199 in the *Integration Guide*
- “Data Extraction Integration” on page 607 in the *Integration Guide*

Document Template Localization

For information on localizing document templates, see:

- “Localizing Templates” on page 57 in the *Globalization Guide*
- “Document Localization Support” on page 62 in the *Globalization Guide*

IMPORTANT The base configuration *Sample Acrobat* document (*SampleAcrobat.pdf*) uses Helvetica font. If you intend to create a document that uses Unicode characters (for example, one that uses an East Asian language), then the document template must support a Unicode font. Otherwise, the document does not display Unicode characters correctly.

Document Creation Examples

Note: See also “Document Management” on page 199 in the *Integration Guide* for more information on working with documents in ClaimCenter.

IMPORTANT The following examples refer to Guidewire ClaimCenter objects and entities. However, these examples illustrate concepts and principals that are applicable to document creation in all Guidewire applications.

The following examples use an HTML template to construct a document that is in reference to an accident that took place in Great Britain. (FSA is the regulator of all providers of financial services in the United Kingdom, thus, the references to FSA regulations in the document.) After creation, the document looks similar to the following:

To:
Peter Smith
535 Main Street
Sutter, CA 12345

Dear Peter Smith,

*This letter is being sent in accordance with FSA requirements ICOB 7.5.1, 7.5.4, and 7.5.5.
Your claim, number 54321, was filed on November 25, 2005. We are currently investigating this claim and will contact you and other involved parties shortly.*

You can expect an update from your claims representative, Mary Putnam, no later than 30 days after the listed claim filing date.

If you need more information, please contact me at +44 555 123 1234.

Sincerely,

*John Sandler
XYZ Insurance*

To construct this document, you need the following files:

- FSAClaimAcknowledgement.gosu.htm
- FSAClaimAcknowledgement.gosu.htm.descriptor

FSAClaimAcknowledgement.gosu.htm

File FSAClaimAcknowledgement.gosu.htm sets the text to use in the document. It contains template variables (%...%) that ClaimCenter replaces with values supplied by FSAClaimAcknowledgement.gosu.htm.descriptor during document production.

```
<html xmlns="http://www.w3.org/TR/REC-html40">

  <head>
    <meta http-equiv=Content-Type content="text/html; charset=windows-1252">
    <title>FSA Claim Acknowledgement Letter</title>
    <style></style>
  </head>

  <body>
    <b>To:</b><br>
    <%=InsuredName%><br>
    <%=InsuredAddress1%><br>
    <%=InsuredCity%>, <%=InsuredState%> <%=InsuredPostalCode%><br>
    <br>Dear <%=InsuredName%>,<br>
    <br>This letter is being sent in accordance with FSA requirements ICOB 7.5.1, 7.5.4, and 7.5.5.<br>
    <br>Your claim, number <%=ClaimNumber%>, was filed on <%=ClaimReportedDate%>.<br>
    <br>We are currently investigating this claim and will contact you and other involved parties
      shortly. You can expect an update from your claims representative, <%=AdjusterName%>,
      no later than 30 days after the listed claim filing date.<br>
    <br>If you need more information, please contact me at +44 555 123 1234.<br>
    <br>Sincerely,<br>
    <%=AdjusterName%><br>
    XYZ Insurance<br>
  </body>

</html>
```

FSAClaimAcknowledgement.gosu.htm.descriptor

The descriptor document serves a number of purposes:

- First, it serves to classify the created document.
- Secondly, it specifies the ContextObject objects that it needs to fill in the document template. These are manually set by the user or ClaimCenter automatically sets these through business rules during document creation.
- It also provides a mapping between the names of templated fields in the destination document and the context objects (FormField objects).

```
<?xml version="1.0" encoding="UTF-8"?>
<DocumentTemplateDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.guidewire.com/schema/claimcenter/document-template.xsd"

  id="FSAClaimAcknowledgement.gosu.htm"
  name="Claim Acknowledgement Letter"
  description="ICOB 7.5.1 acknowledgement letter."
  type="letter_sent"
  lob="GL"
  state="UK"
  mime-type="text/html"
  date-effective="Mar 15, 2004"
  date-expires="Mar 15, 2007"
  keywords="UK, acknowledgement">

  <ContextObject name="To" type="Contact">
    <DefaultObjectValue>Claim.Insured</DefaultObjectValue>
    <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
  </ContextObject>

  <ContextObject name="From" type="Contact">
    <DefaultObjectValue>Claim.AssignedUser.Contact</DefaultObjectValue>
```

```

<PossibleObjectValues>Claim.getRelatedUserContacts()</PossibleObjectValues>
</ContextObject>

<FormFieldGroup name="main">
    <DisplayValues>
        <DateFormat>MMM dd, yyyy</DateFormat>
    </DisplayValues>
    <FormField name="ClaimNumber">Claim.ClaimNumber</FormField>
    <FormField name="ClaimReportedDate">Claim.ReportedDate</FormField>
    <FormField name="InsuredName">To.DisplayName</FormField>
    <FormField name="InsuredPrefix">(To as Person).Prefix</FormField>
    <FormField name="InsuredLastName">(To as Person).LastName</FormField>
    <FormField name="InsuredAddress1">To.PrimaryAddress.AddressLine1</FormField>
    <FormField name="InsuredCity">To.PrimaryAddress.City</FormField>
    <FormField name="InsuredState">To.PrimaryAddress.State</FormField>
    <FormField name="InsuredPostalCode">To.PrimaryAddress.PostalCode</FormField>
    <FormField name="CurrentDate">gw.api.util.DateUtil.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>

```

Method createAndStoreDocumentSynchronously Example 1

In the following example, the `createAndStoreDocumentSynchronously` method takes following parameters to construct the previously shown document:

Parameter	Value	Description
descriptor	<code>FSAClaimAcknowledgement.gosu.htm.descriptor</code>	Template descriptor to use in constructing the document.
parameters	<code>contextObjects</code>	Relevant information related to the document contained in a <code>HashMap</code> object.
document	<code>Document</code>	Document entity to contain the newly constructed document

The following (ClaimCenter) Gosu code creates the document.

```

//Creates a map object and sets document information (ContextObjects) needed for content creation
var parameters = new java.util.HashMap()
parameters.put("Claim", claim)
parameters.put("To", claim.maincontact)
parameters.put("From", claim.AssignedUser.Contact)
parameters.put("CC", null)

//Creates a new Document entity and sets metadata directly on the document object
var document : Document = new Document(claim)
document.Claim = claim
document.Name = "Claim Acknowledgement"
document.Type = "letter_sent"
document.Status = "draft"

//Creates the document template descriptor
var plugin = gw.plugin.Plugins.get(gw.plugin.document.IDocumentTemplateSource)
var descriptor = plugin.getDocumentTemplate("FSAClaimAcknowledgement.gosu.htm",
    \ code -> gw.i18n.ILocale.EN_US)

gw.document.DocumentProduction.createAndStoreDocumentSynchronously(descriptor, parameters, document)

```

Method createAndStoreDocumentSynchronously Example 2

Gosu also provides the ability to test whether document creation is possible before attempting the operation. You can check for the ability to create a document synchronously or asynchronously by using one of the following methods.

- synchronousDocumentCreationSupported
- asynchronousDocumentCreationSupported

These methods take the name of a template descriptor as the lone argument and determine whether the `IDocumentProduction` plugin supports the specified template descriptor. Each method returns `true` if the plugin does support that template descriptor file and `false` otherwise. Each method also return `false` if it cannot find a template descriptor with the specified name.

The following sample (ClaimCenter) Gosu code illustrates the use of these methods:

```
//Creates a map object and sets document information (ContextObjects) needed for content creation
var parameters = new java.util.HashMap()
parameters.put("Claim", claim)
parameters.put("To", claim.maincontact)
parameters.put("From", claim.AssignedUser.Contact)
parameters.put("CC", null)

//Creates a new Document entity and sets metadata directly on the document object
var document : Document = new Document(claim)
document.Claim = claim
document.Name = "Claim Acknowledgement"
document.Type = "letter_sent"
document.Status = "draft"

//Creates the document template descriptor (of type IDocumentTemplateDescriptor)
var plugin = gw.plugin.Plugins.get(gw.plugin.document.IDocumentTemplateSource)
var descriptor = plugin.getDocumentTemplate("SampleAcrobat.pdf",
    \ code -> gw.i18n.ILocale.EN_US)

if (gw.document.DocumentProduction.synchronousDocumentCreationSupported(descriptor)) {
    gw.document.DocumentProduction.createAndStoreDocumentSynchronously(descriptor, parameters, document)
}
```

Troubleshooting

The following issues with document creation can occur on occasion:

- `IDocumentContentSource.addDocument` Called with Null `InputStream`
- `IDocumentMetadataSource.saveDocument` Called Twice
- `UnsupportedOperationException` Exception
- Document Template Descriptor Upgrade Errors
- “Automation server cannot create object” Error
- “`IDocumentProduction` implementation must return document...” Error

`IDocumentContentSource.addDocument` Called with Null `InputStream`

In certain cases, ClaimCenter calls `IDocumentContentSource.addDocument` with a null `InputStream`. This can occur as a result of the following scenario:

1. The user creates a `Document` entity with some content and clicks **Update** in the application interface.
2. The document fails validation for some reason. However, ClaimCenter uploads the document contents to the configured `IDocumentContentSource` before validation is run. Therefore, ClaimCenter has already called `addDocument` and has already read the `InputStream`.
3. The user fixes the problem and clicks **Update** again. At this point, any changes made to the document by the previous call to `IDocumentContentSource.addDocument` have been lost. So, ClaimCenter calls `addDocument` again. This gives the `IDocumentContentSource` implementation a chance to set any fields (such as `DocUID`) that need to be set on the `Document` entity before ClaimCenter stores it.

This scenario assumes that you can match up the document content passed in the first call to `addDocument` with the `Document` entity from the second call to `addDocument`. This is not true of every Document Management

System. Therefore, you might need to add some caching code to your `IDocumentContentSource` implementation so that the connection can be preserved.

`IDocumentMetadataSource.saveDocument` Called Twice

Occasionally, ClaimCenter can call `IDocumentMetadataSource.saveDocument` twice after a call to `createDocumentAsynchronously`. This can happen, for example, if you have a rule such as the following:

```
uses gw.document.DocumentProduction

var document: Document = new Document(Account)
document.Account = Account
document.Name = "Created by a Rule" + java.lang.System.currentTimeMillis()
document.Type = DocumentType.TC_LETTER_RECEIVED
document.Status = DocumentStatusType.TC_DRAFT

var contextObjects = new java.util.HashMap()
contextObjects.put("Account", Account)
contextObjects.put("To", Account.MainContact)
contextObjects.put("From", Account.UpdateUser)
contextObjects.put("CC", null)

DocumentProduction.createDocumentAsynchronously( "SampleDecSheet.rtf", contextObjects, document )
```

In this instance, ClaimCenter saves the document—created by the rule to pass to `createDocumentAsynchronously`—as part of commit of the bundle used by the rule. ClaimCenter saves the document metadata again any time that the asynchronous document creation completes. This is because the semantics of `createDocumentAsynchronously` require that the implementation take care of saving the `Document` entity.

The workaround is to add something like the following to the end of the rule, before the call to the `createDocumentAsynchronously` method:

```
document.setPersistedByDocumentSource( true )
```

Note: This is only an issue with ClaimCenter 4.0.2 and earlier.

`UnsupportedOperationException` Exception

Occasionally, the following exception occurs:

```
java.lang.UnsupportedOperationException: Asynchronous client-side MailMerge-based generation  
is not supported!
```

This exception occurs because the sample `IDocumentProduction` implementation does not support server-side creation of Microsoft Word or Excel documents. Therefore, you cannot create documents asynchronously based on templates of these types.

Document Template Descriptor Upgrade Errors

The following error can occur during document template descriptor upgrade:

```
IOException encountered: Content is not allowed in prolog.
```

The cause of this error, generally, is that one or more descriptor files has some characters before the "`<?xml`" at the beginning of the file. That XML tag must be the very first thing in the file.

"Automation server cannot create object" Error

Occasionally, the following (or similar) error occurs during attempts to create a document from a template.

```
"Automation server cannot create object"
```

Creation of Microsoft Word and Excel documents from templates is done on the user's machine. Therefore, if the user's machine does not have Word or Excel, this kind of error occurs. You can, however, still create a document from a PDF or Gosu template instead.

"IDocumentProduction implementation must return document..." Error

You may see the following error while calling a `DocumentProduction.create*` method:

"The `IDocumentProduction` implementation must return document contents to be called from a rule"

This error message indicates a problem with the `DocumentContentsInfo.ResponseType` property on the return value of the document production plugin document creation code. The exact plugin method is the `IDocumentProduction` plugin method `createDocumentSynchronously`.

In the default configuration, the built-in implementation of this plugin expects the `responseType` field has the response type value `DocumentContentsInfo.DOCUMENT_CONTENTS`.

If you modify the document production plugin or any related code such that it returns another response type such as URL, ClaimCenter displays this error. The built-in `IDocumentProduction` implementation expected an actual document contents as an input stream of bytes, not other response types such as `DOCUMENT_URL`.

However, you might want to support a document production implementation that supports the response type `DOCUMENT_URL`. If so, you may need to modify the built-in document production plugin implementation. In Studio, edit the built-in Gosu class `gw.document.DocumentProduction`. There are multiple method signatures for the method `createAndStoreDocumentSynchronously`, all of which check the response type. You must modify the `DocumentProduction` class to appropriately handle the `DOCUMENT_URL` response type.

Your modifications depend on the meaning of the URL in your system:

- If the URL refers directly to the external Document Management System (DMS), the document is already stored. The document production plugin does not need to do anything directly with returned data. Remove the error checking line that requires the document type `DOCUMENT_CONTENTS`. Check if the value is `DOCUMENT_URL`, and if so skip the following lines that assume the result is an input stream.
- If the URL refers to a separate external resource other than the DMS, the document production plugin must immediately retrieve that external data. Next, submit the resulting input stream to the document content source. Base your code to submit the document on the existing `createAndStoreDocumentSynchronously` code that submits an input stream to the document content source.

If your URL has some other meaning than the choices listed above, contact Guidewire Customer Support.

Large Size Microsoft Word Documents

Occasionally, you see the same document template create Microsoft Word documents with different file sizes. The issue, in this case, is that some users have their individual Microsoft Word applications configured to save documents as `Word97-2003 6.0/95 - RTF.doc`. This causes Word to generate the file as Rich Text Format (RTF), but save the generated file with the `.doc` extension. In general, RTF documents have a much larger file size than the basic `.doc` Word files.

To set a default file format for saving new documents, do the following:

1. On the Word Tools menu, click **Options**, and then click the **Save** tab.
2. In the **Save Word files as** box, click the file format that you want.

This procedure can be different for each version of Microsoft Word.

Note: Guidewire recommends that you check the Microsoft Word documentation specific to your version for exact instructions.

Transaction Set Validation

Insurance companies typically sell vehicle liability insurance as a package, with the insurer's liability limited to each person, each occurrence (incident), and total property damage. For example, a 100/300/50 package limits the maximum payout in an accident for which an insurer would be liable to the following:

- \$100,000 in Bodily Injury (BI) coverage for each person
- \$300,000 in Bodily Injury (BI) coverage for all persons
- \$50,000 for all Property Damage (PD)

By using the ClaimCenter transaction validation rules and library functions, you can track these limits and raise alerts anytime that a transaction exceeds the exposure or incident limit. You can set these rules either to block the transaction from going through or to allow the transaction to continue, but with warnings.

Note: Guidewire provides the Transaction Set Validation rules as sample rules only. Customize these rules to meet your business logic as appropriate (for example, by modifying the reserve definition to meet your business needs).

This topic includes:

- “Transaction Set Validation Rules” on page 161
- “Transaction Set Validation Rules” on page 161

Transaction Set Validation Rules

ClaimCenter provides four sample transaction validation rules that block or warn the user from executing a financial transaction that exceeds the policy limits. In addition, there is a fifth rule that is specific to PIP (Personal Injury Protection) that blocks the user from making payments that exceed the PIP aggregate limit. The rules, all in the Transaction Set Validation rule set, are described later in this topic.

The sample Transaction Set Validation rules execute correctly only if both of the following are true:

- The created exposure is tied to the coverage, which is visible in the Exposure screen.
- The corresponding coverages are defined at the vehicle level on the policy's Vehicles screen.

ClaimCenter runs the validation rules anytime that it updates a ClaimCenter business entity (Claim, Exposure, or TransactionSet, for example) and commits data to the database.

These sample rules calculate payments values using *claim cost*, which includes only costs directly associated with the claim, and not ancillary costs such as expenses. The sample rules also calculate all values by using the financial calculations RemainingReserves value. This value is obtained by calling the getRemainingReserves method:

```
var RemainingReservescache = gw.api.financials.FinancialsCalculationUtil.getRemainingReserves()
```

To work with a different type of reserve value, use a different getter method. For example, to access the AvailableReserves value, use the getAvailableReserves method:

```
var AvailableReservescache = gw.api.financials.FinancialsCalculationUtil.getAvailableReserves()
```

This topic includes

- “TXV01000 – Total payments not greater than Exposure limit” on page 162
- “TXV02000 – Check Exposure limit when increasing reserves” on page 163
- “TXV03000 – Total payments not greater than Incident limit” on page 164
- “TXV04000 – Check Incident limit when increasing reserves” on page 164
- “TXV05000 – PIP” on page 165

See also

- “ClaimCenter Financial Calculations” on page 619 in the *Configuration Guide*
- “Transaction Set Validation Rule Set” on page 84
- “Validation Rule Set Category” on page 74
- “Validation in ClaimCenter” on page 121

TXV01000 – Total payments not greater than Exposure limit

ClaimCenter executes this rule anytime that you attempt to create a new check.

```
transactionset.Subtype == "CheckSet" && transactionset.new && transactionset.ApprovalStatus ==  
"unapproved"
```

Rule	Condition	Action
TXV01000	1. (New payment + Current Total Payments already made) > Exposure Limit 2. Above condition false and (New payment + Current Total Payments already made + Remaining Reserves) > Exposure Limit	1. Error, block 2. Warn, but permit

Use this rule to either block (reject completely) or permit (with warnings) a payment that would cause one of the following conditions:

- Making a new payment and adding it to the payments *already made* would cause the Total Payments to exceed the Exposure limit. See “Example 1: Payment Would Cause Total Payments to Exceed Exposure Limit” on page 162.
- Making a new payment and adding it to the payments *already made plus* the Remaining Reserves would cause the Total Payments to exceed the exposure limit. Note that for this condition to be true, the previous condition must be false. Thus, this condition assumes that the Remaining Reserves is greater than the Exposure limit. See “Example 2: Payment Plus Remaining Reserves Would Cause Total Payments to Exceed Exposure Limit” on page 163.

Example 1: Payment Would Cause Total Payments to Exceed Exposure Limit

Suppose that you have created a claim with a Vehicle Damage exposure. The exposure limit is \$10,000 with a reserve of \$8,000. As you make payments (transactions 1 and 2), you deplete the reserve, but the total payout is

still below both the exposure limit and the reserve limit. An attempted third payment now brings the payment total to \$10,900, which is over the exposure limit of \$10,000.

Exposure Limit	Payment	Amount	Rule action
\$10,000	1	\$2,000	Permit the transaction as payment of \$2,000 < Exposure Limit of \$10,000
	2	\$5,500	Permit the transaction as \$2,000 + \$5,500 < \$10,000
	3	\$3,400	Block the transaction as \$2,000 + \$5,500 + \$3,400 > \$10,000
	Total	\$10,900	

Depending on your business needs, you might want to raise an alert anytime that a payment causes the payment total to exceed a reserve or exposure limit. In some cases, you might want to block the transaction altogether. The sample rules in the Transaction Set Validation rules handle either of these situations easily. See “Transaction Set Validation Rule Set” on page 84.

Example 2: Payment Plus Remaining Reserves Would Cause Total Payments to Exceed Exposure Limit

Exposure Limit	Remaining Reserves	Payment	Amount	Rule action
\$10,000	\$8,000	1	\$1,500	Permit the transaction as the payment of \$1,500 + \$8,000 (Remaining Reserves) < \$10,000 (Exposure limit)
	\$6,500	2	\$1,000	Permit the transaction as \$1,000 + \$1,500 + \$6,500 < \$10,000
	\$5,500	3	\$3,000	Warn the user as \$1,500 + \$1,000 + \$3,000 + \$5,500 > \$10,000, but permit the user to execute the transaction if desired

Two internal variables within the rule, one for each condition, determines the outcome of that condition.

Variable	Effect
warn1	True – warn, but permit; False – reject and block
warn2	True – warn, but permit; False – reject and block

TXV02000 – Check Exposure limit when increasing reserves

ClaimCenter executes this rule anytime that you attempt to create a new reserve (or, more specifically, if you attempt to create a new TransactionSet of type ReserveSet).

```
transactionset.Subtype == "ReserveSet" && transactionset.new
```

Rule	Condition	Action
TXV02000	(New reserve amount + Current Total Payments + Remaining Reserves) > Exposure Limit	Warn, but permit

You can use this rule to either block (reject completely) or permit (with warnings) a reserve increase. For example, you could reject a reserve increase that would cause the sum of the combined current payments, the new reserve amount, and remaining reserve to exceed the exposure limit. To illustrate, suppose that the exposure limit is \$10,000 and the initial reserve limit is \$8,500, and you are attempting to make a payment of \$1,000. The

remaining reserve after this transaction equals \$7,500. Increasing the reserve by \$2,000 would create a combined total of \$10,500 ($\$7,500 + \$2,000 + \$1,000$), which exceeds the exposure limit.

Exposure Limit	Remaining Reserves	Payments	Rule Action
\$10,000	\$8,500	\$1,000	Permit the transaction as $\$1,000 + \$8,500 < \$10,000$, leaving a Remaining Reserves of \$7,500
	\$7,500		
	$\$7,500 + \$2,000 = \$9,500$		Block (or permit with warnings) as attempting to increase the reserves by \$2,000 to \$9,500 fails ($\$9,500 + \$1,000 > \$10,000$)

You set the outcome of this rule (block or warn) by setting an internal variable (warning) within the rule.

Variable	Value
warning	True – warn, but permit; False – reject and block

TXV03000 – Total payments not greater than Incident limit

ClaimCenter executes this rule anytime that you attempt to create a new check.

```
transactionset.Subtype == "CheckSet" && transactionset.new && transactionset.ApprovalStatus == "unapproved"
```

Rule	Condition	Action
TXV03000	<ol style="list-style-type: none"> (New payment + Current Total Payments already made for all exposures tied to the same coverage) > Incident Limit Above condition false <i>and</i> (New payment + Current Total Payments for all exposures tied to the same coverage + Remaining Reserves) > Incident Limit 	<ol style="list-style-type: none"> Error, block Warn, but permit

This sample rule works in a similar manner to TXV01000, except that it applies to incident limits, not exposure limits. See “TXV01000 – Total payments not greater than Exposure limit” on page 162.

Use this rule to either block (reject completely) or permit (with warnings) a payment that would cause one of the following conditions. Two internal variables within the rule (one for each condition) determines the outcome of that condition.

Variable	Value
warn1	True – warn, but permit; False – reject and block
warn2	True – warn, but permit; False – reject and block

Note that this rule handles only one vehicle per incident.

TXV04000 – Check Incident limit when increasing reserves

ClaimCenter executes this rule anytime that you attempt to create a new reserve.

```
transactionset.Subtype == "ReserveSet" && transactionset.new
```

Rule	Condition	Action
TXV04000	(New reserve amount + Current Total Payments for all exposures tied to the same coverage + Remaining Reserves) > Incident Limit	Warn, but permit

This sample rule works in a similar manner to TXV02000, except that it applies to incident limits, not exposure limits. See “TXV02000 – Check Exposure limit when increasing reserves” on page 163.

You set the outcome of this rule (block or warn) by setting an internal variable (`warning`) within the rule.

Variable	Value
<code>warning</code>	True – warn, but permit; False – reject and block

Note that this rule handles only one vehicle per incident.

TXV05000 – PIP

The Rule engine executes this rule anytime that it validates a check.

```
TransactionSet.Subtype == "checkset"
```

Rule	Condition	Action
TXV05000	Payments on PIP coverages exceed one of the four defined PIP Aggregate Limits	Warn, but permit

This sample rule determines whether adding this new amount to the payments already made for this exposure would cause the total payments to exceed one of the following limits. (ClaimCenter calculates these aggregate limits in the Claim enhancement method `sumForPIPAGG`. See the following section for details.)

Limit	Description
PIPNonmedAgg	Non-medical aggregate limit
PIPReplaceAgg	Lost wages and replacement services aggregate limit
PIPPersonAgg	PIP per person aggregate limit
PIPClaimAgg	PIP per claim aggregate limit

Gosu code within this rule looks at each limit and determines whether the new payment total exceeds a specific aggregate limit. If so, the rule permits the transaction, but raises a warning within ClaimCenter. The user must then explicitly cancel or continue the transaction. For example, if the new payment causes the payment total to exceed the PIPPersonAgg limit, it raises the following warning

```
TransactionSet.reject( null, null, "external", "The PIP Per Person Aggregate Limit ($"
+ (each.Exposure.Coverage as VehicleCoverage).PersonAggLimit
+ ") has been exceeded for this coverage, " + each.Exposure.Coverage.Type
+ ". Do you still want to continue? )
```

At the same time, the rule sets a Boolean flag indicating that the sum of all payments, including this new one, now exceeds one of the aggregate limits. There are four of these flags, corresponding to the four aggregate limits:

- PIPNonmedAggLimitReached
- PIPReplaceAggLimitReached
- PIPPersonAggLimitReached
- PIPClaimAggLimitReached

Other business rules can access these values as properties on the `Exposure` entity. For example, a rule in the `Exposure Preupdate` rule set could use the following code to determine if the aggregate limit for each person has been reached for this exposure:

```
var PersonAggLimitReached = exposure.PIPPersonAggLimitReached
```

See “Exposure Preupdate Rules” on page 67.

TXV17000 – Reserve Lines Compatible with Service Requests

The Rule engine executes this rule anytime that it validates a check. It validates all checks linked to a service request invoice to determine if the check's set of reserve lines is compatible with at least one of the service request invoices.

It is compatible if one of the following conditions is met:

- One of the associated service requests is claim-level, or
 - Each exposure-level payment shares an incident with one of the associated service requests.
- ```
transactionSet typeis CheckSet
```

| Rule     | Condition                                                                      | Action       |
|----------|--------------------------------------------------------------------------------|--------------|
| TXV17000 | Check's reserve line is compatible with one of the service request's invoices. | Error, block |

## TXV18000 – Currencies Compatible with Service Requests

The TXV18000 sample rule validates currencies associated with a service request invoice.

```
transactionSet typeis CheckSet
```

| Rule     | Condition                                           | Action       |
|----------|-----------------------------------------------------|--------------|
| TXV17000 | Check and service request invoice currencies match. | Error, block |

## sumForPIPAgg Enhancement Methods

ClaimCenter business rules use the `sumForPIPAgg` enhancement methods on the Claim entity (Classes → libraries → `sumforPIPAgg`) in working with PIP-related transactions. (Note that several of the `sumForPIPAgg` methods, in turn, contain method calls to other enhancement methods.) In the base configuration, these enhancement methods calculate the aggregates sums of the following coverage types per incident:

| Aggregate method     | Sums these coverage types                                                                                                                                                                                |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| sumForPIPClaimAgg    | <ul style="list-style-type: none"> <li>• PIP Medical</li> <li>• PIP Rehabilitation</li> <li>• PIP Lost Wages</li> <li>• PIP Funeral</li> <li>• PIP Death</li> <li>• PIP Extraordinary Medical</li> </ul> |
| sumForPIPNonmedAgg   | <ul style="list-style-type: none"> <li>• PIP Rehabilitation</li> <li>• PIP Lost Wages</li> <li>• PIP Funeral</li> <li>• PIP Death</li> </ul>                                                             |
| sumForPIPPersonAgg   | <ul style="list-style-type: none"> <li>• PIP Medical</li> <li>• PIP Rehabilitation</li> <li>• PIP Lost Wages</li> <li>• PIP Funeral</li> <li>• PIP Death</li> <li>• PIP Extraordinary Medical</li> </ul> |
| sumForPIPRReplaceAgg | PIP Lost Wages                                                                                                                                                                                           |

The TXV05000 sample rule uses these summing methods as methods on the Exposure entity. For example:

```
each.Exposure.Claim.sumForPIPNonmedAgg(Claimant) >
 (each.Exposure.Coverage as VehicleCoverage).NonmedAggLimit)
```

See “TXV05000 – PIP” on page 165.

# Working with Queues

This topic describes the ClaimCenter activity queues. It also provides several examples of how to use an activity queue.

This topic includes:

- “What Are Activity Queues?” on page 167
- “Using a Queue to Handle FNOL Claims: Example” on page 167
- “Using a Queue to Void/Stop Checks: Example” on page 170

## What Are Activity Queues?

Guidewire ClaimCenter provides the concept of *activity queues*. An activity queue is a list of activities that are waiting for someone to take ownership of them. An activity must be assigned to an owner before any action—such as completing or skipping it—can be taken.

The following topics provide examples of how to use queues in Guidewire ClaimCenter:

- “Using a Queue to Handle FNOL Claims: Example” on page 167
- “Using a Queue to Void/Stop Checks: Example” on page 170

### Assignment Review Activities

ClaimCenter does not route activities of type *assignment review* to a queue, even if you explicitly call for this behavior. Therefore, these types of actives do not show in the **Queue** tab within ClaimCenter. Instead, ClaimCenter displays these activities in the **Pending Assignments** screen.

## Using a Queue to Handle FNOL Claims: Example

The following example illustrates how to use a queue to handle FNOL claims. If a claim enters ClaimCenter as an FNOL (First Notification of Loss), it triggers the ClaimCenter Global Assignment rules. However, in this

example, the assignment rules do not assign the claim. Instead, these rules generate review activities, which are queued for response by a team of people who will manually assign the incoming claims.

ClaimCenter associates every queue with a single group, or potentially with a group and its subgroups. ClaimCenter automatically creates a default FNOL queue for each group in the hierarchy. This queue is not visible to the subgroups, by default.

In this example, there are five rules:

1. The first rule assigns the incoming FNOL claim to an intake group that will make the assignment decision on that claim. See “Assign Claim to an Intake Group” on page 168.
2. The second rule then assigns the claim to the group’s supervisor. This “parks” the claim until the final assignment determination. See “Assign Claim to the Group Supervisor” on page 168.
3. The third rule creates the FNOL review activity. See “Create FNOL Review Activity” on page 169.
4. The next rule assigns the FNOL review activity to the intake group designated in the first assignment rule. See “Assign FNOL Review Activity to Intake Group” on page 169.
5. The final rule places the review activity on the FNOL queue of the designated intake group, from which members of the group will pull a review activity. See “Assign FNOL Review Activity to Queue” on page 169.

### [Assign Claim to an Intake Group](#)

The Global Claim Assignment rule ASGCDefault - Assign to Claim Intake Units assigns an incoming claim to a specific group or groups. This group consists of users who will make the assignment decisions. This example uses two groups:

- FNOL Review Group - WC
- FNOL Review Group - GL

There are no conditions for this rule, so the rule fires for every claim that comes into ClaimCenter.

#### **Conditions**

true

#### **Actions**

```
gw.api.util.Logger.logInfo("In Claim No. : " + Claim.Claimnumber + " - In "
+ actions.getRuleSet() + " - In Rule " +actions.getRule())

if (claim.LossType == "WC") {

 if (Claim.CurrentAssignment.assignGroup(Group("cc:94" /* FNOL Review Group - WC */)) {
 gw.api.util.Logger.logInfo(claim.ClaimNumber+ " was assigned to FNOL Review Group - WC")
 actions.exit()
 } else {
 if (Claim.CurrentAssignment.assignGroup(Group("cc:95" /* FNOL Review Group - GL */)) {
 gw.api.util.Logger.logInfo(claim.ClaimNumber+ " was assigned to FNOL Review Group - GL")
 actions.exit()
 }
 }
}
```

### [Assign Claim to the Group Supervisor](#)

The Default Group Claim Assignment rule ASDGCDefault - Assign to Claim Group Supervisor assigns the claim to the supervisor of the group. There are no conditions for this rule, so the rule fires for every claim that comes into ClaimCenter.

#### **Conditions**

true

**Actions**

```
gw.api.util.Logger.logInfo("In Claim No. : "+ Claim.Claimnumber + " - In "
+ actions.getRuleSet() + " - In Rule " + actions.getRule())

claim.CurrentAssignment.assignUser(claim.assignedgroup.supervisor)
```

## Create FNOL Review Activity

The Claim Workplan rule WPWC03 - FNOL Review creates the review activity to be queued. This activity tells the user to review and assign the claim. There are no conditions for this rule, so the rule fires for every claim that comes into ClaimCenter.

**Note:** This rule assumes that activity pattern fnol\_review is defined already. If it does not exist, then you must create it through the ClaimCenter Administration interface by adding it to the **Activity Patterns** page.

**Conditions**

```
true
```

**Actions**

```
gw.api.util.Logger.logInfo("In Claim No. : "+ Claim.Claimnumber + " - In "
+ actions.getRuleSet() + " - In Rule " +actions.getRule())
claim.createActivityFromPattern(null,
ActivityPattern.finder.getActivityPatternByCode("fnol_review"))
```

## Assign FNOL Review Activity to Intake Group

The Global Activity Assignment rule ASGA30 - FNOL Review Assignments assigns the activity to the group that has already been assigned to the claim. The rule condition ensures that only this specific type of activity is assigned. Otherwise, ClaimCenter would assign all activities that hit this rule to the queue. Note that the activity assignment happens as part of the if condition, which returns true if successful.

**Conditions**

```
Activity.ActivityPattern.Code == "fnol_review"
```

**Actions**

```
if (Activity.CurrentAssignment.assignGroup(activity.claim.AssignedGroup)) {
 gw.api.util.Logger.logInfo("In Claim No. : "+ Activity.Claim.Claimnumber + " - In Activity :"
+ Activity.ActivityPattern.Subject + " - In " + actions.getRuleSet()
+ " - In Rule " +actions.getRule() + " - Assigning to " + activity.claim.AssignedGroup.Name)
 actions.exit()
}
```

## Assign FNOL Review Activity to Queue

The Default Group Activity Assignment rule ASDGA01 - Assign New Claim Activity to Queue adds the review activity to the queue. This rule condition also specifies the correct activity type.

**Note:** ClaimCenter defines the default FNOL queue as AssignableQueues[0].

**Conditions**

```
Activity.ActivityPattern.Code == "fnol_review"
```

**Actions**

```
gw.api.util.Logger.logInfo("In Claim No. : "+ Activity.Claim.Claimnumber + " - In "
+ actions.getRuleSet() + " - In Rule " + actions.getRule() + " - In "
+ activity.AssignedGroup.AssivableQueues[0].Name)

activity.CurrentAssignment.assignActivityToQueue(activity.AssignedGroup.AssivableQueues[0])
actions.exit()
```

## Using a Queue to Void/Stop Checks: Example

The following example demonstrates using a queue to handle Void/Stop Payment activity requests. In this scenario, ClaimCenter does not allow random adjusters to void or stop checks. Instead, individual adjusters create an activity requesting a void/stop of a check. The activity enters a queue from which a member of the group of accountants takes the activity and completes it.

This example uses two rules that assign an activity that has been manually created by an end-user. Therefore, there is no need for claim assignment or workplan rules, as there is in “Using a Queue to Handle FNOL Claims: Example” on page 167. All that is necessary are the two activity assignment rules. Note, however, that you must first create a Stop/Void queue if one does not exist.

This example has three steps

1. If necessary, create the void/stop queue. See “Create the Stop/Void Queue” on page 170.
2. The first rule assigns an activity of type `void_or_stop_check` to a particular group, the Accountants group. See “Assign the Activity to the Appropriate Group” on page 170.
3. The second rule adds the activity to the void/stop queue. See “Add the Activity to the Queue” on page 171.

### Create the Stop/Void Queue

ClaimCenter associates each queue with a single group, or potentially with a group and its subgroups. Therefore, the first step is to associate the group that is responsible for voiding/stopping payments with the void/stop queue. If a void/stop queue does not exist, you must create one. In fact, if the group responsible for voiding/stopping payments has not yet been defined, you must define it first.

If the appropriate queue does not exist, use the ClaimCenter **Administration** tab to create one.

- First, select a specific group in the hierarchy.
- Then, select the **Queues** tab for that group.
- Click **Edit** and add the queue.

This example uses the Accountants group to handle the voiding and stopping of checks. (The members of this group are the only ones assigned the proper role to void/stop payments.)

### Assign the Activity to the Appropriate Group

The Global Activity Assignment rule ASGA40 - Void/Stop Payments checks to see if the activity is of type `void_or_stop_check`. If it is, the rule assigns the activity to the Accounts group.

#### Conditions

```
Activity.ActivityPattern.Code == "void_or_stop_check"
```

#### Actions

```
if (Activity.CurrentAssignment.assignGroup(Group("cc:96" /* Accountants */)) {
 gw.api.util.Logger.logInfo("In Claim No. :" + Activity.Claim.Claimnumber + " - In Activity :"
 + Activity.ActivityPattern.Subject + " - In " + actions.getRuleSet() + " - In Rule "
 + actions.getRule() + " - Assigning to " + activity.AssignedGroup.Name)
 actions.exit()
}
```

## Add the Activity to the Queue

The Default Group Activity Assignment rule ASDGA02 - Assign Void/Stop Activity to Queue adds the activity to the queue. This rule condition also requires that the activity be of the correct activity type.

**Note:** The rule refers to `AssignableQueues[1]` because this queue is presumably the first new queue added to this group. If you have multiple queues on a group, you need to choose the correct one.

`AssignableQueues[0]` refers to the default FNOL queue that ClaimCenter creates anytime that you create a new group. See “Assign FNOL Review Activity to Queue” on page 169.

### Conditions

```
Activity.ActivityPattern.Code == "void_or_stop_check"
```

### Actions

```
gw.api.util.Logger.logInfo("In Claim No. : " + Activity.Claim.Claimnumber
+ " - In " + actions.getRuleSet() + " - In Rule " + actions.getRule()
+ " - In " + activity.AssignedGroup.AssachableQueues[1].Name + "Queue")

activity.CurrentAssignment.assignActivityToQueue(activity.AssignedGroup.AssachableQueues[1])
actions.exit()
```

