ClaimCenter Rules Guide

Release 6.0.8

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

Contents

Part II
Advanced Topics

About This Document

This document contains an overview of how Guidewire ClaimCenter and ContactCenter use Guidewire Studio and rules to handle business processes. It describes:

- The basic processes of defining and managing Gosu business rules.
- The rule methodology, rule categories, and of rule syntax for Gosu business rules.
- The interaction between Gosu business rules created through Studio and the ClaimCenter application interface.

This topic includes:

Intended Audience

This document is intended for the following readers:

- Rule writers who actually implement and maintain Gosu rules
- Business analysts who define the business logic to be implemented in rules
- Implementation team members who work with Gosu in any fashion, whether through Gosu business rules and classes, through PCF pages, or through Gosu plugins
- Implementation team members or IT staff who do not directly write rules, but seek a better understanding of how ClaimCenter uses rules to perform business logic

Guidewire encourages ContactCenter users to read this document also. It contains the following information useful for ContactCenter users:

- General information about Guidewire Studio and business rules
- Specific information about ContactCenter and how it uses business rules

Assumed Knowledge

This document assumes that you are already familiar with the Guidewire ClaimCenter application. Guidewire recommends that you have read *ClaimCenter Application Guide* or that you have experience with the ClaimCenter screens and functionality before you begin working with rules.

Related Documents

See the following Guidewire documents for further information:

ClaimCenter Application Guide – Introduces the application, explains application concepts, and provides a high-level view of major features and business goals of ClaimCenter. This is your first place to look when learning about a feature. This book is written for all audiences.

Gosu Reference Guide – Describes the syntax of expressions and statements within ClaimCenter. This document also provides examples of how the syntax is used when creating rules. This document is intended for rule writers who create and maintain rules in Guidewire Studio.

ClaimCenter Gosu Generated Documentation – Documents all types visible from the Gosu type system. This includes Guidewire entities, Gosu classes, utility classes, Gosu plugin definitions, and Java types that are available from Gosu. With a local copy of the product, you can regenerate this documentation. From the `ClaimCenter/bin` directory run the `gwcc regen-gosudoc` command.

ClaimCenter Integration Guide – Provides an architectural overview and examples of how to integrate ClaimCenter with external systems and custom code. This document is a learning tool for explanations and examples with links to the *Java API Reference Javadoc* and *SOAP API Javadoc* for further details. This document is written for integration programmers and consultants.

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

- Java plugin interface definitions (also usable to write Gosu plugins)
- Details of ClaimCenter Java entities, including both the entity data and additional methods called domain methods
- General Java utility classes in a separate Javadoc directory.

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

ClaimCenter Upgrade Guide – Provides instructions to upgrade ClaimCenter.

ClaimCenter Data Dictionary – Describes the ClaimCenter data model, including your custom data model extensions. To generate the dictionary, go to the `ClaimCenter/bin` directory and run the `gwcc regen-dictionary` command. To view the dictionary, open the `ClaimCenter/build/dictionary/data/index.html` file. For more information about generating and using the *Data Dictionary*, see the *ClaimCenter Configuration Guide*.

ClaimCenter Security Dictionary – Documents security permissions, roles, and the relationships between them. Generate the dictionary by going to the `ClaimCenter/bin` directory and running the `gwcc regen-dictionary` command. To view the dictionary, open the `ClaimCenter/build/dictionary/security/index.html` file. For more information about generating the *Security Dictionary*, see the *ClaimCenter Configuration Guide*.

Conventions In This Document

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

Support

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

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

part I

Gosu Business Rules

chapter 1

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

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 "Rule and Rule Set Naming Conventions" on page 33.

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. The Rule engine 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.) The Rule engine 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, the work of the Rule engine 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 the Rule engine 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 92

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 18
- "Generating a Rule Repository Report" on page 91
- "Generating a Rule Execution Report" on page 92

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 the Rule engine 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 97" for more information.
Entity	An entity is a business data model class or instance such as `Claim` or `Policy`. Entities are a subset of objects. (See Object.)
Guidewire Studio	Guidewire Studio is the Guidewire administration tool for managing ClaimCenter resources (PCF pages, business rules, and Gosu classes, for examples).
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 `gw.api.*` packages).
Object	An object is an instance of any class such as `ArrayList`, `Claim`, or `Activity`. (See Entity.)

Rule	A rule is a single decision in the following form:
	`If {`*`some conditions`*`}`
	`Then {`*`take some action`*`}`
	For example, you can create a validation rule that means the following:
	If the claim has no loss date or a loss date in the future, then mark the loss date field as invalid.
	This individual rule is just one of many validation tests that you can conduct on a new claim, for example.

Term	Definition
Rule engine	This is the part of the ClaimCenter application that evaluates rules for a given claim or exposures. It uses the rules and rule sets within Guidewire Studio to make automated decisions.
Rule set	A rule set combines many individual rules into a useful set to consider as a group. For example, the Claim Segmentation 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.

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	Route approval activities for a transaction that requires approval to the appropriate user.
Archive	Determine if ClaimCenter can archive that particular claim.
Assignment	Determine the responsible party for an activity, claim, exposure, or matter.
BulkInvoice Approval	Define the approval requirements for Bulk Invoices.
Closed	Perform an action if you close an activity, claim, exposure, or matter—for example, sending a notification.
Event Message	Handle communication with integrated external applications. See the *ClaimCenter Integration Guide* for information about events and event messaging.
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.
Initial Reserve	Determine the initial financial reserves to create for a new exposure.
Loaded	Trigger automatic actions if the FNOL import interface loads a new claim.
Postsetup	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.
Presetup	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, Assignment, and Workplan rules.
Preupdate	Trigger automatic actions any time that a claim or other high-level entity changes.
Reopened	Trigger automated actions on reopening a claim or exposure.
Segmentation	Categorize a new claim or an exposure based on complexity, severity of damage, or other attributes.
Transaction Approval	Determine if a user has the required authorization to submit a financial transaction.
Validation	Check for missing information or invalid data on Address Book, Claim, Exposure, Matter, Policy and Transaction objects.
Workplan	Add initial activities to a claim, exposure or matter as a checklist of work that various people need to do on the claim.

chapter 2

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

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: • 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
Conditions	A Boolean expression (that is, a series of questions connected by AND and OR logic that evaluates to TRUE or FALSE). For example:
	(This is an auto claim) AND (the policy includes collision coverage)
Actions	A list of actions to take. For example:
	{Mark the claim as flagged}
	{Add an activity for the adjuster to follow up}

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 opens 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 material in the *Gosu Reference Guide*.)

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).

- **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 **Actions** pane.

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

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

- **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 Your 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 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 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 Your Actions

Within the **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", "loadsave",
          "Please provide information about claimant's other coverage.", null, null )

      }

    }
```

(See “Correcting Validation Errors” on page 82 for a discussion of the various `reject` methods.)

Exiting a Rule

At some point in the decision tree, the Rule engine makes the decision that you want. At this point, it is important that the Rule engine not continue. Indeed, if rule checking did continue, if the Rule engine processed the *default* rule, it might overwrite the decision that came earlier. Therefore, you need to be able to instruct the Rule engine 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 the Rule engine what to do next.

Flow control action	Description
`actions.exit`	This is the simplest version. The Rule engine stops processing the rule set as soon as it encounters this action.
`actions.exitAfter`	This action causes the Rule engine to stop processing the rule set after processing any child rules.
`actions.exitToNext`	This action causes the Rule engine 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.
`actions.exitToNextParent`	This action causes the Rule engine 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.
`actions.exitToNextRoot`	This action causes the Rule engine 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), the Rule engine 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, or Constructor” on page 199 in the *Gosu Reference Guide*.

Invoking a Gosu Rule from Gosu Code

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

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

chapter 3

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

ClaimCenter organizes and displays rules as a hierarchy within the center pane of Guidewire Studio, with the rule set itself appearing at the root (top level) of the hierarchy tree. Studio displays the **Rule** menu only if you first select a rule set category in the **Resources** pane. 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 find or replace text in the current view
- To find usages of an expression or statement
- To find or replace text throughout the resource tree
- To change the root entity of a rule

To view or edit a rule

1. Expand the **Rule Sets** drop-down list in the left-hand (**Resources**) pane, and then expand the rule set category.
2. Select the rule set to view or edit. All editing and saving within 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 on the rule that you want to move, 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, then ClaimCenter moves the chosen rule and all of its children as a group. This 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.

- Rules sets are logical groupings of rules that 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. Select **Rule Sets** in the Studio **Resources** tree.
2. Right-click and select **New** → **Rule Set Category** from the menu.
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 rules sets, which organize the business rules by function. It is possible to create new rule sets through Guidewire Studio.

1. Expand the **Rule Sets** node in the Studio **Resources** tree. Although Guidewire labels the node as rule sets, the node itself actually lists rule set categories.
2. Select a rule set category or create a new rule set category.
3. Right-click and select **New** → **Rule Set** from the menu.

4. Enter the following information:

Field	Description
Name	Studio displays the rule name in the middle pane, under **Rules**. For the Guidewire recommendations on rule set names, see "Rule and Rule Set Naming Conventions" on page 33 in the *Rules Guide*. In general, however, if you create a rule set for a custom entity named `Entity_Ext`, then 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 to 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**. (You can also press Ctrl+Shift+N.)
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 on a rule and selecting the command from the drop-down list.

Command	Description	Actions you take
Rule → **Rename**	Renames the currently selected rule	Select the rule that to rename in the center pane of Studio, then select **Rename** from the **Rule** menu (or right-click and select **Rename**). Enter the new name for the rule in the **Input** dialog box. You must save the rule for the change to become permanent.

Command	Description	Actions you take
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**). (There is no secondary dialog window that asks you to confirm your selection.) You can only use the **Delete** command 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 **Resources** tree. ClaimCenter stores the rule definition classes in several different locations, for example:

```
ClaimCenter/modules/pl/config/rules/...
ClaimCenter/modules/cc/config/rules/...
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 in synchronization.

Using Find-and-Replace

The Guidewire Studio **Search** menu provides a number of ways that you can search for text or objects in the active Studio resources. You can also access some of the search commands by using the right-click menu of the currently selected resource. Search commands work in one of three modes:

Find/Replace, Find Next	Searches/replaces text in the currently focused view.
Find Usage	Searches for usages of an expression or statement throughout the active resources.
Find/Replace in Path	Searches/replaces text through all or part of the resource tree.

Viewing Search Results

Studio displays the results of a search in a **Search** pane at the bottom of the screen. This pane contains a **Text Filter** entry field that you can use to further refine your search. For example, suppose that the search returns 105 instances of the use of the word *Activity* in a global search. You can further refine your search by entering `Activity.Subject` in the **Text Filter** field. Studio filters the search results and displays (for example) the three occurrences of the search results that contain this specific text.

To find or replace text in the current view

1. Open a rule or class in its Gosu editor. This is the currently selected view.
2. Select either **Find...** or **Replace...** from the **Search** menu.
   - If you chose **Find...**, you can search for text in the currently selected view.
   - If you chose **Replace...**, you can search for and replace text in the currently selected view. To replace text, enter the text string on which to search, and its replacement in the appropriate fields. If Studio finds a text match, it presents you with the option of replacing the text in this one match or in all matches it finds.

   Notice that you have several search options from which to choose. These include:
   - **Case sensitive** - If checked, Studio searches for an exact text string match, including capitalization.
   - **Regular expression** - If checked, Studio uses the provided text string as a search pattern. For example, entering “[Cc]laim” matches to either “claim” or “Claim”, but not to “CLAIM”.

If it finds a match, Studio highlights the text. To move to the next occurrence of the found text, select **Find Next** from the **Search** menu, or press F3.

To find usages of an expression or statement

1. Place the cursor in an expression or statement within the currently selected view. Studio uses the text at the cursor location as the search item. For example, if you place your cursor in the first word in the following expression:

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

   Studio searches for all usages of the word *Activity*.
2. Select **Find Usages** from the Search menu.
3. Select the scope for the search:
   - **Global** - Searches for usages throughout all active resources.
   - **Current view** - Searches for usages in the current view only.

Studio displays the results of a search in a **Search** pane at the bottom of the screen.

To find or replace text throughout the resource tree

1. Select a resource.
2. Select either **Find in path...** or **Replace in path...** from the **Search** menu, depending on your purpose.
   - If you chose **Find in path...**, then you can search for text or all/part of a resource name.
   - If you chose **Replace in path...**, then you can both search for and replace text. To replace text, enter the text string on which to search, and its replacement in the appropriate fields. If Studio finds a text match, it presents you with the option of replacing the text in this one match or in all matches it finds.

   Notice that you have multiple options for the search. These include:
   - **Search all active resources** - If checked, Studio searches for all active resources, not just the currently selected view.
   - **Search in view** - If checked, Studio searches in the current view only.
   - **Case sensitive** - If checked, Studio searches for an exact text string match, including capitalization.
   - **Regular expression** - If checked, Studio uses the provided text string as a search pattern. For example, entering “[Cc]laim” matches to either “claim” or “Claim”, but not to “CLAIM”.
   - **Recursive search** - If unchecked, Studio performs the search in the current node only. If checked Studio performs the search in the current node and all its subnodes.

Studio displays the results of the search in a **Search** pane at the bottom of the screen.

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. This indicates 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
   ```

   Note: 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 as the declarations failed to parse. **Guidewire does not recommend the use of 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 /*startOOrule*/true/*endOOrule*/
     }
     ...
   }
   ```
   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 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** for 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 replace occurrences of the old entity in the function declarations of all the files in the rule set with the new entity. This 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.*

IMPORTANT The intent of this command is to enable you to edit a rule that you otherwise cannot open in Studio as the declarations failed to parse. Guidewire does **not** recommend the use of this command in other circumstances.

Validating Rules and Gosu Code

Guidewire Studio provides several means to verify the validity of the rules and Gosu code that you construct. The simplest and most visible is to view the validation status indicators located at the bottom of the right-most Studio pane (or subpanes, for rules). The validation status indicates the following conditions:

- It shows *green* if the code you enter passes the Studio internal validation.
- It shows *yellow* if the code is valid but contains deprecated references.
- It shows *red* if the Gosu code does not pass validation.

If you enter non-valid code, Studio provides an error message next to the validation status indicator.

Making a Rule Active or Inactive

The Rule engine 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.

Use the following menu command to make a rule active or inactive.

Command	Description	Actions you take
Rule → **Active**	Toggles the rule active status	Select a rule in center pane, then select **Active** from the **Rule** menu to make it active. Active rules have a double check mark next to **Active** in the **Rule** menu besides a check in the box next to the rule name in the center pane. You can also do the following: • Check the box next to the rule to make it active. • Uncheck the box next to the rule to make it inactive.

chapter 4

Writing Rules: Best Practices

This topic describes Guidewire-suggested best practices for naming rules, functions and methods. It also includes information on adding comments to rules code to enhance readability and how to log debug information.

This topic includes:

Rule and Rule Set Naming Conventions

Note: For information on the Guidewire guidelines for naming data model extensions, see "Naming Guidelines for Extensions" on page 236 in the *Configuration Guide*.

Rule Set Naming Conventions

There is one important naming convention that you must follow if you create a new rule set. This is especially true under the following conditions:

1. You are creating a rule set that contains rules that interact with a custom (non-base configuration) entity.
2. You are creating a Preupdate or Validation rule set.

The rule is that you use the name of the entity and append the rule set name to it. In other words, use the following syntax to name rule sets:

```
<CustomEntity><RuleSetName>
```

If there is any doubt about what you need to append to the entity name, view the existing rule sets in that category. Thus, you see in Studio, for example, if you expand the **Rule Sets** node and then the individual rule set nodes:

Rule set category	Rule set name to append
Assignment	AssignmentRules
Preupdate	Preupdate
Validation	ValidationRules

Notice that not all of the rule sets follow the same pattern. If there is any doubt, simply duplicate the pattern that currently exists for that rule set category.

For example, suppose that you create the following entity:

```
Abc_Ext
```

You want to run preupdate rules against this entity and decide to create a special rule set to hold your preupdate rules. In this case, you must name your rule set as follows:

```
Abc_ExtPreupdate
```

If you want to create a validation rule set for this entity, then name it accordingly:

```
Abc_ExtValidationRules
```

IMPORTANT If you specify a name for the rule set other than the mandated name, ClaimCenter does **not** trigger the rules properly.

Rule Naming Conventions

Each rule name within a rule set must be unique. As a best practice, Guidewire strongly recommends that you use the following naming convention for each Guidewire Studio rule that you create to avoid problems during configuring and testing rules:

8 character alphanumeric identifier + hyphen + description

Use this format for the eight character alphanumeric identifier:

ABC12345

In this format, *ABC* is the unique rule set identifier and *12345* is the numeric ID. Use the first two digits after the alpha characters to identify the root rules. Use the next digit to identify the next nested rule, the next digit by its nested rule, and so forth. For example:

```
ABC01000 - Parent rule
ABC02000 - Parent rule
ABC03000 - Parent rule
ABC04000 - Parent rule
ABC05000 - Parent rule
  ABC05100 - Level 2 rule
  ABC05200 - Level 2 rule
...
  ABC05700 -Level 2 rule
    ABC05710 - Level 3 rule
    ABC05720 - Level 3 rule
      ABC05721 - Level 4 - rule
      ABC05722 - Level 4 - rule
      ABC05730 - Level 3 - rule
      ABC05741 - Level 4 - rule
      ABC05742 - Level 4 - rule
```

IMPORTANT ClaimCenter limits rule name length to a maximum of 60 characters as you create the rule.

Use the following principals in naming your rules.

Rule 1: Create Unique Rules

The first important concept to enforce during rule creation is rule uniqueness. *Each rule name within a rule set must be unique.*

Rule 2: Create Easily Identifiable Rules

The second important concept to enforce during rule creation is easy identification of rules during configuring and testing rules. It is possible to create parent and child rules that have the same first eight characters, but Guidewire does not recommend this practice. It is better to create rules in which there is no possible confusion between the different rules.

For example, suppose that you want to use a `print` statement within an action block to identify the currently executing rule:

```
print( “Rule:  “ + gw.api.util.StringUtil.substring( actions.getRule() as java.lang.String, 0,8) )
```

The use of this library function outputs the rule name to the console window as Studio executes the rule. If multiple rules have the same first eight characters, then identification of the correct rule becomes much more difficult.

Rule 3: Create Consistently Named Rules

The third important concept is that different rule naming schemes are acceptable as long as rule names are consistent. Guidewire recommends that you appoint one individual in your organization to develop a simple naming convention and to enforce that naming standard.

Adding Comments to Rules

Guidewire strongly recommends that you comment your Gosu code as you create it. There are four styles of implementation comments:

- Block
- Single-line
- Trailing
- End-of-line

Guidewire recommends that you insert block comments for every method that you write. Use the block comment to briefly describe the method. Use the other comments for better readability.

Block Comments

Use block comments to provide descriptions of classes, functions, and enhancements. Guidewire recommends that you insert block comments at the beginning of each file and before each method. They can also be used in other places, such as within methods. Indent block comments inside a method to the same level as the code they describe. The following is an example of a comment block.

```
/*
* Here is a block comment with some very special
* formatting.
*
* one
* two
* three
*/
```

Single-Line Comments

Short comments appear on a single line indented to the level of the code that follows. If a comment cannot be written in a single line, follow the block comment format. The following is an example of a single-line comment:

```
if(condition) {
  /* Handle the condition. */
  ...
}
```

Trailing Comments

Very short comments can appear on the same line as the code they describe, but shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, indent all comments to the same tab setting. The following is an example of a trailing comment:

```
if(a == 2) {
  return true    /* special case */
}
else {
  return false   /* works only for odd a */
}
```

End-Of-Line Comments

Use the `//` comment delimiter to comment out a complete or partial line. Do **not** use it on consecutive multiple lines for text comments. You can use it, however, on consecutive multiple lines for commenting out sections of code. The following shows examples of this usage:

```
if(someExpression > 1) {
  // Do something useful.
  ...
}
else {
  return false // Explain why here.
}
```

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.
- Use validation warning and error messages embedded in the ClaimCenter validation pane to provide information as appropriate.

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 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")
```

Logging Debugging Information

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

- logDebug(Text)
- logError(Text)
- logInfo(Text)
- logTrace(Text)
- logWarn(Text)

You set the message to log in the *Text* parameter. You set the importance (severity) level by the method you choose. Then, depending on how you configure your application-wide logging settings, ClaimCenter sends messages of a certain severity level to the designated location as they occur.

These methods correspond to the following severity levels (in order of increasing information):

Level	Description
TRACE	Provides messages about processes that are about to start or that completed. Trace logging has no or minimal impact on system performance.
DEBUG	Messages that test a provable and specific theory intended to reveal some system malfunction. For example dumping the contents of an XML document is fine.
INFO	Messages intended to convey a sense of correct system operation such as a component has started, perhaps a user has entered or left ClaimCenter, and so forth.
WARN	Messages that denote something potentially out of ordinary is happening. Examples include: assignment rules that do not end in an assignment, a plugin call that took 90 seconds, and so forth.
ERROR	Messages that denote something is wrong. For example, a remote system has refused a connection within a plugin, or ClaimCenter can not complete some operation even with a default.

By default, ClaimCenter sends the output from the `gw.api.util.Logger.*` logging methods to both the application system console and to file `cclog.log` (typically found in the `Guidewire/ClaimCenter/logs` directory). Unless instructed otherwise, it also sends information concerning rule operations to these same locations.

You set general application logging for the ClaimCenter application server in the following file:

```
config/logging/logging.properties
```

This file controls how much information is written to the log and to which location it is sent.

Note: For more information, see “Configuring Logging” on page 35 in the *System Administration Guide* for details.

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 *Gosu Reference Guide* for information about creating and using Gosu class methods.
- See “Script Parameters” on page 111 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(),
  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
```

Adding Comments

Guidewire strongly recommends that you comment your rules as you create them. This improves readability, and provides information useful in future debugging efforts. See “Adding Comments to Rules” on page 35 for information on creating and adding comments to your rules.

Adding Validation Warning and Errors

ClaimCenter provides the ability to display warnings and errors directly within ClaimCenter as visual messages. See “Validation” on page 79 for details on how to add validation warnings and errors to the ClaimCenter interface.

Handling Errors

The `try...catch...finally` statement blocks provides a way to handle some or all of the possible errors that can occur in a given block of code during runtime. It has the following syntax:

```
try {
  [try statements]
} catch(exception) {
  [catch statements]
} finally {
  [finally statements]
}
```

The elements of this construction have the following meanings:

`try`	Statements in which an error can occur. The use of a `try` expression is optional. However, you cannot have a `catch` block or a `finally` block without an initial `try` block.
`exception`	Any variable name. The initial value of `exception` is the value of the thrown error. You must have an `exception` name if you have a `catch` statement.
`catch`	Statements used to handle any errors that occurred in the preceding `try` statements. The use of a `catch` statement is optional.

`finally`	Statements that are unconditionally executed after all other error processing has occurred. The use of a `finally` block is optional.

You use the `try...catch...finally` blocks to handle any errors that might occur in the `try` statements. If an error occurs in the `try` block, the Rule engine passes program control to the `catch` statements. The value of exception is the value of the error that occurred in the `try` statements. If no error occurs, the `catch` statements are not executed. For errors that you do not handle in the `catch` statements, the Rule engine generates an application error.

Sometimes you cannot handle a specific error in the `catch` statements associated with the `try` statements in which the error occurred. In this case, use the `throw` statement to rethrow the error to higher-level error handling. One set of `try...catch...finally` blocks can be nested within a higher-level `try` statement. If you throw an exception from lower-level `catch` statements, it is caught by the higher level `catch` statements.

After all statements in `try` statements have been executed and any error processing has been completed in the `catch` statements, `finally` statements are unconditionally executed. The code inside the `finally` statement block always executes. Even if the `catch` statements rethrows the error, or if a `return` statement occurs inside the `catch` or `try` blocks, `finally` statements are guaranteed to run. The only exception to this if you have an unhandled error condition. In this case, ClaimCenter processes this run-time error by logging the error condition.

Note: See also “Exception Handling” in the *Gosu Reference Guide* for more information on how Gosu handles errors and exceptions.

Example

The following is an example of using nested `try...catch...finally` block statements:

```
//Some class
function ex_printinfo() {
  try {
    print("In outer try block...")

    try {
      throw "Error 325 from inner block."
    } catch(e) {
        print("Inner catch block caught " + e)
        throw e + " Rethrown from inner block."
    } finally {
        print("In inner finally block...")
    } //End inner try-catch-finally block

  } catch(e) {
      print("Outer catch block caught " + e)
  } finally {
      print("In outer finally block...")
  } //End outer try-catch-finally block
}

//Output
In outer try block...
  Inner catch block caught Error 325 from inner block.
  In inner finally block...
Outer catch block caught Error 325 from inner block. Rethrown from inner block.
  In outer finally block...
```

In the preceding example, the first `throw` statement passes the message “Error 325 from inner try block” from the inner `try` block to the `inner` catch block. The second `throw` (inside the inner `catch` block) adds the string “Rethrown from inner catch block” to the original error message, and throws it to the outer `catch` block. The outer `catch` block adds the passed error message to the string “Outer catch block caught” and prints it.

chapter 5

Writing Rules: Examples

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

This topic includes:

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” 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 the Rule engine 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 the Rule engine finds an exception, you need to instruct the Rule engine 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:

Conditions

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

Actions

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

In this example, the Rule engine 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 the Rule engine 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 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"))
```

Note: 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 Entity Data Changed in a Bundle" on page 285 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.

Condition

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

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
```

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

chapter 6

Rule Set Categories

This topic reviews the sample rules that Guidewire provides as part of the base ClaimCenter application. It also includes the minimal set of sample rules that Guidewire provides with Guidewire ContactCenter. The text identifies which rule sets are specific to ContactCenter. You can access ContactCenter rule sets only from within ContactCenter Studio. However, in all cases, you access the Gosu business rules through **Studio** → **Resources** → **Rule Sets**.

This topic includes:

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.
Archive	Determine if ClaimCenter can archive that particular claim.
Assignment	Determine the responsible party for an activity, claim, exposure, or matter.
BulkInvoice Approval	Define the approval requirements for Bulk Invoices.
Closed	Perform an action anytime that you close an activity, claim, exposure, or matter—for example, sending a notification.
Event Message	Handle communication with integrated external applications. See the *ClaimCenter Integration Guide* for information events and event messaging.
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.
Initial Reserve	Determine the initial financial reserves that to create for a new exposure.
Loaded	Trigger automatic actions anytime that you load a new claim using the FNOL import interface.
Postsetup	Trigger automatic actions anytime that you add a new claim within ClaimCenter. These rules run after the Assignment and Workplan rules complete their execution.
Presetup	Trigger automatic actions anytime that you add a new claim, exposure, matter or transaction within ClaimCenter. These rules run prior to the execution of the Segmentation, Assignment, and Workplan rules.
Preupdate	Trigger automatic actions anytime that a claim or other high-level entity changes.
Reopened	Trigger automated actions on reopening a claim or exposure.
Segmentation	Categorize a new claim or an exposure based on complexity, severity of damage, or other attributes.
Transaction Approval	Verify that a user has the appropriate authorization to submit a financial transaction.
Validation	Check for missing information or invalid data on Address Book, Claim, Exposure, Matter, Policy and Transaction objects.
Workplan	Add initial activities to a claim, exposure or matter as a checklist of work that various people need to do on the claim.

Approval Routing

Approval routing rules route an approval activity for a transaction that requires approval. The Rule engine uses approval routing rules to determine which person to assign the approval activity. (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 and Transaction Approval rule set categories form a pair. The Rule engine calls these rule sets sequentially.

- The Rule engine calls the Transaction Approval rule set anytime that you submit any kind of financial transaction.

- The Rule engine calls the Approval Routing rule set if the outcome of transaction approval is Yes, or, if the transaction authority limit fails such that the transaction needs approval.

IMPORTANT Be careful not to create approval rules that generate an infinite loop. For example, suppose a rule first assign 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. The Rule engine 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. (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
`approveByGroupSupervisor`	Use to assign an approval active to a group supervisor based on the exposure or exposures associated with the transaction: • If the transaction set has transactions associated with *only one* 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 *more than one* exposure—the method assigns the approval activity to the supervisor of the group assigned to the associated claim. In either case, if the supervisor of the group is the same as the user requesting approval, then the activity remains unassigned. *If you use this method, you must explicitly handle this case.*
`approveByUserSupervisor`	Use to assign an approval activity to a group supervisor based on the user who submitted the transaction: • 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 *one and only one* group, then the approval goes to the supervisor of that group. Otherwise, approval goes to the supervisor of the owning group. 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 *defaultowner*.
`setApprovingUser`	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 (`user.credential.Active != true`) and returns `false` if either of these conditions exist. It also returns `false` if the user or group is missing. It returns `true` otherwise. **IMPORTANT** Only call this method from within the Approval Routing rule set.

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

Condition

```
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 Rules Assignment rules and the BulkInvoice Approval rule sets form a pair in a similar fashion to the Approval Routing and Transaction Approval rule sets. ClaimCenter runs the BulkInvoice Approval Assignment rule set for each BulkInvoice to determine to whom to assign it for approval.

In the following example, the rule sets approval for bulk invoice payments to user Karen Brown in the Headquarters group.

Conditions

```
true
```

Actions

```
BulkInvoice.setApprovingUser( User("user:44" /*Karen Brown*/), Group("group:64" /*Headquarters*/))
```

Archive

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 “The Archive Source Plugin” on page 286 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 allows you to search for archived claims in a similar fashion as you search for non-archived claims in the main database. Within 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 restores an archived claim (after you click the **Restore** button) any time that you want to work with an archived claim found through a search.

See also

- “Archiving” on page 87 in the *Application Guide* – information on archiving claims, searching for archived claims, and restoring archived claims.
- “Archive Parameters” on page 39 in the *Configuration Guide* – discussion on the configuration parameters used in claims archiving.
- “Archiving Claims” on page 665 in the *Configuration Guide* – information on configuring claims archiving, selecting claims for archiving, and archiving and the object (domain) graph.
- “Archiving Integration” on page 285 in the *Integration Guide* – describes the archiving integration flow, storage and retrieval integration, and the `IArchiveSource` plugin interface.

- "Archive" on page 48 in the *Rules Guide* – information on base configuration archive rules and their use in detecting archive events and managing the claims archive and restore process.
- "Logging Successfully Archived Claims" on page 37 in the *System Administration Guide*.
- "Purging Unwanted Claims" on page 58 in the *System Administration Guide*.
- "Archive Info" on page 160 in the *System Administration Guide*.
- "Upgrading Archived Entities" on page 62 in the *Upgrade Guide*.

IMPORTANT To increase performance, most customers find increased hardware more cost effective than archiving unless their volume exceeds one million claims or more. Guidewire strongly recommends that you contact Customer Support before implementing archiving to help your company with this analysis.

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 restored 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 restored claims from other claims.

Skipped Claims Versus Excluded Claims

Through the Archive rules, you can do the following:

Skip a claim	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: `claim.skipFromArchiving(String)` **IMPORTANT** Calling this method on a claim terminates rule set execution.
Exclude a claim	Excluding a claim from archiving makes that claim unavailable for archiving during this and future archiving passes. This makes the claim 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: `claim.reportArchivingProblem(String)` Calling this method on a claim does **not** terminate rule set execution.

Detecting Excluded Claims

There are several cases in which ClaimCenter excludes claims from archiving. They are:

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

`ClaimInfo.ExcludeFromArchive`	(Boolean) If `true`, ClaimCenter excluded this claim from the archiving process.
`ClaimInfo.ExcludeReason`	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, this can be for a foreign key violation. In this case, ClaimCenter populates the following `ClaimInfo` properties, which you can use to view more details of the failure.

`ClaimInfo.ArchiveFailure`	Provides a one-line summary of why the claim failed the archive process
`ClaimInfo.ArchiveFailureDetails`	Provides the full stack trace related to the failure.

Managing Skipped or Excluded Claims

You can view information about skipped and excluded claims in the **Server Tools** → **Info Pages** → **Archive Info** page. This page 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 159 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 Rules	Checks...
ARC01000 - Claim State Rule	*If the claim is closed?* If it 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 Draft or Not Valid status, or a status of: • Approved • Check Pending Approval • Awaiting Submission If so, ClaimCenter skips this claim. The intent is not to force a user to restore 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. This is because the claim can retain one of these statuses long after a Bulk Invoice item is escalated and cleared.

Default Group Claim Archiving Rules	Checks...
ARC04000 - Open Activities Rule	*If the claim has open activities.* If so, then ClaimCenter skips the claim. **Note** The Work Queue writer does not typically mark a claim with open activities for archiving. 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 there are incomplete reviews on the claim.* If so, ClaimCenter skips the claim.
ARC06000 - Unsynced Review Rule	*If a claim has reviews than have not been synchronized with ContactManager.* If so, ClaimCenter skips the claim.
ARC07000 - Transaction State Rule	*If a claim has transactions that have yet to be escalated or acknowledged.* If so, ClaimCenter skips the claim.

It is possible for you to add your own, additional, archive-related rules to this rule set, or to modify the sample rules that Guidewire provides. Thus, it is possible to write archiving rules that reflect your unique business conditions. For example, you can write rules to do the following:

- Do not archive a sensitive claim, or one with restricted access control.
- Do not archive a claim that meets a certain condition, such as having medical payments over some amount.
- Do not archive a claim whose claimant has other open claims pending.

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 `claim.PurgeDate` to seven years after `Claim.CloseDate`. Guidewire disables this rule in the base configuration.
Claim Reopened	CRO03000 - Sample rule to remove purge date	Clears (sets to `null`) the `claim.PurgeDate` 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 inactivate) 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 `claim.DateEligibleForArchive` to a date created by adding the value of configuration parameter `DaysClosedBeforeArchive` to the current system date.
Claim Reopened	CRO04000 - Clear archive eligibility date	Sets `claim.DateEligibleForArchive` 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 inactivate) the two rules in conjunction with each other.

Assignment

Guidewire refers to certain business entities as *assignable* entities. This means that 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

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 97.

BulkInvoice Approval

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 in a similar fashion to the Approval Routing Rules and Transaction Approval rule sets.

See Also

- For information on bulk invoice web service APIs, validating bulk invoices, and bulk invoice processing performance, see "Bulk Invoice Integration" on page 221 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 contains a sample approval rule for bulk invoices that always requires approval. Guidewire ships this rule disabled for obvious reasons. As indicated, 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 a check for the subtype of the transaction set (for example, `TransactionSet.subtype == "checkset"`). This is 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 such a check in their rule conditions. This is because the only implicit entity that ClaimCenter ever passes to the bulk approval rules is a bulk invoice.
- As the previous item implies, 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 is different from the Transaction Approval rules, which have an implicit reference to the transaction set being approved.

In the following example, the rule requires that a human manually approve each bulk invoice. (A rule in the BulkInvoice Approval Assignment Rules determines who can actually approve bulk invoices.) As noted, Guidewire ships this rule disabled in the base configuration.

Conditions

```
true
```

Actions

```
BulkInvoice.requireApproval( "Bulk invoices require approval." )
```

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

These rule sets fire anytime that you close an activity, claim, exposure, or matter. They fire as immediately as you close one (not before). You can use these rules to send a notification or to generate additional activities, for example. The following two examples illustrate how to use these rule sets.

- The first example rule closes the claim file from within the Exposure Closed rule set.
- The second example rule closes a claim automatically if all of its Exposures are in a status of “Closed”. It runs in the Claim Closed rule set.

Condition

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

Actions

```
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 rule set) works in conjunction with the previous example. The rule closes any open activities that are not available to closed claims. This means that if ClaimCenter closes a a claim, the rule closes all open activities associated with the claim as well.

Condition

```
true
```

Actions

```
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 as last exposure closed." }
    else { act.Description = act.Description + "Closed by ClaimCenter as last exposure closed." }
  }

}
```

Activity Closed

The Rule engine 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.

Conditions

```
Activity.ActivityPattern.Code == “recover_vehicle” and Activity.Status =="complete"
```

Actions

```
if (Activity.Exposure <> null) {
  Activity.Exposure.VehicleIncident.DateVehicleRecovered = gw.api.util.DateUtil.currentDate()
}
```

Claim Closed

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
CCCA0001 - Suspend AutoSync for Related Contacts	Helps facilitate the synchronization of contact information between ClaimCenter and ContactCenter. See “Synchronizing Contacts between ContactCenter and ClaimCenter” on page 39 in the *Contact Management Guide* for more information.
CCCA0002 - 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 → CCCA0002 - Sample rule to remove purge date* See “Archive Claim Purge Rules” on page 51 for more information.

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

Conditions

```
true
```

Actions

```
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 as last exposure closed."
    }
    else {
      act.Description = act.Description + "Closed by ClaimCenter as last exposure closed."
    }
  }
}
```

Exposure Closed

These rules execute immediately after the Exposure Closed Validation Rules run. Use these rules to take automated actions on the close. In the following example, the rule closes a claim anytime that a user closes the last open exposure associated with the claim.

Condition

```
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

A Matter is the set of data organized around a single lawsuit or potential lawsuit. This 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 close.

Event Message

IMPORTANT ClaimCenter runs the Event Message rules as part of the database bundle commit process. **Only** use these rules 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, the Rule engine:

- Runs the rules in the Event Fired rule set once for every event for which a message destination has registered interest.
- Runs the Event Fired rule set once for each destination that is listening for that particular event. Thus, it is possible for the Rule engine to run the Event Fired rules sets multiple times for each event, once for each destination interested in that event.

Messaging Events

ClaimCenter automatically generates certain events for most top-level objects. (Guidewire calls these events *standard* events.) In general, this occurs for any addition, modification, or 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)
```

For information on...

- Base configuration events, see “List of Messaging Events in ClaimCenter” on page 156 in the *Integration Guide*.
- Custom events, see “Triggering Custom Events” on page 161 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 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. Thus:

- 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. As this 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 availabe 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 )
}
```

A Word of Warning

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. Only change fields that are not modifiable from the user interface. For example, set custom data model extension flags only used 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 of the Event Message rule set.

See Also

- "Messaging Overview" on page 140 in the *Integration Guide*
- "List of Messaging Events in ClaimCenter" on page 156 in the *Integration Guide*
- "Using the Messaging Editor" on page 161 in the *Configuration Guide*

Detecting Object Changes

It is possible that you want to listen for a change in 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 (an update of the address, perhaps). However, in this case, 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 in itself 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 anytime that the address changes. In actual practice, you would use a different message destination, of course.)

Condition

```
//DestID 68 is the Console Message Logger
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

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 151 for information on how to construct a document within Guidewire ClaimCenter/

Conditions

```
MessageContext.Root typeis Recovery and (MessageContext.Root as Recovery).Status == "submitting"
```

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["CBO"]          = rec.CurrentAssignment.AssignedGroup.RootGroup
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 the *ClaimCenter Integration Guide*. See also the Claim Preupdate rules (Metropolitan Police Report Request Rules).

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 73.
- 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.

Exception

Use the Exception rule sets to specify an action to take if an Activity or Claim is overdue and is in the escalation state. You can use this rule set to send e-mail, create an activity or to set a flag, among other actions.

The Rule engine runs these rule sets at regularly scheduled intervals. See the *ClaimCenter System Administration Guide* for information on exception batch processes.

Activity Escalation Rules

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

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.

ClaimCenter runs scheduled process `activityesc` every 30 minutes (by default). This system process runs against all open activities within ClaimCenter to find activities that need to be escalated using 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, the Rule engine 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 enters the escalated condition. 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 intervene and check on why the work is not yet complete. Possibly, you want to mark the claim as a flagged claim as well. (See “Claim Exception Rules” on page 59 for how to set a claim flag.) You can also decide that there is no follow-up action needed for certain kinds of activities.

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

Condition

```
Activity.Priority == “high“
```

Action

```
activity.claim.setFlag(“Overdue high priority item: “ + 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 135.

ClaimCenter monitors claims for unusual or potentially concerning conditions so that adjusters or supervisors can be alerted to look into it. 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:

- *It must have changed significantly* — 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.)
- *It must have been idle for a length of time* — The claim is idle (aging without any significant change to it) for a certain length of time. The `IdleClaimThresholdDays` configuration parameter defines the number of days after which ClaimCenter considers a non-modified claim as idle.

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

To run separate schedules for changed claims and idle aging claims, set the configuration parameter `SeparateIdleClaimExceptionMonitor` to `true` (in `config.xml` within Studio). The following table lists the configuration parameters that specifically affect changed and idle claims. See the *ClaimCenter Configuration Guide* for a discussion of ClaimCenter configuration parameters.

Configuration parameter	Description	Default value
`IdleClaimThresholdDays`	Number of days without significant change after which ClaimCenter considers a claim idle.	7
`SeparateIdleClaimExceptionMonitor`	If true, ClaimCenter runs separately scheduled processes for determining changed claims and idle aging claims.	true

The following table lists the batch processes that work with changed and idle claims. See the *ClaimCenter System Administration Guide* for a discussion on the ClaimCenter batch processes.

Batch process	Description	Default time
`claimexception`	Runs the claim exception rules on claims that have been modified significantly since the last time that ClaimCenter ran this process.	Every day at 2:00 A.M.
`idleclaimexception`	Runs the claim exception rules on claims that are idle. ClaimCenter considers a claim idle if the exception rules have not been run on it in the number of days specified by the `IdleClaimThresholdDays` configuration parameter.	Every Sunday at 12:00 noon.

There are many different kinds of exceptions for which you can possibly 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, then 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 the properties that you can use with flagged claims. ClaimCenter displays the first three listed properties in the **New Loss** screen for the claim.

Property	Description
`Flagged`	Sets the flagged status of a claim: • isFlagged • neverFlagged • wasFlagged After set, use to determine a status of a claim.

Property	Description
`FlaggedReason`	Provides a reason for the flag to help the adjuster understand the nature of the exception noticed.
`FlaggedDate`	Indicates the date on which a flag was set.
`clearFlag`	Toggles the `Flagged` property and clears the `FlaggedReason` field. Use this method after the claim is no longer considered to have any exceptions.

In a similar fashion to the example shown in the Activity Escalation Rules, you can also use the `setFlag` method to set the flagged reason text.

```
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 43, be mindful in writing exception rules to prevent ClaimCenter from reporting the same exceptions repeatedly.

In the following example, the rule first checks if indemnity payments have been created. If not, it creates a First Party Payment alert and sends it to the supervisor.

Conditions

```
not exists( activity in Claim.Activities where activity.ActivityPattern.Code == "8day_review" )
        and exists( Exposure in Claim.Exposures where Exposure.PrimaryCoverage=="COLL")
        or exists( Exposure in Claim.Exposures where Exposure.PrimaryCoverage=="COMP")
        and gw.api.util.DateUtil.daysSince( Claim.AssignmentDate) > 7
```

Actions

```
if (not exists(payment in Claim.Transactions where payment.CostCategory== "indemnity") ) {
  Claim.createActivityFromPattern( null,
          ActivityPattern.finder.getActivityPatternByCode( "8day_review" ) )
  gw.api.util.Logger.logDebug( "@@@@@@@  A 8 day review activity has been created" )
  actions.exit()
}
```

Claim Exception: CER04000 - Recalculate claim metrics

In the base configuration, Guidewire provides claim exception rule **CER04000 - Recalculate claim metrics** that you can use to verify that ClaimCenter has created the following on every claim:

- claim metrics
- claim indicators
- exposure metrics

The effect of executing this rule is to add any missing metrics or indicators to the claim. *Guidewire recommends that you execute this rule only if all of the following conditions are true:*

- You added a new metric or indicator to the system.
- You want all existing claims to have the new metric or indicator.

Guidewire disables this rule in the base configuration. To use this rule, you must do the following:

1. Enable the **CER04000 - Recalculate claim metrics** rule in Studio.
2. Run batch process *Claim Health Calculations* (`ClaimHealthCalc`) after you enable the rule.

See Also

- "Configuring Claim Health Metrics" on page 645 in the *Configuration Guide*.
- "Batch Processes and Distributed Work Queues" on page 134 in the *System Administration Guide*.

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 the follow-up actions for each exception found. ClaimCenter identifies groups that have been changed or which have not been inspected for a certain period of time, and runs these rules on each group chosen. (For information on users and groups, see the *ClaimCenter System Administration Guide*.)

Note: ClaimCenter batch process `groupexception` runs the Group Exception rules every day at 4:00 a.m. (by default) on all groups within 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. (For information on users and groups, see the *ClaimCenter System Administration Guide*.)

Note: ClaimCenter batch process `userexception` runs the user exception rule sets on all users in within ClaimCenter every day at 3:00 a.m. by default.

Initial Reserve

ClaimCenter uses these rules 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.

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, you use the `createInitialReserve` method to automate the creation of the reserve.

Initial Reserve

ClaimCenter uses these rules determine the initial financial reserves to create for a new exposure. ClaimCenter does not associate a TransactionSet with any reserves added through this rule 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*. To accomplish this, the rule calls an enhancement method on Exposure. (You can modify this rule to automatically create a reserve of any amount for any kind of exposure.)

Condition

```
Exposure.CoverageSubType.DisplayName == "PIP Death"
```

Actions

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

Loaded

ClaimCenter runs these rules anytime that a FNOL import loads a new claim. Because Loaded rules execute only in concert with the FNOL import interface, only create rules that relate to imported claims in the Loaded category. Place rules that apply to claims that ClaimCenter create and that must execute before Segmentation in the Presetup rule set category.

Claim Loaded

Claim Loaded rules execute anytime that a 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. This happens, typically, by logging the transaction. The following rule illustrates this.

Condition

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

Actions

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

This is 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

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 a 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.

Conditions

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

Actions

```
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

Postsetup rules run automatically anytime that ClaimCenter adds a new claim. The Rule engine runs these rules after the Assignment and Workplan rules complete their execution, and immediately before the Initial Reserve or Preupdate rule sets.

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. You can design these rules to trigger based on specific conditions that the setup rules create, for example.

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 66 for more information.

Activity Postsetup

The Activity Postsetup rules fire just prior to completing the setup process and saving any changes to activities. 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.

Conditions

```
Activity.ActivityPattern.Code == "fraud_review"
```

Actions

```
Activity.Claim.SIUStatus="Under_Investigation"
```

Note: See “Detecting Claim Fraud”, on page 135 for a discussion of SIU (Special Investigation Unit) rules.

Claim Postsetup

The Claim Postsetup rules fire just prior to completing the setup process and saving any changes to claims. In the following example, the rule creates a new activity for FNOL.

Conditions

```
Claim.isValid( "newloss", true )
```

Actions

```
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

The Exposure Postsetup rules fire just prior to completing the setup process and saving any changes to exposures. In the following example, the rule first checks that this is an auto accident involving a rear-end collision

with the other party at fault. If so, the rule assigns the role of “subrogation owner” to the user assigned to the exposure. It then creates an activity for the subrogation owner to follow up on the claim for possible subrogation. In actual practice, you would need to set up a condition to prevent the rule from firing repeatedly during exposure updates. See “Taking Actions on More Than One Subitem” on page 43 for a way to handle this situation.

Conditions

```
Exposure.Claim.LossType == "AUTO" and Exposure.Claim.LossCause == "rearend" and
        Exposure.LossParty == "third_party"
```

Actions

```
var strGroup : String
var strUser : String
strGroup = Exposure.AssignedGroup.PublicID
strUser = Exposure.AssignedUser.PublicID
var grpName : Group
var y : Group

//Find the Group ID of the User assigned to the exposure
var resultset1 = find (var test in Group where test.PublicID == strGroup)
for (d in resultset1.iterator()){
  y = d as Group
  grpName = y
}

//Find the User ID of the User assigned to the exposure
var usrName : User
var u : User
var resultset2 = find (var test2 in User where test2.PublicID == strUser)
for (e in resultset2.iterator()){
  u = e as User
  usrName = u
}

var curExposure = Exposure
Exposure.Claim.createActivityFromPattern( curExposure,
        ActivityPattern.finder.getActivityPatternByCode( "subro_check" ) )
Exposure.assignUserRole( usrName, grpName,  "subrogationowner")
```

Matter Postsetup

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. This is to initiate a legal review of a bodily injury claim 10 days before a scheduled mediation meeting with the insured.

Condition

```
true
```

Actions

```
var act = Matter.Claim.newActivity( ActivityPattern.finder.getActivityPatternByCode( "legal_review" ),
        Exposure( 17 /* Bodily Injury */ ) )
act.TargetDate = gw.api.util.DateUtil.addDays( act.Matter.MediationDate, -10 )
act.assignToClaimOwner()
```

Transaction Postsetup

The Transaction Postsetup rules fire under the following conditions:

- Immediately after the Exposure Closed rule set after closing an exposure, if there is a transaction associated with that exposure.
- Immediately after the Transaction Approval Rules rule set **except** if you are in the process of creating a reserve.

These rules also fire under the following conditions:

- Anytime that ClaimCenter approves a TransactionSet
- Anytime that ClaimCenter voids, stops, or escalates a check

- Anytime that ClaimCenter records a payment
- Anytime that ClaimCenter voids a recovery
- Anytime that ClaimCenter closes an exposure

In the following example, the rule creates an activity if a reserve for a Workers' Compensation claim increases three or more times in a six month period:

Conditions

```
TransactionSet.Subtype == "reserveset"
        and (TransactionSet.Claim.LossType == "WC" )
        and TransactionSet.getAmount() > 0
```

Actions

```
var reserveChgCount = 0

for (tran in TransactionSet.Claim.Transactions) {
  if ( (tran.Subtype == "reserve" and tran.CreateTime == null and tran.TransactionSet.getAmount() > 0)
          or (tran.Subtype == "reserve" and (gw.api.util.DateUtil.daysSince( tran.CreateTime ) < 180)
          and tran.TransactionSet.getAmount() > 0) ) {
    reserveChgCount = reserveChgCount +1
  }
}

if ( reserveChgCount > 2 and not exists(act in TransactionSet.Claim.Activities
        where act.ActivityPattern.Code == "Reserve_Increase_Alert" and act.Status == "open") ) {
  TransactionSet.Claim.createActivityFromPattern( null,
          ActivityPattern.finder.getActivityPatternByCode("Reserve_Increase_Alert" ) )
}
```

Presetup

The Rule engine runs the Presetup rules automatically anytime 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 Segmentation, Assignment, and Workplan creation.) 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.

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. Thus, ClaimCenter typically performs this kind of setup under the following circumstances:

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

Set Up Operations Performed for the addFNOL SOAP Call

ClaimCenter performs the following setup tasks anytime that you add a new claim through the `addFNOL` SOAP call. ClaimCenter:

1. First adds the FNOL claim.
2. Invokes the Loaded rules on the FNOL claim.
3. Creates an "import" history event.

4. Performs the sequence of actions classified as “save and setup” (see the following discussion), setting the claim status to “open” at the end of the process.
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 Preupdate, Validation, and Event Message rules, validating the claim at the `loadsave` level.

The “save and setup” process. Guidewire ClaimCenter performs the following steps for a “save and setup” operation on a FNOL claim. ClaimCenter:

1. Creates a new claim number (if one does not exist).
2. Creates a claim snapshot.
3. Runs the Presetup rules on the claim and each exposure.
4. Runs the Segmentation rules on each exposure, then the claim.
5. Runs the Assignment rules on the claim, then each exposure.
6. Runs the Workplan rules on the claim, then each exposure. (This process generates the workplan.)
7. Runs the Postsetup rules on the claim and each exposure.
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 a FNOL claim, ClaimCenter returns the public ID of the newly created claim.

Set Up Operations Performed for Exposures

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

1. Sets the exposure's claim order.
2. Runs the Presetup rule set on the exposure.
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 the Postsetup rule set on the exposure.
8. Sets the exposure's status to “open”.
9. Create initial reserves.
10. Commits the exposures's bundle (which contains the exposure, activities, and other objects creating during the call).

Transaction Presetup

If you create financial transactions within the ClaimCenter interface, then ClaimCenter does all the bookkeeping for you. It adjusts aggregate limits, updates all T-accounts, changes all summary financial amounts, and so on. However, if you were to try to use Gosu methods in rules to directly create or modify transactions and checks,

ClaimCenter would not properly perform the necessary bookkeeping. For example, you can cause accounting errors by doing the following:

- Using `var = new reserve()` to create a new reserve
- Setting the amount of its line item through Gosu methods

This causes errors to occur as you have not updated any T-accounts in this way.

To perform bookkeeping properly within rules, you *must* use the Transaction Presetup rule set. This rule set is the **only** appropriate place for you to create, modify, or remove transactions using rules. ClaimCenter executes this rule set under the following circumstances:

- If you click the **Finish** or **Update** button at the end of the **Check** wizard.
- If you close a **New Reserves**, **New Recovery Reserves**, or **New Recoveries** screen. (This is the correct time to make all bookkeeping changes.)

Using rules in this rule set, you can:

- Add to or remove a payment from a check
- Add to or remove a line item from a transaction (Note that reserves and recovery reserves only have one line item.)
- Modify the amount of a line item

The Transaction Presetup rule provides support for modifying the amounts of already existing single and recurring checks from rules. It also provides support for adding new payment line items through the user interface. For example, after a payment is submitted, you might want to add a new payment to a check with the *Awaiting Submission* status, both from the user interface and programmatically. You might want to do this, for example:

- If you wish to adjust a future payment
- If you wish to add a line item for a certain type of tax computed in rules

Performing your business logic in the Transaction Presetup rule set causes the T-Accounts to update properly.

Guidewire provides the following methods to simplify writing rules in this rule set:

```
Transaction.addNewLineItem(Amount, Comments, LineCategory)
Check.addNewPayment(Exposure, CostType, CostCategory)
RecoveryReserveSet.addNewRecoveryReserve(Exposure, CostType, CostCategory)
ReserveSet.addNewReserve(Exposure, CostType, CostCategory)
```

Guidewire does not provide a `RecoverySet.addNewRecovery` method as a recovery set could possibly contain only one recovery.

To modify an existing transaction

To modify an existing transaction, or one you created with one of the provided methods, use the standard `getLineItems` method to retrieve the line item. Then, call any of the standard setters on the line item—for example, `lineitem.setAmount`—but again, only within this rule set so that the T-accounts will be correct.

To repeat:

- Use the Transaction Pre-set rule set *only* to modify the amount of a line item.
- Use the Transaction Pre-set rule set *only* to add or remove a payment on a check.
- Use the Transaction Pre-set rule set *only* to add or remove a Reserve or RecoveryReserve from a TransactionSet of the appropriate subtype.

To modify the level of available reserves or open recovery reserves

If you simply wish to create a a rule to modify the level of available reserves or open recovery reserves, then use the following methods:

- `setAvailableReserves`
- `setOpenRecoveryReserves`

Remember that simply creating these transaction entities directly and adding line items to them does not update T-accounts. If a rule has access to the `ReserveLine` on which the `Reserve` or `RecoveryReserve` change is desired (`Claim.ReservesLines`), then you can optionally use the following methods to change these values:

- `ReserveLine.setAvailableReserves`
- `ReserveLine.setOpenRecoveryReserves`

Check Scheduled Send Date Only Modifiable in Special Situations

The check property for the schedule send date, `ScheduledSendDate`, is only 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. In other words, which one of the following:

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

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

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

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 WC 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 for only medical costs, and the employer's WC 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.

One way to add these exposures to a Workers' Compensation claim is to create Presetup rules that automatically create the exposures, based on the severity type assigned to the claim. As soon as the user saves the claim, the Presetup rules evaluate the claim's severity, and create the appropriate exposures. The following example would be placed in the Claim Presetup rule set.

Condition

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

Action

```
claim.createWcDefaultExposures()
```

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

```
function createWcDefaultExposures() : void {

  if( this.ClaimInjuryIncident.Severity == "wc-ell" ) {
    if( !this.hasExposureOfType( this.getNewExposureType( "wl_el" ) ) ) {
      this.createAndSetupExposure("WL","wl_el")
    }
  } else {
    if( !this.hasExposureOfType( this.getNewExposureType( "wm_wcid" ) ) ) {
      this.createAndSetupExposure("WM", "wm_wcid")
    }

    if( (this.ClaimInjuryIncident.Severity != null) &&
            (this.ClaimInjuryIncident.Severity!= "contract-medical")  &&
```

```
            (this.ClaimInjuryIncident.Severity !="medical_only") &&
            !this.hasExposureOfType( this.getNewExposureType( "wi_lw" ) ) ) {
        this.createAndSetupExposure("WI","wi_lw")
      }
    }
  }
```

Preupdate

Note: See "Validation in ClaimCenter", on page 125 for a discussion of how the Validation and Preupdate rules work together in performing object validation.

Use the Preupdate rule sets to perform domain logic or validation that must be committed 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 after the Initial Reserve rules, but before the Validation rule set. In a similar fashion to the Postsetup rules, 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.

Triggering Preupdate Rules

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

- `Activity`
- `Claim`
- `Contact`
- `Exposure`
- `Group`
- `Matter`
- `Policy`
- `Region`
- `TransactionSet`
- `User`

For an entity to trigger the preupdate 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.

All entities that implement the `Validatable` delegate trigger the validation rules as well. If configuration parameter `UseOldStylePreupdate="true"`, then 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:

- 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.

IMPORTANT In running the Preupdate rule set, the Rule engine 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.

This is also the case for entities newly created within a preupdate rule. For example, if you create a new activity within a preupdate rule, the Rule engine 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 197 in the *Configuration Guide*.
- For information on configuration parameter `UseOldStylePreupdate`, see the "Environment Parameters" on page 52 in the *Configuration Guide*.
- For information on custom entities and validation rules, see "Triggering Validation Rules" on page 81.

Preupdate Rules and Custom Entities

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

To create an extension entity that triggers preupdate rules

For an entity to trigger a preupdate rule:

1. The entity must implement the `Validatable` delegate. In short, any extension entity that you create that you want to trigger a preupdate rule must implement the following code:

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

2. The preupdate and validation rule sets that you want the custom entity to trigger must conform to the following naming convention:

   - Place your preupdate rules in the *Preupdate* rule set category and name the rule set `<entity name>Preupdate`.
   - Place your validation rules in the *Validation* rule set category and name the rule set `<entity name>ValidationRules`.

   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 place 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 rules sets, see "Working with Rules" on page 25 in the *Configuration Guide*. See especially the section entitled "To create a new rule set".

ABContact Preupdate

Note: You access this rule set through Guidewire ContactCenter Studio.

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

Activity Preupdate

Use the Activity Preupdate rules to modify activities and related entities. By default, anytime that you reassign a claim, ClaimCenter reassigns the activities associated with the claim to the new claim owner. However, it is possible that this is not always the desired 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.)

Conditions

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

Actions

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

Notice the following:

- If the `activity.getOriginalValue("AssignedUserId") != null` condition evaluates to `true`, then this is an already assigned activity (meaning, it is not a new activity), so continue.
- If the `activity.Claim.isFieldChanged("AssignedUser")` condition 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

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.

Conditions:

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

Actions:

```
var dbClaim : Claim = Claim.CreateTime == null ? null : Claim( (claim as Key), "dbClaim" )

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 : String = Claim.UpdateUser.Contact.DisplayName
  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 )

}
```

Catastrophe-Related Rules

In the base configuration, Guidewire provides several rules related to catastrophe categorization in the Claim Preupdate rule set. These 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.
`CPU1300 - Catastrophe History`	Determines if there is a change to the claim **Catastrophe** field. If there is, the rule updates History and marks any open *Review for Catastrophe* activity complete.

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

See Also

- "Catastrophes" on page 115 in the *Application Guide*
- "Working with Catastrophe Bulk Associations" on page 639 in the *Configuration Guide*

Metropolitan Police Report Request Rules

The Metropolitan Reporting Bureau (`www.metroreporting.com`) provides a nationwide (United States) service for obtaining accident and incident reports. The Claim Preupdate `MetroReportRequest` rule starts an internal process to determine the type of each report requested (individually). It also 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 Metropolitan report status to validated. Once the report status becomes valid, ClaimCenter changes the Metropolitan report status to `Sending Order`.

IMPORTANT Guidewire 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 389 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 135. See also “Claim Exception Rules” on page 59.

Exposure Preupdate

Use the Exposure Preupdate rules to modify exposures and related entities. In the following example, the rule creates an activity to review the claim exposure for potential salvage value (if a review activity does not already exist).

Conditions

```
not exists(activity in exposure.claim.Activities where activity.Status == "open" and
  activity.ActivityPattern.Code == "review_salvage_potential")
```

Actions

```
if( Exposure.VehicleIncident.VehicleDriveable <> true
  and Exposure.VehicleIncident.TotalLoss ==  true ) {
  exposure.claim.createActivityFromPattern( exposure,
           ActivityPattern.finder.getActivityPatternByCode( "review_salvage_potential" ) )
}
```

Group Preupdate

Use Group Preupdate rules to modify groups and related entities. ClaimCenter runs the Group Preupdate (and Group Validation) 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. For information on users and groups, see the *ClaimCenter System Administration Guide*.

Matter Preupdate

Use the Matter Preupdate rules to modify matters and related entities. 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.

Conditions

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

Actions

```
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

Use the Policy Preupdate rules to modify policies and related entities. 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

```
true
```

Actions

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

Transaction Set Preupdate

The Rule engine runs the Transaction Set preupdate rules any time that a use adds or edits a transaction.

IMPORTANT Do **not** use the `new RecoveryReserve` constructor to create a `RecoveryReserve` transaction from a Transaction Set *Preupdate* rule. Guidewire *only* recommends creating new transactions and transaction sets using this method if you are adding a `Transaction` to the `TransactionSet` in the Transaction *Presetup* rules. The use of the Transaction Presetup rules guarantees that ClaimCenter calls the `prepareForSave` method on the `TransactionSet`, which properly sets up the T-accounts.

In the following example, the rule removes (nulls out) unwanted fields on a check if the user changes the payment method from check to EFT (Electronic Check Transfer). *Guidewire strongly recommends that you use this rule to clean up data if the payment method changes.*

Condition

```
transactionSet.Subtype == "CheckSet"
```

Actions

```
for (check in (transactionSet as CheckSet).Checks) {
  if (check.New or check.isFieldChanged("PaymentMethod")) {
    check.removeUnusedPaymentMethodRelatedFields()
  }
}
```

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	`BankAccountNumber`
	`BankName`
	`BankAccountType`
	`BankRoutingNumber`

Electronic funds transfer	`MailToAddress`
	`MailTo`
	`DeliveryMethod`

In the base configuration, the method code looks similar to the following:

```
public function removeUnusedPaymentMethodRelatedFields() {
  if (this.PaymentMethod == PaymentMethod.TC_CHECK) {
    this.BankAccountNumber = null
    this.BankName = null
    this.BankAccountType = null
    this.BankRoutingNumber = null
  } else if (this.PaymentMethod == PaymentMethod.TC_EFT) {
    this.MailToAddress = null
    this.MailTo = null
    this.DeliveryMethod = null
  }
}
```

User Preupdate

Use the User Preupdate rules to modify users and related entities. ClaimCenter runs the User Preupdate (and User Validation) rules upon a save of the User entity. Any exceptions found during validation cause the bounding database transaction to roll back, thus effectively vetoing the update. For information on users and groups, see the *ClaimCenter System Administration Guide*.

Reopened

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

Claim Reopened

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
CCCA0001 - Allow AutoSync for Related Contacts	Helps facilitate the synchronization of contact information between ClaimCenter and ContactCenter. See “Synchronizing Contacts between ContactCenter and ClaimCenter” on page 39 in the *Contact Management Guide* for more information.
CCCA0002 - Sample rule to remove purge date	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: *Claim Closed → CCCA0002 - Sample rule to set purge date* See “Archive Claim Purge Rules” on page 51 for more information.

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

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

Actions

```
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

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 111 in the *Configuration Guide*.)

Conditions

```
Exposure.ExposureType == "WCInjuryDamage"
```

Actions

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

Matter Reopened

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

The goal of the segmentation rules is to set the Segment and Strategy properties on new claims and exposures. The Rule engine runs segmentation rules just prior to running assignment rules anytime 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, the Rule engine segments the claim first by the claim line and then by the severity 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.Severity == "severe-injury" )
```

Action

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

Strategy

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

```
Exposure.Strategy = strategy_value
```

There is no reason to repeat all of 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

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".

Conditions

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

Actions

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

Exposure Segmentation

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.

Conditions

```
Exposure.ExposureType == "VehicleDamage" and Exposure.Segment == null
```

Actions

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

Transaction Approval

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, the Rule engine can ensure that the transaction be marked as requiring approval.

The Rule engine 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. The Rule engine calls these rule sets sequentially.

- The Rule engine calls the Transaction Approval rule set anytime that you submit any kind of financial transaction.
- The Rule engine calls the Approval Routing rule set 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.

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 the Rule engine runs the rules. For example, suppose that Andy Applegate initiates the reserve, and the Rule engine 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. Never, *ever*, 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. The Rule engine runs these rules for each transaction to determine whether the transaction requires approval. Note that in a typical TransactionSet approval rule, the rule condition requires a check for the subtype of the transaction set (for example, `TransactionSet.subtype == "checkset"`). This is 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 previously entered into ClaimCenter).

Condition

```
TransactionSet.Subtype=="checkset"
```

Actions

```
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

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

ClaimCenter incorporates the concept of validation levels as a way of ensuring 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. In other words, as a claim gains data, it can be viewed as achieving 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 only enforces rule errors if set at or below the level the object has achieved.

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
loadsave	*Load and Save*	400
newloss	New Loss	300
ISO	Solid for ISO	250
external	Sent to External System	200
payment	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
loadsave	400	The loosest possible validation level against which ClaimCenter validates claims. *All data must pass "loadsave" to be saved to the database.*
newloss	300	The lowest level at which a claim can exit the New Claim wizard.
payment	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
iso	250	The level at which the claim passes ISO validation.
external	200	The level at which a claim must pass before it can be submitted to an external system. Validation occurs anytime that an external submission or decline is triggered by a workflow button.

Validation warnings. The Rule engine 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. (See Correcting Validation Errors following for a discussion of the `rejectField` method.)

Condition

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

Actions

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

Validation errors. The Rule engine 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. (See Correcting Validation Errors following for a discussion of the `rejectField` method.)

Conditions

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

Actions

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

In this example, the Rule engine 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. In configuring validation levels, you can do the following:

System validation levels. You can override the *name* and the *priority* of the three system validations levels (`loadsave`, `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` prop-

erty reflects the previous validation level. However, this is not true for Event Message rules, which ClaimCenter runs after validation.

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:

- Place 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`, place validation rules for that entity in the following rule set that you create in the *Validation* rule set category:

```
MyEntitityExtValidationRules
```

See Also

- For information on preupdate rules and custom entities, see “Preupdate Rules and Custom Entities” on page 71.
- For information on how to create an entity that implements the `Validatable` delegate, see “Delegate Data Objects” on page 197 in the *Configuration Guide*.

Validation in ContactCenter

Note: See “Validation in ContactCenter”, on page 133for more information on how validation in ContactCenter differs from validation in ClaimCenter.

Validation in ContactCenter works in a similar fashion to validation in ClaimCenter, with the following difference. In the base configuration, ContactCenter uses only two levels of validation instead of the five levels used in the base ClaimCenter configuration. (It is possible, however, to increase the number of validation levels by extending the data model by modifying `ValidationLevel.xml`). See the *ClaimCenter Configuration Guide* for details.)

The following table lists the ContactCenter validation levels:

loadsave	Load and Save
external	Send to External System

Validation and Preupdate Rules

The Rules engine 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 the Rule engine 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 125.
- For information on how to create custom entities that trigger validation (and preupdate) rules, see "Preupdate Rules and Custom Entities" on page 71.

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. 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
`reject`	Indicates a problem and provides an error message, but does not point the user to a specific field.
`rejectField`	Indicates a problem with a particular field (`relatedObject`), provides an error message, and directs the user to the correct field to fix.

Method form	Description
`rejectSubField`	Similar to `rejectField`, 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
`errorLevel`	Corresponding level effected by the validation error.
`relatedObject`	Object related to the root entity that is causing the error.
`strErrorReason`	Message to show to the user indicating the reason for the error.
`strRelativeFieldPath`	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.
`strWarningReason`	Message to show to the user indicating the reason for the warning.
`warningLevel`	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
```

ABContact Validation Rules

Note: You access this rule set through Guidewire ContactCenter Studio.

The ABContact Validation rules describe complex data requirements that cannot be encoded as required fields, single-field validations, or edit masks. ABContact validation rules can ensure that contact data stored in ClaimCenter is sufficient and valid for downstream processing. Guidewire provides several contact-related validation rules in the base ClaimCenter configuration. This includes the following rules:

Rule	Throws a validation error if...	Active
Require Phone Number	The user does not enter a contact phone number.	No
Require Primary Number	There are two contact phones number and neither is marked as primary.	No
Require Past DOB	The date of birth for the contact is not in the past.	Yes

In the following example, the rule requires a user to specify which entered phone number is the primary number.

Conditions

```
ABContact.PrimaryPhone == null
```

Actions

```
var count = 0
if (ABContact.WorkPhone != null)
  count = count+1
if (ABContact.HomePhone != null)
  count = count+1
if (ABContact.FaxPhone != null)
  count = count+1
if ((ABContact typeis ABPerson) && ((ABContact as ABPerson).CellPhone != null))
  count = count+1
if (count >= 2 )
  ABContact.reject( "loadsave", "Please specify which phone number is primary", null, null )
```

Activity Validation

The Activity Validation rules describe complex data requirements that cannot be encoded as required fields, single-field validations, or edit masks.

Bulk Invoice Validation

Guidewire ClaimCenter handles bulk invoice validation through a call out to a bulk invoice plugin. See the section on BulkInvoice Integration in the *ClaimCenter Integration Guide* for more information.

Claim Closed Validation Rules

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

- They execute before a new entity is inserted into the database or an existing entity is updated.
- They do **not** execute if validation rules deem the entity invalid.

In the following example, the rule prevents claim closure if the claim has open activities associated with it.

Conditions

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

Actions

```
claim.reject( "newloss", "There are still open activities for this claim. "
  + "Please complete or skip these activities before closing the claim.", null, null)
```

Claim Reopened Validation Rules

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

- They execute before a new entity is inserted into the database or an existing entity is updated.
- They do **not** execute if validation rules deem the entity invalid.

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

Condition

```
true
```

Actions

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

Claim Validation Rules

The Claim Validation rules describe complex data requirements that cannot be encoded as required fields, single-field validations, or edit masks. Claim validation rules can 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

Conditions

```
( Claim.LossType=="WC" ) and ( Claim.Policy.PolicyType != "comp" )
```

Actions

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

Exposure Closed Validation Rules

The Exposure Closed Validation rules execute anytime that an exposure is closed. (The Exposure Validation Rules run at the end of the execution flow.) In the following example, the rule prevents closure of an exposure if the exposure has open activities.

Conditions

```
exists( Activity in Exposure.Claim.Activities
  where (Activity.Exposure.ClaimOrder == Exposure.ClaimOrder and
    Activity.Status=="open" and Activity.ActivityPattern != null and
    Activity.ActivityPattern.ClosedClaimAvlble == "false")
)
```

Actions

```
exposure.reject( "newloss", "There are still open activities related to the exposure you are trying
  to close. " + "Please complete or skip these activities before closing the exposure.", null, null )
```

Exposure Reopened Validation Rules

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

- They execute before a new entity is inserted into the database or an existing entity is updated.
- They do **not** execute if validation rules deem the entity invalid.

In the following example, the rule does not permit reopening an exposure if the associated claim is closed.

Condition

```
Exposure.Claim.State<>"open"
```

Actions

```
exposure.reject( "newloss", "You can only reopen an exposure for an open claim.", null, null )
```

Exposure Validation Rules

The Exposure Validation rules describe complex data requirements that cannot be encoded as required fields, single-field validations, or edit masks. You can use Exposure Validation rules to ensure that exposure data stored in ClaimCenter is sufficient and valid for downstream processing.

Group Validation Rules

The Group Validation rules describe complex data requirements that cannot be encoded as required fields, single-field validations, or edit masks. 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. Guidewire recommends that you use the LoadSave validation level for rules pertaining to validation for Users and Groups.

For information on users and groups, see the *ClaimCenter System Administration Guide*.

Matter Closed Validation Rules

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

- They execute before a new entity is inserted into the database or an existing entity is updated.
- They do **not** execute if validation rules deem the entity invalid.

Matter Reopened Validation Rules

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

- They execute before a new entity is inserted into the database or an existing entity is updated.
- They do **not** execute if validation rules deem the entity invalid.

Matter Validation Rules

The Matter Validation rules describe complex data requirements that cannot be encoded as required fields, single-field validations, or edit masks. Matter validation rules can ensure that matter data stored in ClaimCenter is sufficient and valid for downstream processing. The Rule engine does not enforce validation levels on Matter objects, meaning the Matter Validation rule set will always fire, regardless of the validation level.

Policy Validation Rules

The Policy Validation rules describe complex data requirements that cannot be encoded as required fields, single-field validations, or edit masks. Policy validation rules can ensure that policy data stored in ClaimCenter is sufficient and valid for downstream processing. The Rule engine does not enforce validation levels on Policy objects, meaning the Policy Validation Rule Set will 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.

Conditions

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

Actions

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

Transaction Validation Rules

Note: See “Transaction Validation”, on page 183 for a discussion of the transaction-related validation rules.

The Rule engine runs the Transaction Validation rules anytime that a TransactionSet is saved. The Rule engine does not enforce validation levels on Transaction Objects, meaning the Transaction Validation Rule Set will always fire, regardless of the validation level.

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

Condition

```
TransactionSet.claim.isClosed() and transactionset.subtype=="checkset"
```

Actions

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

User Validation Rules

The User Validation rules describe complex data requirements that cannot be encoded as required fields, single-field validations, or edit masks. ClaimCenter runs the User Validation (and User Preupdate) rules upon 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.

For information on users and groups, see the *ClaimCenter System Administration Guide*.

Workplan

A “workplan” adds initial activities to a Claim, Exposure or Matter object as a checklist of work that various people need to perform. Activities are broadly divided into the following categories:

- Activities that are performed once 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. (For more information on activity patterns, see the *ClaimCenter Configuration Guide*.)

Condition

```
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 `true` if there are other claims that are possible duplicates of the claim being checked. The `Claim.DuplicateClaimNumbers` property contains a comma-delimited list of claim numbers that might be duplicates of the current claim, or `null` if no duplicates were found.

Claim Workplan

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

```
Claim.LossType == "PR" and Claim.LossCause == "fire"
```

Action

```
Claim.createActivityFromPattern( null,
        ActivityPattern.finder.getActivityPatternByCode( “police_report” ) )
```

Exposure Workplan

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.

Conditions

```
not exists( activity in ( Exposure.claim.Activities )
where ( activity.ActivityPattern ==
        ActivityPattern.finder.getActivityPatternByCode( “10_day_review” ) ) )
```

Actions

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

Matter Workplan

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

```
true
```

Action

```
matter.claim.createActivityFromPattern( null,
        ActivityPattern.finder.getActivtyPatternByCode( "Review_Legal_Expl" ) )
```

chapter 7

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

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 application `/bin` directory.
2. At a command prompt, type:

```
gwcc regen-rulereport
```

This command generates the following XML files:

```
build/cc/rules/RuleRepositoryReport.xml
build/cc/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.

   Note: Guidewire also provides a way to enable web profiling directly from within the ClaimCenter interface. You can also use ALT+SHIFT+p 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 155 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 `NewUser -> UserDetailPage` 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**
- **Name**
- **Frame (ms)**
- **Elapsed (ms)**
- **Properties and Counters**

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

chapter 8

Assignment in ClaimCenter

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

This topic includes:

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`
- `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. For an example of how to make an entity assignable, see the "Creating an Assignable Extension Entity" on page 263 in the *Configuration Guide*.

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 ClaimCenter 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 deprecates the `assignToQueue` methods on `Assignable` in favor of the `assignActivityToQueue` method. This makes the restriction more clear. See "Queue Assignment" on page 108 for more information.

See Also

- "ClaimCenter Assignment" on page 98
- "Primary and Secondary Assignment" on page 102
- "Assignment Success or Failure" on page 104
- "Assignment Method Reference" on page 106

ClaimCenter Assignment

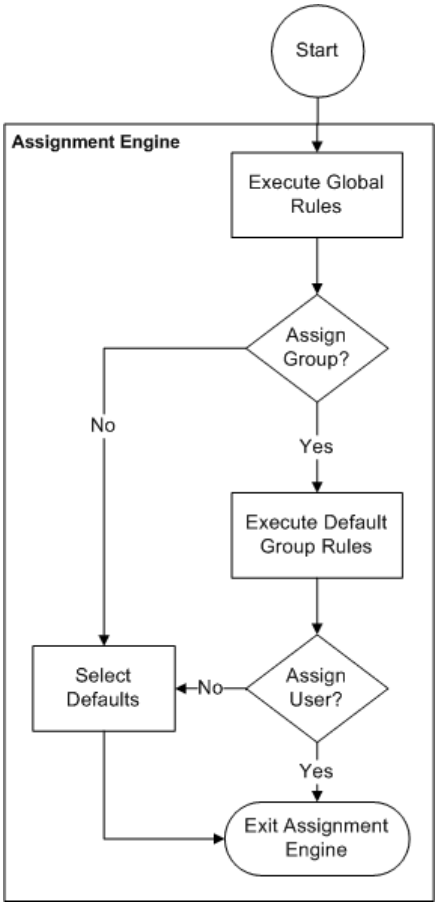
Assignment that uses the Assignment engine works within the context of the Assignment rules, which ClaimCenter groups into the following categories:

- Global Assignment Rules
- Default Group Assignment Rules

In general, ClaimCenter uses the global assignment rules to determine the group to which to assign the entity. It then the 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, the Rule Engine 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 **strongly** 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 101

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, and matters.

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. (The Rule engine runs Segmentation rules just prior to running assignment rules anytime that you select automated assignment for a new claim or exposure. See "Segmentation" on page 76 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 **strongly** 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.

Note: 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 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 strongly 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`:

```
getDefaultGroupIDFromExecutionSession
```

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 `getDefaultGroupIDFromExecutionSession` method to retrieve the current `GroupBase` value in `ExecutionSession`.

The returned `GroupBase` value from the `getDefaultGroupIDFromExecutionSession` 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 `getDefaultGroupIDFromExecutionSession`, 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.

Note: As the return value of `getDefaultGroupIDFromExecutionSession` 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 (User-based) Assignment	Also known as user-based assignment, this type of assignment assigns an object that implements the `Assignable` delegate to a particular user.
Secondary (Role-based) Assignment	Also known as role-based assignment, this type of assignment assigns an object that implements the `RoleAssignments` 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.)

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`
- `UserRoleAssignment`

Note: 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 `RoleAssignmentss` array:

- `Claim`
- `Exposure`

Note: 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 117 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	`entity.currentAssignment` 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 `CurrentRoleAssignment` property.
Secondary	`entity.currentRoleAssignment` This property returns `null` 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 for those that invoke the Assignment engine*—return a `Boolean` value that is `true` if an assignment was successfully found and `false` otherwise. Thus, you have the following cases:

- Assignment Methods that Use the Assignment Engine
- Assignment Methods that Do Not Use the Assignment Engine

Assignment Methods that Use the Assignment Engine

You can invoke the Assignment engine by calling the `autoAssign` method from outside the Assignment rules, which invokes the Assignment engine automatically.

If you invoke the Assignment engine and it does not successfully assign a user, it then assigns the assignable item to the group supervisor. If it is not possible to assign the item to the group supervisor, then the Assignment engine assigns the assignable item to the ClaimCenter default owner (`defaultowner`). (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 found, then you need to 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, then the method simply returns `true` (if it has found a valid assignment) or `false` otherwise. If `false`, then 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 the 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 (one of the `assignGroup` methods, for example). In this case, it can be necessary to call another assignment method to assign the item to an actual user.
- If you use the Assignment engine within the context of the Assignment rules, you can exit the rule (not consider any further rules), or perform other actions before exiting.

Logging Assignment Activity

Guidewire recommends also that you log the action anytime that you use one of the assignment methods. The following example—called from within the context of the Assignment rules—illustrates 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 the Assignment engine to continue to the next rule if current rule is not successful. That is, if it cannot choose a group or person to which it can assign the item. It is

extremely important that you plan an effective exit from within a rule. Otherwise, the Assignment engine can make further unanticipated and unwanted assignments. You need only use the actions.`exit` method if performing assignment from within the Assignment Rules.

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	`AssignmentAdded`	`Assignable.ASSIGNMENTADDED_EVENT`
Assigned	Assigned	`AssignmentChanged`—if a field changes, for example, the assigned user, group, or date	`Assignable.ASSIGNMENTCHANGED_EVENT`
Assigned	Unassigned	`AssignmentRemoved`	`Assignable.ASSIGNMENTREMOVED_EVENT`

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 following general categories:

- Assignment by Assignment Engine
- Group Assignment (within the Assignment Rules)
- Queue Assignment
- Immediate Assignment
- Claim-Based Assignment
- Condition-Based Assignment
- Proximity-Based Assignment
- Round-robin Assignment
- Dynamic Assignment
- Manual Assignment

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* from within the Global Assignment rule set (which is only responsible 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 which perform both assignments. These methods include the following:

- assignGroup
- assignGroupByLocation
- assignGroupByRoundRobin

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 primarygrouptype = result[0]
var secondarygrouptype = 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:

Use a finder method. You can use one of the following finder methods to return an `AssignableQueue` object:

```
AssignableQueue.finder.findVisibleQueuesForUser( User )
AssignableQueue.finder.findVisibleQueuesInUserAndAncestorGroups( User )
```

Consult the Gosu API Reference for details of how to use these finder methods.

Use 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 AssignalbeQueue 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 and 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 following methods perform “immediate” or direct assignment to the specified user or group.

- `assign`
- `assignUserAndDefaultGroup`
- `assignToIssueOwner`
- `assignToCreator`
- `assignToPreviousOwner`

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. These include the following:

- `assignToClaimOwner`
- `assignToClaimUserWithRole`

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 a Claim UserRole.
3. If not assigning an activity associated with an exposure, the method then searches for a match with a UserRole 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
- Doctor
- Independent Appraiser
- Nurse Case Manager
- Police
- Property Inspector
- Related User
- Repair Shop
- Reporter
- Salvage Owner
- SIU
- SIU Investigator
- Subrogation Owner
- Supervisor
- Vendor
- Vehicle Inspector

Note: Guidewire defines the roles that a user can have on an assignable object in typelist `UserRole`.

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 the following:

- `assignByUserAttributes`
- `assignUserByLocation`
- `assignUserByLocationAndAttributes`

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 speak 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:

GroupID	If set, restricts the search to the indicated group. This can be null.
AttributeCriteria	An array of AttributeCriteriaElement 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 performs a Boolean AND on the restrictions.

The AttributeCriteriaElement entity has a number of fields, all of which are all optional. These fields can interact in very dependant ways, depending on the value of UserField.

Field	Description
UserField	The the AttributeCriteriaElement behaves differently depending on whether the UserField property contains an actual value: • If set, then UserField must be the name of a property on the User entity. The method imposes a search restriction using the Operator and Value fields to find users based on their value for this field. • If null, 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:
AttributeField	If set, this is the name of a property on the Attribute entity. The method imposes a search restriction using the AttributeValue field to find users based on the user having the appropriate value for the named field for some attribute.
AttributeType	If set, then the method tightens the AttributeField-based restriction to Attributes only of the indicated type.
AttributeValue	If set, then the method restricts the search to users that have the specified AttributeValue only.
State	If set, then the method restricts the search to users that have an Attribute with the indicated value for State.
Value	If set, then the method restricts the search to users that have the specified Value for an Attribute 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	Works with attributes	Search algorithm
`assignUserByLocationUsingProximitySearch`	No	Searches the Group tree
`assignUserByLocationUsingProximityAndAttributes`	Multiple attributes	Searches the Group tree
`assignUserByProximityWithSearchCriteria`	Single attribute only	Uses proximity search
`assignUserByProximityWithAssignmentSearchCriteria`	Single attribute only	Uses proximity search

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 "Proximity Search and Geocoding with Gosu" 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. It 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 `AttributeCrieriaElement` 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. • 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 not-`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	• **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).
number	• **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	• **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. This parameter takes its value from UnitOfDistance typelist. • UnitOfDistance.TC_KILOMETER • UnitOfDistance.TC_MILE

Distance-based 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 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.UserSearchCriteria.Contact.Address.City = "Pasadena"
usc.UserSearchCriteria.Contact.Address.PostalCode = "94404"

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.

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. • 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 not-`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 `GeocodeScriptHelper.setupAssignmentProximitySearch` method to set up the parameters for the proximity search. It then sets up the French-speaking attribute on the `assignmentSearchCriteria` object.

```
uses gw.api.geocode.CCGeocodeScriptHelper

// Set up an distance-based proximity search for claim assignment
if (Claim.LossLocation <> null) {

  var asc = GeocodeScriptHelper.setupAssignmentProximitySearch(Claim.LossLocation, true, 10,
          UnitOfDistance.TC_MILE )

  // Set the custom attribute match.
  asc.UserSearchCriteria.AttributeName = "French"
  asc.UserSearchCriteria.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 102 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 getCandidateUsers(assignable, group, includeSubGroups)
public Set 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. It 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 provides the following manual-based assignment methods:

- `assignManually`
- `confirmManually`
- `assignManuallyByRoundRobin`
- `confirmManuallyByRoundRobin`

Guidewire recommends, before attempting to assign the assignable, that you test if it is possible to assign the assignable manually. 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 manually assign this assignable as it is not possible to create an assignment review activity for it. Otherwise, it returns the owning claim object.

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 `assignManuall`, 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 = find (u in User where exists (c in User.Credential where c.UserName == "bbaker"))
    var user = users.getAtMostOneRow()
    assignableBean.assign( user.GroupUsers[0].Group, user )
  }

  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` contain fields that you can access through Gosu to determine the total number of items of a given type assigned to an individual or group. They are:

- `OpenActivityCount`
- `OpenClaimCount`
- `OpenExposureCount`
- `OpenMatterCount`

For example, the following Gosu 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 129 in the *System Administration Guide*.

chapter 9

Validation in ClaimCenter

This topic describes rule-based validation in Guidewire ClaimCenter. It also describes the validation graph.

This topic includes:

What is Validation?

Note: The terms *object* and *entity*—while not absolutely identical—are used interchangeably throughout this section. See "Important Terminology" on page 17 for a description of each.

The Rules engine runs pre-update and validation rules every time that it commits data to the database. (This is known as a database bundle commit.) The validation rules execute after the Rule engine runs all pre-update call-backs and pre-update rules. The Rule engine runs the validation rules as the last step before ClaimCenter actually writes data to the database.

Before applying rules during the bundle commit operation, the Rule engine builds a validation object graph according to configuration settings. The graph contains possible targets for rule execution and is used by both the pre-update and validation rules.

Pre-update rules, however, are slightly different from validation rules in that it is possible for pre-update rules to modify additional objects during execution. If this is the case, there can be additional objects that need to be constructed after the pre-update rules run and before the validation rules run.

If the pre-update rules modify an entity that is not in the commit bundle...

ClaimCenter can possibly run additional validation rules. (That is, if the entity being modified also triggers validation.)

If the pre-update 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 pre-update rules ran.

If, however, the pre-update 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 which are validated can be a superset of the entities for which pre-update rules are run.

Note: The Pre-update rules are non-recursive, meaning that they do not run a second time on objects modified during execution of the Pre-update rules. ClaimCenter simply adds any objects modified by the pre-update rules to the list of objects needing validation as described by the validation graph.

Overview of ClaimCenter Validation

For an entity to have pre-update or validation rules associated with it, the entity must be validatable. This means that the entity implements the `Validatable` delegate using `<implementsEntity name="Validatable"/>` in the entity definition. In the base configuration, Guidewire ClaimCenter comes preconfigured with a number of high-level entities that trigger validation (meaning that they are validatable). These entities are (in alphabetic order):

- `Activity`
- `Claim`
- `Contact`
- `Exposure`
- `Group`
- `Matter`
- `Policy`
- `Region`
- `TransactionSet` (and all its subclasses)
- `User`

See Also

- "<implementsEntity>" on page 224 in the *Configuration Guide*

Entity Validation Order

The Rule engine 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.

Note: Guidewire does not guarantee that the Rule engine 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, the Rules engine 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.

The Rule engine 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 the Rules engine to validate the claim as well.

The Rule engine 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, the Rule engine does the following to find any claims that references a changed address. The Rule engine 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, the Rule engine applies the pre-update 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 reasonably expect the pre-update and validation rules to execute before the Rule engine 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, implementation (at the metadata level) of a user's credential is a pointer (link) from a `User` entity to a `Credential` entity.

The standard way to trigger the pre-update and validation rules on linked objects is by setting the `triggersValidation` pointer (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 the Rule engine runs the pre-update 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.)

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.

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 an foreign key to B
- Entity B has an owned array of C
- Array C has an 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 → B and C → D as triggering validation. What happens with the B → C link?

- If the B → 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 → 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

WARNING Guidewire places the base entity definition files in *`ClaimCenter`*`/modules/cc/config/metadata`. Do **not** modify these files in any way. Any attempt to modify files in this directory can cause damage to the ClaimCenter application sufficiently severe that it prevents the application from starting thereafter.

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 those entities that trigger validation in the base ClaimCenter configuration. If a table cell is empty, then there are no additional entities associated with that top-level entity that trigger validation (`triggersValidation="true"`) in the base configuration.

Validatable entity	Foreign key	Array	One-to-one
`Activity`			

Validatable entity	Foreign key	Array	One-to-one
Claim	ClaimWorkComp EmploymentData LocationCode LossLocation Policy	Activity ClaimContact ClaimText ConcurrentEmployment ContribFactors Document Evaluation Exposure Incident Matter MetroReport Negotiation Note Official OtherBenefit SIUAnswerSet UserRoleAssignment	PropertyFireDamage PropertyWaterDamage SubrogationSummary
Contact	PrimaryAddress	ContactAddresses	
Exposure	Address Benefits CompBenefits Coverage DeathBenefits DisBenefits Incident LifePensionBenefits NewEmpData PIPDeathBenefits PIPVocBenefits PPDBenefits PriorEmpData PTDBenefits RSBenefits SSDIBenefits StatLine TempLocation TPDBenefits TTDBenefits WCBenefits VocBenefits	BenefitPeriod ClaimContactRole Document ExposureText IMEPerformed MedicalAction Note OtherCoverageDetail Settlement UserRoleAssignment	
Group			
Matter		Roles StatusTypeLines	
Policy		ClaimContact ClaimContactRole ClassCode Endorsement PolicyCoverage PolicyLocation RiskUnit StatCode	
Region			
TransactionSet			
User	Contact Credential UserSettings	Attributes	

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 only override validation on certain objects associated with a base configuration 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
<array>	<array-override>	"<array>" on page 210 in the *Configuration Guide*
<foreignkey>	<foreignkey-override>	"<foreignkey>" on page 220 in the *Configuration Guide*
<onetoone>	<onetoone-override>	"<onetoone>" on page 227 in the *Configuration Guide*

See "Working with Attribute Overrides" on page 239 for details on working with these override elements.

Note: It is not necessary to override your own extension entities as you simply set `triggersValidation` to the desired value as you define the entity.

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 **strongly** 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 the Rule engine 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 the Rule engine 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 claim contact) 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 131.

Links Between Top-level Objects

As previously described, it is legal to have top-level entities trigger validation on each other. This, however, necessarily 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. (You can logically consider contacts and vehicles on a claim, for instance, 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. (The `matter` on a `ClaimContact`, for instance, 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 as triggering validation. Any attempt to do so results in an error being reported at application start up. Similarly, any array or link property defined at the application level (such as the `Driver` link on `Claim`) that is not in the database 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

> **Note:** For more information on application logging, see “Configuring Logging” on page 35 in the *System Administration Guide*.

To view a text version of the validation graph, open file `logging.properties` (from within Guidewire Studio, navigate to the **Other Resources** → **logging** folder) and add the following entry:

```
log4j.category.com.guidewire.pl.system.bundle.validation.ValidationTriggerGraphImpl=DEBUG
```

This 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.

chapter 10

Validation in ContactCenter

This topic describes the similarities and differences between ContactCenter validation and ClaimCenter validation. ContactCenter validation is similar to ClaimCenter validation in that both construct a virtual graph of validatable entities that ContactCenter must validate while performing a database commit. For a description of how validation works in general, see “Validation in ClaimCenter”, on page 125.

This topic includes:

Overview of ContactCenter Validation

For an entity to have pre-update or validation rules associated with it, the entity must be validatable. This means that the entity implements the `Validatable` delegate using `<implementsEntity name="Validatable"/>` in the entity definition. In the base configuration, Guidewire ContactCenter comes preconfigured with a number of high-level entities that trigger validation (meaning that they are validatable). These entities are (in alphabetic order):

- `ABContact`
- `Activity`
- `Contact`
- `Group`
- `Region`
- `User`

ContactCenter validates these entities in no particular order.

See Also

- “<implementsEntity>” on page 224 in the *Configuration Guide*

The Validation Graph in Guidewire ContactCenter

During database commit, the Rules engine performs validation on the following:

- Any validatable entity that is itself either updated or inserted.
- Any validatable entity that refers to a entity that has been updated, inserted, or removed.

ContactCenter gathers all the entities that reference a changed entity into a virtual graph. This graph maps all paths from each type of entity to the top-level validatable entity such as `ABContact`. These paths are queried in the database or in memory to determine which validatable entities (if any) refer to the entity that was inserted, updated, or removed.

Note: See “Validation in ClaimCenter” on page 125 for details on how the Rules engine constructs the validation graph.

Top-level ContactCenter Entities that Trigger Validation

WARNING Guidewire places the base entity definition files in *`ClaimCenter`*`/modules/ab/config/metadata`. Do **not** modify these files in any way. Any attempt to modify files in this directory can cause damage to the ClaimCenter application sufficiently severe that it prevents the application from starting thereafter.

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 those entities that trigger validation in the base ContactCenter configuration. If a table cell is empty, then there are no additional entities associated with that top-level entity that trigger validation (`triggersValidation="true"`) in the base configuration.

Validatable entity	Foreign key	Array
ABContact	PrimaryAddress	ContactAddresses SourceRelatedContacts TargetRelatedContacts
Contact	PrimaryAddress	ContactAddresses
User	Contact Credential UserSettings	Attributes

chapter 11

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

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 the Rule engine:

- 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, the Rule engine 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, the Rule engine 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, the Rule engine 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 the Rule engine to run the Claim Preupdate rules.
2. The Rule engine evaluates the SI trigger rules, 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. The Rule engine 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 the Rule engine 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 the Rule engine 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, the Rule engine 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.

SIU Action	Rule set	Rules
Set the Initial Life Cycle Stage	Claim Preupdate	Set SIU Life Cycle
Advance the Life Cycle Stage	Claim Exception Rules	Setting SIU Life Cycle State
Evaluate SIU Triggers	Claim Preupdate	SIU Life Cycle Stage 1 SIU Life Cycle Stage 2
	Exposure Preupdate	SI Triggers
Escalate Claim to Supervisor	Claim Preupdate	Create Supervisor Review Activity
	Default Group Assignment Rules (Activity)	Assign claim review activity to supervisor
Assign Activity to SIU Group	Global Assignment Rules (Activity)	Assign SIU escalation activity to SIU group
	Default Group Assignment Rules (Activity)	Assign Escalation Activity to named SIU user Default SI Escalation activity routing
Add Assigned User in SIU Role	Activity Postsetup	Add user in SIU Role to claim if necessary

Set the Initial Life Cycle Stage

The Rule engine uses claim property `SIULifeCycleState` to track a claim's stage. Claim Preupdate rule (`Set SIU Life Cycle`) sets the initial claim stage to 1 (`step1`) the first time that the Rule engine runs the pre-update rules against a claim. The Rule engine uses the value of a claim's state to determine which SIU rules to run against the claim.

Conditions

```
Claim.SimpleClaimTimer == null and Claim.State !="draft"
```

Actions

```
Claim.SIULifeCycleState="step1"
```

Advance the Life Cycle Stage

A ClaimCenter Exception rule advances the claim stage based on the number of days which have passed since claim inception. ClaimCenter automatically runs the claim exception rules on a daily basis (by default, 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 easily 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: The Rule engine evaluates claims not touched “today” on the standard Exception rule schedule.

Evaluate SIU Triggers

Each time that ClaimCenter runs the Claim Preupdate rules for a claim, the Rule engine evaluates the rules tied to the relevant stage. For example, suppose that the Rule engine pushes a claim to stage 2 during the evening’s Exception process. Then, the next time that the Rule engine evaluates the Claim Preupdate rules, it will also evaluate the SIU rules associated with stage 2.

Guidewire ships several sample stage 1 Claim Preupdate 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 Claim PreupdateSIU 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, the Rule engine runs several sample Exposure Pre-update 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 pre-update SIU rule calls an enhancement method (one each for claims and exposures) that determines whether the Rule engine 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 to meet your business needs.

Escalate Claim to Supervisor

In the base configuration, ClaimCenter ships with a special SIU script parameter called `SpecialInvestigation_CreateActivityForSupervisorThreshold` set to a value of 5. The Rule engine 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 within Guidewire ClaimCenter. However, thereafter, you modify this value through the administration interface within ClaimCenter. See "Script Parameters" on page 111 in the *Configuration Guide* for details.

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.

chapter 12

Sending Emails

This topic describes how to send email messages from Guidewire ClaimCenter.

This topic includes:

Guidewire ClaimCenter and Email

The Guidewire platform includes support for sending emails from within ClaimCenter. You can access this capability from any Gosu code. For example, you can access the email functionality from within Gosu rules or even Gosu embedded in the ClaimCenter PCF screens.

ClaimCenter provides the following email functionality:

- Support for different 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 of 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
`Subject`	Subject of the email
`Body`	Body of the email
`Sender`	EmailContact
`ReplyTo`	EmailContact (It is possible for this to be different from the Sender.)
`ToRecipients`	List of EmailContacts
`CcRecipients`	List of EmailContacts
`BccRecipients`	List of EmailContacts
`Documents`	List of DocumentBase entities to be attached to the email

gw.api.email.EmailContact

The `EmailContact` class contains three fields:

Field	Description
`Name`	Name of contact
`EmailAddress`	Email address of contact
`Contact`	Contact entity, which can be null. If this is set, it sets the `Name` and `EmailAddress` 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 within 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 (a specific claim, exposure, or activity, for example). ClaimCenter only uses this parameter during processing the email for transmission. (See “Email Transmission” on page 143.)

Emails that Use an Email Object

This variation of the `sendEmailWithBody` method requires that you create a `gw.api.email.Email` entity, 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. Within Gosu, you can obtain `Contact` objects from various places. For example, in a claim rule, to send an email *from* an insurance company employee and *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 this variation of the `sendMailWithBody` method if you do not have a full `Contact` object for a recipient or sender. (Perhaps, it was generated dynamically through some other application.). This method uses a a name and email address instead of entire `Contact` records. It does not require that you have access to a `Contact` record.

In the following method, all arguments are `String` objects:

```
gw.api.email.EmailUtil.sendEmailWithBody( Entity, toName, toEmail, fromName, fromEmail, subject, body)
```

Email Transmission

Guidewire ClaimCenter sends emails asynchronously (from the user's perspective) 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`, then ClaimCenter adds the `Message` entity to the `entity` bundle. Thus, ClaimCenter persists the `Message` entity any time that it creates the bundle.
- If the `entity` parameter is `null`, then ClaimCenter persists the `Message` entity immediately.

You must configure a `MessageTransport` class to consume the email Messages and do the actual sending. (See “Configuring ClaimCenter to Send Emails” on page 146.) ClaimCenter processes messages one at a time, and sends out the emails associated with that message.

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. This means that they are 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 291 in the *Gosu Reference Guide*
- "Data Extraction Integration" on page 279 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 143
- “Creating an Email Template” on page 144
- “Localizing an Email Template” on page 145
- “Gosu Templates” on page 291 in the *Gosu Reference Guide*
- “Data Extraction Integration” on page 279 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 517 in the *Configuration Guide*
- “Creating Localized Documents, Emails, and Notes” on page 518 in the *Integration Guide*

The *IEmailTemplateSource* Plugin

In the base configuration, Guidewire defines an `IEmailTemplateSource` plugin 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:

`locale`	Locale on which to search.
`valuesToMatch`	Values to test. You can include multiple values to match against, including: • topic • name • key words • available 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 161 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
`SMTPHost`	Name of the SMTP email application. ClaimCenter attempts to send mail messages to this server. For example, this might be `EmailHost1.acmeinsurance.com`.
`SMTPPort`	SMTP email port. Email servers normally listen for SMTP traffic on port 25 (the default), but you can change this if necessary.
`defaultSenderName`	Provides a default name for outbound email, such as `XYZ Company`.
`defaultSenderAddress`	Provides a default *From* 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 139 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:

`wkSmtpHost`	SMTP host name to use for sending mail
`wkSmtpPort`	SMTP host port number
`email`	`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:

`wkSmtpHost`	SMTP host name to use for sending mail
`wkSmtpPort`	SMTP host port number
`email`	`Email` object
`out`	`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	`UseMessageCreatorAsUser`
Value	`true`

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 143). You then need to send the email using one of the `sendEmailWithBody` methods described in “Email Utility Methods” on page 142.

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 (and recipient information, and other information) of an email as a document in the configured DMS application. Since the `sendEmailXXX` methods all take all of the sending information explicitly, this can be done simply through 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 See “Document Creation” on page 151. for details of the `DocumentProduction` class methods.

```
// First, construct and send the email
var toEmailAddress : String = "Recipient emaail address"
var toName = "Recipients 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)
```

chapter 13

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 249 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

Guidewire ClaimCenter supports several different types of document creation:

- *Synchronous* document creation completes *immediately* after it 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 `IDocumentProduction` 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 106 in the *Integration Guide*
- "Document Management" on page 249 in the *Integration Guide*

Integrating Document Functionality with ClaimCenter

Note: For information on how to integrate document-related functionality with Guidewire ClaimCenter, see the *ClaimCenter Integration Guide*. Specifically, see information on plugins in "Document Management" on page 249 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...
`IDocumentContentSource`	Storage and retrieval of document contents. This is document *contents* only, with no metadata.
`IDocumentMetadataSource`	Storage and retrieval of document metadata. This is document *metadata* 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: • 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 *atomically* so that either both exist or neither exists.
`IDocumentProduction`	Document creation. See "The Document Production Class" on page 157.
`IDocumentTemplateDescriptor`	Describe the templates used to create documents. See "The IDocumentTemplateDescriptor Interface" on page 153 for details of this interface.

Interface	Used for...
`IDocumentTemplateSource`	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: *`ClaimCenter`*`/modules/configuration/config/resources/doctemplates` If you desire to use a different path, then you need to supply the following parameters to specify an alternate location: • `templates.path` • `descriptors.path` 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. The default implementation also checks the files system for template files if a user performs a search or retrieval operation. 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.

The following table summarizes information about the minor plugin interfaces (those used less frequently).

Interface	Used for...
`IDocumentTemplateSerializer`	Customizing the reading and writing of `IDocumentTemplateDescriptor` objects. Use this sparingly. 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.
`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 106 in the *Integration Guide*
- "Document Management" on page 249 in the *Integration Guide*

The *IDocumentTemplateDescriptor* Interface

A *document template descriptor* works in conjunction with an 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)
Context Objects	Set of values that are referenced by the values to be inserted into the document template, known as *Context Objects*. This includes both default values and a set of legal alternative values for use in the document creation user interface.

Category	Contains
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
`getDateEffective` `getDateExpiration`	`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 *expired* 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.
`getDateModified` `setDateModified`	`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 `IDocumentTemplateSource` implementation.
`getDescription`	`String`	Human-readable description of the template or the document it creates.
`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 249 in the *Integration Guide* 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.

getXXX method	Return type	Returns
getKeywords	String	Set of keywords that you can use in a search for the template.
getMetadatePropertNames 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 249 in the *Integration Guide*.
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: • **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
getTemplateType	String	Value corresponding to a type code from the DocumentType typelist. If you create a document using this template, ClaimCenter sets the type 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 type rather than as templateType.

getXXX method	Return type	Returns
getDefaultSecurityType	String	Value corresponding to a type code from the DocumentSecurityType typelist. If you create a document using this template, ClaimCenter sets the securityType 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
getContextObjectNames	String[]	Set of context object names that the document template defines. See "The Document Production Class" on page 157 for an example of how template descriptor files display context objects.
getgetContextObjectType	String	Type of the specified context object. Possible values include the following: • String • Text • Bean • Name of any system entity—Claim, for example
getContextObjectAllowsNull	Boolean	True if null is a legal value, false otherwise.
getContextObjectDisplayName	String	Human-readable name for the given context object. ClaimCenter displays this name in the document creation interface.
getContextObjectDefaultValueExpression	String	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.
getContextObjectPossibleValuesExpression	String	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
getFormFieldDisplayValue	String	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 getFormFieldValueExpression). 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.
FormFieldNames	String[]	Set of form fields that the document template defines. See "The Document Production Class" on page 157 for an example of how template descriptor files show form fields.
getFormFieldValueExpression	String	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
`getLocale`	`String`	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 Document Production 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
• `asynchronousDocumentCreationSupported` • `createDocumentAsynchronously`	• `synchronousDocumentCreationSupported` • `createAndStoreDocumentSynchronously` • `createDocumentSynchronously`

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 Asynchronous Document Creation Supported 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 Synchronous Document Creation Supported 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 Create Document Asynchronously 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 Create and Store Document Synchronously 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 Create Document Synchronously 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 persists 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 291 in the *Gosu Reference Guide*
- "Document Management" on page 249 in the *Integration Guide*
- "Data Extraction Integration" on page 279 in the *Integration Guide*

Document Template Localization

For information on localizing document templates, see:

- "Localizing Templates" on page 517 in the *Configuration Guide*
- "Document Localization Support" on page 522 in the *Configuration 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 249 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 templatized fields in the destination document and the context objects (`FormField` objects).

Note: See "Document Template Descriptors" on page 258 in the *Integration Guide* for more information on template descriptor files.

```
<?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>
```

createAndStoreDocumentSynchronously - Example 1

In the following example, the `createAndStoreDocumentSynchronously` method takes following parameters to construct the previously shown document:

Parameter	Value	Description
`descriptor`	`FSAClaimAcknowledgement.gosu.htm.descriptor`	Template descriptor to use in constructing the document.
`parameters`	`contextObjects`	Relevant information related to the document contained in a `HashMap` object.

Parameter	Value	Description
document	Document	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)
```

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.
```

This means 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

Occasionally, the following error

```
"The IDocumentProduction implementation must return document contents to be called from a rule"
```

occurs while calling a `DocumentProduction.create*` method.

This error message is objecting to the `responseType` field value on the `DocumentContentsInfo` returned by calling `IDocumentProduction.createDocumentSynchronously`. The specific cause is that the `responseType` field is not set to `DocumentContentsInfo.DOCUMENT_CONTENTS`.

In other words, to successfully call document creation methods from rules, the `IDocumentProduction` implementation must return actual document contents. It must not return an HTML page, or JScript, or some other result.

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.

chapter 14

Financial Calculations

This topic discusses the pre-defined financial calculations that Guidewire provides in the default configuration of Guidewire ClaimCenter.

This topic includes:

ClaimCenter Financials

Guidewire ClaimCenter provides the ability to create calculated financial values that can be used any place that Gosu is accessible. (This is, for example, in the ClaimCenter user interface, in Gosu rules and classes, and in Gosu plugins, and similar places.) These calculated values provide various critical views of the financial state of a claim and its exposures.

You can define your own financial calculations out of a set of financial building blocks (pre-built financial calculations) that Guidewire provides. You can then use these custom financial calculations any place that you can use the standard base configuration financial calculations.

For more information on ClaimCenter financials, see the following documentation:

For information on	See	Section
Working with financials in ClaimCenter (checks, reserves, and similar financial entities)	*ClaimCenter Application Guide*	Claim Financials
Tracking and controlling financial transaction in Guidewire ClaimCenter	*ClaimCenter Integration Guide*	Financials Integration
Configuring financial parameters in ClaimCenter	*ClaimCenter Configuration Guide*	Financial Calculations
Running the `financialsescalation` batch process	*ClaimCenter System Administration Guide*	Batch Processes and Work Queues

Pre-Defined Financial Calculations

ClaimCenter provides the following pre-defined financial calculations in the base configuration:

Calculation	Description
Available Reserves	Similar to Open Reserves except that it includes pending approval and future eroding payments. It is defined as the sum of all submitted and awaiting submission reserves *minus* the sum of all approved eroding payments and all pending approval eroding payments. = (Submitted and Awaiting Submission Reserves) - (Eroding Pending Approval Payments) - (All Approved Eroding Payments) This calculation can legitimately have a negative value. (See “Financial Calculations with a Negative Value” on page 169 for an example.)
ErodingPaymentsForeignExchangeAdjustments	Total foreign exchange adjustment for *eroding payments* only.
ForeignExchangeAdjustments	Total foreign exchange adjustments for both eroding and non-eroding payments
Future Payments	Sum of all future payments (for example, those approved payments whose scheduled send date is after today). = (Open Reserves) - (Remaining Reserves)
Gross Total Incurred	Sum of Remaining Reserves *plus* Eroding Future Payments *plus* Total Payments. This calculation is the same as Net Total Incurred except that it does not account for Recoveries. = (Remaining Reserves) + (Eroding Future Payments) + (Total Payments) = (Open Reserves) + (Total Payments) = (Submitted Reserves) + (Non-eroding Total Payments)
Net Total Incurred	Sum of Remaining Reserves *plus* Eroding Future Payments *plus* Total Payments *minus* Total Recoveries. (Note that Recovery Reserves do not affect the Net Total Incurred value.) = (Open Reserves) + (Total Payments) - (Total Recoveries) = (Total Reserves) + (Total Non-eroding Payments) - (Total Recoveries)
Net Total Incurred Minus Open Recovery Reserves	Sum of Remaining Reserves *plus* Eroding Future Payments *plus* Total Payments *minus* Total Recoveries *minus* Open Recovery Reserves. = (Remaining Reserves) + (Eroding Future Payments) + (Total Payments) - (Total Recoveries) - (Open Recovery Reserves) = (Open Reserves) + (Total Payments) - (Total Recovery Reserves)
NonErodingPayments ForeignExchangeAdjustments	Total foreign exchange adjustments for *non-eroding payments* only.
Open Recovery Reserves	Sum of all submitted recovery reserves *minus* sum of all submitted recoveries on the claim. (Note that in the base configuration, recovery reserves are always automatically approved, so, in essence, it includes all recovery reserves.)

Calculation	Description
Open Reserves	Sum of all submitted and awaiting submission reserves *minus* the sum of all submitted and awaiting submission eroding payments whose scheduled send date is today or earlier. = (Submitted and AwaitingSubmission Reserves) - (Submitted Eroding Payments) - (AwaitingSubmission Eroding Payments with Scheduled_Send_Date <= today) = (Remaining Reserves) + (Future Eroding Payments)
Remaining Reserves	Similar to Open Reserves except that it includes future eroding payments. It is defined as the sum of all submitted and awaiting submission reserves *minus* all approved eroding payments. = (Submitted and Awaiting Submission Reserves) - (Eroding Approved Payments) = (Open Reserves) - (Eroding Future Payments) This calculation can legitimately have a negative value. (See “Financial Calculations with a Negative Value” on page 169 for an example.)
Total Payments	Sum of all submitted and awaiting submission payments whose scheduled send date is today or earlier.
Total Payments with Pending	Sum of all submitted, awaiting submission, future, and pending approval payments.
Total Recoveries	Sum of all submitted recoveries on the claim.
Total Recovery Reserves	Sum of all submitted recovery reserves on the claim.
Total Reserves	Sum of all submitted and awaiting submission reserves.
Total Reserves with Pending	Sum of all submitted, awaiting submission, future, and pending approval reserves.

Financial Calculations with a Negative Value

Only two of the pre-defined financial calculations can legitimately have a negative value:

- Available Reserves
- Remaining Reserves

The reason for this is that it is possible that both of these calculations subtract-out payments for which offsetting reserves have not yet have been created. The following example illustrates this concept:

Consider a claim or exposure with the following transactions:

- Submitted reserve: $500.00
- Submitted payment: $300.00
- Awaiting submission payment: $400.00, scheduled send date is one week from the current day.

Note that for this example to be possible, you need to configure ClaimCenter to allow payments to exceed reserves. That being said, the RemainingReserves in this case are -$200.00. ClaimCenter does not automatically create the necessary offsetting reserves for the payment that exceeds reserves (the $400.00 payment) until the scheduled send date of the payment check is reached. Since ClaimCenter counts these payments against remaining reserves, the remaining reserves value ends up negative.

Now, consider if an additional pending approval payment were created:

- Pending approval payment - $350

In this case, the calculated values would be:

- Remaining reserves = -$200.00
- Available reserves = -$550.00

Available reserves are even further into negative territory in this case because the pending approval payment is also subtracted.

Eroding and Non-eroding Payments

Note that:

- N*on-supplemental* payments are eroding by default, which means they draw down the reserves for their reserve line. This can be changed by either of the following methods:
  - Setting the payment as non-eroding in the payment creation step of the New Check wizard (if this field is exposed)
  - Calling the `setAsNonEroding` domain method for a payment from either the Preupdate or Postsetup rules. (Once set as non-eroding, you can reset the payment to be eroding by using the `setAsEroding` domain method.)
- S*upplemental* payments are non-eroding by definition. Any partial or final payment can also be marked as non-eroding using one the previously described methods.

Multicurrency Transactions

Note: See "Multiple Currencies" on page 171 in the *Application Guide* for information on how ClaimCenter handles multiple currencies. See also "Claim Financials Web Services" on page 200 in the *Integration Guide* for information on using methods on `IClaimFinancialsAPI` that are specific to foreign-exchange adjustments.

ClaimCenter supports multicurrency transactions. You can carry out transactions (create reserves and recovery reserves, make payments, and collect recoveries), as well as write checks, in more than one currency in a single claim. ClaimCenter supports a single default currency and multiple secondary transaction currencies.

- **Default currency**: ClaimCenter defines its default currency with `DefaultApplicationCurrency` (a configuration parameter in `config.xml`) on server startup. Currently, the terms *claim currency* and *reporting currency* also refer to this *default currency*. However, it is possible that a future ClaimCenter will change this so that these terms will no longer be synonymous. (The `ExchangeRate` entity contains a `BaseCurrency` typekey. Do not confuse this with the default currency.)
- **Transaction currencies**: ClaimCenter maintains a list of all of the available currencies in the Currencies typelist. (You access the Currencies typelist in Studio through the **Typelist** editor.)

Foreign-Exchange Adjustments

For foreign-currency payments, it is possible that the exchange rate can change during a transaction. For example, the exchange rate between the payment's currency and the claim's base currency might change. This could happen, perhaps, in between the time that the payment was created and the time its associated check finally clears the bank. For this reason, ClaimCenter provides the ability to update the converted amount of a check and its payments once the final converted amount is known.

ClaimCenter supports this update process by having the user specify the final converted amount of the check or its payments, as opposed to the final exchange rate. ClaimCenter then uses this final converted amount to calculate a "foreign-exchange adjustment" for the payments. It then updates the underlying accounts appropriately. ClaimCenter makes the adjustment for each payment at the level of the `TransactionLineItems`, making each adjustment proportional to the line items' amounts.

For example, suppose that you have a check with a single payment that has three line-items with the following transaction and claim amounts:

Line-item	Transaction Amount	Claim Amount
1	60	50
2	35	30

Line-item	Transaction Amount	Claim Amount
3	22	20
Total Amount	**117**	**100**

At some future time, the payment's check clears, and the converted “claim” amount is actually $110, rather than $100. ClaimCenter applies the foreign-exchange adjustment proportionally across all the line-items, resulting in the following:

Line-item	Transaction amount	Claim amount	Foreign Exchange Adjustment	Cleared Claim Amount
1	60	50	5	55
2	35	30	3	33
3	22	20	2	22
Total Amount	**117**	**100**	**10**	**110**

ClaimCenter provides methods for foreign-exchange adjustments to be made on either a payment-by-payment basis, or for an entire check at once. In the latter case, the adjustment is split proportionally between all the “active” payments on the check (excluding any recoded, offsetting or canceled payments). Then, for each such payment, its adjustment is split amongst its line-items as discussed in the previous example.

Applying a foreign exchange adjustment to an existing payment or check is equivalent to setting a custom exchange rate on the payment or payments.

Foreign-Exchange Transactions and Calculated Values

Guidewire ClaimCenter treats foreign-exchange adjustments as non-eroding. This means the following:

- Foreign-exchange adjustments *do not* decrease calculated values such as Open, Remaining and Available Reserves.
- Foreign-exchange adjustments *do* increase all total incurred calculations (Net and Gross), as well as overall Total Paid and Total Paid Non-Eroding.

In the previous example, assume that the only other transaction on the payment's reserve line is a reserve transaction for $150 in the base currency. When the payment is first escalated, you see the following calculated values:

- Open, Remaining and Available Reserves - $50.00
- Total Incurred (gross and net) - $150.00
- Total Paid - $100.00
- Total Paid Non-eroding - $0.00
- Total Paid Eroding - $100.00

After ClaimCenter applies the foreign exchange adjustment, the calculated values become:

- Open, Remaining and Available Reserves - $50.00 (unchanged)
- Total Incurred (gross and net) - $160.00 (+$10.00)
- Total Paid - $110.00 (+$10.00)
- Total Paid Non-eroding - $10.00 (+$10.00)
- Total Paid Eroding - $100.00 (unchanged)

To work with foreign-exchange calculations, Guidewire provides three expressions accessible from `gw.api.financials.FinancialsCalculationUtil`:

```
getForeignExchangeAdjustmentsExpression
getErodingPaymentsForeignExchangeAdjustmentsExpression
getNonErodingPaymentsForeignExchangeAdjustmentsExpression
```

See “Using Custom Calculation Building Blocks” on page 178 for more information on using expressions in custom calculations.

Foreign-Exchange Adjustments on Claims and Payments

Note: Foreign exchange adjustments only affect total incurred and total paid calculations. They do not further erode reserves.

Guidewire ClaimCenter provides the following methods to apply a foreign-exchange adjustment to a check or a payment within the context of the ClaimCenter business rules:

- `Check.applyForeignExchangeAdjustment`
- `Payment.applyForeignExchangeAdjustment`

Check.applyForeignExchangeAdjustment

```
Check.applyForeignExchangeAdjustment( newClaimAmount : BigDecimal ) : void
```

This method applies a foreign-exchange adjustment to this Check's claim currency amount. The parameter `newClaimAmount` specifies the new amount for this check in the claim currency. It cannot be `null`.

The amount of the adjustment is the percentage difference between the new passed-in amount and the current amount for claim currency. Suppose, for example, that the original check has three payments of $50, $20 and $10 for a total claim amount of $80, and the new claim amount is $100. Then, every payment will get a 25% offset, making them $62.50, $25 and $12.50, for a total of $100.

Only use this method on a check that is in a committed but uncanceled state, meaning that the check status must be one of the following:

- Requesting
- Requested
- Issued
- Cleared

Payment.applyForeignExchangeAdjustment

```
Payment.applyForeignExchangeAdjustment( newClaimAmount : BigDecimal ) : void
```

This method applies a foreign exchange adjustment to this Payment's claim currency amount. The parameter `newClaimAmount` specifies the new amount for this payment in the claim currency. It cannot be `null`.

The amount of the adjustment is the percentage difference between the new passed-in amount and the current amount for claim currency. Suppose, for example, that the original payment has three line items of $50, $20 and $10 for a total claim amount of $80, and the new claim amount is $100. Then, every line-item will get a 25% offset, making them $62.50, $25 and $12.50, for a total of $100.

Only use this method on a payment that is an offsetting payment. The payment must also be in a committed but uncanceled state, meaning that the payment status must be one of the following:

- Submitting
- Submitted

Applying Foreign-Exchange Adjustments Multiple Times

You can apply a foreign-exchange adjustment to a payment or check multiple times. However, each application will undo and replace the prior one. To illustrate this using the previous example, assume that you make a second

call to the `applyForeignExchangeAdjustment` method on the payment. This time you pass in a value of $120.00. You would then end up with the following three line-items:

- $60.00 ($10.00 adjustment)
- $36.00 ($6.00 adjustment)
- $24.00 ($4.00 adjustment).

The ClaimCenter database maintains a history of both of these adjustments in order to maintain an audit trail. However, it only applies the last adjustment to calculated values.

TransactionLineItem Fields

Guidewire also provides the following fields on the `TransactionLineItem` entity for use with foreign-exchange adjustments:

- `ClaimForExAmount`
- `ReportingForExAmount`

as well as the following fields on the `Transaction` entity:

- `ClaimForExAdjustmentAmount`
- `ReportingForExAdjustmentAmount`

Using 'CheckCreator'

Guidewire provides methods on `gw.api.financials.CheckCreator` to use as you create new `Check` objects in Gosu business rules. This class provides a builder-like interface to assist you in creating `Check` entities and their subobjects.

Use one of the following properties to create a `CheckCreator` object:

- `Claim.CheckCreator`
- `Exposure.CheckCreator`

Use the builder-like methods on `CheckCreator` to modify a new `Check` object before you submit the check for approval and before ClaimCenter updates any affected TAccount objects. For example, you can add additional payments or line items to the check before you submit the check.

The `CheckCreator` methods start typically with `with` and provide a way for you to specify parameters such as the cost type or the mail-to-address of the check. For example:

```
CheckCreator.withPayee(payee)
            .withPayeeRole(payeeRole)
            .withRecipient(recipient)
            .withMailToAddress(mailToAddress)
            .withReportabilityType(reportabilityType)
            .withCostType(costType)
            ...
```

The following example illustrates how to create a new `CheckCreator` object and pass in various parameters. You need, of course, to define or to retrieve values for each of the supplied parameters.

```
var claimCheck = claim.CheckCreator
claimCheck.withPayee(payee)
          .withPayeeRole(payeeRole)
          .withRecipient(recipient)
          .withMailToAddress(mailToAddress)
          .withReportabilityType(reportabilityType)
          .withCostType(costType)
          .withCostCategory(costCategory)
          .withPaymentType(paymentType)
          .withLineCategory(lineCategory)
          .withCheckAmount(checkAmount)
          .withComments(comments)
          .withMemo(memo)
          .withPayTo(payTo)
          .withScheduledSendDate(scheduledSendDate)
          .withRequestingUser(requestingUser)
```

In actual practice, you need supply only those parameters that meet your business needs. However, at a minimum, supply valid values for the following `Check` parameters. You cannot leave these values `null`.

- `Payee`
- `PayeeRole`
- `CostType`
- `CostCategory`
- `PaymentType`
- `CheckAmount`
- `ScheduledSendDate`

In a similar manner, you can create a check for an exposure. The following examples illustrate how to do this.

```
exposure.CheckCreator.withPayee(payee)
                     .withPayeeRole(payeeRole)
                     .withRecipient(recipient)
                     .withMailToAddress(mailToAddress)
                     .withReportabilityType(reportabilityType)
                     .withCostType(costType)
                     .withCostCategory(costCategory)
                     ...
```

After you create a `CheckCreator` object, you then use one of the following methods on `CheckCreator` to create the check:

Check creation methods	Description
createCheck	Creates a check defined by the current state of `CheckCreator`. It is possible to modify the check and any subentities of the check after you create the check. The `createCheck` method call returns a new `Check` object. It also stores the new `Check` object inside the `CheckCreator` for the subsequent call to `prepareForCommit`. You must call `CheckCreator.prepareForCommit` to submit the check for approval and to update the appropriate TAccount objects.
prepareForCommit	Prepares the previously created check for commit to the database. Call this method after calling `CheckCreator.createCheck` and after performing any modifications to the check, such as adding additional Payments or Transaction Line Items. This methods submits the check for approval and updates the appropriate TAccount objects. It is not possible to modify the check after you call this method, except for extension fields.
createAndPrepareForCommit	Use to prepare and commit the check in one operation. This method calls the following methods in the listed order: • `createCheck` • `prepareForCommit`

For example, on a specific exposure:

```
var cc = exposure.CheckCreator.withPayee(payee1)
                              .withPayeeRole("checkpayee")
                              .withCostType("claimcost")
                              .withCostCategory("autoglass")
                              .withPaymentType("final")
                              .withCheckAmount(100.00)
                              .withScheduledSendDate("2012-01-31" as java.util.Date)

var thisCheck = cc.createCheck()
thisCheck.BankAccountNumber = "123456789"
cc.prepareForCommit()
```

Note: As always, consult the GosuDoc for the latest information on Gosu types and methods.

Using the Pre-defined Financials Calculations

Guidewire makes the pre-defined financial calculations available in the Gosu Financial Calculations library, which you can access in Gosu through the `gw.api.financials.FinancialCalculationsUtil` Gosu library. This library contains multiple `getFinancialCalculation` methods, each of which return a

FinancialCalculation object that encapsulates the associated financial calculation. In the base configuration, Guidewire ships the following methods (which correspond to the calculations defined in “Pre-Defined Financial Calculations” on page 168):

- getAvailableReserves
- getOpenRecoveryReserves
- getOpenReserves
- getRemainingReserves
- getTotalIncurredGross
- getTotalIncurredNet
- getTotalIncurredNetMinusOpenRecoveryReserves
- getTotalPayments
- getTotalPaymentsIncludingPending
- getTotalRecoveries
- getTotalReserves
- getTotalReservesIncludingPending

As described previously, each of these methods returns an instance of FinancialCalculation that exposes methods of two types:

- getAmount
- getTransactionIds

getAmount

The getAmount method contains multiple overloaded versions. All return a monetary value for the calculation on which they are called, limited by the passed-in parameters. The following examples illustrate these methods. Note that all examples assume a code prefix of gw.api.financials.FinancialsCalculationUtil.

getTotalReserves().getAmount(claim)	Returns the Total Reserves amount for the passed-in Claim.
getTotalReserves().getAmount(exposure)	Returns the Total Reserves amount only for the passed-in Exposure. For example, it returns only those reserve transactions coded to this exposure.
getOpenReserves().getAmount(exposure, CostType)	Returns the amount of Open Reserves only for the passed-in Exposure and CostType values. It returns only those reserves and payments coded to this exposure and cost type, regardless of cost category coding.
getAmount(claim: Claim, exposure: Exposure, costType: CostType, costCategory: CostCategory)	Accepts null values for everything except Claim, and it matches those values explicitly. So, if this method is called with a null value for “exposure”, it only matchs claim-level transactions.

Note: Review the ClaimCenter GosuDoc on these method for more details on how each works.

getTransactionIds

The getTransactionIds method is overloaded in many of the same ways as the getAmount method. However, rather than returning the monetary amount of the calculation, it returns the IDs of the transactions that contribute to this calculated value. So, for example, a call to following

```
gw.api.FinancialCalculationsUtil.getOpenReserves().getTransactionIds
```

would return the IDs of all transaction objects that contribute to the open reserves calculated values (for example, reserves and payments).

The parameter list determines which open reserves to include. For example, if you pass in only the claim, then the method returns the IDs of all the transactions that contribute to the OpenReserves for the entire claim. If you

use the version that takes an Exposure only, then it returns the IDs of all the transactions that contribute to Open-Reserves for that exposure *only*. (It does not return the IDs of transactions for other exposures on the same claim.) Other versions of this method allow you to account only for certain CostTypes and/or CostCategories.

Working with Reserves

You read (retrieve) the values for Available Reserves and Open Recovery Reserves using the following methods in the `gw.api.financials.FinancialCalculationsUtil` package:

- `getAvailableReserves`
- `getOpenRecoveryReserves`

However, to set these values, you must call the following methods directly on a `Claim` or `Exposure` entity. For example:

- `Claim.setAvailableReserves`
- `Claim.setOpenRecoveryReserves`
- `Exposure.setAvailableReserves`
- `Exposure.setOpenRecoveryReserves`

Alternately, you can call the `setAvailableReserves` method on a reserve line directly. For details, see the “Methods on Reserve Line Objects” on page 177.

Setting Reserve Values

Guidewire provides the following methods to create reserves on `Claim` and `Exposure` objects:

- `setAvailableReserves`
- `setOpenRecoveryReserves`

IMPORTANT The `setAvailableReserves` and `setOpenRecoveryReserves` on `Claim` and `Exposure` objects do not support the creation of multicurrency Reserve and Recovery Reserve transactions with a non-Claim currency. See “Creating Reserve Transactions Directly” on page 178 for information on creating reserve transactions that support non-Claim currency.

setAvailableReserves

```
Claim.setAvailableReserves( costType, costCategory, amount, submittingUser )
```

This method sets the available reserves for this claim or exposure to the given amount by creating a reserve that increases or decreases the currently available reserves.

Available Reserves are similar to Open Reserves except that they include pending approval and future eroding payments. Thus, they are the sum of all submitted and awaiting submission reserves minus the sum of all approved eroding payments and all pending approval eroding payments. For more information, see the definition of Available Reserves at “Pre-Defined Financial Calculations” on page 168.

The method takes the following parameters:

Parameter	Description
`costType`	The cost type for the reserve. This value cannot be null, but it can be unspecified.
`costCategory`	The cost category for the reserve. This value cannot be null, but it can be unspecified.
`amount`	The amount to which to set the available reserves. The amount must be zero or greater, and cannot be negative.
`submittingUser`	The user submitting this reserve.

Executing the `setAvailableReserves` method from Gosu code provides significantly different behavior than performing the equivalent action from within ClaimCenter. If you execute this method within Gosu code, the method:

- Creates a separate `ReserveSet` object.
- Ignores any existing reserve set objects for the giving `ReserveLine` that have a status of *pending approval*.
- Returns the new Reserve, or `null` if ClaimCenter was unable to create the new reserve.

setOpenRecoveryReserves

```
Exposure.setOpenRecoveryReserves( costType, costCategory, recoveryCategory, newRecoveryReserveAmount,
  submittingUser )
```

This method sets the open recovery reserves for this claim or exposure to the given amount by creating a recovery reserve that increases or decreases the current open recovery reserves. It takes the following parameters:

Parameter	Description
`costType`	The cost type for the reserve. This value cannot be null, but it can be unspecified.
`costCategory`	The cost category for the reserve. This value cannot be null, but it can be unspecified.
`recoveryCategory`	The recovery category for the recovery reserve. This value can be null.
`newRecoveryReserveAmount`	The amount to which to set the open recovery reserves. The amount must be non-null and zero or greater, and cannot be negative.
`submittingUser`	The user submitting this recovery reserve.

The method returns new Recovery Reserve, or `null` if ClaimCenter was unable to create the new Recovery Reserve

Methods on Reserve Line Objects

ClaimCenter also provides methods for use with the `ReserveLine` objects that mirror the methods described for use with `Reserve` objects. For example:

```
TransactionSet.Transactions[0].ReserveLine.setAvailableReserves( "1000", User( "1" /* A. Brown*/ ) )
```

This method sets the available reserves for this reserve line to the given amount by creating a reserve that increases or decreases the currently available reserves.

Available Reserves are similar to Open Reserves except that they include pending approval and future eroding payments. Thus, they are the sum of all submitted and awaiting submission reserves minus the sum of all approved eroding payments and all pending approval eroding payments. For more information, see the definition of Available Reserves at “Pre-Defined Financial Calculations” on page 168.

The method takes the following parameters:

Parameter	Description
`newReserveAmount`	The amount to which to set the new reserve. The amount must be non-null and zero or greater, and cannot be negative.
`submittingUser`	The user submitting this recovery reserve.

There is a similar method to set open recovery reserves on a `ReserveLine` object: `setOpenRecoveryReserves`.

Creating Reserve Transactions Directly

ClaimCenter also provides a more flexible API (than those listed in the topic on Working with Reserves) that you can use to create reserve transactions directly. These methods include the following:

- `Claim.newReserveSet()`
- `ReserveSet.newReserve( exposure, costType, costCategory )`
- `Reserve.addNewLineItem( amount, comments, lineCategory )`
- `ReserveSet.prepareForCommit()`

For example:

```
var reserveSet = claim.newReserveSet()
var reserve = reserveSet.newReserve(exposure, costType, costCategory)

// Modify the new Reserve transaction however you want - for example:
reserve.Currency = Currency.TC_EUR

// Set the amount to 100 EUR.
// A Reserve transaction must have only one TransactionLineItem, and its LineCategory must be null.
reserve.addNewLineItem( CurrencyAmount.getStrict( 100, Currency.TC_EUR) , null, null)

// Add more reserve transactions if required

reserveSet.prepareForCommit()
```

The `newReserve` method returns the new transaction entity so that you can modify it. At minimum, you must call the `addNewLineItem` method to add a `TransactionLineItem` with the `Amount` of the transaction. You can also create a multicurrency `Reserve` transaction by modifying the `Currency` field on the transaction.

After you finish creating transactions, call the `ReserveSet.prepareForCommit` method to submit the reserve set for approval and to update the financials calculations. After calling the `prepareForCommit` method, it is not possible to modify the transaction amount, currency, exchange rates, or other key fields on the transaction.

Using Custom Calculation Building Blocks

To support custom financial calculations, ClaimCenter provides a set of financial building blocks from which you can construct these calculations. Each is an instance of `FinancialsExpression` and you access each using a *getter* method on `gw.api.financials.FinancialCalculationsUtil`. Most of the building blocks are similar to the predefined calculations, except for the fact that they can be combined with each other. The following list describes the Guidewire-supplied custom calculations building blocks. Note that all are prefixed with `gw.api.financials.FinancialCalculationsUtil`.

gw.api.financials.FinancialCalculationUtil.get*	Description
getAvailableReservesExpression	Same as Available Reserves for the pre-defined calculations.
getErodingPayments ForeignExchangeAdjustmentsExpression	Same as ErodingPaymentsForeignExchangeAdjustments for the pre-defined calculations.
getForeignExchangeAdjustmentsExpression	Same as ForeignExchangeAdjustments for the pre-defined calculations.
getFuturePaymentsExpression	Same as Future Payments for the pre-defined calculations.
getGrossTotalIncurredExpression	Same as Gross Total Incurred for the pre-defined calculations.
getNonErodingPayments ForeignExchangeAdjustmentsExpression	Same as NonErodingPayments ForeignExchangeAdjustments for the pre-defined calculations.
getOpenRecoveryReservesExpression	Same as Open Recovery Reserves for the pre-defined calculations.
getOpenReservesExpression	Same as Open Reserves for the pre-defined calculations.
getPendingApprovalErodingPaymentsExpression	Sum of all pending approval payments that are eroding.
getPendingApprovalNonErodingPaymentsExpression	Sum of all pending approval payments that are non-eroding.

gw.api.financials.FinancialCalculationUtil.get*	Description
getPendingApprovalPaymentsExpression	Sum of all pending approval payments.
getPendingApprovalReservesExpression	Sum of all pending approval reserves.
getRemainingReservesExpression	Same as Remaining Reserves for the pre-defined calculations.
getTotalIncurredNetRecoveriesExpression	Same as Net Total Incurred for the pre-defined calculations.
getTotalIncurredNetRecoveryReservesExpression	Same as Net Total Incurred Minus Open Recovery Reserves for the pre-defined calculations.
getTotalPaymentsExpression	Same as Total Payments for the pre-defined calculations.
getTotalRecoveriesExpression	Same as Total Recoveries for the pre-defined calculations.
getTotalRecoveryReservesExpression	Same as Total Recovery Reserves for the pre-defined calculations.
getTotalReservesExpression	Same as Total Reserves for the pre-defined calculations.

You manipulate these building blocks using two basic operations: *plus* and *minus*. Use these operations to combine the `FinancialsExpression` objects to produce custom expressions from which you can obtain the desired value. For example:

```
FinancialsExpression.plus(FinancialsExpression)
FinancialsExpression.minus(FinancialsExpression)
```

Note that you combine `FinancialsExpression` objects, not `FinancialCalculation` objects. Therefore, the following is valid:

```
//Valid expression using FinancialExpression objects
gw.api.financials.FinancialsCalculationUtil.getTotalIncurredNetRecoveriesExpression().
  minus( gw.api.financials.FinancialsCalculationUtil.getOpenRecoveryReservesExpression() )
```

but not the following:

```
//Invalid expression using FinancialsCalculation objects
gw.api.financials.FinancialsCalculationUtil.getTotalIncurredNetRecoveries().
  minus( gw.api.financials.FinancialsCalculationUtil.getOpenRecoveryReserves() )
```

Forming Composite Custom Expressions

You can combine the supplied `FinancialsExpression` instances to form composite custom expressions. For example, if ClaimCenter did not supply a `TotalIncurredNetRecoveryReserves` expression, you could create such an expression in either of the following ways:

```
//Example 1
gw.api.financials.FinancialsCalculationUtil.getTotalIncurredNetRecoveriesExpression().
  minus(gw.api.financials.FinancialsCalculationUtil.getOpenRecoveryReservesExpression() )

//Example 2
gw.api.financials.FinancialsCalculationUtil.getGrossTotalIncurredExpression().
  minus(gw.api.financials.FinancialsCalculationUtil.getTotalRecoveryReservesExpression() )
```

In either case, the method returns a new `FinancialsExpression` instance that is equivalent to the `TotalIncurredNetRecoveryReserves` expression.

However, to be useful, you need to obtain a calculated value for the financial expression. To do this, you must generate an instance of a `FinancialCalculation` that is backed by this custom expression. This is done with a specialized method on `gw.api.financials.FinancialsCalculation.Util`:

```
getFinancialsCalculation(FinancialsExpression)
```

Thus, to generate a `FinancialCalculation` for the example `FinancialExpression`, you would use:

```
var TotalIncurredNetRecoveryReserves = gw.api.financials.FinancialsCalculationUtil.
  getFinancialsCalculation( gw.api.financials.FinancialsCalculationUtil.
  getTotalIncurredNetRecoveryReservesExpression().
  minus( gw.api.financials.FinancialsCalculationUtil.getOpenRecoveryReservesExpression() ) )
```

Creating Custom Financial Gosu Classes

Clearly, having to repeat this long string each time that you want to make use of a custom calculation is very tedious. It is also error prone and inefficient from a performance point of view. Instead, Guidewire recommends that you create a custom Gosu utility class that contains all your custom financial calculations, which insures that they are all defined in one location. You can then reference these calculations each place that you reference Gosu code (rules, Gosu classes and enhancements, and PCF files).

Guidewire provides an example of how to do this in Studio, under the `Classes` folder: `util.financials.CustomCalcs`:

```
package util.financials

class CustomCalcs {
  //Clas constructor
  function CustomCalcs() {   }

  private static var lib application = gw.api.financials.FinancialsCalculationUtil
  private static var calcMyTotalIncurredNet application =
    lib.getFinancialsCalculation( lib.getGrossTotalIncurredExpression().
      minus( lib.getTotalRecoveryReservesExpression() ) )

  public static property get MyTotalIncurredNet() : FinancialsCalculation {
    return calcMyTotalIncurredNet
  }
}
```

This Gosu class defines the sample custom calculation as a static property of the class with “application” scope. This makes it simple to reference the calculation from any place in the application. If you create a custom Gosu class that defines all the desired custom calculations as static properties, then it is a simple matter to use those custom calculations. You can use them, for example, in rules, PCF files, Gosu plugins, or any place else that ClaimCenter supports Gosu.

You would reference the previous example class thusly:

```
util.financials.CustomCalcs.MyTotalIncurredNet.getAmount(...)
```

The Guidewire-provided sample rules show an example usage of the `CustomCalcs` class in the Transaction Approval rule set. Since `MyTotalIncurredNet` is merely an instance of a `FinancialCalculationt` object, the `getAmount` method works as described previously for the pre-defined calculations.

Condition

```
TransactionSet.Subtype == "CheckSet"
```

Actions

```
var totalIncurredAmt = util.financials.CustomCalcs.MyTotalIncurredNet.getAmount(TransactionSet.Claim)
  if (totalIncurredAmt > 20000) {
    TransactionSet.requireApproval( "Total Incurred on the claim exceeds $20,000" )
}
```

Transaction Status

> **Note:** See “Configuring Financials” on page 599 in the *Configuration Guide* for more information. This provides a discussion on financial transaction statuses and how to use them to track and control the flow of a transaction or check in ClaimCenter.

Guidewire ClaimCenter represents the current lifecycle state of a financial transaction or check using the `Status` property (a `TypeKey` value of type `TransactionStatus`) on the `Transactions` object. You can access this property from both the ClaimCenter interface and from within Gosu rules. Use it to indicate the current business state of a transaction, as well as act on the business state of the transaction. The following are all valid values for the `Transaction.Status` property.

- Awaiting Submission
- Pending Approval
- Recoded
- Submitted (transactions only)

- Cleared
- Draft
- Issued
- Notifying
- Pending Recode
- Pending Stop
- Pending Transfer
- Pending Void
- Rejected
- Requesting (checks only)
- Requested (checks only)
- Stopped
- Submitting (transactions only)
- Transferred
- Void

Submitted Transactions

Submitted transactions are a subset of Approved transactions. A transaction is considered to be **Submitted** if its `Status` is any of the following:

- Pending
- Pending Stop
- Pending Transfer
- Pending Void
- Recoded
- Stopped
- Submitted
- Submitting
- Transferred
- Voided

Awaiting Submission Transactions

A transaction is considered to be **Awaiting Submission** if its `Status` is `AwaitingSubmission`. Only payments and reserves can have this status:

- *AwaitingSubmission payments* are those payments whose containing Check has not yet been escalated and has a scheduled send date of the current day or earlier.
- *AwaitingSubmission reserves* are those reserves that are tied to an AwaitingSubmission payment in order to act as an offsetting reserve.

Payments with a `Status` of `AwaitingSubmission` can also be future-dated. *FutureDated payments* are those approved payments whose containing Check has a scheduled send date after the current day.

Pending Approval Transactions

A transaction is considered to be **Pending Approval** if its `Status` is `PendingApproval`.

Approved Transactions

A transaction is considered to be **Approved** if its `Status` is any of the following:

- AwaitingSubmission
- Pending recode
- Pending stop
- Pending transfer
- Pending void
- Recoded
- Stopped
- Submitted
- Submitting
- Transferred
- Voided

chapter 15

Transaction Validation

This topic discusses the Transaction Validation rules and how to work with them.

This topic includes:

Overview of Transaction Validation

Insurance companies typically sell vehicle liability insurance as a package, with the insurer's liability limited per person, per occurrence (incident), and per 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 to either reject (block) the transaction from going through or to allow the transaction to continue, but with warnings.

Relevant Transaction Validation Rules

The following table lists the sample Transaction Validation rules that relate to transaction limits.

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
TXV02000	(New reserve amount + Current Total Payments + Remaining Reserves) > Exposure Limit	Warn, but permit

Rule	Condition	Action
TXV03000	**1.** (New payment + Current Total Payments already made for all exposures tied to the same coverage) > Incident Limit	**1.** Error, block
	2. Above condition false *and* (New payment + Current Total Payments for all exposures tied to the same coverage + Remaining Reserves) > Incident Limit	**2.** Warn, but permit
TXV04000	(New reserve amount + Current Total Payments for all exposures tied to the same coverage + Remaining Reserves) > Incident Limit	Warn, but permit
TXV05000	Payments on PIP coverages exceed one of the four defined PIP Aggregate Limits	Warn, but permit

Consider the following example, which illustrates the use of sample rule TXV01000 (Total payments not greater than 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 wish to raise an alert anytime that a payment causes the payment total to exceed a reserve or exposure limit. In some cases, you might wish to block the transaction altogether. The sample rules in the Transaction Validation Rules handle either of these situations easily.

Note: Guidewire provides the Transaction 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).

The Transaction Validation Rules

Note: For information on the validation rules in general, see “Validation” on page 79. For specific information on how ClaimCenter handles validation in rules, see “Validation in ClaimCenter”, on page 125.

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. These rules are all in the Transaction Validation Rules rule set:

- TXV01000 (Total payments not greater than Exposure limit)
- TXV02000 (Check Exposure limit when increasing reserves)
- TXV03000 (Total payments not greater than Incident limit)
- TXV04000 (Check Incident limit when increasing reserves)
- TXV05000 (PIP)

These sample rules execute correctly only if both of the following are true:

- The created exposure is tied to the coverage (in the Exposure page).
- The corresponding coverages are defined at the vehicle level on the Policy’s Vehicles page.

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.

Note that these sample rules calculate payments values using “claim cost”, which only includes costs directly associated with the claim, and not ancillary costs such as expenses. The sample rules also calculate all values using the financial calculations `RemainingReserves` value, using the following “getter” 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 following:

```
var AvailableReservescache = gw.api.financials.FinancialsCalculationUtil.getAvailableReserves()
```

Note: For more information on working with financial calculations with ClaimCenter, see “Financial Calculations”, on page 167.

TXV01000 (Total payments not greater than Exposure limit)

The Rule engine executes this rule anytime that you attempt to create a new check.

```
transactionset.Subtype == “CheckSet” && transactionset.new && transactionset.ApprovalStatus ==
  "unapproved"
```

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 the Overview of Transaction Validation for an example of this.)
- 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. The following example illustrates this.

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)

The Rule engine 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
```

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)

The Rule engine executes this rule anytime that you attempt to create a new check.

```
transactionset.Subtype == “CheckSet” && transactionset.new && transactionset.ApprovalStatus ==
  "unapproved"
```

This sample rule works in a similar manner to that of TXV01000 (Total payments not greater than Exposure limit) except that it applies to incident limits, not exposure limits. 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)

The Rule engine executes this rule anytime that you attempt to create a new reserve.

```
transactionset.Subtype =="ReserveSet" && transactionset.new
```

This sample rule works in a similar manner to that of TXV02000 (Check Exposure limit when increasing reserves) except that it applies to incident limits, not exposure limits. 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

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"
```

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 rules could use the following code to determine if the per person aggregate limit has been reached for this exposure:

```
var PersonAggLimitReached = exposure.PIPPersonAggLimitReached
```

The 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	• PIP Medical • PIP Rehabilitation • PIP Lost Wages • PIP Funeral • PIP Death • PIP Extraordinary Medical
sumForPIPNonmedAgg	• PIP Rehabiliation • PIP Lost Wages • PIP Funeral • PIP Death

Aggregate method	Sums these coverage types
sumForPIPPersonAgg	• PIP Medical • PIP Rehabiliation • PIP Lost Wages • PIP Funeral • PIP Death • PIP Extraordinary Medical
sumForPIPReplaceAgg	PIP Lost Wages

The TXV05000 (PIP) 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)
```

chapter 16

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?

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 sections provides examples of how to use queues in Guidewire ClaimCenter. These include:

- Using a Queue to Handle FNOL Claims: Example
- Using a Queue to Void/Stop Checks: Example

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 a 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:

- The first rule assigns the incoming FNOL claim to an intake group that will make the assignment decision on that claim.
- The second rule then assigns the claim to the group's supervisor. This "parks" the claim until the final assignment determination.
- The third rule creates the FNOL review activity.
- The next rule assigns the FNOL review activity to the intake group designated in the first assignment rule.
- 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.

The following table summarizes these tasks.

Task	Rule set	Rule
1. Assign Claim to an Intake Group	Global Claim Assignment	ASGCDefault
2. Assign Claim to the Group Supervisor	Default Group Claim Assignment	ASDGCDefault
3. Create FNOL Review Activity	Claim Workplan	WPWC03
4. Assign FNOL Review Activity to Intake Group	Global Activity Assignment	ASGA30
5. Assign FNOL Review Activity to Queue	Default Group Activity Assignment	ASDGA01

The following sections describe these tasks in more detail.

Assign Claim to an Intake Group

The (Global Claim Assignment) `ASGCDefault - Assign to Claim Intake Units` rule 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. This means that this rule fires for every claim that comes 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"  )
        acions.exit()
      }
}
```

Assign Claim to the Group Supervisor

Next, the (Default Group Claim Assignment) `ASDGCDefault - Assign to Claim Group Supervisor` rule assigns (as the name suggests) the claim to the supervisor of the group. Again, this will happen for every incoming claim.

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) `WPWC03 - FNOL Review` rule creates the review activity to be queued. (This activity tells the user to review and assign the claim.) And again, this happens for every claim.

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) `ASGA30 - FNOL Review Assignments` rule 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, the Rule engine 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) `ASDGA01 - Assign New Claim Activity to Queue` rule finally adds the review activity to the queue. This rule condition also specifies the correct activity type. Note that 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.AssignableQueues[0].Name )

activity.CurrentAssignment.assignActivityToQueue( activity.AssignedGroup.AssignableQueues[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.

In this example, there are only two rules:

- The first rule assigns an activity (of type `void_or_stop_check`) to a particular group (the Accountants group).
- The second rule adds the activity to the void/stop queue.

As these rules merely assign an activity that has been manually created by an end-user, there is no need for claim assignment or workplan rules as in the previous example. 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.

The following table summarizes these tasks.

Task	Rule set	Rule
1. Create the Stop/Void Queue	Not applicable	Not applicable
2. Assign the Activity to the Appropriate Group	Global Activity Assignment	ASGA40
3. Add the Activity to the Queue	Default Group Activity Assignment	ASDGA02

The following sections describe these tasks in more detail.

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) `ASGA40 - Void/Stop Payments` rule 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) `ASDGA02 - Assign Void/Stop Activity to Queue` rule adds the activity to the queue. This rule condition also requires that the activity be of the correct activity type.

Note that the rule refers to `AssignableQueues[1]` as this is (presumably) the first new queue added to this group. If you have multiple queues on a group, you will need to choose the correct one. (As shown in the previous example, `AssignableQueues[0]` refers to the default FNOL queue that ClaimCenter creates anytime that you create a new group.)

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.AssignableQueues[1].Name + "Queue" )

activity.CurrentAssignment.assignActivityToQueue( activity.AssignedGroup.AssignableQueues[1] )
actions.exit()
```