Guidewire PolicyCenter®

PolicyCenter Rules Guide

RELEASE 8.0.3

Guidewire®
Deliver insurance your way™

Product Name: Guidewire PolicyCenter
Product Release: 8.0.3
Document Name: *PolicyCenter Rules Guide*
Document Revision: 18-November-2014

Contents

About PolicyCenter Documentation

The following table lists the documents in PolicyCenter documentation.

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

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

Conventions in This Document

Text style	Meaning	Examples
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. In code blocks, bold formatting highlights relevant sections to notice or to configure.	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 Portal – `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 PolicyCenter 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 PolicyCenter. 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 PolicyCenter rules work. It can also help you make decisions about changing your rules as your use of PolicyCenter 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 PolicyCenter. You typically organize these rules sets into a hierarchy that fits your business model. Guidewire strongly recommends that you implement a rule-naming scheme as you create rules and organize these rules into a hierarchy. This can be similar to that described in "Generating Rule Debugging Information" on page 27.

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 PolicyCenter 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 PolicyCenter. However, make each specific to the rule set to which it belongs.

Rule Execution

The hierarchy of rules in PolicyCenter mirrors a decision tree that you might diagram on paper. PolicyCenter 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.) PolicyCenter moves through the hierarchy according to the following algorithm. Recursively navigating the rule tree, it processes the parent and then its children, before continuing to the next peer of the parent.

- Start with the first rule
- Evaluate the rule's conditions. If true…
  - Perform the rule's actions
  - Start evaluating the rule's children, starting with the first one in the list.

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

To illustrate how to use a rule hierarchy, take the business logic used to determine the type and number of forms that a certain policy requires as an example. You would probably expect to describe a series of questions to figure out in which forms are mandatory and required by each policy. For convenience, you would want to describe these questions hierarchically. For example, if your answer to 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 a Workers' Compensation policy**, go through a list of more detailed rules for this policy line …
  - If this is the state of Washington, [done]
  - Otherwise…If this is the state of California and business is not construction-related …
    - If payroll more than $100,000[done]
    - Else, if payroll less than $1,000,000 [done]
    - Else, if payroll more than $1,000,000 [done]
  - Otherwise, consider construction-related (high hazard) rules …
    - If the number of employees less than 50 done]
    - Else, if the number of employees less than 150 [done]
    - Else, if the number of employees greater than 150 [done]
- **If this is a BOP policy**, then go through a list of more detailed rules for this policy line …
- **If all else fails [Default]**, use the default rule action [done]

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

See also

- "Generating a Rule Execution Report" on page 48

Rule Management

Guidewire strongly recommends that you tightly control access to changing rules within your organization. Editing rules can be complicated. Because PolicyCenter 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 PolicyCenter.
- **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 PolicyCenter.

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 PolicyCenter Rule Sets" on page 14
- "Generating a Rule Repository Report" on page 47
- "Generating a Rule Execution Report" on page 48

Important Terminology

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

Term	Definition
entity	An entity is a type of business data configured in the data model configuration files, for example `Policy`. You can use the Data Dictionary to review the entity types and their properties. For an entity type, this documentation refers to an instance as an *entity instance* or, for brevity, *object*.
Guidewire Studio	Guidewire Studio is the Guidewire administration tool for managing PolicyCenter resources, such as PCF pages, business rules, and Gosu classes.
library	A library is a collection of functions (methods) that you can call from within your Gosu programs. Guidewire provides a number of standard library functions (in the `gw.api.*` packages).
object	An object refers to any of the following: • an instance of an *entity type*, such as `Policy` or `Activity`. For an entity type, an object is also called an *entity instance*. • an instance of an *Gosu class* • an instance of an *Java class, such as* `java.util.ArrayList`.

rule	A rule is a single decision in the following form:
	`If {some conditions}`
	`Then {take some action}`
	For example, a validation rule might mean: *If the auto policy does not list a Vehicle Identification Number (VIN) for all covered automobiles, then mark the policy as invalid.* This individual rule is just one of many validation tests that you can conduct on a new policy, for example.

rule set	A rule set combines many individual rules into a useful set to consider as a group.
job	A job encapsulates a specific set of tasks that can be performed on a `PolicyPeriod` object.
policy	A Policy encapsulates all the information about a risk underwritten by a carrier. You can also think of a policy as a container of logical policy periods (with a specific range of effective time) for a policy.

Term	Definition
policy period	A logical period of time for which a policy is in effect. For example, a typical year-long personal auto policy is one logical policy period. If you modify that policy several times, PolicyCenter keeps each version, but represents all of them with the same logical policy period. PolicyCenter represents each *logical policy period* by a single identifier called a fixed ID. PolicyCenter represents each *version of the logical period* by a `PolicyPeriod` entity instance, which stores its period ID in a field called `PeriodID`. Note that policy period as a concept and the `PolicyPeriod` entity type are not exactly the same.
workflow	A workflow is a Guidewire mechanism to run custom business processes asynchronously, optionally with multiple states that transition over time. Studio stores individual workflows as XML files. However, you manage workflow directly through Guidewire Studio user interface. In the base configuration, PolicyCenter provides workflow for policy Cancellation and Renewal jobs only. These two jobs usually occur asynchronously over time.

Overview of PolicyCenter Rule Sets

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

Rule set category	Contains rules to
Assignment	Manage the policy archive process. See “Assignment” on page 34.
Assignment	Determine the responsible party for an activity, for example. See “Assignment” on page 34.
Audit	Generate activities related to audits. See “Audit” on page 34.
Event Message	Handle communication with integrated external applications. See the *PolicyCenter Integration Guide* for information about events and event messaging. See “Event Message” on page 35.
Exception	Specify an action to take if an Activity or Job is overdue and is in the escalated state. See “Exception” on page 38.
Renewal	Govern behavior at decision points in the Renewal job flow, for example, accepting or rejecting a renewal request. See “Renewal” on page 41.
Validation	Check for missing information or invalid data on non-policy-related (platform) objects. See “Validation” on page 42.

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 PolicyCenter.

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	PolicyCenter
All	If New Submission	Require the following fields: • Primary Insured Name • Primary Insured Phone Number • Primary Insured Address • Policy Effective Date • Producer Agency • Producer Code	Complete the required fields	New Submission Wizard

Rule Structure

The basic structure of every rule has the following syntax:

`IF` {some conditions}

`THEN` {do some actions}

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

Pane	Requirements

Rule Conditions	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 a policy renewal) AND (no one has reviewed the policy in the last three years
	It is also possible to insert a statement list, instead of a simple expression. However, the statement list must contain a `return` statement. For example:
	`uses java.util.HashSet`
	`uses gw.lang.reflect.IType`
	
	`var o = new HashSet<IType>() {A, B, C, ...}`
	`return o.contains(typeof(...))`

Rule Actions	A list of actions to take. For example: • Mark the renewal as flagged • Add an activity for the underwriter 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, PolicyCenter gives you access only to conditions and actions that are appropriate for the type of your current rule.

Rule Syntax

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

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

- Assignment statements
- `If` statements
- Iteration statements

All expressions follow the dot notation to reference fields and subobjects. For example, to retrieve the latest `PolicyPeriod` (of possibly several branches), do the following:

```
var period = Job.LatestPeriod
```

An expression can consist of one or many statements.

IMPORTANT Use only Gosu expressions and statements to create PolicyCenter 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 (policies, for example).

- **Rule Conditions** – A collection of expressions that provide true/false analysis of an entity. If the condition evaluates to `true`, then Studio runs the activities in the **Rule Actions** pane.
- **Rule Actions** – A collection of action expressions that perform business activities such as making an assignment or marking a field as invalid. These actions occur only if the corresponding condition is `true`.
- **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 method in testing whether a vehicle driver is 18 years of age or older:

  ```
  var mustBeBornOnOrBefore = gw.api.util.DateUtil.addYears( today , -18 )
  ```
- **Entities** – The collection of business object entities supported by PolicyCenter. Studio objects use the familiar "dot" notation to reference fields and objects.

  For example, you can use the following object conditions to automatically raise evaluation issues for the specified location.

  ```
  PolicyPeriod.autoRaiseLocationEvalIssues(location)
  ```

Defining the Rule Conditions

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

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

This example demonstrates some important basics:

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

Combining Rule Conditions

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

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

(See the "Gosu Operators and Expressions" on page 65 in the *Gosu Reference Guide* for details on operator precedence and other syntactical information.)

IMPORTANT The rule condition statement must evaluate to either Boolean `true` or `false`. If you create a condition statement that evaluates to `null`, PolicyCenter 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 PolicyCenter never executes the associated rule actions.

Defining a Statement List as Rule Conditions

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

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

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

Defining the Rule Actions

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

Exiting a Rule

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

Flow control action	Description
`actions.exit`	This is the simplest version. PolicyCenter stops processing the rule set as soon as it encounters this action.
`actions.exitAfter`	This action causes PolicyCenter to stop processing the rule set after processing any child rules.
`actions.exitToNext`	This action causes PolicyCenter 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 PolicyCenter 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 PolicyCenter 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, suppose that you set up an *Add Forms* rule to determine the issuing state for a policy. Then, you decide on the forms to attach using the results of this determination. You can structure the rule set as follows:

- Always process my child rules to evaluate the policy line…
  - If (conditions) Then (Add forms) [Done but go to next top-level rule]
  - Otherwise…
- Always process my child rules to evaluate the policy state…
  - If (conditions) Then (Add forms) [Done; no need to go further]
  - Otherwise…

After making the first decision (setting forms for that policy line), PolicyCenter 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 if further forms are required by the state). After setting the segment, the rule set is finally complete, so it can just use the simple exit method.

Gosu Annotations and PolicyCenter Business Rules

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

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

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

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

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

Studio displays a list of validation annotations.

See also

- For information on working with annotation, including creating your own annotations, see "Annotating a Class, Method, Type, Class Variable, or Argument" on page 221 in the *Gosu Reference Guide*.

Invoking a Gosu Rule from Gosu Code

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

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

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

See also

- "Using the Rules Editor" on page 21
- "Working with Rules" on page 21

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

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

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

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

To view or edit a rule

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

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

To change rule order

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

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

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

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

PolicyCenter 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 specific to a business function with Guidewire PolicyCenter.
- Rules contain the Gosu code (condition and action) that perform the actual business logic.

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

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

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

To create a new rule set

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

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

4. Enter the following information:

Field	Description
Name	Studio displays the rule name in the middle pane. For the Guidewire recommendations on rule set names, see “Generating Rule Debugging Information” on page 27. In general, however, if you create a rule set for a custom entity named `Entity_Ext`, you must name your rule set `Entity_Ext<RuleSet>`. Thus, if you want the custom entity to invoke the Preupdate rules, then name your rule set `Entity_ExtPreupdate`. There are some variations in how to name a rule set. See the existing rule sets in that category to determine the exact string to append and follow that same pattern with new rule sets in that category.
Description	Studio displays the description in a tab at the right of the Studio if you select the rule set name in the middle pane. Guidewire recommends that you make the description meaningful, especially if you have multiple people working on rule development. In any case, a meaningful rule description is particularly useful as time passes and memories fade.
Entity Type	PolicyCenter uses the entity type as the basis on which to trigger the rules in this rule set. For example, suppose that you select a rule set, then a rule within the set. Right-click and select **Complete Code** from the menu. Studio displays the entity type around which you base the rule actions and conditions.

To create a new rule

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

To access a rule set from Gosu code

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

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

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

Renaming or Deleting a Rule

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

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

Renaming a Rule

At a structural level, Guidewire PolicyCenter stores each rule as a separate Gosu class, with a `.gr` extension. The name of the Gosu class corresponds to the name of the rule that you see in the Studio **Project** window. PolicyCenter stores the rule definition classes in the following location:

```
PolicyCenter/modules/configuration/config/rules/...
```

If you rename a rule set, PolicyCenter 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, PolicyCenter ensures that the rule class names and the file names are always synchronized.

Changing the Root Entity of a Rule

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

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

   ```
   var test =
   ```

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

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

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

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

Why Change a Root Entity?

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

Do not use this command in other circumstances.

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

1. You create a new entity in PolicyCenter, 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, however, does not remove references to the entity that currently exist in the rules files.
6. You update the database by stopping and restarting the application server.
7. You stop and restart Studio.

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

To change the root entity of a rule

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

After you supply a name:

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

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

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

Making a Rule Active or Inactive

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

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

chapter 4

Writing Rules: Testing and Debugging

This topic describes ways to test and debug your rules.

This topic includes:

Generating Rule Debugging Information

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

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

Printing Debugging Information

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

```
print("This is an 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().DisplayName + " Rule Set")
```

Logging Debugging Information

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

- "Logging" on page 613 in the *Integration Guide*

- “Configuring Logging” on page 23 in the *System Administration Guide*

Using Custom Logging Methods

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

- See the for information about creating and using Gosu class methods.
- See “Script Parameters” on page 100 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 Actions:

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

CustomLogging Class

```
package util

class CustomLogging {

  construct() { }

  public function logRule( policy:PolicyPeriod, 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### POLICYCENTER RULE EXECUTION LOGGER (Policy #: "
              + policy.PolicyNumber + ") ###" + "\n " + ruleInfo + message + "\n " + "Timestamp: "
              + gw.api.util.StringUtil.formatTime(gw.api.util.DateUtil.currentDate(), "full") + "\n" )
    }
    return true
  }
}
```

Result

```
### POLICYCENTER RULE EXECUTION LOGGER (Policy #: 25-708090) ###
    Rule Set: PreQualification
    Rule Name: Raise Pre-Qual Issues
    Message: Custom text...
    TimeStamp: 2:27:36 PM PST
```

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 PolicyCenter 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 primary contact for an account:

```
Account.Accountcontacts.Contact.LastName.compareTo("Smith")  //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:

```
(Account.Contacts.PrimaryContact[0] as Person).LastName.compareTo( "Smith" )
```

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 policy has any unfinished activities that are overdue, you need to look at all activities on the policy. Clearly, you need something more complicated than `Policy.*` to express this condition. PolicyCenter provides an `exists(...in...where...)` syntax to describe these tests.

For example:

```
exists ( Activity in Account.AllJobs where Activity.Status == "open"
  and Activity.AssignmentDate < gw.api.util.DateUtil.currentDate() )
```

This condition evaluates to `true` if there are one or more activities connected to an account that meet the conditions described. It is also common to use the `exists` method to look for a certain kind of endorsement within a policy. For example, “does this Workers’ Compensation policy have the correct forms for California?”

See “Existence Testing Expressions” on page 72 in the *Gosu Reference Guide* for more information on Gosu `exists(...in...where)` expressions.

Taking Actions on More Than One Subitem

PolicyCenter provides a `for(... in ...)` syntax and an `if(...) { then do something }` syntax that you can use to construct fairly complicated actions. In the following example, the rule iterates through the various lines on a policy to determine if a line contains a certain Coverage Group.

```
for (eachline in p.Lines) {
  for (eachcov in eachline.AllCoverages) {
    if (eachcov.CoveragePatternGroup == "BAPHiredGrp") {
      Print( "PolicyLine \"" + PolicyLine + "\" has BAPHiredGroup coverage" )
    }
  }
}
```

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 line on the policy.
- The inner set brackets the commands to do “if” the specified coverage group is found.

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

```
PolicyPeriod.Lines.Length
```

See “Gosu Conditional Execution and Looping” on page 94 in the *Gosu Reference Guide* for more information on the `for(... in ...)` Gosu syntax.

Checking Permissions

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

- PolicyCenter exposes static permissions that are non-object-based (like the permission to create a user) as Boolean properties.
- PolicyCenter exposes permissions that take an object (like the permission to edit a claim) as methods that take an entity as their single parameter.
- PolicyCenter 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. PolicyCenter always evaluates permissions relative to the current user unless specifically instructed to do otherwise. You can use permissions anywhere that you can use Gosu (in PCF files, rules, and classes) and there is a current user.

For example:

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

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

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

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

chapter 6

Rule Set Categories

This topic describes the sample rules that Guidewire provides as part of the PolicyCenter base configuration.

This topic includes:

Rule Set Summaries

PolicyCenter Studio contains the following sample rule sets:

Rule set category	Contains rules to	See
Assignment	Determine the responsible party for an activity, for example.	"Assignment" on page 34
Audit	Generate activities related to audits.	"Audit" on page 34
Event Message	Handle communication with integrated external applications. See the *PolicyCenter Integration Guide* for additional information.	"Event Message" on page 35
Exception	Specify an action to take if an Activity or Job is overdue and enters the escalated state.	"Exception" on page 38
Renewal	Govern behavior at decision points in the Renewal job flow, for example, accepting or rejecting a renewal request.	"Renewal" on page 41
Validation	Check for missing information or invalid data on non-policy-related (platform) objects.	"Validation" on page 42

Assignment

Guidewire refers to certain business entities as *assignable* entities. For assignable entities, it is possible:

- To determine the party responsible for that entity.
- To assign that entity through the use of Gosu assignment methods. For a description of these assignment methods, see “Assignment in PolicyCenter” on page 53.

The Assignment Engine

Each of the assignable entities in the base configuration has a rule set called the *<Entity> Assignment Rules*. You can run these rules by invoking the Assignment engine. The Assignment engine then runs the rules until either the assignment completes or it runs out of rules to run.

The Assignment engine process consists of two phases, one of which uses global assignment rules, the other of which uses default assignment rules.

Global Assignment Rules

First, the engine invokes the “Global” assignment rules for the entity type in question. (There is a rule set for each assignable entity type) The Assignment engine runs the global rules a single time, with a three legal outcomes possible:

1. One of the rules assigns both a group and a user. In this case, the assignment process is done and the Assignment engine exits.
2. One of the rules assigns a group. In this case the assignment process update the execution session with the assigned group and continue with the “default” assignment rules for that entity.
3. None of the rules perform any assignment. In this case the Assignment engine assigns the entity to a default user and group and then exits.

Default Assignment Rules

There is again one rule set for each assignable entity type. After the rules have been invoked, there are again three possibilities:

1. A rule assigns the user. In this case, the assignment process is complete and the Assignment engine exits.
2. No rule assigns a user, but a rule assigns a different group. In this case, the engine updates the execution session with the assigned group and runs the Default Assignment rules for that entity again.
3. No rule assigns a user, nor does a rule change the assigned group. In this case, the assignment fails and the assignment engine exits.

Audit

PolicyCenter supports two subtypes of audits:

- *Final audit*, which covers the entire policy term. A final audit begins on the policy effective date and ends on the policy expiration or cancellation date.
- *Premium reporting*, which are a series of non-overlapping periodic audits that you schedule and bill within the coverage period.

See “Premium Audit Policy Transaction” on page 143 in the *Application Guide* for information related to audits in PolicyCenter.

Reporting Trend Analysis

The Reporting Trend Analysis rule set checks to see if the reporting trend analysis ratio is within an acceptable range. In the default configuration, the acceptable range is from 90% to 110%. If the ratio is outside the acceptable range and the number of reporting days is greater than 60, the code creates a `RatioOutOfRange` activity and assigns it to an underwriter.

See also

- For information about reporting trend analysis, see "Premium Report Trend Analysis" on page 147 in the *Application Guide*.

Event Message

IMPORTANT PolicyCenter 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

PolicyCenter 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:

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

Messaging Events

PolicyCenter 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. PolicyCenter 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 PolicyCenter" on page 309 in the *Integration Guide*.
- Custom events, see "Triggering Custom Event Names" on page 316 in the *Integration Guide*.

Message Destinations and Message Events

You use the Studio Messaging editor to define message destinations and message events.

- 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 PolicyCenter that is of possible interest to an external system. For example, this can be adding, changing, or removing a Guidewire entity.

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

- `ContactAdded`
- `ContactChanged`
- `ContactRemoved`

If you modify, add, or remove a `Contact` object (among others), PolicyCenter generates the relevant event. Then, during a bundle commit of the `Contact` object, PolicyCenter 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.

IMPORTANT After the PolicyCenter application server starts, PolicyCenter initializes all message destinations. PolicyCenter saves a list of events for which each destination requested notifications. As this happens at system start-up, you must restart the PolicyCenter 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 an external submission batch completes successfully.

```
var batch : SubmissionGroup = messageContext.Root as SubmissionGroup
var subs : Submission[] = batch.Submissions

var payload : String
payload = "External Submission Batch "
payload = payload + "(" + batch.Submissions.length + " Submissions):"

for (sub in subs) {
  payload = payload + " " + sub.JobNumber
}

MessageContext.createMessage(payload)
```

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 and Events" on page 289 in the *Integration Guide*
- "List of Messaging Events in PolicyCenter" on page 309 in the *Integration Guide*
- "Using the Messaging Editor" on page 131 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, PolicyCenter 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 PolicyCenter 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 constructs a message containing the account number any time that the application adds an account.

Rule Conditions

```
// Only fire if adding account
messageContext.EventName == "AccountAdded"
```

Rule Actions

```
// Create handle to root object.
var account = messageContext.Root as Account
```

```
// Print a message to the application log
print("Account [" + account.AccountNumber + "] added")

// Create the message
messageContext.createMessage("Account [" + account.AccountNumber + "] added")
```

Exception

The Exception rule set category encompasses both exception rules and escalation rules.

- PolicyCenter runs *escalation* rules on entities that have been active past a certain date.
- PolicyCenter runs *exception* rules on all instances of an entity with the intent of finding and handling instances with some unusual condition. (However, the query can prioritize some kinds of instances over others.) One common use of the exception rules is to find policies that look unusual for some reason and generate activities directing someone to deal with them.

Both rule types involve batch processes invoking rule sets on entities. PolicyCenter runs these rule sets at regularly scheduled intervals. See “Batch Processing” on page 99 in the *System Administration Guide* for information on exception batch processes.

Activity Escalation Rules

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.

PolicyCenter runs scheduled process `activityesc` every 30 minutes (by default). This process runs against all open activities within PolicyCenter 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.

PolicyCenter marks each activity that meets the criteria as escalated. After the batch process runs, PolicyCenter runs the escalation rules for all the activities that have hit their escalation date.

This rule set is empty by default. However, you can use this rule set to specify what actions to take any time that the activity enters the escalated condition. For example, you can use this rule set to reassign escalated activities. In the following example, the rule re-assigns the activity to the supervisor of the underwriting group if the escalated activity priority is set to urgent.

Condition

```
Activity.Priority == “urgent“
```

Action

```
var currentGroup = Activity.Job.getAssignmentByRole( "Underwriter" ).AssignedGroup
Activity.assign( currentGroup, currentGroup.Supervisor )
```

Group Exception Rules

PolicyCenter runs the Group Exception rules on a scheduled basis. It looks for certain conditions on groups that might require further attention and to define the follow-up actions for each exception found. PolicyCenter 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. This rule set is empty by default.

PolicyCenter batch process `groupexception` runs the Group Exception rules every day at 4:00 a.m. (by default) on all groups within PolicyCenter.

Policy Exception Rules

PolicyCenter runs the Policy Exception rules on a scheduled basis. It looks for certain conditions on `PolicyPeriod` entities that might require further attention and to define the follow-up actions for each exception found. PolicyCenter identifies `PolicyPeriod` entities that have been changed or which have not been inspected for a certain period of time and runs these rules on each `PolicyPeriod` chosen.

This rule set is empty by default. Implementers are free to create rules in this rule set category as necessary. However, it is important to understand:

- PolicyCenter persists any change made to the `PolicyPeriod` or its related objects through this rule set to the database.
- PolicyCenter runs the rules in this rule set category—one at a time—over a potentially large number of `PolicyPeriod` entities. This can have an adverse impact on performance.

Validation

The Policy Exception rules perform no implicit validation. Any validation that you wish to perform must be executed explicitly by calling the appropriate validation method. Guidewire strongly recommends that you take care to validate *only* as needed, for performance reasons. For instance, if the only change you are making to a `PolicyPeriod` is to raise an alert in the form of a Note, Activity, or email, do not validate.

PolicyPeriod Entities

PolicyCenter runs the Policy Exception rules over every `PolicyPeriod` entity in the database *except* for bound periods that are no longer the most recent model (`PolicyPeriod.MostRecentModel == false`). Thus, you can run exception rules on withdrawn, declined, and bound revisions.

PolicyPeriod and the Data Model

To store the time at which the exception rules were last run, PolicyCenter maintain a separate `PolicyException` entity with two fields of interest:

- A non-nullable foreign key to `PolicyPeriod`
- A datetime field “ExCheckTime” indicating the time at which the exception rules last ran

A `PolicyPeriod` never has more than one associated `PolicyException` entity. If `PolicyPeriod` entity has no associated `PolicyException` entity, it means that the exception rules have not yet been run on that particular `PolicyPeriod` entity.

Updating ExCheckTime

Any time a batch process (described in “Scheduling” on page 40) decides to run exception rules on a given `PolicyPeriod` entity, the process does the following:

1. It runs the exception rules on the `PolicyPeriod`.
2. It finds the `PolicyPeriod` entity’s associated `PolicyException` entity, or it creates one if one is not found.
3. It sets `PolicyException.ExCheckTime` to the current system clock time.
4. It saves (commits) the changes to the database.

Scheduling

Three separate batch processes control the time at which Rule engine runs the exception rules:

`openpolicyexception`	Runs on all `PolicyPeriod` entities that are in an *open* status—Draft, Quoted, and Binding, for example
`boundpolicyexception`	Runs on all `PolicyPeriod` entities in a *bound* status—except those which are not the latest in model time
`closedpolicyexception`	Runs on all `PolicyPeriod` entities in a *closed* status—Withdrawn, Declined, and NotTaken, for example

All three processes invoke the same rule set. Therefore, if you want the exception behavior to vary based on the `PolicyPeriod` status, you need to test that status within the rules.

You can configure the batch execution time and how each batch process runs in `scheduler-config.xml`. To disable a process, comment it out in this file.

In the default configuration:

- `openpolicyexception` runs every night at 2 a.m.
- `boundpolicyexception` runs on the first Saturday of each month at 12:00 noon
- `closedpolicyexception` runs on the first Sunday of each month at 12:00 noon

You can use configuration parameters (in `config.xml`) to schedule these batch processes. You can also limit how often PolicyCenter runs the exception rules against a single `PolicyPeriod` entity. Use the following parameters:

- `OpenPolicyThresholdDays` for `openpolicyexception`
- `BoundPolicyThresholdDays` for `boundpolicyexception`
- `ClosedPolicyThresholdDays` for `closedpolicyexception`

These parameters limit how many days must pass before the exception rules run a second time on any given `PolicyPeriod` entity. For instance, suppose that a given `PolicyPeriod` has a withdrawn status and `ClosedPolicyThresholdDays` is set to 30. After PolicyCenter runs the exception rules on that `PolicyPeriod`, the `closedpolicyexception` rules does not run rules on that particular policy again until at least 30 days have passed. Thus, these configuration parameters are useful as throttles for performance.

The default values for the three configuration parameters are:

- 1 day for `OpenPolicyThresholdDays`
- 14 days for `ClosedPolicyThresholdDays`
- 60 days for `BoundPolicyThresholdDays`

The following table summarizes this information:

Process	Default Schedule	Criteria
`openpolicyexception`	Every night at 2 a.m.	PolicyPeriod entities that are not locked and have a `PolicyException` whose `ExCheckTime` is earlier than `OpenPolicyExceptionThresholdDays` days ago.
`boundpolicyexception`	First Saturday noon	PolicyPeriod entities that are bound (and `MostRecentModel == true`) and have a `PolicyException` whose `ExCheckTime` is earlier than `BoundPolicyExceptionThresholdDays` days ago.
`closedpolicyexception`	First Sunday noon	PolicyPeriod entities that are locked but not bound and have a `PolicyException` whose `ExCheckTime` is earlier than `ClosedPolicyExceptionThresholdDays` days ago.

Exception Handling

Generally, if any of the exception rules throws an error, PolicyCenter logs the exception. PolicyCenter rolls back any changes made by the exception rules to that `PolicyPeriod`. (This is not true, however, if you made the changes in a separate bundle, which Guidewire *explicitly* does not recommend.) PolicyCenter handles each `PolicyPeriod` over which it runs the exception rules in a separate transaction. Therefore, if PolicyCenter encounters an error in `PolicyPeriod` B, it does not roll back changes to `PolicyPeriod` A.

A special loophole exists in the case of the `ConcurrentDataChangeError` error, which occurs any time that two separate transactions try to change the same data. This can happen quite easily in exception rules. For instance, imagine that someone is editing a `PolicyPeriod` at the same time that the exception rules are changing it. If there is a `ConcurrentDataChangeError`, the batch process still rolls back the changes, but remembers the ID of the `PolicyPeriod`. It then tries to run exception rules on it again at a later time. This is standard behavior across all exception rules.

Performance Issues

It is extremely easy to configure the exception rules so as to run in to performance issues, because they run on all `PolicyPeriod` entities of a given type. Guidewire recommends the following:

- Try to put as little expensive logic into the exception rules as possible. The suggested approach is to check for a condition (which is relatively cheap to inspect). Then, if the condition is true, perform a relatively low-cost action (such as creating an Activity).
- If you can disable one or more of exception processes (`openpolicyexception`, `closedpolicyexception`, and `boundpolicyexception`) by removing it from `scheduler-config.xml`, Guidewire recommends that you do so.
- Try to schedule the processes to run as rarely as possible, and use the highest values possible for a `XXPolicyThresholdDays` configuration parameter.

Logging and Debugging

For logging and debugging purposes, Guidewire recommends the following:

- Include the following print statement in your Gosu code as appropriate:

```
print("Ran exception rules on " + PolicyPeriod)
```

- Set the `Server.BatchProcess` logging level to the `DEBUG` level.

User Exception Rules

PolicyCenter runs the User Exception rules on a scheduled basis. It looks for certain conditions on users that might require further attention and to define the follow-up actions for each exception found. PolicyCenter identifies users that have been changed or which have not been inspected for a certain period of time, and runs these rules on each user chosen. This rule set is empty by default.

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

Renewal

A policy renewal takes place before the policy period ends, on an active policy. (It cannot be used to reinstate a canceled policy.) A policy renewal continues the insurance for another time period. It can include changes to the following:

- Coverage
- Underwriting company

- Form update
- Industry or class code

Renewal AutoUpdate

This rule set fires during creation of a `Renewal` job. You can use this rule set to perform any automatic changes to policy data that need to take place during a renewal.

In the following example, the rule updates building limits for all buildings on a BOP policy line renewal.

Rule Conditions

```
PolicyPeriod.BOPLineExists
```

Rule Actions

```
for (bld in PolicyPeriod.BOPLine.BOPLocations.Buildings) {
    var increase = bld.BOPBuildingCov.BOPBldgAutoIncreaseTerm.TypekeyValue.Code as double
    if (increase <> 0) {
        bld.BOPBuildingCov.BOPBldgLimTerm.Value =
        bld.BOPBuildingCov.BOPBldgLimTerm.Value * (1 + (increase/100));
        bld.BOPPersonalPropCov.BOPBPPBldgLimTerm.Value =
        bld.BOPPersonalPropCov.BOPBPPBldgLimTerm.Value * (1 + (increase/100));
    }
}
```

Validation

IMPORTANT PolicyCenter 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.)

Guidewire generally divides data object validation in PolicyCenter into two categories: *class*-based and *rule*-based:

Validation type	Performed on	Using	At what time
Class-based	Policy objects	Gosu classes	As designated; you choose the time and how you validate one of these data objects by specifying the validation you want in a specific Gosu class.
Rule-based	Non-policy objects	Gosu rules	At the time PolicyCenter commits a data “bundle” containing that object to the database; specific “validatable” entities trigger execution of rule-based validation.

For a discussion of these two different strategies for performing validation, see the following:

- “Performing Class-Based Validation” on page 69
- “Performing Rule-based Validation” on page 87

The following section discusses rule-based validation.

Many, if not most, of the validation rule sets are empty by default. Guidewire expects you to create your own business rules to meet your business needs.

Validation in the User Interface

Use the validation rules to flag problems with data that need to fixed before allowing the user to submit the item for further processing. You can use validation rules to do the following:

- Ensure that the user enters data that makes sense

- Ensure that the user enters all necessary data
- Manage relationships between data fields

For example, one validation check verifies that a producer code (501-002554, for example) has at least one role associated with it. If it does not, PolicyCenter displays a warning:

ProducerCode has no roles. A role is required to permit users to be assigned to '501-002554'.

Guidewire makes the following distinction between warnings and errors.

Warning	Warnings are non-blocking, for information purposes only. The user can ignore (clear) a warning and continue.
Error	Errors are blocking, and cannot be ignored or cleared. The user must fix all errors before continuing.

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 some of the more common forms of the `reject` method.

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, provides an error message, and directs the user to the correct field to fix.
`rejectFieldWithFlowStep`	Similar to `rejectField`, but also provides a mechanism to filter out errors not associated with a particular Job wizard step (`flowStepId`). PolicyCenter uses this field only if validating against the `default` validation level. Otherwise, it ignores this field.

The method signature can take one of the following forms:

```
reject(errorLevel, strErrorReason, warningLevel, strWarningReason)
rejectField(strRelativeFieldPath, errorLevel, strErrorReason, warningLevel, strWarningReason )
rejectFieldWithFlowStep(strRelativeFieldPath, errorLevel, strErrorReason, warningLevel,
        strWarningReason, flowStepId)
```

The following table lists the `reject` method parameters and describes their use. Identified issues or problems are either errors (blocking) or warnings (non-blocking). You can indicate a 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.
`flowStepId`	Job wizard step to which this error applies, if any. PolicyCenter uses this field only if validating against the `default` validation level. Otherwise, it ignores this field. If used, PolicyCenter filters out the validation error unless the user is on that particular wizard step. • To specify multiple wizard steps, use a comma-separated list. • To indicate this step and all later steps, place a dash at the end of the `flowStepId`.
`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 within PolicyCenter to indicate the reason for the warning.
`warningLevel`	Corresponding level effected by the validation warning.

Validation Rules and Entity Types

If you define a validation rule, Guidewire recommends that you define it in a rule set declared for a specific entity type. For example, define it in a rule set that declares `Policy`, `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 PolicyCenter would lose this particular reject.

Error Text Strings

Guidewire recommends that you use the `displaykey` class to return the value to use for the `strErrorReason` and `strWarningReason`. 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}
```

Using the following code (`GroupName` has been retrieved already):

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

returns

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

Validatable Entities

PolicyCenter executes rule-based validation automatically any time that there is a bundle commit (to the database) with a direct or indirect change to a validatable entity. For example, if a user saves a newly created or modified activity (one of the validatable entities), PolicyCenter runs the Activity validation rule set. If there are any errors, the commit fails and PolicyCenter displays error messages to the user.

PolicyCenter executes the validation rules on indirect changes based on the Boolean attribute `triggersValidation` on foreign keys and arrays defined in a validatable entity's metadata definition file. For example, PolicyCenter defines the `Account` entity in file `Account.eti` and marks the `Assignment` array off the `Account` entity as triggering validation. Any change to the `Assignment` array causes PolicyCenter to execute the Account validation rules during a bundle commit to the database.

PolicyCenter defines the following objects as validatable entities:

• Account	• Group	• Region
• Activity	• Organization	• User
• Contact	• ProducerCode	

PolicyCenter runs full validation (that transverses the validation graph) on the following entities (in the listed order) as they are modified.

- `Account` entities
- `Group` and `User` entities
- `Activity` entities

Validation Levels

PolicyCenter defines a `ValidationLevel` typelist that rule writer can use to set how valid data must be before continuing. (It is possible for you to create your own validation levels.) PolicyCenter 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 `bindable` with a priority of 6000 is more restrictive than `quotable` with a priority of 7000. For clarity, this discussion uses the terminology of tighter and looser (small numbers are tighter) to indicate more or less restrictive validation levels.

The validate-on-commit process (see "Validation Levels: A Review" on page 73) always checks against the special validation level `default`, which Guidewire defines to be the second-loosest possible level (`priority=8,000`).

In the base configuration, Guidewire defines three immutable validation levels. You cannot delete these validation levels as the internal PolicyCenter 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	10000	The loosest possible validation level against which PolicyCenter validates policies entering PolicyCenter from an external application. *All data must pass "loadsave" to be saved to the database.*
default	8000	The default validation level against which PolicyCenter runs the ordinary validate-on-commit process. PolicyCenter executes the flowstep filter at this level.
quotable	7000	The level a Policy must pass before it can be sent to a rating engine.

PolicyCenter 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
bindable	6000	The level a Policy must pass before it can be bound. (Binding finalizes a given Job and promotes an associated PolicyPeriod to the main branch.
quickquotable	7200	The level a policy must pass before it is ready for a quick quote.
readyforissue	5000	The level a Policy must pass before it can be issued. (Issuance is the special Job that occurs after Submission, causing the Policy to become legally in effect.)

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:

Validation level	Description
System	You can override the `name` and the `priority` of the three system validations levels: • `loadsave` • `default` • `quotable`
Non-system	You can modify or delete any non-system validation level at will.
Custom	You can add new validation levels to the *ValidationLevel* typelist. To do so: **1.** Open the *ValidationLevel* typelist. **2.** Click **`Add`** and fill in the required information.

Triggering Validation

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

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 PolicyCenter 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 how to create an entity that implements the `Validatable` delegate, see "Delegate Data Objects" on page 161 in the *Configuration Guide*.

The validate Method

Guidewire PolicyCenter provides a `validate` method that you can use to trigger validation on a number of important business entities. You can use this method any place that you can use Gosu, including user interface PCF pages, Gosu rules, and Gosu classes. For example:

```
entity.Account.validate()
```

The `validate` method returns a `ValidationResult` object. (PolicyCenter does not persist this object to the database.) You can use this object, for example, with the following base configuration Gosu function to display validation errors in the user interface:

```
gw.api.web.validation.ValidationUtil.showValidationErrorsOnCurrentTopLocation( ValidatableBean,
  ValidationResult, ValidationLevel )
```

Account Validation Rule Example

The Account rule set contains validation rules for `Account` objects (as expected). Accounts are a special type of validatable entities. PolicyCenter does not automatically validate `Account` objects as it validates other high-level entities. Unlike other objects, changes to policy data do not automatically trigger account validation on bundle commit in rules-based validation. PolicyCenter validates an `Account` entity only on bundle commits that actually affect the `Account` entity, such as in the **Account Setup** pages.

In the following example, the rule rejects an account if the account holder contact information does not contain a value for the work phone number.

Condition

```
account.AccountHolderContact.WorkPhone ==  null
```

Action

```
Account.reject( "loadsave", "You must provide a work phone number for the primary contact.",
  null, null)
```

chapter 7

PolicyCenter Rule Execution

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

This topic includes:

Generating a Rule Repository Report

To facilitate working with the Gosu business rules, PolicyCenter 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 `PolicyCenter/bin` directory.
2. At a command prompt, type:

```
gwpc regen-rulereport
```

This command generates the following files:

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

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

Generating a Rule Execution Report

The Guidewire Profiler provides information about the runtime performance of specific application code. It can also generate a report listing the business rules that individual user actions trigger within Guidewire PolicyCenter. 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 PolicyCenter using a user account with access to the Server Tools.
2. Access the Server Tools and click **Guidewire Profiler** on the menu at the left-hand side of the screen.
3. On the Profiler **Configuration** page, click **Enable Web Profiling for this Session**. This action enables profiling for the current session. Profiling provides information about the runtime performance of the application, including information on any rules that the application executes.

   Guidewire also provides a way to enable web profiling directly from within the PolicyCenter interface. To sue the alternative method, press ALT+SHIFT+p within PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 138 in the *System Administration Guide*

Interpreting a Rule Execution Report

After you enable the profiler and perform activity within PolicyCenter, 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 PolicyCenter 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 PolicyCenter executed as a result of user actions within the PolicyCenter 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 PolicyCenter, 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 PolicyCenter executed in the **Properties and Counters** column in the order in which they occurred.

part II

Advanced Topics

chapter 8

Assignment in PolicyCenter

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

This topic includes:

Understanding Assignment

Guidewire refers to certain business entities as *assignable* entities. Thus, it is possible to designate a specific user as the party responsible for that entity. In the PolicyCenter base configuration, Guidewire defines certain entities as either *primarily* or *secondarily* assignable:

Assignment type	Meaning	Entities affected
Primary assignment	The entity can have a single owner only. The entity must implement the `Assignable` and `PCAssignable` delegates.	`Activity`

Secondary assignment	The entity does not have a single owner, but instead has a set of users—with different roles—associated with it. The entity must implement the `RoleAssignments` array.	`Account`
		`Job` (and all its subtypes)
		`Policy`

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

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

Only one entity—`Activity`— implements primary assignment in the Guidewire PolicyCenter base configuration. All other assignable entities use secondary assignment.

Assignment Persistence

PolicyCenter 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 PolicyCenter interface or through some other mechanism.

Assignment Queues

Each group in Guidewire PolicyCenter has a set of `AssignableQueue` entities associated with it. It is possible to modify the set of associated queues to meet your business needs. Currently, Guidewire only supports assigning `Activity` entities to queues. If you assign an activity to a queue, you cannot simultaneously assign it to a user as well.

See also

- “Primary and Secondary Assignment” on page 54
- “Role Assignment” on page 57
- “Assignment Success or Failure” on page 59
- “Assignment Method Reference” on page 60

Primary and Secondary Assignment

At its core, the concept of assignment in Guidewire PolicyCenter is basically equivalent to ownership. The user to whom you assign an activity, for example, is the user who owns that activity, and who, therefore, has primary responsibility for it. Ownership of a `Policy` entity, for example, works in a similar fashion. Guidewire defines an entity that someone can own in this way as *assignable*.

Assignment in Guidewire PolicyCenter 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.

Guidewire PolicyCenter distinguishes between two different types of assignment:

Type of assignment	Description	See
Primary (user-based) assignment	Also known as user-based assignment, this type of assignment assigns an object that implements the `Assignable` and `PCAssignable` delegates to a particular user.	“Primary (User-based) Assignment” on page 55
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.)	“Secondary (Role-based) Assignment” on page 55

Note: Only one entity—`Activity`— implements primary assignment in the Guidewire PolicyCenter base configuration. All other assignable entities use secondary assignment.

Primary (User-based) Assignment

PolicyCenter uses primary assignment in the context of ownership. For example, only a single user (and group) can own an activity. Therefore, an activity is primarily assignable. *Primary* assignment takes place any time that PolicyCenter assigns an item to a *single* user.

Primary assignment objects *must* implement the `Assignable` and the `PCAssignable` delegates. In the PolicyCenter base configuration, the following object implements the required delegates:

- `Activity`

It is common for PolicyCenter to implement the `Assignable` delegate in the main entity definition file and the `PCAssignable` delegate in an entity extension file.

Secondary (Role-based) Assignment

PolicyCenter 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 PolicyCenter, with each role assigned to a specific person. Since each of the roles can be held by only a single user, PolicyCenter 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 PolicyCenter base configuration, the following objects all contain a `RoleAssignments` array:

- `Account`
- `Job` (and all its subtypes)
- `Policy`

For example, an entity—such as `Account`—does not have a single owner. Instead, an account has a set of `User` objects associated with it, each of which has a particular `Role` with respect to the `Account`. As only a single `User` can hold each `Role`, PolicyCenter represents the relationships by a set of `UserRoleAssignment` entities.

To state this differently, it is possible to assign several different users to the same entity, but with different roles. It is also possible for the same user to have several roles on an entity. (Some common role types are Creator, Underwriter, and Producer.)

It is possible for an entity to be both primarily and secondarily assignable.

Determining the Available Roles on an Account, Job, or Policy Object

PolicyCenter uses the following method to determine the available roles to display in the assignment drop-down as you edit an assignment on an account, job, or policy (the `owner` parameter):

```
gw.assignment.AssignmentUtil.filterAssignableRoles(owner, role)
```

The method filters the typekeys defined in the `UserRole` typelist by the specified role type (the `role` parameter). It returns `true` for those typekeys it determines are available in the assignment drop-down. For example, suppose that you start with the existing assignments on an account:

- *Creator* – Bruce Baker
- *Producer* – Bruce Baker
- *Underwriter* – Alice Applegate

If you want to add a new participant (assign a new role), then you need to do the following:

- Determine the set of roles in the `UserRole` typelist (Auditor, AuditExaminer, Creator, Producer, and so on).
- Pass each individual role type and the account (the owner in this case) to the `filterAssignableRoles` filter method.

As you call the method for the auditor role, the filter method determines that there is no currently existing user with that role on the account. It returns `true`. PolicyCenter then displays the **Auditor** role in the assignment drop-down in the PolicyCenter interface. However, if you pass the producer role to the filter method, it determines that this role does currently exist on this account. It returns `false`. Thus, PolicyCenter does not display the **Producer** role in the assignment drop-down in the PolicyCenter interface.

Permitting Multiple Users to Share the Same Role on an Entity

It is not possible—in the base configuration—for different users to share the same role on an entity. PolicyCenter enforces this behavior by setting a `UserRoleConstraint` category on each role in the `UserRole` typelist. For example, PolicyCenter associates the following `UserRoleConstraint` categories with the `AuditExaminer` role:

- `UserRoleConstraint.accountexclusive`
- `UserRoleConstraint.jobexclusive`
- `UserRoleConstraint.policyexclusive`

These constraints mandate that an account (or job or policy) can have at most one user assigned to the `AuditExaminer` role. Internally, the `filterAssignableRoles` method uses these constraints to enforce this behavior. Thus, to permit multiple users to share the same role on an entity, you need to remove that restriction (category) from that role type in the `UserRole` typelist. For example, to remove the restriction that an account can have only one associated user with the `AuditExaminer` role, simply remove that category (`UserRoleConstraint.accountexclusive`) from the `AuditExaminer` role.

To determine the categories set on a role, select a typecode (a role) from the **UserRole** typelist and view the **Categories** pane. See “Dynamic Filters” on page 280 in the *Configuration Guide* for information on typecode category filters.

Secondary Assignment and Round-robin Assignment

Secondary assignment uses the assignment owner to retrieve the round-robin state. Thus, 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 64 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. Of these, only the Activity assignment rules use primary assignment. All other assignment rules use 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 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)
```

PolicyCenter 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.

Role Assignment

In the default configuration of the PolicyCenter application, Guidewire assigns specific roles to specific entities. In actuality, there are many more role available that you can use for assignment. If you need—or want—more roles, you can create them. Role assignment is completely configurable.

Submission Jobs

Assigning submission jobs entails involves the following:

- "Setting the Producer" on page 57
- "Setting the Underwriter" on page 57

As PolicyCenter binds the submission, it passes the Underwriter, Producer, and Creator roles to the created Policy. (By default, PolicyCenter automatically assigns the roles of Creator and Requestor to the user that creates the initial policy submission.)

Setting the Producer

The process of assigning a producer to a submission job involves the following decision tree:

1. First, attempt to assign a user (with a user type of Producer) using the `ProducerCode` entered during job creation:
   - Check if the `CurrentUser` is a Producer for that `ProducerCode`. If so, preferentially choose that `CurrentUser`.
   - Otherwise, use round-robin on the Group to search for users with the required Producer user type.
2. If the assignment method cannot find a suitable user, try to set a Producer from the Account producer codes.
3. If the assignment method cannot find a valid producer, set the Producer to superuser and log a warning that the method only performed a default assignment.

Setting the Underwriter

The process of assigning an underwriter to a submission job involves the following decision tree:

1. First, attempt to assign the `PreferredUnderwriter` from the `ProducerCode` as the default underwriter for the Submission job.
2. If the assignment method cannot find an underwriter, attempt to set an Underwriter from `ProducerCode.Branch`.
   - Check if the `CurrentUser` is an underwriter for that branch. If so, preferentially choose that `CurrentUser`.
   - Otherwise, use round-robin on the Group to find a user. If round-robin selects a user that does not have a user type of underwriter, look for any user in the group that is an underwriter. (Guidewire strongly recommends that you put only underwriters into the branch to take advantage of the load-balancing properties of round-robin assignment.)
3. If the assignment method cannot find a valid user, set the Underwriter to superuser and log a warning that the method only performed a default assignment.

Non-submission Jobs

If the `ProducerCode` changes during any job, PolicyCenter updates the Policy with the new `ProducerCodeOfService`. For any of the following (new) jobs started from the policy, PolicyCenter adds the policy's current Underwriter and Producer role assignments to the job by default.

- Cancellation
- PolicyChange
- Reinstatement
- Renewal
- Rewrite

Audit is a special case. Instead of assigning a producer or an underwriter, PolicyCenter assigns an auditor at the beginning of the job. PolicyCenter chooses this auditor from the default organization.

Account

Account assignment is relatively simple. PolicyCenter merely sets the Creator role to `CurrentUser` and sets nothing else.

Policy/Pre-renewal Direction

Guidewire exposes the assignment methods underlying the Pre-Renewal Direction process in the **Pre-Renewal Direction** pages. Use method `getSuggestedPreRenewalOwners` to populate the **RelatedTo** drop-down list. Selecting a value assigns it to `PreRenewalOwner`. The end result of assigning the Pre-Renewal Direction is a role assignment to `PreRenewalOwner`. Thus, the Policy maintains the assignments of Underwriters and Producers *and* maintains the assignment of the Pre-Renewal Owner as well.

Pre-Renewal Owner is set using explicit assignment and property "getter" methods, which all exist in `gw.assignment.PolicyAssignmentEnhancement`:

- `PreRenewalOwner` property getter
- `PreRenewalOwner` property setter
- `getSuggestedPreRenewalOwners` method

Activity Assignment

To create a new activity and assign it, use the `JobAssignmentEnhancement.createRoleActivity` method, which assigns the activity to the user with the given role on a job. This method has the following signature:

```
createRoleActivity(role, pattern, subject, description)
```

This creates an activity with the given activity pattern, then assigns a user with the given role to that activity.

Gosu Support for Assignment Entities

Each entity that can have a primary assignment—including the `UserRoleAssignment` entities representing secondary assignments—must implement the following interfaces:

- `Assignable`
- `PCAssignable`

To facilitate the process of assigning entities, Guidewire provides the following Gosu classes and enhancements in the `gw.assignment` package:

Class or enhancement	Description
`AssignmentUtil`	Contains shared code for checking roles, logging assignments, and setting the default user.
`AuditAssignmentEnhancement`	Assign an auditor to an Audit job.
`JobAssignmentEnhancement`	Contains shared job assignments such as underwriter and producer, pushes assignments from job to policy, and contains `Activity` assignment helpers.
`PolicyAssignmentEnhancement`	Contains Pre-Renewal Owner assignment and suggestion.
`ProducerCodeAssignmentEnhancement`	Retrieves the assignment group for a user within the `ProducerCode`.
`UserAssignmentEnhancement`	Retrieves the default assignment group from the user directly or for a given list of group types.

Assignment Success or Failure

All assignment methods return a `Boolean` value that is `true` if PolicyCenter successfully makes an assignment, and `false` otherwise. It is your responsibility to determine what to do if PolicyCenter does not make an assignment. You can, for example, assign the item to the group supervisor or invoke another assignment method.

IMPORTANT Do not attempt to make direct assignments to `defaultowner`.

Activity Assignment to Roles

To assign activities to roles, PolicyCenter uses `AssignmentUtil.DefaultUser` as the default user to use if the assignment fails. You can modify `AssignmentUtil.DefaultUser` to change the default user throughout PolicyCenter. You can also modify individual PCF files to use a different default user than `AssignmentUtil.DefaultUser`.

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 is 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 might be necessary to call another assignment method to assign the item to an actual user.

Logging Assignment Activity

Guidewire recommends also that you log the action any time that you use one of the assignment methods. Class `gw.assignment.AsssignmentUtil` provides several helper methods for logging assignment changes. These include the following:

- `assignAndLogUserRole`
- `logUserRoleAssignment`

To see examples of the use of these logging methods, see `JobAssignmentEnhancement` in the same package. For example, the following method uses `assignAndLogUserRole` to log information about the assignment process:

```
/**
 * Assign roles from the Policy to the Job
 */

function assignRolesFromPolicy() {

  var producer = this.Policy.getUserRoleAssignmentByRole( "producer" )
  if (producer != null) {
    AssignmentUtil.assignAndLogUserRole( this, producer.AssignedUser, producer.AssignedGroup,
            "producer", "Job.assignRolesFromPolicy()" )
```

```
  }

  var underwriter = this.Policy.getUserRoleAssignmentByRole( "underwriter" )
  if (underwriter != null) {
    AssignmentUtil.assignAndLogUserRole( this, underwriter.AssignedUser, underwriter.AssignedGroup,
            "underwriter", "Job.assignRolesFromPolicy()" )
  }
}
```

Assignment Events

Any time that the assignment status changes on an assignment, PolicyCenter fires an assignment event. There are multiple possible events that 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

Guidewire divides the assignment methods into the following general categories:

- “Queue Assignment” on page 60
- “Immediate Assignment” on page 61
- “Condition-based Assignment” on page 62
- “Round-robin Assignment” on page 64
- “Dynamic Assignment” on page 64

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 selecting the F1 keyboard key.

IMPORTANT Within the Studio interface, it is possible to see several different assignment methods that involve some aspect of proximity searching (the `assignUserByLocationUsingProximitySearch` method, for example). Studio displays these assignment methods as part of its underlying platform assignment configuration. However, Guidewire expressly does not support these types of assignment methods for Guidewire PolicyCenter. If you attempt to use one of these methods, PolicyCenter fails to perform an assignment.

Queue Assignment

Each group in Guidewire PolicyCenter 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 supports assigning `Activity` entities to a Queue only.

Within PolicyCenter, an administrator can define and manage queues through the PolicyCenter **Administration** screen.

assignActivityToQueue

```
boolean assignActivityToQueue(queue, currentGroup)
```

Use this method to assign an activity to the specified queue. The activity entity then sits in the queue until a group supervisor reassigns the activity manually or a user chooses 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:

Using a Finder Method to Define an AssignableQueue Object

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.

Using the Group Name to Define an Assignable Queue Object

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.

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 User and Default Group Method” on page 61

Assign User and Default Group Method

The `assignUserAndDefaultGroupMethod` 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. Generally, you use this method only if a user belongs to a single group, or if the assigned group really does not matter.

```
boolean assignUserAndDefaultGroup(user)
```

It is possible that the assigned group can affect visibility and permissions. Therefore, Guidewire recommends that you 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)
```

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.

It is important to understand the following:

- Condition-based assignment methods tie 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 always re-uses the same round-robin sequence. However, two different sets of restrictions always result in *distinct* round-robin sequences, even if the set of resulting users is the same.
- Condition-based assignment methods that use round-robin assignment do not use the load factors maintained in the PolicyCenter **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:

- "Assign by User Attributes Method" on page 62
- "Assign User by Location Method" on page 63
- "Assign User by Location and Attributes Method" on page 64

Assign by User Attributes Method

The `assignByUserAttributes` method assigns an assignable item to the user who best matches the set of user attribute constraints defined in the `attributeBasedAssignmentCriteria` parameter. It is possible that you need to assign some policies to specific users, such as those who speak French or those who have some sort of additional qualification. These users can exist across an organization, rather than exist as members of a single group.

```
boolean assignByUserAttributes(attributeBasedAssignmentCriteria, includeSubGroups, currentGroup)
```

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. Thus, you can use this method to maintain group-specific assignment rotations.
2. After the method selects a user, it determines the best group for the assignment. Then, the method assigns the entity to this group or one of its subgroups—if the method finds 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 ANDs the restrictions together.

The AttributeCriteriaElement entity has a number of fields, which are all optional. These fields can interact in very dependent 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 AttributValue of *French*. Notice also that the search does not include subgroups and that the method uses AssignmentEngineUtil.getCurrentGroupFromES to retrieve the current group.

```
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,
        gw.api.assignment.AssignmentEngineUtil.getCurrentGroupFromES() )
```

Assign User by Location Method

The assignUserByLocation method uses the location-based assigner to assign an assignable entity based on a given address.

```
boolean assignUserByLocation(address, includeSubGroups, currentGroup)
```

The method matches users first by zip, then by county, then by state. The first match wins. If one or more users match at a particular location level, then the method 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 postal code, 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.

The assignUserByLocation method bases state persistence in round-robin assignment on the specified location information. For this reason, it is preferable to use a partially completed location, such as one including only the postal code, rather than a specific house location.

The following example assigns a user based on the primary location of the account associated with the activity.

```
Activity.assignUserByLocation( Activity.Account.PrimaryLocation.Address, false,
        Group( "default_data:1" /* Default Root Group */ ) )
```

Assign User by Location and Attributes Method

The assignUserByLocationAndAttributes method 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. (This is the address of a policy holder, for example.) You can then impose of additional restrictions, such as the requirement to handle large dollar amounts or to speak a foreign language, for example.

```
boolean assignUserByLocationAndAttributes(address, attributeBasedAssignmentCriteria, includeSubGroups,
        currentGroup)
```

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,
        gw.api.assignment.AssignmentEngineUtil.getCurrentGroupFromES() )
```

Round-robin Assignment

The round-robin algorithm rotates through a set of users, assigning work to each in sequence. See “Secondary (Role-based) Assignment” on page 55 for a discussion on secondary assignment and round-robin states.

Assign User by Round Robin Method

The `assignUserByRoundRobin` method uses the round-robin user selector to choose the next user from the current group or group tree to receive the assignable.

```
boolean assignUserByRoundRobin(includeSubGroups, currentGroup)
```

If the `includeSubGroups` parameter is true, the selector 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 policies that you want to assign to a particular group. If you want all of the users within the group to share the workload, then set the `includeSubGroups` parameter is 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 you must lock the rows in the database.
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 must assign.
6. Commit the transaction. This updates and unlocks the lock entities.

   Dynamic assignment is not complete after these steps. 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 at a later save of the item.

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 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 the current machine obtains and holds 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 so 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 `findUserToAssign` 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 must 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 PolicyCenter commits the modifications at the same time as it commits the assignment change.

rollbackAssignment

Guidewire provides the final two methods on the API to deal with situations in which, after the assignment flow, some problem in the bundle commit blocks the assignment. This can happen, for example, if a validation causes 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, PolicyCenter 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 readjust the load numbers.

getAssignmentToken

If the `rollbackAssignment` method returns `false`, then PolicyCenter 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 an assignment does not commit, your logic does one of the following:

- PolicyCenter rolls back any database changes that have already been made.
- PolicyCenter preserves the assignment in the event that the you invoke the assignment logic again.

Sample DynamicUserAssignmentStrategy Implementation

The following code shows the implementation of a `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. It simply looks 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
  }
}
```

Using Assignment Methods in Assignment Pop-ups

In Guidewire PolicyCenter, 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 activity 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 modify the upper part of this popup to call a specific assignment method. 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 business logic to assign the passed-in `PCAssignable` object. The following is an example of this.

```
package gw.api

class ExtensionAssignee implements gw.api.assignment.Assignee {

  construct()

  override function assignToThis(assignableBean : com.guidewire.pl.domain.assignment.Assignable) {
    var users = gw.api.database.Query.make(User).compare("PublicID", Equals, userID)
    var usr = users.select().AtMostOneRow
    assignableBean.autoAssign( usr.GroupUsers[0].Group, usr )
  }

  function toString() : String {
    return "Bakeriffic Assignment"
  }
}
```

2. Modify the PCF files to include this new `Assignee` object, removing others as necessary. One option, for example, is 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 = this.SuggestedAssignees
  assignees[0] = new gw.api.ExtensionAssignee()
  return assignees
}
```

chapter 9

Performing Class-Based Validation

This topic describes the class-based validation in Guidewire PolicyCenter, how to use it, and how to configure it.

This topic includes:

What is Class-Based Validation?

Note: The terms *object* and *entity*—while not absolutely identical—are used interchangeably throughout this topic. See "Important Terminology" on page 13 for a description of each.

Guidewire generally divides data object validation in PolicyCenter into two categories: *class*-based and *rule*-based:

Validation type	Performed on	Using	At what time
Class-based	Policy objects	Gosu classes	As designated; you choose at what time and how you validate one of these data objects by specifying the validation you want in a specific Gosu class.
Rule-based	Non-policy objects	Gosu rules	At the time PolicyCenter commits a data "bundle" containing the object to the database; specific "validatable" entities trigger execution of rule-based validation.

For a discussion of these two different strategies for performing validation, see the following:

- "Class-Based Validation: An Overview" on page 70
- "Rule-based Validation: An Overview" on page 88

In the base configuration, Guidewire performs class-based validation on the following policy-related objects:

- PolicyContact
- PolicyLocation
- PolicyPeriod
- Line-of-business entities
- Policy-specific entities

It is important to understand, however, that you can perform class-based validation on basically any business object (including those objects that you create by extending a base object). The listed objects are simply those for which Guidewire provides validation classes in the base configuration.

IMPORTANT Guidewire does not recommend, nor support, the use of class-based validation as a feedback mechanism for user action. Instead, if you need to provide that feedback, use confirmation dialog boxes and user-viewable exceptions. Class-based validation is not appropriate in all contexts, nor in all situations—use it carefully and thoughtfully.

Class-Based Validation: An Overview

PolicyCenter performs class-based validation on policy-related entities any time that you call for validation on the entity. In the base configuration, Guidewire provides the following validation classes that you can use with a specific entity. (The name of the validation class contains the name of the entity to which it applies or its purpose.)

All Lines of Business

- AnswerValidation
- FormPatternValidation
- InvariantValidation
- PolicyContactRoleValidation
- PolicyContactRoleForSameContactValidation
- PolicyHoldValidation
- PolicyLineValidation
- PolicyLocationValidation
- PolicyPeriodValidation

Business Auto

- BALineValidation
- BusinessVehicleValidation

Businessowners

- BOPBuildingValidation
- BOPLineValidation
- BOPLocationValidation
- BOPPolicyInfoValidation

Commercial Property

- CostOverrideValidation
- CPBlanketValidation
- CPBuildingValidation
- CPLineValidation

General Liability
- GLExposureValidation
- GLLineValidation

Inland Marine
- ContractorsEquipmentPartValidation
- IMARPartValidation
- IMLineValidation
- IMSignPartValidation

Personal Auto
- PALineAssignmentValidator
- PALineCoveragesValidator
- PALineDriversValidator
- PALineQuickQuoteValidation
- PALineStepsValidator
- PALineValidation
- PALineVehiclesValidator
- PersonalVehicleValidation

Workers' Compensation
- WCJurisdictionValidation
- WCLineValidation
- WCPolicyInfoValidation

It is important to understand that there is nothing particularly special about these entities. Thus, they are not `validatable` entities in the sense that rule-based validation entities are defined as validatable. Guidewire provides these validation classes to illustrate how to create validation classes for specific purposes. For example, the base configuration validation classes illustrate:

- How to create a Gosu class to validate any defined entity in the PolicyCenter data model. `PALineValidation.gs` is an example of a class that validates a specific entity, in this case, the `PersonalAutoLine` entity.
- How to create a Gosu class to validate a wizard step (for example, the **Policy Info** step in a LOB submission wizard). `WCPolicyInfoValidation.gs` and `BOPPolicyInfoValidation.gs` are examples of this type of validation.

It is possible to write your own Gosu validation classes and implement your own validation logic on any entity that you choose. However:

- Each validation class that you create must implement the `PCValidation` interface. A convenient way to do this is to extend `PCValidationBase` or one of its subtypes.
- Each validation class that you create must provide an overridden `validate` function.

To summarize:

- PolicyCenter can perform validation on any entity—not just those designated as `validatable` in the base configuration—using an appropriate Gosu class. See “Performing Rule-based Validation” on page 87 for a discussion of the base configuration entities marked as `validatable`.
- PolicyCenter can perform validation on a specific wizard step using an appropriate Gosu class. See “Invoking Class-Based Validation” on page 83 for an example of how to invoke validation from a wizard step.
- PolicyCenter performs validation *as designated* (by the appropriate Gosu class), not only at the time that it commits data to the database.
- PolicyCenter displays validation error and warning messages in the user interface *only* if they apply to the validation level of the given context. (See “Validation Levels: A Review” on page 73 for information on validation levels.)

- PolicyCenter permits you to create and add arbitrary validation logic to the (rule-based) validation graph using Gosu classes.

See also

- “Class-Based Validation Configuration” on page 74

Field-Level Validation: A Review

Note: PolicyCenter field-level validation is not part of the class- or rule-based validation framework. Field-level validation does not involve Gosu business rules or classes. It is simply part of the PCF rendering framework.

The PolicyCenter rendering framework builds the user interface from sets of PCF files. During the rendering process, PolicyCenter performs any *field-level* checks that have been configured in the PCF files. For example, if you configure a specific field in a PCF as required, then if the user leaves that required field empty, the rendering framework generates a user error.

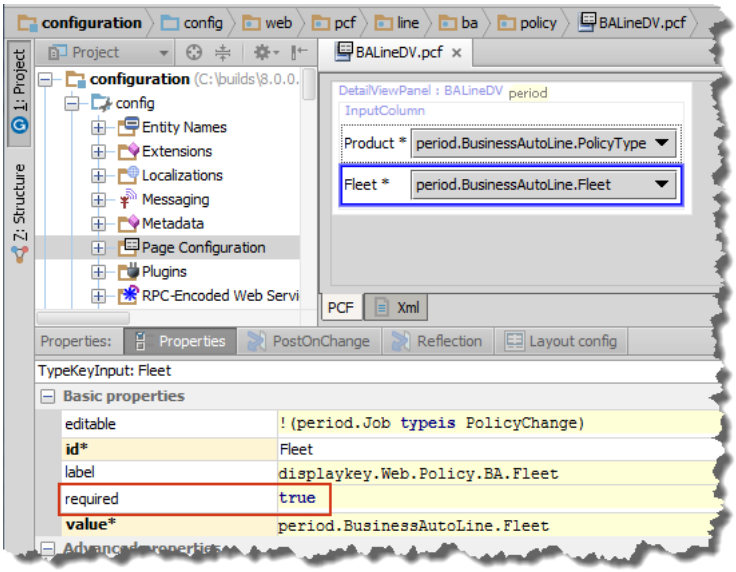
This type of field-level validation is independent of class- or rule-based validation. PolicyCenter performs this type of validation before policy validation and displays problems that occur at the PCF level before it runs policy validation or other processes.

Suppose, for example, that a user does not enter a value for the **Fleet** field on the (Submission wizard) **Policy Info** page. In this case, the rendering framework automatically generates a field-level validation error as the user attempts to move to the next step in the Submission wizard.

PolicyCenter generates the error message text string from the `MissingRequired` display key, substituting the field name for the `{0}` variable:

```
Java.Validation.MissingRequired = Missing required field "{0}"
```

In this particular instance, you specify that the **Fleet** field in the `BALineDV.pcf` PCF file is required. You do this by setting the `required` property to `true`.



Validation Levels: A Review

In the base configuration, PolicyCenter defines a set of validation levels (in typelist `ValidationLevel`) that you can use to verify how *valid* data is before continuing. (It is possible for you to create your own validation levels.) PolicyCenter uses the type key *priority* attribute to order the validation levels. Lower priority numbers override higher priority numbers. This is sometimes confusing. Thus, in the base configuration. `bindable` with a priority of 6000 is more restrictive than `quotable` with a priority of 7000. You can also think of this as *tighter* and *looser* (smaller numbers are tighter) to indicate more or less restrictive validation levels.

In the base PolicyCenter configuration, Guidewire defines the following immutable validation levels. (As Guidewire defines these levels at the platform level, you cannot modify them.)

Level	Priority	Description
`loadsave`	10,000	The loosest possible validation level against which PolicyCenter validates policies entering PolicyCenter from an external application. All data must pass “loadsave” to be saved to the database.
`default`	8,000	The default validation level against which PolicyCenter runs the validate-during-commit-cycle process. PolicyCenter executes the flowstep filter at this level.
`quotable`	7,000	The level a Policy must pass before it can be quoted.

PolicyCenter 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
`quickquotable`	7,200	The level a Policy must pass before it can be quoted in a Quick Quote job.
`bindable`	6,000	The level a Policy must pass before it can be bound.
`readyforissue`	5,000	The level a Policy must pass before it can be issued.

Validation Levels and Class-based Validation

Through the Gosu validation classes, Guidewire provides methods to help you write checks to determine if one of the policy-specific entities is passing validation at a certain priority level.

For example, you can check “`if Context.levelIsAtLeast(“bindable”)`” to see if the validation level being tested is at least at the bindable level. Depending on the result of your validation checks, you can configure PolicyCenter to display an error or warning message to the user.

To illustrate, suppose that one class-based check validates that a billing plan must be entered before a policy is bound. Thus, you define the following condition in a validation class to enforce the constraint and generate an error message in PolicyCenter if the condition is not met.

```
var atBindable = Context.levelIsAtLeast(“bindable”)
if ( (Period.BillingPlanDetail == null) && (atBindable == true) {
  Result.addError(Period, "bindable", displaykey.web.Policy.MustHaveBillingPlan)
}
```

PolicyCenter then displays an error if both of the following are `true`:

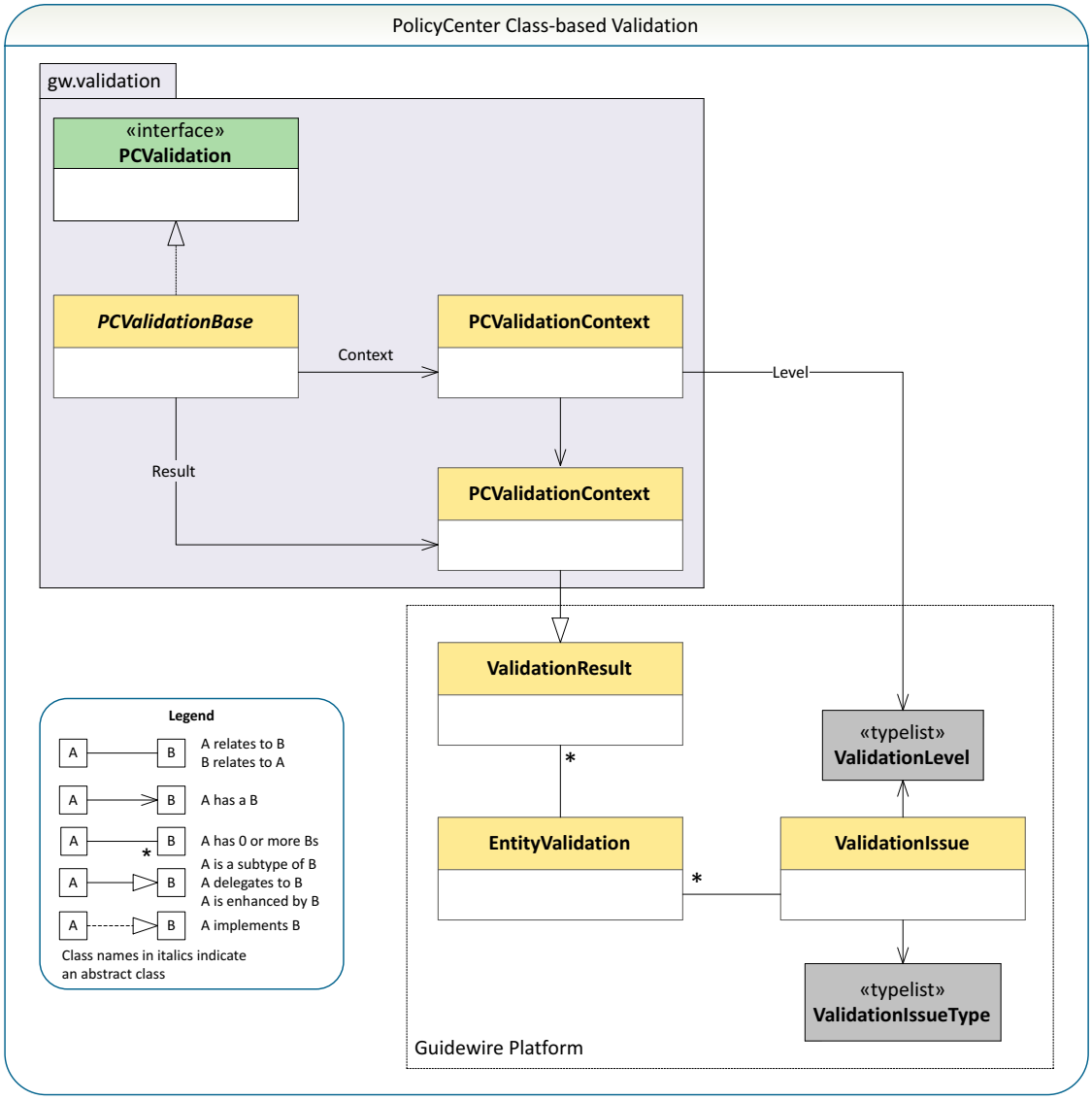
- A billing plan does not exist (`Period.BillingPlanDetail == null`)
- The policy level being checked is at least at the `bindable` level (`atBindable == true`)

Class-Based Validation Configuration

Guidewire provides the following validation-related classes and interfaces in package `gw.validation`.

Class or Interface	Description
`PCValidation`	Interface that all validation classes must implement.
`PCValidationContext`	Class that takes a `ValidationLevel` and creates a new `PCValidationResult` object during initialization.
`PCValidationBase`	Abstract convenience class that takes a `PCValidationContext` instance.
`PCValidationResult`	Class that contains the object methods to use in generating warnings and errors.

The following object diagram illustrates these relationships.



PCValidation

All Gosu validation classes must implement the `PCValidation` interface. Any Gosu class (or any interface defined in Gosu) that implements `PCValidation` can perform validation logic. This interface contains a single `validate` method.

Classes that implement this interface can create methods that test for a single issue and call those method from their implementation of the `validate` method. For example (from `PALineDriversValidator.gs`):

```
override function validate() {
  Context.addToVisited(this, "validate")
  qualifiedGoodDriver()
  appliedGoodDriverDiscount()
  licenseInfoRequired()
  licenseNumberUnique()
  licenseStateMatchesGarageState()
  primaryAddressRequiredFields()
  accountDriverNumberOfIncidents()
  verifyAndOrderMVRs()
  verifyBirthDateAndIncidents()
}
```

PCValidationBase

Class `PCValidationBase` is a convenience class that implements `PCValidation`. Its constructor takes a validation context (`PCValidationContext`) instance that holds the level at which to perform validation *and* the validation results (warnings and errors).

```
protected construct(valContext : PCValidationContext) {
    _context = valContext
}
```

`PCValidationBase` provides a number of *getter* property methods. Some of the more important are:

```
property get Context() : PCValidationContext
property get Level()   : ValidationLevel
property get Result()  : PCValidationResult
```

PCValidationContext

Class `PCValidationContext` takes a `ValidationLevel` and creates a new `PCValidationResult` during initialization. Its constructor takes the following form:

```
construct(valLevel : ValidationLevel) {
  _result = new PCValidationResult()
  _level = valLevel
}
```

This class has several important methods for use in managing validation. The following list describes each briefly. However, for the most complete information, consult the Gosu documentation associated with the method. To see the documentation, place your cursor in the method signature and press F1 on the keyboard.

Method	Description
`addToVisted(validation, methodName)`	Use to track a complete listing of the checks that PolicyCenter performed during the validation. The method returns `false` if the given `validation.Name` and `methodName` have been visited before. You can use this to determine that this particular validation method does not need to be checked again. This method only checks against the class name, not the validation object itself (using `hasVisited`). Thus, for example, if you are validating multiple vehicles, you can not discern whether a method has been visited for a specific vehicle.
`hasVisited(className, methodName)`	Use to test whether the given combination of validation class name and method name have been seen before. If so, the method returns `true`.
`isAtLeast(ValLevel)`	Use to perform a test to determine if the level specified by `ValidationContext` is at least `valLevel`. This method does not actually perform validation. Instead, you use the information to determine whether you want to perform validation.

Method	Description
`showVisited()`	Use to produce a string that lists the validation methods that were visited as validation was performed with the provided `Context`. You can then use this string for debugging.
`resetVisited()`	Resets the set of visited validation methods.
`raiseExceptionIfProblemsFound()`	Throws an `EntityValidationException` if either errors or warnings have been added to the validation context.
`raiseExceptionIfErrorsFound()`	Throws an `EntityValidationException` only if errors have been added to the validation context.

PCValidationResult

Class `PCValidationResult` holds the warnings and errors added by the validation implementation classes as problems are discovered. A warning is non-blocking. The user can clear the warning and continue. In contrast, an error blocks any further progress until the user resolves the problem.

`PCValidationResult` contains a number of object methods that you use in generating warnings and errors. The following list describes the more important ones briefly. However, for the most complete information, consult the Gosu documentation associated with the method. To see the documentation, place your cursor in the method signature and press F1 on the keyboard.

Method	Description
`addError`	Use to add a general error message. This method takes the following arguments, of which the first three are mandatory and the last optional: • keyable bean (entity) that is the source of the error • validation level associated with the error • error reason to display to users • ID of the wizard step where the error can be corrected If you supply a `wizardstepID`, the method creates a link to the wizard step with ID `wizardStepId` if the error is not in the current step.
`addFieldError`	Use to add an error message specifically associated with a field on the given keyable bean as defined by the relative `FieldPath`. This method takes the following arguments: • keyable bean (entity) that is the source of the error • field path to the field that must be changed to resolve the error • validation level associated with the error • error reason to display to users • ID of the wizard step where the error can be corrected If you supply a `wizardstepID`, the method creates a link to the wizard step with ID `wizardStepId` if the error is not in the current step.
`addWarning`	Similar to the `addError` method, except that it generates a warning rather than an error.
`addFieldWarning`	Similar to the `addFieldError` method except that it generates a warning rather than an error.

For example:

```
Result.addError(paLine, "quotable", "Policy cannot have both mutually exclusive coverages.", "PALine")
Result.addFieldError(paLine.Vehicles[0], "Make", "quotable", "The make of the vehicle is unknown.",
        "Vehicles")
Result.addWarning(paLine, "loadsave", "The policy must have at least one vehicle.", "Vehicles")
Result.addFieldWarning(paLine.Vehicles[0], "Model", "quotable", "The model of vehicle is unknown.",
        “Vehicles”)
```

Note: The previous code examples use hard-coded text strings for the error messages. This is for illustration purposes only. In actual practice, Guidewire strongly recommends that you use display keys rather than hard-coded text strings.

PCValidationResult inherits several different forms of the reject method from the (platform) ValidationResult class (which it subclasses).

Method form	Description
reject	Indicates a problem and provides an error message, but does not point to a specific field.
rejectField	Indicates a problem with a particular field, provides an error message, and indicates the correct field to fix.

Base Configuration Validation Classes

In the PolicyCenter base application, class PCValidationBase implements interface PCValidation. All validation classes must subclass PCValidationBase or one of its subclasses. The following list shows the validation classes that PolicyCenter provides in its base configuration. The list indentation indicates the level of interface implementation or class extension.

```
PCValidation (Interface)
    PCValidationBase (Abstract Class)
        AbstractBuildingValidation (Abstract Class)
            BOPBuildingValidation
            CPBuildingValidation
        AnswerValidation
        BOPLocationValidation
        BOPPolicyInfoValidation
        BusinessVehicleValidation
        ContractorsEquipmentPartValidation
        CostOverrideValidation
        CPBlanketValidation
        CPBuildingValidation
        GLExposureValidation
        FormPatternValidation
        IMARPartValidation
        IMSignPartValidation
        InvariantValidation
        PersonalVehicleValidation
        PolicyContactRoleForSameContactValidation
        PolicyContactRoleValidation
        PolicyHoldValidation
        PolicyLineValidation
            BALineValidation
            BOPLineValidation
            CPLineValidation
            GLLineValidation
            IMLineValidation
            PALineAssignmentValidator
            PALineCoveragesValidator
            PALineDriversValidator
            PALineQuickQuoteValidation
            PALineValidation
                PALineStepsValidator
```

WCLineValidation
PALineVehiclesValidator
PolicyLocationValidation
PolicyPeriodValidation
WCJurisdictionValidation
WCPolicyInfoValidation

Validation Chaining

In the base configuration, the `gw.policy.PolicyPeriodValidation` class is an example of policy-graph validation in which one validation class successively calls (chains to) other validation classes to perform additional validation. For example, while running its validation checks for the policy period, `PolicyPeriodValidation` chains to:

- `gw.policy.PolicyContactRoleValidation` to validate the contact roles on the policy and to `gw.policy.PolicyContactRoleForSameContactValidation` to validate multiple roles on the same contact.
- `gw.policy.PolicyLocationValidation` to validate each policy location.
- `gw.policy.PolicyLineValidation` to validate each of its lines of business.
- `gw.policy.ModifierValidation` to validate that all specified modifiers are valid modifier type codes.

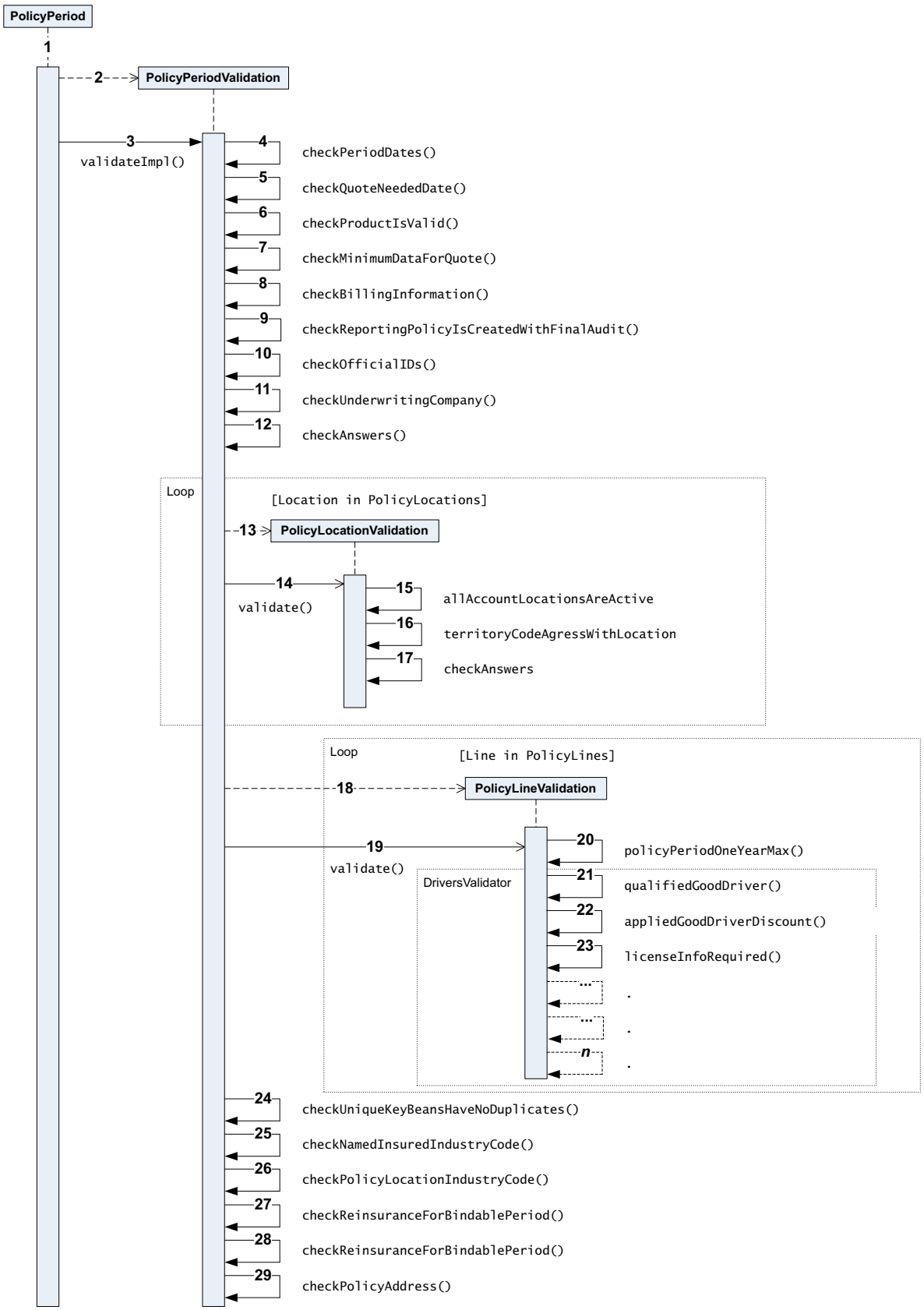
Validation chaining is the process of one validation class calling another validation class to perform additional validation checks. Thus, to chain validation:

- A `validate` method in one validation class first calls the validation methods defined in its class.
- This `validate` method then invokes the `validate` method on another validation class (often by looping through a set of objects).
- In turn, these classes chain to validations of entities they hold. For example, to validate all vehicles on a policy for a Personal Auto policy period, `PolicyPeriodValidation` chains to `PALineValidation` which then chains to `PALineVehiclesValidator`. `PALineVehiclesValidator` calls `PALineCoveragesValidator`, `PALineDriversValidator`, `PALineVehiclesValidator` and `PALineAssignmentValidator`.
- After executing the `validate` methods in each of the related classes, control returns to the calling class. The calling class can then perform additional validation checks.

It is important to understand that validation chaining is not automatic. For a class to perform validation chaining, you must specifically construct it to do so. In this case, Guidewire specifically designed the `PolicyPeriodValidation` class to perform validation chaining.

Validation Chaining Example

The following graphic illustrates the concept of validation chaining. This is a simplified diagram—it does not show every validation method call in the actual implementation. The actual implementation provides an easy way to perform a full validation of the policy period.


PolicyPeriod
1
2
PolicyPeriodValidation
3
validateImpl()
4
checkPeriodDates()
5
checkQuoteNeededDate()
6
checkProductIsValid()
7
checkMinimumDataForQuote()
8
checkBillingInformation()
9
checkReportingPolicyIsCreatedWithFinalAudit()
10
checkOfficialIDs()
11
checkUnderwritingCompany()
12
checkAnswers()
Loop
[Location in PolicyLocations]
13
PolicyLocationValidation
14
validate()
15
allAccountLocationsAreActive
16
territoryCodeAgressWithLocation
17
checkAnswers
Loop
[Line in PolicyLines]
18
PolicyLineValidation
19
validate()
DriversValidator
20
policyPeriodOneYearMax()
21
qualifiedGoodDriver()
22
appliedGoodDriverDiscount()
23
licenseInfoRequired()
...
...
n
24
checkUniqueKeyBeansHaveNoDuplicates()
25
checkNamedInsuredIndustryCode()
26
checkPolicyLocationIndustryCode()
27
checkReinsuranceForBindablePeriod()
28
checkReinsuranceForBindablePeriod()
29
checkPolicyAddress()

Initially, the validateImpl method defined in PolicyPeriodValidation (3 in the graphic) executes a number of validation methods defined in PolicyPeriodValidation, each of which tests for a certain condition. These methods (labeled 4 through 12) are:

- checkPeriodDates
- checkQuoteNeededDate
- checkProductIsValid
- checkMinimumDataForQuote
- checkBillingInformation
- checkReportingPolicyIsCreatedWithFinalAudit
- checkOfficialIDs
- checkUnderwritingCompany
- checkAnswers

After these checks complete—still in the validateImpl method—the code executes a loop through all of the Location objects in PolicyLocations (labeled 13). For each Location, it calls the PolicyLocationValidation validate method on that object (labeled 14). The validation methods within PolicyLocationValidation (labeled 15 through 17) are:

- allAccountLocationsAreActive
- territoryCodeAgreesWithLocation
- checkAnswers

The PolicyPeriodValidation validateImpl method continues by invoking the validate method for each policy line to perform line-specific validation:

```
Period.Lines.each(\ line -> validateLine(line, \ validator -> validator.validate() )
```

For example, if validating a Personal Auto line policy period, the code calls the validate method in PALineValidation.gs (labeled 19). This validation override first calls the local method policyPeriodOneYearMax. It then chains to validate methods in a series of other validation classes:

- CoveragesValidator.doValidate
- AssignmentValidator.doValidate
- VehiclesValidator.doValidate
- DriversValidator.doValidate

Each of these validation classes contains its own override of doValidate. Among these, only a few of the methods in DriversValidator are shown in diagram (labeled 21 through *n*). These methods perform specific checks for driver qualifications, including:

- qualifiedGoodDriver
- appliedGoodDriverDiscount
- licenseInfoRequired

Each of the validators loops as needed to validate all of the coverages, assignments, vehicles, and drivers that apply within the policy period being validated.

Finally, control returns to the original validateImpl method in PolicyPeriodValidation and calls additional local validation methods (labeled 24 through 29), including:

- checkUniqueKeyBeansHaveNoDuplicates
- checkNamedInsuredIndustryCode
- checkPolicyLocationIndustryCode

The following topics discuss parts of this example in more detail:

- “PolicyPeriodValidation: validateImpl Method” on page 81
- “PolicyPeriodValidation Validation Checks” on page 81
- “Invariant Validation Checks” on page 82
- “Static Validation Checks” on page 82

For an explanation of Gosu block definition and syntax, refer to “Basic Block Definition and Invocation” on page 234 in the *Gosu Reference Guide*.

PolicyPeriodValidation: validateImpl Method

Ideally, the `validateImpl` method simply directs the flow of logic. Rather than check for problems itself, Guidewire recommends instead that the `validateImpl` method call other methods, each with a specific purpose. The `PolicyPeriodValidation.validateImpl` method demonstrates this approach. (The following is a simplified version of the method code.)

```
override function validate() {
  if (not Context.addToVisited(this, "validate")) {
    return
  }
  checkPeriodDates()
  checkQuoteNeededDate()
  checkProductIsValid()
    ...
  checkAnswers()
    ...
  Period.PolicyContactRoles.each(\ role -> new PolicyContactRoleValidation(Context, role).validate())
  accountContact.Values.each(\ roles -> new PolicyContactRoleForSameContactValidation(Context,
                roles).validate())
  locs.each(\ loc -> new PolicyLocationValidation(Context, loc, validatedTerritoryCodes).validate())
  Period.Lines.each(\ line -> validateLine(line, \ validator -> validator.validate()))
  checkUniqueKeyBeansHaveNoDuplicates()
  checkNamedInsuredIndustryCode()
  checkPolicyLocationIndustryCode()
  checkReinsuranceForBindablePeriod()
  checkPolicyAddress()
  modifiers.each(\ m -> new ModifierValidation(Context, m).validate() )
  new InvariantValidation(Context, Period).validate()
}
```

First, `validateImpl` registers the fact that it has been called. Notice how the `validateImpl` method perform no checks itself. Instead the `validateImpl` method calls other methods such as `checkPeriodDates` and `checkPolicyAddress`, all of which have a very narrowly focused purpose. Also notice that `validateImpl` chains to validations for entities held by `PolicyPeriod`:

- `PolicyContactRoleValidation`
- `PolicyContactRoleForSameContactValidation`
- `PolicyLocationValidation`
- `PolicyLineValidation`

PolicyPeriodValidation Validation Checks

The `PolicyPeriodValidation` check methods test for specific conditions. Guidewire strongly recommends that the method name summarize the test condition. The following code is for method `checkOfficialIDs`. This test ensures that if the validation level is at or above *bindable*, all required official IDs for the named insured have been set for all states on the policy.

Notice that the check method always registers itself with the `Context`, whatever the validation level. This is useful for debugging as it makes it clear that the method was called during the course of validation. This is true even if no message was added, which would be the case if no problems were discovered.

```
private function checkOfficialIDs() {
  Context.addToVisited(this, "checkOfficialIDs")
  var linePatterns = Period.Lines*.Pattern

  var states = Period.Lines*.CoveredStates.toSet()
  if (Context.isAtLeast("bindable")) {
    // check official IDs for each covered state
    for (covstate in states) {
      var checkOfficialIDs = Period.getNamedInsuredOfficialIDsForState
              (StateJurisdictionMappingUtil.getStateMappingForJurisdiction(covstate))
        .where(\id -> id typeis PCOfficialID and linePatterns.contains(id.Pattern.PolicyLinePattern))
      for (officialID in checkOfficialIDs) {
        officialID.validateOfficialID(Period, Result)
      }
    }
```

```
    }
}
```

Invariant Validation Checks

The `validateImpl` method in the `gw.policy.PolicyPeriodValidation` class also calls the following code:

```
new InvariantValidation(Context, Period).validate()
```

This code initiates what Guidewire calls *invariant* checks. PolicyCenter uses these checks to determine whether a policy has been modified in ways that violate implied constraints to the data model.

The method call creates a new `InvariantValidation` object, whose `validate` method initiates the actual invariant tests and checks.

In the base configuration, these tests perform the following entity checks:

Entity	PolicyCenter checks that...
`Coverage`	• Required coverage terms are not `null`. • Direct coverage terms are within minimum and maximum constraints.
`BusinessAutoLine`	If the Business Auto Line exists, then the correct policy line pattern is applied.
`CoverageSymbolGroup`	• The coverage symbol group is compatible with policy line pattern. • The coverage symbol is compatible with coverage symbol group pattern. • The coverage symbol pattern does not have multiple instances. (Each coverage symbol must appear only once).
`Modifier`	The rate modifier value is within the allowed minimum and maximum values.
`PolicyLine`	• A policy line subtype exists for a policy line. • A policy line pattern exists for a policy line. • The policy line type matches the subtype of the policy pattern.
`Product`	The referenced policy line pattern is compatible with this product.
`RateFactor`	• The rate factor pattern is compatible with the schedule credit pattern. • The rate factor is within the allowable minimum and maximum values

Constraint Errors

If a check indicates that the user-defined product model violates one or more constraints, PolicyCenter generates an appropriate error message.

PolicyCenter stores the invariant errors in display keys. In Studio, in the **Project** window in **Project** view, navigate to **configuration** → **config** → **Localizations** → **en_US** (assuming you are modifying the display key for English). Open display.properties and search for:

```
Java.Invariant.*
```

For example:

```
Java.Invariant.Modifier.RateAboveUpperBound
```

contains the display key:

The rate modifier value “{0}” cannot exceed the allowed maximum “{1}”.

During product model verification, PolicyCenter replaces the variables `{0}` and `{1}` with the actual values.

Static Validation Checks

Within a validation class, you can define static methods that define the specific validation checks to perform at a specific wizard step, reducing the need for full-policy inspections. For example, the **Drivers** steps on the submission wizard for Personal Auto contains the following validation code (on the `beforeSave` property of the `LineWizardStepSet.PersonalAuto.pcf`):

```
gw.lob.pa.PALineStepsValidator.validateDriversStep(policyPeriod.PersonalAutoLine)
```

This code invokes the `validateDriversStep` method in `PALineStepsValidator` before committing the Drivers wizard step.

```
static function validateDriversStep(paLine : PersonalAutoLine) {
    PCValidationContext.doPageLevelValidation(\ context -> {
      var validator = new PALineValidation(context, paLine)
      validator.CoveragesValidator.validate()
    })
}
```

Notice the following:

1. Wrapper method `doPageLevelValidation` provides the exception reporting necessary if a validation check fails to pass. It is the `PCValidationContext.doPageLevelValidation` method that calls the `raiseExceptionIfProblemsFound` method if any of the `validator.DriversValidator` validation checks fail. The `raiseExceptionIfProblemsFound(context)` method instructs the PolicyCenter rendering framework to display the appropriate warning or error automatically if there is an exception condition.
2. The actual validation takes place in the `validate` override located in the `CoveragesValidator` class.

For a discussion of how to use this validation method to raise exceptions within PolicyCenter, see "Example: Invoking Validation in a Job Wizard Step" on page 83.

Invoking Class-Based Validation

PolicyCenter does *not* automatically invoke class-based validation by virtue of the database commit cycle. Instead, you must explicitly invoke class-based validation in configuration code. (The PolicyCenter user interface framework, however, automatically handles the rendering of all errors and warnings *raised through exceptions*.)

PolicyCenter runs full validation on the policy graph at critical points in the job processes, such as before a quote and before binding. At any time, however, you can also invoke class-based validation from PCF files, job processes, wizard steps, workflow steps, integration plugins, or, from any Gosu code.

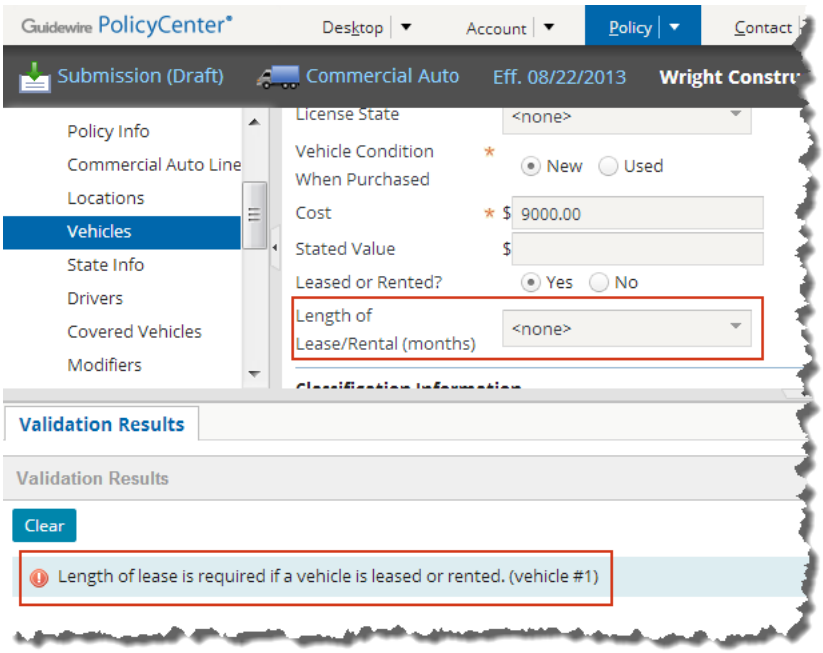
Use the following to invoke validation from a particular PolicyCenter location:

Location	Use
Wizard step	The `beforeSave` property for that step
Pop-up	The `beforeCommit` property for that pop-up

Example: Invoking Validation in a Job Wizard Step

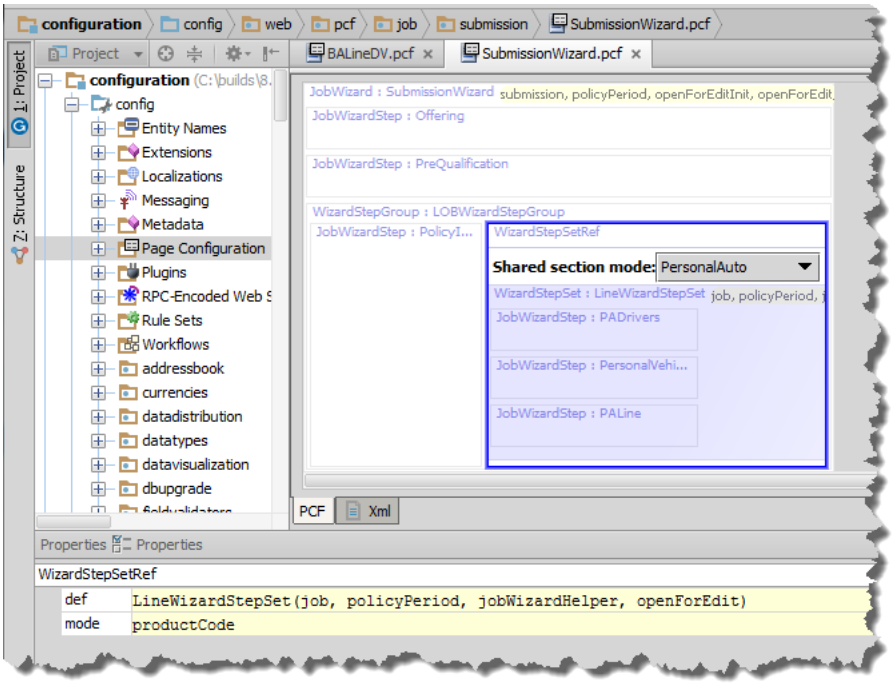
The following example walks through the process of invoking a static validation method from a Job wizard step. Specifically, it illustrates how to link the various Gosu validation classes and methods to the **Vehicles** step in the Personal Auto (PA) **Submission** wizard.

The following graphic shows the **Vehicles** step in the PolicyCenter Submission wizard. The page failed a validation check that occurred as the user clicked **Next**. (This is because there is no value set for the **Length of Lease/Rental (months)** field.)

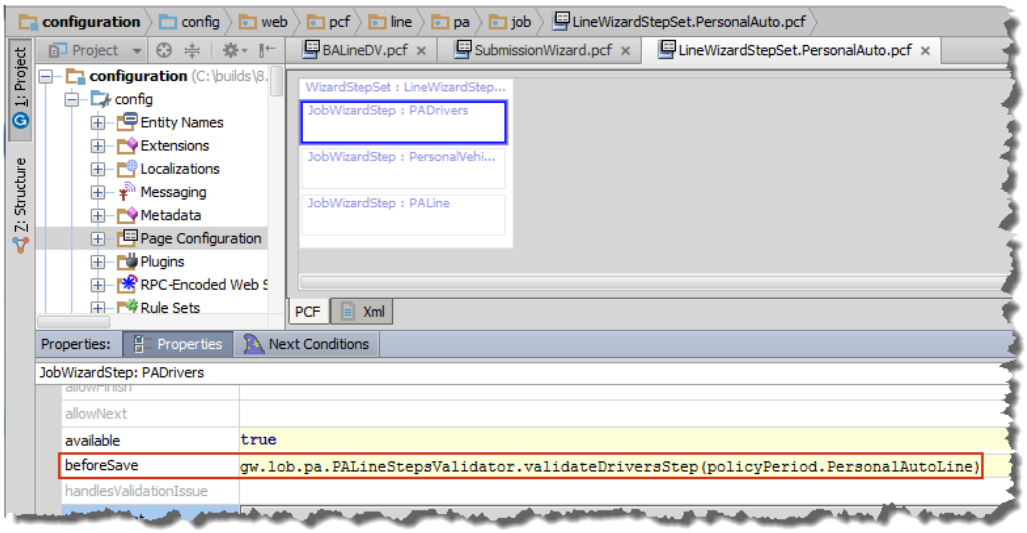


Tracing the Validation Path

Starting with `SubmissionWizard` is a logical first step as the **Vehicles** page is part the PolicyCenter Submission wizard. Examining `SubmissionWizard` PCF file (within Studio) shows that it contains a referenced PCF file named `LineWizardStepSet.PersonalAuto`, which governs the PA **Vehicles** page in PolicyCenter.



It is the value of the `beforeSave` property within `LineWizardStepSet.PersonalAuto` that invokes validation on the **Vehicles Details** card of the **Vehicles** page. (You must open the `LineWizardStepSet.PersonalAuto` PCF in its own **View** tab before you can access its properties.)



The `beforeSave` value consists of the following expressions:

```
gw.lob.pa.PALineStepsValidator.validateDriversStep(policyPeriod.PersonalAutoLine)
```

The `validateDriversStep` method starts the validation chain for the Personal Auto **Vehicles** screen. You can use similar techniques for class-based validation chaining in other screens.

chapter 10

Performing Rule-based Validation

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

This topic includes:

What is Rule-based Validation?

> **Note:** The terms *object* and *entity*—while not absolutely identical—are used interchangeably throughout this topic. See "Important Terminology" on page 13 for a description of each.

Guidewire generally divides data object validation in PolicyCenter into two categories, *class*-based and *rule*-based:

Validation type	Performed on	Using	At what time
Rule-based	Non-policy objects	Gosu rules	At the time PolicyCenter commits a data "bundle" containing that object to the database. Specific "validatable" entities trigger execution of rule-based validation.
Class-based	Policy objects	Gosu classes	As designated—You choose at what time and how you validate one of these data objects by specifying the validation you want in a specific Gosu class.

For a discussion of these two different strategies for performing validation, see the following:

- "Rule-based Validation: An Overview" on page 88
- "Class-Based Validation: An Overview" on page 70

Rule-based Validation: An Overview

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

- `Account`
- `Activity`
- `Contact`
- `Group`
- `Organization`
- `ProducerCode`
- `Region`
- `User`

PolicyCenter validates these entities in no particular order.

Commit-cycle Validation

Rule-based validation is also known as *commit-cycle* validation because it occurs only during a data bundle commit.

See also

- "<implementsEntity>" on page 193 in the *Configuration Guide*

The Validation Graph

During database commit, PolicyCenter 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.

PolicyCenter 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 `Account` and `ProducerCode`. PolicyCenter 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.

PolicyCenter determines the validation graph by traversing the set of foreign keys and arrays that trigger validation. For example, the data model marks the `ProducerCodeRoles` array on `ProducerCode` as triggering validation. Therefore, any changes made to a producer code role causes PolicyCenter to validate the producer code as well.

PolicyCenter 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 `Policy` → `Contact` → `ContactAddress` → `Address`. (To actually trigger validation, each link in the chain—`Address`, `ContactAddress`, and `Contact`—must be marked as triggering validation, and `Policy` must be marked as validatable.)

PolicyCenter stores this path in reverse in the validation graph. Thus, if an address changes, PolicyCenter transverses the tree in reverse order from address to contact address. It then moves to the policy contact, and finally to policy (`Address` → `ContactAddress` → `PolicyContact` → `Policy`).

Traversing the Validation Graph

If a entity is validatable, PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter to validate that entity and all its revisioning parents.

The `triggersValidation` attribute is also available on foreign keys and arrays that have been added to custom subtypes, custom entities, or to core entities using extensions. The attribute defaults to `false`. You must specifically change the attribute to `true` if adding a new foreign key or array that you want to trigger validation to a validatable object.

Owned Arrays

PolicyCenter treats owned arrays (arrays attached to a validatable entity) and marked as `triggersValidation` in a special fashion. PolicyCenter performs validation based on the validation graph. If an entity changes, PolicyCenter 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 PolicyCenter 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 PolicyCenter to validate B and A. This is because PolicyCenter treats the change as a change to B directly. A change to D, however, does not cause PolicyCenter to validate anything, the graph stops at C. As C is not actually changed, PolicyCenter 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 PolicyCenter to validate both B and A.

Top-level Entities that Trigger Full Validation

WARNING Guidewire places the base entity definition files in *PolicyCenter*`/modules/pc/config/metadata`. Do not modify these files in any way. Any attempt to modify files in this directory can cause damage to the PolicyCenter application severe enough to prevent 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 PolicyCenter 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 entity	Array name
Account		RoleAssignments
Activity		
Contact	Address	ContactAddresses
Group		
Organization		
ProducerCode		ProducerCodeRoles
Region		
User	Credential UserContact UserSettings	Attributes

ValidationTrigger Example

In the base configuration, PolicyCenter runs the validation rules on the `Account` object during database commit. Guidewire defines the `Account` entity in `Account.eti`. The PolicyCenter base configuration first defines the `Account` entity as *validatable*, then defines the `RoleAssignments` array as triggering validation.

```
<entity entity="Account" table="account" type="retireable" lockable="true">
  ...
  <implementsEntity name="Validatable"/>
  ...
  <array arrayentity="UserRoleAssignment"
         cascadeDelete="true"
         desc="Role Assignments for this account."
         exportable="false"
         name="RoleAssignments"
         triggersValidation="true"/>
  ...
</entity>
```

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 PolicyCenter base data model that does not currently trigger validation.
- You do not want to trigger validation on entities on which PolicyCenter performs validation in the base configuration.

You can only override validation on certain objects associated with a base configuration entity. (It is not necessary to override your own extension entities as you simply set `triggersValidation` to the desired value as you define the entity.)

The following tables lists the data objects for which you can override validation, the XML override element to use, and the location of additional information.

Data field	Override element	See
`<array>`	`<array-override>`	"<array>" on page 178 in the *Configuration Guide*
`<foreignkey>`	`<foreignkey-override>`	"<foreignkey>" on page 190 in the *Configuration Guide*
`<onetoone>`	`<onetoone-override>`	"<onetoone>" on page 196 in the *Configuration Guide*

See "Working with Attribute Overrides" on page 216 for details on working with these override elements.

Validation Performance Issues

There are three ways in which validation can cause performance problems:

- The rules themselves are too performance intensive.
- There are too many objects being validated.
- The queries used to determine which objects to validate take too long.

The following sections discuss the various performance issues.

Administration Objects

Guidewire recommends that you never mark foreign keys and arrays that point to administration objects—such as `User` or `Group` objects—as triggering validation. (An administration object is basically any object that can be referenced by a large number of other objects.)

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 PolicyCenter must follow each of those chains of relationships during validation. The longer the paths through the tables, the more expensive the queries that PolicyCenter 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 account contact) can result in much longer and unintended paths. For example, a number of entities point to AccountContact. 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 PolicyCenter 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 92.

Links Between Top-level Objects

As previously described, it is legal to have top-level entities trigger validation on each other. This can, however, unnecessarily increase the number of paths and the number of objects that must be validated on any particular commit.

Graph Direction Consistency

In general, Guidewire strongly recommends that you consistently order the validation graph to avoid the previously mentioned problems.

- Arrays and foreign keys that represent some sort of “containment” are candidates for triggering validation.
- Foreign keys that are referenced as arrays or that are merely there to indicate association are not considered candidates for triggering validation.

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 `Policy`) that is not in the database cannot be set as triggering validation. (An array can be considered to be “in the database” if the foreign key that forms the array is in the database.)

Debugging the Validation Graph

To view a text version of the validation graph, open file `logging.properties` (in Guidewire Studio navigate to **configuration** → **config** → **logging**) 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 application system console, but only after application deployment, and after you have opened any policy in that deployment.

chapter 11

Sending Emails

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

This topic includes:

Guidewire PolicyCenter and Email

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

PolicyCenter provides the following email functionality:

- Support for various types of email recipients (To, CC, and BCC)
- Support for templates that can be used to populate the subject and body of the email
- Support for attaching documents stored in the configured DMS (Document Management System) to the email message

Because email messages are sent using the same integration infrastructure as event-based messages, you can use the same administrative tools for monitoring the status of messages. You can view unsent messages in the PolicyCenter **Administration** interface.

The Email Object Model

Guidewire PolicyCenter uses the following two classes to define email messages:

- `gw.api.email.Email`
- `gw.api.email.EmailContact`

Both these classes are simple data containers with almost no logic.

gw.api.email.Email

The `Email` class contains the following fields, most of which are self-explanatory:

Field	Description
`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 Gosu:

```
gw.api.email.EmailUtil.sendEmailWithBody( KeyableBean entity, Email email )
gw.api.email.EmailUtil.sendEmailWithBody( KeyableBean entity,
                                          Contact to,
                                          Contact from,
                                          String subject,
                                          String body )
gw.api.email.EmailUtil.sendEmailWithBody( KeyableBean entity,
                                          String toEmailAddress,
                                          String toName,
                                          String fromEmailAddress,
                                          String fromName,
                                          String subject,
                                          String body )
```

All three methods take an entity as the first parameter. This parameter can be `null`. However, if specified, use the application entity to which this email is related, such as a specific policy or activity. PolicyCenter uses this parameter only while processing the email for transmission. See “Email Transmission” on page 95.

Emails that Use an Email Object

This variation of the `sendEmailWithBody` method requires that you create a `gw.api.email.Email` entity, and then define its properties to build the `Email` entity. For example:

```
...
var testEmail : gw.api.email.Email
testEmail.Body = "This is a test."
testEmail.Subject = "Test"
...
gw.api.email.EmailUtil.sendEmailWithBody( thisClaim, testEmail)
```

Emails that Use Contact Objects

The second variation of the `sendEmailWithBody` method uses `Contact` objects for the `to` and `from` parameters. In Gosu, you can obtain `Contact` objects from various places. For example, in a policy rule, to send an email from an insurance company employee to the insured, do the following:

- Set the `to` parameter to `Policy.insured`.
- Set `from` to `Policy.AssignedUser.Contact`.

The following Gosu example generates an email from the current assigned user to that user’s supervisor:

```
gw.api.email.EmailUtil.sendEmailWithBody(thisPolicy, thisPolicy.AssignedGroup.Supervisor.Contact,
        thisPolicy.AssignedUser.Contact, "A policy got a PolicyValid event", "This is the text." )
```

Emails that Use an Email Address

Use the following variation of the `sendMailWithBody` method if you do not have a full `Contact` object for a recipient or sender. The contact might have been generated dynamically through some other application. The `sendMailWithBody` method uses a name and email address instead of entire `Contact` records and does not require that you have access to a `Contact` record. In the following example, all arguments are `String` objects:

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

Email Transmission

Guidewire PolicyCenter, from the user's perspective, sends emails asynchronously by using the PolicyCenter Messaging subsystem. If there is a method call for one of the `EmailUtil.sendEmail` methods, PolicyCenter creates a `Message` entity with the contents and other information from the `Email` object.

- If the `entity` parameter is non-null, PolicyCenter adds the `Message` entity to the `entity` bundle. PolicyCenter persists the `Message` entity any time that it creates the bundle.
- If the `entity` parameter is `null`, PolicyCenter persists the `Message` entity immediately.

You must configure a `MessageTransport` class to consume the email Messages and do the actual sending. PolicyCenter processes messages one at a time, and sends out the emails associated with that message. For more information, see “Configuring PolicyCenter to Send Emails” on page 98.

Understanding Email Templates

You use email templates to create the body of an email message. Unlike Document templates (but like Note templates), PolicyCenter performs no Gosu interpretation on the Email templates themselves. Thus, email templates are generally only suitable for boilerplate text that does not require modification, or for presenting in the application interface as a starting point for further modification. You cannot use Email templates for mail-merge-style operations.

WARNING Do not add, modify, or delete files from any directory other than the `modules/configuration` application directory. Otherwise, you can cause damage to PolicyCenter.

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 351 in the *Gosu Reference Guide*
- “Data Extraction Integration” on page 607 in the *Integration Guide*

Creating an Email Template

PolicyCenter 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 96
- “Creating an Email Template” on page 97
- “Localizing an Email Template” on page 97
- “Gosu Templates” on page 351 in the *Gosu Reference Guide*
- “Data Extraction Integration” on page 607 in the *Integration Guide*

Localizing an Email Template

Localizing an email template is generally a straight-forward process. To localize an email template:

- Create a locale folder for the template files in following location:

  Other Resources → **emailtemplates** → ***locale-folder***

- Within the locale folder, place localized versions of the email template file and its associated template descriptor file.

To use a localized email template in PolicyCenter, you must perform a search for the localized template as you create a new email.

See also

- "Localizing Templates" on page 69 in the *Globalization Guide*
- "Creating Localized Documents, Emails, and Notes" on page 70 in the *Globalization Guide*

The IEmailTemplateSource Plugin

In the base configuration, Guidewire defines an `IEmailTemplateSource` plugin implementation that provides a mechanism for Guidewire PolicyCenter 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 PolicyCenter to Send Emails

To configure Guidewire PolicyCenter to send emails, you must first define an email server *destination* to process the emails. In the base configuration, PolicyCenter defines an *email* destination with ID 65. In PolicyCenter, 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 131 in the *Configuration Guide*.

You also need to configure the base configuration messaging transport that actually sends the email messages. The PolicyCenter 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. PolicyCenter 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.

PolicyCenter 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 289 in the *Integration Guide*.

Class EmailMessageTransport

In the base configuration, PolicyCenter 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 PolicyCenter itself, then PolicyCenter does not send any messages until you resolve the problem. This ensures that PolicyCenter does not lose any emails.
- If some of the email addresses are bad, then PolicyCenter skips the emails with the bad addresses.
- If all of the email address are bad, then PolicyCenter marks the message with an error message and skips that particular message. PolicyCenter reflects this skip in the message history table.

Class JavaxEmailMessageTransport

In the base configuration, PolicyCenter 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 PolicyCenter 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, PolicyCenter 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 96). You then need to send the email using one of the `sendEmailWithBody` methods described in “Email Utility Methods” on page 94.

Saving an Email Message as a Document

PolicyCenter does not store email objects created through the `Email` class in the database. However, it is possible to save the contents, recipient information, and other information of an email as a document in the configured DMS application. Because the `sendEmailXXX` methods all take all the sending information explicitly, you can save the email information by using the following process in Gosu code:

1. Create the email subject and body, and determine recipients.
2. Send the email by calling the appropriate `EmailUtil` method.
3. If PolicyCenter does not encounter an exception, create a document recording the details of the sent email.

Create Document from Email Example

See “Document Creation” on page 103 for details of the `DocumentProduction` class methods.

```
// First, construct and send the email
var toEmailAddress : String = "Recipient email address"
var toName = "Recipient's Name"
var fromEmailAddress : String = "Sender email address"
var fromName : String = "Sender's name"
var subject : String = "Email Subject"
var body : String = "Email Body"
gw.api.email.EmailUtil.sendEmailWithBody( null, toEmailAddress, toName, fromEmailAddress, fromName,
        subject, body )

// Next, create the document recording the email
var contextObjects = new java.util.HashMap()
```

```
contextObjects.put("To", toEmailAddress)
contextObjects.put("From", fromEmailAddress)
contextObjects.put("Subject", subject)
contextObjects.put("Body", body)
...
var document : Document = new Document()
document.Name = "EmailSent"

// Set other properties as needed
var template : gw.plugin.document.IDocumentTemplateDescriptor
var templateName =  "EmailSent.gosu"
...

// Call this method to create and store the document for later retrieval
gw.document.DocumentProduction.createDocumentSynchronously(template, contextObjects, document)
```

chapter 12

Document Creation

This topic describes synchronous and asynchronous document creation in Guidewire PolicyCenter. It briefly describes the integration points between a document management system and Guidewire PolicyCenter. (For detailed integration information, see “Document Management” on page 191 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 PolicyCenter 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.

PolicyCenter 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 PolicyCenter. 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 PolicyCenter upon successful creation of the document.

See also

For additional information regarding PolicyCenter document creation and management, see the following sections in the *PolicyCenter Integration Guide*:

- "Summary of All PolicyCenter Plugins" on page 141 in the *Integration Guide*
- "Document Management" on page 191 in the *Integration Guide*

Integrating Document Functionality with PolicyCenter

Note: For information on how to integrate document-related functionality with Guidewire PolicyCenter, see the information on plugins in "Document Management" on page 191 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 DocumentProduction Class" on page 109.

Interface	Used for...
`IDocumentTemplateDescriptor`	Describe the templates used to create documents. See “The IDocumentTemplateDescriptor Interface” on page 105 for details of this interface.
`IDocumentTemplateSource`	Retrieval of document templates, which are a part of document creation. In the base configuration, PolicyCenter 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: *`PolicyCenter`*`/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 PolicyCenter Plugins” on page 141 in the *Integration Guide*
- “Document Management” on page 191 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 PolicyCenter how to populate the fields on the template. Within PolicyCenter, the `IDocumentTemplateDescriptor` interface defines the API that any object that represents a document template descriptor must implement.

A document template descriptor contains the following different kinds of information:

Category	Contains
Template Metadata	Metadata about the template itself (for example, the template ID, name, and similar items)
Document Metadata	Metadata defaults to apply to any documents created from the template (for example, the document status)

Category	Contains
Context Objects	Set of values that are referenced by the values to be inserted into the document template, known as *Context Objects*. This includes both default values and a set of legal alternative values for use in the document creation user interface.
Form Fields	Set of field names and values to insert into the document template, including some formatting information.
Document Locale	Locale in which to generate the document.

PolicyCenter 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, PolicyCenter 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, PolicyCenter 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.

getXXX method	Return type	Returns
getDocumentProductionType	String	If present, you can use this property to control which IDocumentProduction implementation to use to create a new document from the template. See “Document Management” on page 191 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.
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. PolicyCenter 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, PolicyCenter 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 191 in the *Integration Guide*.
getName	String	A human-readable name for the template.
getPassword	String	If present, this property holds the password that PolicyCenter 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 PolicyCenter requires for a user to use this template. PolicyCenter 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 PolicyCenter interface • **ui** – Indicates you must only use this template from the PolicyCenter 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, PolicyCenter 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`.
`getDefaultSecurityType`	`String`	Value corresponding to a type code from the `DocumentSecurityType` typelist. If you create a document using this template, PolicyCenter 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 DocumentProduction Class” on page 109 for an example of how template descriptor files display context objects.
`getContextObjectType`	`String`	Type of the specified context object. Possible values include the following: • `String` • `Text` • `Bean` – this is means any entity type • Name of any entity type, for example `Policy`.
`getContextObjectAllowsNull`	`Boolean`	`true` if `null` is a legal value, `false` otherwise.
`getContextObjectDisplayName`	`String`	Human-readable name for the given context object. PolicyCenter displays this name in the document creation interface.
`getContextObjectDefaultValueExpression`	`String`	Gosu expression that evaluates to the desired default value for the context object. PolicyCenter 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. PolicyCenter 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 DocumentProduction Class” on page 109 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 DocumentProduction Class

Guidewire Studio provides a helper `DocumentProduction` Gosu class (**Classes** → **gw** → **document** → **DocumentProduction**) with the following public methods to facilitate working with document creation:

Asynchronous methods	Synchronous methods
• `asynchronousDocumentCreationSupported`	• `synchronousDocumentCreationSupported`
• `createDocumentAsynchronously`	• `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 PolicyCenter 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

PolicyCenter provides the following methods that you can use to determine if the `IDocumentTemplate` plugin supports the required document creation mode:

- `asynchronousDocumentCreationSupported`
- `synchronousDocumentCreationSupported`

The asynchronousDocumentCreationSupported Method

Call the `asynchronousDocumentCreationSupported` method to determine whether the `IDocumentProduction` implementation supports asynchronous creation for the specified template. This method returns `true` if it is possible, and `false` otherwise. It also returns `false` if a template cannot be found with the specified name or if you pass in `null` for the template name.

This method has the following signature:

```
public static function asynchronousDocumentCreationSupported(template :
        IDocumentTemplateDescriptor) : boolean
```

The synchronousDocumentCreationSupported Method

Call the `synchronousDocumentCreationSupported` method to determine whether the `IDocumentProduction` implementation supports synchronous creation for the specified template. This method returns `true` if it is possible, and `false` otherwise. It also returns `false` if PolicyCenter cannot find a template with the specified name or if you pass in `null` for the template name.

This method has the following signature:

```
public static function synchronousDocumentCreationSupported(template : IDocumentTemplateDescriptor) :
        boolean
```

Asynchronous Document Creation Methods

Guidewire provides the following methods for asynchronous document creation:

- `createDocumentAsynchronously`

The createDocumentAsynchronously Method

Call the `createDocumentAsynchronously` method to create a new document asynchronously, based on the named template and the supplied parameters. (The method returns immediately, but the actual document creation takes place over an extended period of time.) It is the responsibility of the external document production system to create a `Document` entity (if desired) after document creation is complete.

The last method parameter (`fieldValues`) provides a set of field name to value mappings to set in the metadata of the new `Document` after you create it.

This method has the following signature:

```
public static function createDocumentAsynchronously(template : IDocumentTemplateDescriptor,
        parameters : Map, fieldValues : Map) : String
```

Note: The `IDocumentProduction` implementation is responsible for persisting both the document contents and creating a new `Document` entity if necessary.

Synchronous Document Creation Methods

Guidewire provides the following methods for synchronous document creation:

- `createAndStoreDocumentSynchronously`
- `createDocumentSynchronously`

The createAndStoreDocumentSynchronously Method

Call the `createAndStoreDocumentSynchronously` method to create and store a document without any further user interaction. This method creates a document synchronously and passes it to the `IDocumentContentSource` plugin for persistence.

This method has the following signature:

```
public static function createAndStoreDocumentSynchronously(template : IDocumentTemplateDescriptor,
        parameters : Map, document : Document)
```

The createDocumentSynchronously Method

Call the `createDocumentSynchronously` method to create a new document synchronously, based on the named template and the supplied parameters. PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 351 in the *Gosu Reference Guide*
- "Document Management" on page 191 in the *Integration Guide*
- "Data Extraction Integration" on page 607 in the *Integration Guide*

Document Template Localization

For information on localizing document templates, see:

- "Localizing Templates" on page 69 in the *Globalization Guide*
- "Document Localization Support" on page 74 in the *Globalization Guide*

IMPORTANT The base configuration *Sample Acrobat* document (`SampleAcrobat.pdf`) uses Helvetica font. If you intend to create a document that uses Unicode characters (for example, one that uses an East Asian language), then the document template must support a Unicode font. Otherwise, the document does not display Unicode characters correctly.

Document Creation Examples

Note: See also "Document Management" on page 191 in the *Integration Guide* for more information on working with documents in PolicyCenter.

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 PolicyCenter 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).

```
<?xml version="1.0" encoding="UTF-8"?>
<DocumentTemplateDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.guidewire.com/schema/claimcenter/document-template.xsd"

   id="FSAClaimAcknowledgement.gosu.htm"
   name="Claim Acknowledgement Letter"
   description="ICOB 7.5.1 acknowledgement letter."
   type="letter_sent"
   lob="GL"
   state="UK"
   mime-type="text/html"
   date-effective="Mar 15, 2004"
   date-expires="Mar 15, 2007"
   keywords="UK, acknowledgement">

  <ContextObject name="To" type="Contact">
    <DefaultObjectValue>Claim.Insured</DefaultObjectValue>
    <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
```

```
  </ContextObject>

  <ContextObject name="From" type="Contact">
    <DefaultObjectValue>Claim.AssignedUser.Contact</DefaultObjectValue>
    <PossibleObjectValues>Claim.getRelatedUserContacts()</PossibleObjectValues>
  </ContextObject>

  <FormFieldGroup name="main">
    <DisplayValues>
      <DateFormat>MMM dd, yyyy</DateFormat>
    </DisplayValues>
    <FormField name="ClaimNumber">Claim.ClaimNumber</FormField>
    <FormField name="ClaimReportedDate">Claim.ReportedDate</FormField>
    <FormField name="InsuredName">To.DisplayName</FormField>
    <FormField name="InsuredPrefix">(To as Person).Prefix</FormField>
    <FormField name="InsuredLastName">(To as Person).LastName</FormField>
    <FormField name="InsuredAddress1">To.PrimaryAddress.AddressLine1</FormField>
    <FormField name="InsuredCity">To.PrimaryAddress.City</FormField>
    <FormField name="InsuredState">To.PrimaryAddress.State</FormField>
    <FormField name="InsuredPostalCode">To.PrimaryAddress.PostalCode</FormField>
    <FormField name="CurrentDate">gw.api.util.DateUtil.currentDate()</FormField>
    <FormField name="ClaimNoticeDate">Claim.LossDate</FormField>
    <FormField name="AdjusterName">From.DisplayName</FormField>
    <FormField name="AdjusterPhoneNumber">From.WorkPhone</FormField>
    <FormField name="InsuranceCompanyName">Claim.Policy.UnderwritingCo</FormField>
    <FormField name="InsuranceCompanyAddress">From.PrimaryAddress.AddressLine1</FormField>
    <FormField name="InsuranceCompanyCity">From.PrimaryAddress.City</FormField>
    <FormField name="InsuranceCompanyState">From.PrimaryAddress.State</FormField>
    <FormField name="InsuranceCompanyZip">From.PrimaryAddress.PostalCode</FormField>
  </FormFieldGroup>

</DocumentTemplateDescriptor>
```

Method createAndStoreDocumentSynchronously Example 1

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

Parameter	Value	Description
`descriptor`	`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.
`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)
```

Method createAndStoreDocumentSynchronously Example 2

Gosu also provides the ability to test whether document creation is possible before attempting the operation. You can check for the ability to create a document synchronously or asynchronously by using one of the following methods.

- `synchronousDocumentCreationSupported`
- `asynchronousDocumentCreationSupported`

These methods take the name of a template descriptor as the lone argument and determine whether the `IDocumentProduction` plugin supports the specified template descriptor. Each method returns `true` if the plugin does support that template descriptor file and `false` otherwise. Each method also return `false` if it cannot find a template descriptor with the specified name.

The following sample (ClaimCenter) Gosu code illustrates the use of these methods:

```
//Creates a map object and sets document information (ContextObjects) needed for content creation
var parameters = new java.util.HashMap()
parameters.put("Claim", claim)
parameters.put("To", claim.maincontact)
parameters.put("From", claim.AssignedUser.Contact)
parameters.put("CC", null)

//Creates a new Document entity and sets metadata directly on the document object
var document : Document = new Document(claim)
document.Claim = claim
document.Name = "Claim Acknowledgement"
document.Type = "letter_sent"
document.Status = "draft"

//Creates the document template descriptor (of type IDocumentTemplateDescriptor)
var plugin = gw.plugin.Plugins.get(gw.plugin.document.IDocumentTemplateSource)
var descriptor = plugin.getDocumentTemplate("SampleAcrobat.pdf",
        \ code -> gw.i18n.ILocale.EN_US)

if (gw.document.DocumentProduction.synchronousDocumentCreationSupported(descriptor)) {
  gw.document.DocumentProduction.createAndStoreDocumentSynchronously(descriptor, parameters, document)
}
```

Troubleshooting

The following issues with document creation can occur on occasion:

- IDocumentContentSource.addDocument Called with Null InputStream
- IDocumentMetadataSource.saveDocument Called Twice
- UnsupportedOperationException Exception
- Document Template Descriptor Upgrade Errors
- “Automation server cannot create object” Error
- “IDocumentProduction implementation must return document...” Error

IDocumentContentSource.addDocument Called with Null InputStream

In certain cases, PolicyCenter 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, PolicyCenter uploads the document contents to the configured `IDocumentContentSource` before validation is run. Therefore, PolicyCenter 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, PolicyCenter 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 PolicyCenter 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, PolicyCenter 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, PolicyCenter saves the document—created by the rule to pass to `createDocumentAsynchronously`—as part of commit of the bundle used by the rule. PolicyCenter 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 PolicyCenter 2.0.0 and earlier.

UnsupportedOperationException Exception

Occasionally, the following exception occurs:

```
java.lang.UnsupportedOperationException: Asynchronous client-side MailMerge-based generation
        is not supported!
```

This exception occurs because the sample `IDocumentProduction` implementation does not support server-side creation of Microsoft Word or Excel documents. Therefore, you cannot create documents asynchronously based on templates of these types.

Document Template Descriptor Upgrade Errors

The following error can occur during document template descriptor upgrade:

```
IOException encountered: Content is not allowed in prolog.
```

The cause of this error, generally, is that one or more descriptor files has some characters before the `"<?xml"` at the beginning of the file. That XML tag must be the very first thing in the file.

“Automation server cannot create object” Error

Occasionally, the following (or similar) error occurs during attempts to create a document from a template.

```
"Automation server cannot create object"
```

Creation of Microsoft Word and Excel documents from templates is done on the user's machine. Therefore, if the user's machine does not have Word or Excel, this kind of error occurs. You can, however, still create a document from a PDF or Gosu template instead.

"IDocumentProduction implementation must return document..." Error

You may see the following error while calling a `DocumentProduction.create*` method:

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

This error message indicates a problem with the `DocumentContentsInfo.ResponseType` property on the return value of the document production plugin document creation code. The exact plugin method is the `IDocumentProduction` plugin method `createDocumentSynchronously`.

In the default configuration, the built-in implementation of this plugin expects the `responseType` field has the response type value `DocumentContentsInfo.DOCUMENT_CONTENTS`.

If you modify the document production plugin or any related code such that it returns another response type such as `URL`, PolicyCenter displays this error. The built-in `IDocumentProduction` implementation expected an actual document contents as an input stream of bytes, not other response types such as `DOCUMENT_URL`.

However, you might want to support a document production implementation that supports the response type `DOCUMENT_URL`. If so, you may need to modify the built-in document production plugin implementation. In Studio, edit the built-in Gosu class `gw.document.DocumentProduction`. There are multiple method signatures for the method `createAndStoreDocumentSynchronously`, all of which check the response type. You must modify the `DocumentProduction` class to appropriately handle the `DOCUMENT_URL` response type.

Your modifications depend on the meaning of the URL in your system:

- If the URL refers directly to the external Document Management System (DMS), the document is already stored. The document production plugin does not need to do anything directly with returned data. Remove the error checking line that requires the document type `DOCUMENT_CONTENTS`. Check if the value is `DOCUMENT_URL`, and if so skip the following lines that assume the result is an input stream.
- If the URL refers to a separate external resource other than the DMS, the document production plugin must immediately retrieve that external data. Next, submit the resulting input stream to the document content source. Base your code to submit the document on the existing `createAndStoreDocumentSynchronously` code that submits an input stream to the document content source.

If your URL has some other meaning than the choices listed above, contact Guidewire Customer Support.

Large Size Microsoft Word Documents

Occasionally, you see the same document template create Microsoft Word documents with different file sizes. The issue, in this case, is that some users have their individual Microsoft Word applications configured to save documents as `Word97-2003 6.0/95 - RTF.doc`. This causes Word to generate the file as Rich Text Format (RTF), but save the generated file with the `.doc` extension. In general, RTF documents have a much larger file size than the basic `.doc` Word files.

To set a default file format for saving new documents, do the following:

1. On the Word **Tools** menu, click **Options**, and then click the **Save** tab.
2. In the **Save Word files as** box, click the file format that you want.

This procedure can be different for each version of Microsoft Word.

> **Note:** Guidewire recommends that you check the Microsoft Word documentation specific to your version for exact instructions.