Guidewire PolicyCenter™

Rules Guide

Release 10.0.0

GUIDEWIRE
Adapt and succeed™

Product Name: Guidewire PolicyCenter
Product Release: 10.0.0
Document Name: Rules Guide
Document Revision: 26-September-2018

Contents

About PolicyCenter documentation 7
Conventions in this document 8
Support 8

Part 1
PolicyCenter rules

1 Rules: a background 13
Introduction to PolicyCenter rules 13
Guidewire business rules, Guidewire Gosu rules 13
Gosu rule hierarchy 13
Gosu rule execution 14
Gosu rule management 14
Gosu sample rules 15
Gosu rule terminology 15

2 Rules overview 17
Rules, rule sets, and rule set categories 17
Rule structure 17
Gosu rule syntax 18
Rule members 18
Understanding Gosu rule conditions 19
Understanding Gosu rule actions 20
Exiting a Gosu rule 20
How to use exitToNextRoot 21
Gosu annotations and PolicyCenter Gosu rules 21
The effect of annotations on Gosu rules 21
Determine the annotations on elements of Gosu code 21
Invoking a Gosu rule from gosu code 22

3 Using the rules editor 23
Working with rules 23
View or edit a rule 23
Changing rule order 23
Create a new rule set category 24
Create a new rule set 24
Create a New Rule 25
Accessing a rule set from Gosu code 25
Renaming or deleting a rule 25
Making a rule active or inactive 26
About changing the root entity of a rule 26
Change the root entity of a rule 27

4 Writing rules: testing and debugging 29
Generating rule debugging information 29
Printing debugging information 29
Generating logging information in Gosu rules 29
Printing the complete package path of a rule 31

5 Writing rules: examples . . . 33
Accessing fields on subtypes . . . 33
Taking action on more than one subitem . . . 33
Checking entity permissions . . . 34

6 PolicyCenter Rule Set Categories . . . 35
Assignment . . . 35
Audit . . . 36
Reporting Trend Analysis rule set . . . 36
Event Message rules . . . 36
Messaging events . . . 37
Message destinations and message events . . . 37
Message destinations and message plugins . . . 37
Generating messages . . . 38
Listening for object changes that trigger events . . . 38
Example Event Fired rule . . . 39
Example asynchronous document storage failed Event Fired rule . . . 39
Exception . . . 39
Activity Escalation rules . . . 40
Group Exception rules . . . 40
Policy Exception rules . . . 40
Premium Report Escalation rules . . . 43
User Exception rules . . . 43
Renewal . . . 44
Renewal AutoUpdate . . . 44
Validation . . . 44
Validation in PolicyCenter . . . 45
PolicyCenter validatable entities . . . 46
How an entity triggers Validation rules . . . 47
Triggering Validation rules on custom entities . . . 47
The validate method . . . 47
Account Validation rule example . . . 48

7 PolicyCenter rule reports . . . 49
About the Rule Repository report . . . 49
Generate a Rule Repository Report . . . 49
About the Profiler Rule Execution report . . . 49
Generate a Profiler Rule Execution Report . . . 50
Understanding the Profiler Rule Execution Report . . . 50
View rule information in the Profiler Chrono report . . . 51

Part 2
Advanced topics

8 Assignment in PolicyCenter . . . 55
Understanding assignment in Guidewire PolicyCenter . . . 55
Assignment Entity Persistence . . . 55
Assignment Queues . . . 55
Primary and secondary assignment . . . 56
Primary (user-based) assignment entities . . . 56
Secondary (role-based) assignment entities . . . 56
Assignment within the Assignment rule sets . . . 58
Assignment outside the Assignment rule sets . . . 58
Role assignment . . . 58
Role assignment and Submission Jobs . . . 59

Role assignment and non-Submission jobs . . . 59
Gosu support for assignment entities . . . 60
Assignment success or failure . . . 60
Logging assignment activity . . . 61
Assignment events . . . 61
Assignment method reference . . . 62
Queue assignment . . . 62
Immediate assignment . . . 63
Condition-based assignment . . . 63
Round-robin assignment . . . 66
Dynamic assignment . . . 66

9 Rule-based validation . . . 71
Overview of rule-based validation . . . 71
About the validation graph . . . 72
How PolicyCenter traverses the validation graph . . . 72
How PolicyCenter works with owned arrays . . . 73
Top-level entities that trigger full validation . . . 73
Validation trigger example . . . 74
Overriding validation triggers . . . 74
Validation performance issues . . . 75
View a text version of the validation graph . . . 76

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.
Database Upgrade Guide	Describes the overall PolicyCenter upgrade process, and describes how to upgrade your PolicyCenter database 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.
Configuration Upgrade Guide	Describes the overall PolicyCenter upgrade process, and describes how to upgrade your PolicyCenter configuration 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. The *Configuration Upgrade Guide* is published with the Upgrade Tools and is available from the Guidewire Community.
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 for PolicyCenter. The intended readers are all IT staff and configuration engineers.
PCF Reference Guide	Describes PolicyCenter PCF widgets and attributes. The intended readers are configuration engineers.
Data Dictionary	Describes the PolicyCenter data model, including configuration extensions. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
Security Dictionary	Describes all security permissions, roles, and the relationships among them. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are 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 Guidewire Studio for PolicyCenter. 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.

Document	Purpose
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.
Java API Reference	Javadoc-style reference of PolicyCenter Java plugin interfaces, entity fields, and other utility classes. The intended readers are system architects and integration programmers.
Gosu Reference Guide	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
Gosu API Reference	Javadoc-style reference of PolicyCenter Gosu classes and properties. The reference can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers, system architects, and integration programmers.
Glossary	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.
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	Indicates a term that is being defined, added emphasis, and book titles. In monospace text, italics indicate a variable to be replaced.	A *destination* sends messages to an external system. Navigate to the PolicyCenter installation directory by running the following command: `cd installDir`

bold	Highlights important sections of code in examples.	`for (i=0, i<someArray.length(), i++) {`
		`  newArray[i] = someArray[i].getName()`
		`}`

narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Click **Submit**.
`monospace`	Code examples, computer output, class and method names, URLs, parameter names, string literals, and other objects that might appear in programming code.	The `getName` method of the `IDoStuff` API returns the name of the object.
`monospace italic`	Variable placeholder text within code examples, command examples, file paths, and URLs.	Run the `startServer` *`server_name`* command. Navigate to `http://`*`server_name`*`/index.html`.

Support

For assistance, visit the Guidewire Community.

Guidewire customers

https://community.guidewire.com

Guidewire partners

https://partner.guidewire.com

part 1

PolicyCenter 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 PolicyCenter rules

In general, Guidewire strongly recommends that you develop and document the functional 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.

Guidewire business rules, Guidewire Gosu rules

Guidewire provides two different types of rules in the base PolicyCenter configuration.

PolicyCenter business rules	You use business rules to create, edit, and manage underwriting issues. You work with business rules in the **Business Rules** screens, in PolicyCenter**Administration**→**Business Settings**. The target users of business rules are PolicyCenter administrators and business analysts. It is possible to create and deploy PolicyCenter business rules without restarting the application server. You use business rules for defining underwriting issue types. In the underwriting rules you can: • Test for the existence of an underwriting issue on a policy • Define the course of action to take for an underwriting issue • Define whether to block the progress of a policy transaction until the underwriting issue receives approval.
Gosu rules	You create and manage Gosu rules entirely in PolicyCenter Studio. Gosu rules require in-depth domain knowledge and technical expertise to create. After you make changes to Gosu rules, you typically need to restart the application server.

See also

- For information about the PolicyCenter business rules, see the *Application Guide*.
- For basic overview information about the PolicyCenter Gosu rules, see "Rules: a background" on page 13 and "Rules overview" on page 17.

Gosu 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.

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 hierarchy specific to the rule set to which it belongs.

Gosu 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 following 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.)

Gosu rule management

Guidewire strongly recommends that you tightly control access to changing rules within your organization. Editing rules is a complicated process. 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.

Typically, several different kinds of people manage the PolicyCenter Gosu rules:.

Business analysts	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	One or more rule writers who are generally more technical than business analysts. 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 Gosu rules. These are rules that you can actually implement with the information available to PolicyCenter.

Gosu 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

- "PolicyCenter Rule Set Categories" on page 35

Gosu rule terminology

Guidewire uses the following terminology in discussing Gosu 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, Gosu 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, Guidewire also calls an object an *entity instance*. • An instance of an Gosu class • An instance of a 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}`
	The following example illustrates the logic of a validation rule:
	If the auto policy does not list a Vehicle Identification Number (VIN) for all covered automobiles, then mark the policy as invalid.

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 PolicyCenter can perform on a `PolicyPeriod` object.
policy	A Policy encapsulates all the information about a risk underwritten by an insurer. You can also think of a policy as a container of logical policy periods (with a specific range of effective time) for a policy.
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*.

Term	Definition
	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`. Although similar, a 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. You create and develop workflow within Guidewire Studio.

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.

Rules, rule sets, and rule set categories

In the base configuration, Guidewire provides a number of sample rules as examples of how to use rules to perform business logic. 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.

See also

- "Gosu sample rules" on page 15

Rule structure

Every Gosu rule has the following basic syntax:

`IF` {some conditions}

`THEN` {do some actions}

Each Gosu rule divides the rule functionality into different blocks of code, with a specific label. The following table summarizes each of these parts.

Rule block	Description

USES	A list of packages or classes that this rule uses. For example:
	`uses java.util.HashSet`
	`uses gw.lang.reflect.IType`

CONDITION	A Boolean expression (that is, a series of questions connected by AND and OR logic that evaluates to TRUE or FALSE). For example: (This is an auto claim) AND (the policy includes collision coverage) It is also possible to insert a statement list, instead of a simple expression. For example: `var o = new HashSet<IType>() {A, B, C, ...}`

Rule block	Description
	`return o.contains(typeof(...))` The `CONDITION` block must contain a `return` statement that returns a Boolean value.
ACTION	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 to conditions and actions that are appropriate for the type of your current rule.

Gosu rule syntax

Guidewire Gosu rules use a programming language called Gosu, which is a high-level language tailored for expressing Gosu 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 use a dot notation to reference fields, subobjects, and methods. 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 Gosu rules. For example, do not attempt to define a Gosu function in a Gosu 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.

See also

- See the *Gosu Reference Guide* for more information about how to write Gosu statements.

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

Rule conditions are 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 `ACTION` block. For example, you can use the following condition to test whether the contact information includes a home phone number:

```
contact.HomePhone == null
```

Rule actions

Rule actions are 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 in the `CONDITION` block evaluates

to `true`. For example, if a contact information is missing a home phone number, you can use the following code to raise an error in PolicyCenter:

```
contact.rejectField("HomePhoneCountry",
      ValidationLevel.TC_LOADSAVE, displaykey.Validator.Phone.Home.CountryCode.Null, null, null)
```

Rule APIs

Rule APIs are a collection of Gosu methods accessed through the `gw.api.*` package. They include many standard mathematic, date, string, and financial methods. For example, you can use the following code to determine if an activity (`act`) is more than 15 days old and thus needs additional handling:

```
gw.api.util.DateUtil.differenceInDays(act.CreateTime, gw.api.util.DateUtil.currentDate()) > 15
```

Rule entities

Rule entities are 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 condition to automatically raise evaluation issues for the specified location.

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

Understanding Gosu 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 reference 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 reference 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 using standard Boolean logic operators (`and`, `or`, `not`). For example:

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

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 as `false`. This can happen inadvertently, especially if you create a condition statement with multiple conditions to evaluate. If your condition evaluates to `null` (`false`), 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 `CONDITION` block. Every `CONDITION` block must contain a `return` statement that evaluates to a Boolean value of `true` or `false`. For example:

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

See also

- See the the *Gosu Reference Guide* for details on operator precedence and other syntactical information.

Understanding Gosu rule actions

Within the rule `ACTION` block, you create the outcome for the criteria identified in the rule `CONDITION` block. 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.

For example, suppose that you want to verify that the country field exists on a contact's home phone number and is valid. The following syntax creates this action:

```
if (contact.HomePhoneCountry == null) {
  contact.rejectField("HomePhoneCountry", ValidationLevel.TC_LOADSAVE, displaykey.Validator.Phone.Home
        .CountryCode.Null, null, null)
}

if (contact.HomePhoneCountry == PhoneCountryCode.TC_UNPARSEABLE) {
    contact.rejectField("HomePhoneCountry", ValidationLevel.TC_LOADSAVE, displaykey.Validator.Phone.Home
          .CountryCode.Unparseable, null, null)
}
```

See also

- See "Validation" on page 44 for a discussion of the various `reject` methods.

Exiting a Gosu rule

At some point in the rule decision tree, PolicyCenter makes the decision that you want. At this point, it is important that PolicyCenter not continue to execute the remaining rules in the rule set. Indeed, if rule checking did continue, and if PolicyCenter processed the rule set 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:

Exit – 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 of an exit action. PolicyCenter stops processing the rule set as soon as it encounters this action.
`actions.exitAfter`	This exit action causes PolicyCenter to stop processing the rule set after processing any child rules.
`actions.exitToNext`	This exit 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. You use this exit action only rarely, within very complicated action sections.
`actions.exitToNextParent`	This exit 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 exit 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, a rule lists an exit action as the last action in the rule so that it occurs only after the rule executes all other defined actions.

See also

- See "How to use exitToNextRoot" on page 21 for more information.

How to use 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 PolicyCenter makes the first decision (setting forms for that policy line), the decision logic is complete. However, PolicyCenter needs to move on to the next major block of decision-making, which is deciding if the giving state requires further forms. After PolicyCenter sets the segment, the rule set is finally complete, so the rule can just use the simple exit method.

Gosu annotations and PolicyCenter Gosu rules

Guidewire PolicyCenter uses annotation syntax to add metadata to Gosu classes, constructors, methods, and properties. For example, you can add an annotation to a method to indicate its return type or what exceptions it might throw.

The variation that annotations cause in the visibility of classes, constructors, methods, and properties can make it seem that your Gosu code is incorrect. The location of your Gosu code might be the issue, not the code itself.

See also

- For information on working with and creating annotations, see the *Gosu Reference Guide*.

The effect of annotations on Gosu rules

Guidewire marks certain Gosu code in the base application with the `@scriptable-ui` annotation. That annotation restricts the usage, or *visibility*, of the code to non-rules Gosu code. The converse is the `@scriptable-all` annotation, which makes a class, constructor, method, or property visible in Gosu code everywhere.

Within the Gosu Rules editor, 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. The Rules editor does not recognize the property descriptor for that property. However, the Gosu compiler does recognize the property in other editors in Studio, such as the Gosu editor for classes and enhancements.

Determine the annotations on elements of Gosu code

About this task

To determine the annotations on an element of Gosu code, such as a class, constructor, method, or property, do the following:

Procedure

1. Place your cursor at the beginning of the line directly above the affected code.
2. Type an `@` sign.
   Studio displays a list of valid annotations.

Invoking a Gosu rule from gosu code

It is possible to invoke Gosu 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 the use of the `invoke` method on a rule set triggers 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` block expression evaluates to `true`, then PolicyCenter executes the Gosu code in the `ACTION` block for that rule.

See also

- "Using the rules editor" on page 23
- "Working with rules" on page 23

chapter 3

Using the rules editor

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

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.

View or edit a rule

Procedure

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.

Changing rule order

If you want to change the order of your Gosu rules, you can drag and drop rules within the rule hierarchy in Guidewire Studio. 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 you can use 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.

Action	Description
Move a rule	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.
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**.

Action	Description
Move a rule to a different level	Select the rule to move and 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.

Create a new rule set category

Procedure

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.

Result

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

Create a new rule set

Procedure

1. In the Studio **Project** window, expand **configuration→config→Rule Sets**.

   Although the label is **Rule Sets**, the primary children of this folder are actually rule set categories.
2. Do one of the following:
   - If an appropriate rule set category for your rule set does not exist, first perform the steps listed in "Create a new rule set category" on page 24. After creating a rule set category, move to the next step.
   - If an appropriate rule set category for your rule set does exist, move to the next step.
3. Select a rule set category, then select **New→Rule Set** from the context menu.
4. Enter the following information:

Field	Description
Name	Studio displays the rule name in the middle pane. 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.

Create a New Rule

Procedure

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

Result

Studio creates the new rule as the last child rule of the currently selected rule (or rule set).

Accessing 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 commands to rename a rule or to delete it entirely from the rule set. You 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**. PolicyCenter does not require you to confirm your decision before deleting the rule. You use the **Delete** command to delete rules only.

About 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 in the installation directory:

```
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 in synchronization. Always use Guidewire Studio to rename a rule. Never rename a file using your local file system.

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.

About 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 Validation rule set category, you have Contact Validation rules and Person Validation rules. These rule set names indicate that the root entity for each rule set is—respectively—the `Contact` object and the `Person` object.

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. 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 any 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 /*start00rule*/true/*end00rule*/
     }
     ...
   }
   ```

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

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

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

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

6. You update the database by stopping and restarting the application server.
7. You stop and restart Studio.

As Studio reopens, it presents you with an error message dialog. The message states that Studio can 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.

See also

- "Change the root entity of a rule" on page 27

Change the root entity of a rule

About this task

The intent of the Rules editor **Change Root Entity** menu command is to enable you to edit a rule that you otherwise cannot open in Studio. This can happen, for example, if PolicyCenter is unable to parse the rule declarations. Do not use this command in any other circumstances.

Procedure

1. Select a rule within a rule set.
2. Right-click the rule name 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.

Result

After you complete this command:

- Studio performs a check to verify that the provided name 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 is 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.

See also

- "About changing the root entity of a rule" on page 26

chapter 4

Writing rules: testing and debugging

This topic describes ways to test and debug your rules.

See also

- “Generate a Rule Repository Report” on page 49
- “Generate a Profiler Rule Execution Report” on page 50
- “View rule information in the Profiler Chrono report” on page 51
- *System Administration Guide*

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 class `gw.api.system.PCLoggerCategory` 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 Gosu `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("RULE EXECUTION: " + actions.getRuleSet().DisplayName)
```

Generating logging information in Gosu rules

To trigger the flow of information from a Gosu rule to a log file, you need to configure the following:

- The appender definition (`<RollingFile>`) for the rule log file in `log4j.xml`.
- The logger definition (`<Logger>`) associated with the rule file in `log4j.xml`.
- The activation trigger for the logging category in a Gosu rule using `gw.api.system.PCLoggingCategory`.

Logging category definitions

A logging category defines how the Apache log4j logging utility handles different types of logging requests. In the base configuration, Guidewire provides a number of readily usable logging categories.

To define how PolicyCenter interacts with the ASSIGNMENT category, add something similar to the following code in file `log4j2.xml`.

```
<!-- Rolling file definition... -->
<RollingFile name= "RuleEngineLog" fileName="${guidewire.logDirectory}/ruleengine.log"
      filePattern="${guidewire.logDirectory}/ruleengine.log%d{.yyyy-MM-dd}">
 <PatternLayout pattern="${file.defaultPattern}" charset="UTF-8"/>
 <TimeBasedTriggeringPolicy/>
</RollingFile>

<!-- Logger definition... -->
<Logger name="Assignment" additivity="false" level= "info">
  <AppenderRef ref="RuleEngineLog">
</Logger>
```

Notice that this code does the following:

- It defines a rolling log file named `ruleengine.log` in the `<RollingFile>` element.
- It then links the `ruleengine.log` file to the `Assignment` logger category in the `<Logger>` element.
- It sets a default logging level of INFO.

Base configuration logging categories

There are several ways to view the base configuration logging categories:

- From the **Set Log Level** Server Tools screen.
- By running the `system_tools` command from a command prompt and adding the `-loggercats` option.

Triggering a logging category from a Gosu rule

For example, to use this API in Gosu code to perform assignment logging, do something similar to the following:

```
uses gw.api.system.PCLoggerCategory
...
var logger = PCLoggerCategory.ASSIGNMENT
...
logger.info("Print out this message.")
```

Logging example

The following code is an example of the use of a logging category in a Gosu rule. This code assumes that the necessary appender and logger definitions exist in file `log4j.xml`.

```
var logger=gw.api.system.PCLoggerCategory.ContextualLogger

// If there is an address, assign by location using that address
if (addr != null) {

  if ( policy.CurrentRoleAssignment.assignGroupByLocation("branch", addr, false,
      assignment.CurrentAssignment.AssignedGroup) ) {

    // Then attempt to assign to the proper group within this branch
    if ( policy.CurrentRoleAssignment.assignGroupByRoundRobin("branchuw", true,
          assignment.CurrentAssignment.AssignedGroup) ) {
      logger.debug("##### This is the Global Pre-renewal Assignment rule " + actions.getRule().DisplayName)
      logger.debug("Assigned Group is " + policy.CurrentRoleAssignment.AssignedGroup.DisplayName)
      actions.exit()
    }

  }

}
```

See also

- *System Administration Guide*

Printing the complete package path of a rule

Use the following Gosu code determine the full package path for an executing rule:

```
gw.rules.ExecutionSession.getCurrent().RunningRuleType.Name
```

For example, you can place the code in a `print` statement in the rule:

```
print(gw.rules.ExecutionSession.getCurrent().RunningRuleType.Name)
```

Placing the example `print` statement in a Preupdate Activity rule generates something similar to the following text in the console log:

```
rules.Preupdate.ActivityPreupdate_dir.ACT_1000345
```

chapter 5

Writing rules: examples

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

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 to particular subtypes only, you cannot reference them in rules by using the primary root object. For example, the following illustrates an invalid way to reference the primary contact for an account:

```
Company.PrimaryContact.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:

```
(Company.PrimaryContact as Person).LastName.compareTo( "Smith" )
```

Taking action on more than one subitem

Gosu 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 (line in policyPeriod.Lines) {
  for (cov in line.AllCoverages) {
    if (cov.CoverageCategory == "BAPHiredGrp") {
      print( "PolicyLine \"" + line + "\" has BAPHiredGroup coverage" )
    }
  }
}
```

Notice that the curly braces ({}) mark the beginning and end of a block of Gosu statements:

- The outer pair of braces contain statements to perform "for" each line on the policy.
- The outer pair of braces contain statements to perform "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 also

- For information on the syntax to use to construct `for` and `if` statements, see the *Gosu Reference Guide*.

Checking entity 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 = u.Roles.hasMatch(\role -> role.Role.Privileges.hasMatch(\perm -> perm.Permission == p))
```

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

PolicyCenter Rule Set Categories

As part of the PolicyCenter base configuration, Guidewire provides sample rules in the following rule set categories.

Category	Contains rules to…
Assignment	Determine the responsible party for an activity, for example.
Audit	Generate activities related to audits.
Event Message	Handle communication with integrated external applications.
Exception	Specify an action to take if an Activity or Job is overdue and enters the escalated state.
Renewal	Govern behavior at decision points in the Renewal job flow, for example, accepting or rejecting a renewal request.
Validation	Check for missing information or invalid data on non-policy-related (platform) objects.

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 assignment methods

Guidewire divides the assignment rules into two categories:

- Global assignment rules
- Default assignment rules

In the base configuration, there is both a global and default assignment rule for each assignable entity.

See also

- For information on global and default assignment rules and assignment in general, see "Assignment in PolicyCenter" on page 55.
- For a description of the available assignment methods, see "Assignment method reference" on page 62.

Audit

PolicyCenter supports two subtypes of audits:

- **Final audit** – 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** – A series of non-overlapping periodic audits that you schedule and bill within the coverage period.

See the *Application Guide* for information related to audits in PolicyCenter.

Reporting Trend Analysis rule set

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 the *Application Guide*.

Event Message rules

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 the occurrence of events

PolicyCenter calls the Event Fired rules if an entity involved in a bundle commit triggers an event of interest to a message destination. A message destination registers an event to indicate its interest in that event.

As part of the event processing:

- PolicyCenter runs the rules in the Event Fired rule set a single time for every event of interest to a message destination.
- PolicyCenter runs the Event Fired rule set a single time 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, one time for each destination interested in that event.

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

- *Integration Guide*
- *Configuration Guide*

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)
```

IMPORTANT Do not create custom code that generates update events for any entity type that can possibly be part of the archive bundle.

See also

- *Configuration Guide*
- *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 interested in the event of the occurrence of the event. You can 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.

Note: You must register your plugin in two different editors in Studio, first in the Plugin Registry, and then in the Messaging editor.

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 = messageContext.Root as SubmissionGroup
var subs = 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)
```

Listening for object changes that trigger events

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.)

```
USES
uses gw.api.util.ArrayUtil

CONDITION
//DestID 68 is the Console Message Logger
return messageContext.DestID == 68 and messageContext.EventName == "ContactChanged"

ACTION
var message = messageContext.createMessage( "Event: " + messageContext.EventName )
var contact = messageContext.Root as Contact
var fields = contact.PrimaryAddress.ChangedFields
print( ArrayUtil.toString( fields ) )
```

Example Event Fired rule

In the following example, the rule constructs a message containing the account number any time that the application adds an account.

```
CONDITION (messageContext : entity.MessageContext):
// Only fire if adding account
return messageContext.EventName == "AccountAdded"

ACTION (messageContext : entity.MessageContext, actions : gw.Rules.Action):
// 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")
```

Example asynchronous document storage failed Event Fired rule

Guidewire recommends that, in working with asynchronous document storage, that you create an Event Fired rule to handle the failure of the database to store a document properly. If a document fails to store properly, PolicyCenter creates a `FailedDocumentStore` event on `Document`.

To be useful, you need to add the `FailedDocumentStore` event to a messaging destination in `messaging-config.xml`. For example, you can add this event to message destination 324, which already tracks the `DocumentStore` event.

You then need to create an event rule to process the event as you see fit. You can, for example:

- Email a notification message to the creator of the `Document` entity.
- Create an activity on the primary or root object.
- Attempt to correct the document issues and create a new `DocumentStore` message.

The following example rule illustrates a possible Event Fired rule to handle the failure of the database to store a document.

```
CONDITION (messageContext : entity.MessageContext):
return messageContext.DestID == 324 and messageContext.EventName == "FailedDocumentStore"

ACTION (messageContext : entity.MessageContext, actions : gw.Rules.Action):
var document = messageContext.Root as Document
var fixedIssue = false
var sendEmail = false

if (fixedIssue) {
  messageContext.createMessage( "DocumentStore" )
}
else if (document.CreateUser.Contact.EmailAddress1 != null && sendEmail) {
  gw.api.email.EmailUtil.sendEmailWithBody(document,
  document.CreateUser.Contact.EmailAddress1,
  document.CreateUser.Contact.getDisplayName(),
  null, null, "Failed Document Store",
  //CREATE MESSAGE BODY HERE )
}
```

See also

- *Configuration Guide*

Exception

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

- PolicyCenter runs *escalation* rules on entities that are 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. You can configure whether and how often PolicyCenter runs these rule sets. See the *System Administration Guide* for information on exception batch processes.

Activity Escalation rules

PolicyCenter associates two specific dates with an activity.

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 batch process runs against all open activities in PolicyCenter to find activities to escalate, using the following criteria:

- The activity has an escalation date that is non-null.
- The escalation date is in the past.
- PolicyCenter has not yet escalated the activity.

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 group supervisor if the escalated activity priority is set to urgent.

```
CONDITION (activity : entity.Activity):
return activity.Priority == Priority.TC_URGENT

ACTION (activity : entity.Activity, actions : gw.Rules.Action):
activity.assign( activity.AssignedGroup, activity.AssignedGroup.Supervisor )
```

See also

- *System Administration Guide*

Group Exception rules

PolicyCenter runs the Group Exception rules on a scheduled basis to look for certain conditions on groups that might require further attention. You can use these Gosu rules to define the follow-up actions for each exception found. This rule set is empty in the base configuration. To work with this rule set, navigate in the Studio **Project** window to **configuration→config→Rule Sets→Exception→GroupExceptionRules**.

In the base configuration, Guidewire schedules the Group Exception work queue to run the group exception rules on all groups in PolicyCenter every day at 4:00 a.m. However, Guidewire disables the Group Exception work queue by default. You must enable the work queue by setting the number of workers for this work queue to a number greater than zero in file `work-queue.xml`.

Policy Exception rules

In the base configuration, Guidewire disables the Policy Exception rules by disabling the batch processes that run these Policy Exception rules.

If enabled, the rules looks for certain conditions on `PolicyPeriod` entities that possibly require further attention. You can then define follow-up actions for each exception found. First, PolicyCenter identifies `PolicyPeriod` entities that changed since the last inspection, or, that it has not inspected for a long time. PolicyCenter then runs these rules on each `PolicyPeriod` chosen.

This rule set is empty by default in the base configuration. 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 and Policy Exception rules

The Policy Exception rules perform no implicit validation. Any validation that you want to perform you must execute 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 to a `PolicyPeriod` entity is to raise an alert in the form of a Note, Activity, or email, do not validate.

Policy Exception rules and 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.

Data model and the PolicyPeriod entity

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

- A non-nullable foreign key to `PolicyPeriod`
- A datetime field named `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.

How PolicyCenter updates ExCheckTime

Any time a batch process runs 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 a `PolicyException` entity if it does not find one.
3. It sets `PolicyException.ExCheckTime` to the current system clock time.
4. It saves (commits) the changes to the database.

How PolicyCenter schedules the Policy Exception rules

Three separate batch processes control the time at which PolicyCenter 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—only those that are the latest in model time
`ClosedPolicyException`	Runs on all `PolicyPeriod` entities in a *closed* status—Withdrawn, Declined, and Not Taken, 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.

In the base configuration, Guidewire disables these batch processes by default. To enable a batch process, remove the comments around its entry in `scheduler-config.xml`. If enabled, PolicyCenter runs the default entries at the following times:

- `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 limit how often PolicyCenter runs the exception rules against a single `PolicyPeriod` entity. Use the following parameters:

Batch process	Use parameter...	Default (days)
`OpenPolicyException`	`OpenPolicyThresholdDays`	1
`BoundPolicyException`	`BoundPolicyThresholdDays`	14
`ClosedPolicyException`	`ClosedPolicyThresholdDays`	14

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` process 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 following table summarizes this information:

Process	Default Schedule	Criteria
`OpenPolicyException`	Every night at 2 a.m.	`PolicyPeriod` entities that are unlocked and that 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.

See also

- *System Administration Guide*

Errors generated by the Policy Exception rules

Generally, if any of the exception rules throw 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 the process of editing a `PolicyPeriod` entity at the same time that the exception rules try to make changes to 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.

Minimizing performance issues with the Policy Exception rules

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

- 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).
- Only enable an exception process (`OpenPolicyException`, `ClosedPolicyException`, or `BoundPolicyException`) in `scheduler-config.xml` if you have Policy Exception rules for that particular policy exception type. For example, if you have bound policy exception rules, then only enable the `BoundPolicyException` process.
- Schedule the processes to run as rarely as possible, and use the highest values possible for a *`XX`*`PolicyThresholdDays` configuration parameter.

Policy Exception rule 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.

Premium Report Escalation rules

In the base configuration, PolicyCenter runs the Premium Report Exception rules at 4:30 a.m. every morning. This rule set contains one default rule in the base configuration. The rule generates an activity and attempts assign the activity to the policy underwriter if the premium report is overdue. If the assignment attempt fails, the rule assigns the activity to the Default User.

See also

- *System Administration Guide*

User Exception rules

In the base configuration, `UserException` batch processing runs the User Exception rule sets on all users within PolicyCenter at 3:00 a.m. every morning.

User Exception batch processing looks for certain conditions on users that possibly require further attention. You can then use these rules to define follow-up actions for each exception found. This rule set is empty by default in the base configuration.

First, PolicyCenter identifies users that have changed since the last inspection, or, that it has not inspected for a certain period of time. PolicyCenter then runs User Exception rules on each chosen user.

See also

- *System Administration Guide*

Renewal

The Renewal rules implement business logic around policy renewal. A policy renewal takes place before the policy period ends, on an active policy. A policy renewal continues the insurance for another time period. It can include changes to the following:

- Coverage
- Form update
- Industry or class code
- Underwriting company

It is not possible to use policy renewal to reinstate a canceled policy.)

Renewal AutoUpdate

The Renewal AutoUpdate rule set triggers 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.

```
CONDITION (policyPeriod : entity.PolicyPeriod):
return policyPeriod.BOPLineExists

ACTION (policyPeriod : entity.PolicyPeriod, actions : gw.Rules.Action):
for (bld in policyPeriod.BOPLine.BOPLocations*.Buildings) {
  if (bld.BOPBuildingCov != null) {
    var increase = bld.BOPBuildingCov.BOPBldgAnnualIncreaseTerm.Value
    if (increase != 0) {
      bld.BOPBuildingCov.BOPBldgLimTerm.Value =
            bld.BOPBuildingCov.BOPBldgLimTerm.Value * (1 + (increase/100))
      if (bld.BOPPersonalPropCov != null) {
        bld.BOPPersonalPropCov.BOPBPPBldgLimTerm.Value =
              bld.BOPPersonalPropCov.BOPBPPBldgLimTerm.Value * (1 + (increase/100))
      }
    }
  }
}
```

Validation

PolicyCenter performs rule-based validation as it commits a data bundle containing a validatable entity to the database. Rules-based validation works with non-policy objects only. To validate policy-related objects, use class-based validation.

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

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.)

See also

- “Rule-based validation” on page 71
- *Configuration Guide*

Validation in PolicyCenter

Within PolicyCenter, use the validation rules to identify problems with data that the user must fix before PolicyCenter permits 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) 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. The user cannot ignore or clear an error. 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. Issues or problems that the rules identify 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.

Method parameter	Description
`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. Thus, during validation, it is possible that some traversal of the objects to be validated can reset that entity's previous result. If this traversal is before the rule engine executes the entity's own rules, PolicyCenter would lose this particular reject.

Constructing Error Text Strings

Guidewire recommends that you use the `DisplayKey` class to return the value to use for the `strErrorReason` and `strWarningReason`. For example, the following code returns the message, "User has duplicate roles".

```
USES:
uses gw.api.locale.DisplayKey

CONDITION (...):
...

ACTION (...):
DisplayKey.get("Java.Admin.User.DuplicateRoleError")
```

This also works with display keys that require a parameter or parameters. For example, `display_`*`LanguageCode`*`.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}
```

Code `DisplayKey.get("Java.UserDetail.Delete.IsSupervisorError", `*`GroupName`*`)` then returns the message (with the rule retrieving the value of *`GroupName`* already):

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

PolicyCenter validatable entities

PolicyCenter defines the following non-policy objects as validatable entities that trigger rules-based validation:

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

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

How an entity triggers Validation rules

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

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, in the base configuration, Guidewire defines 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.

Triggering Validation rules on custom entities

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

For an entity to trigger a validation rule:

1. The entity must implement the `Validatable` delegate. This entity must be a root entity, not a subtype of an entity. In short, any extension entity that you create that you want to trigger a validation rule must implement the following code:

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

2. A validation rule must exist in the *Validation* rule set category that conforms to the following naming convention:

   ```
   <custom_entity_name>ValidationRules
   ```

To summarize, a validation rule must exist in a validation rule set that reflects the custom entity's name and ends with `ValdiationRules`. In other words, you must create a rule set with the same name as the extension entity and add the word `ValidationRules` to it as a suffix. You then place your rules in this rule set.

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

```
NewEntityExtValidationRules
```

See also

- For information on how to create an entity that implements the `Validatable` delegate, see the *Configuration Guide*.
- For information on creating new rules sets, see "Working with rules" on page 23. See especially the section entitled "To create a new rule set".

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 : entity.Account):
return account.AccountHolderContact.WorkPhone ==  null

ACTION (account : entity.Account, actions : gw.Rules.Action):
account.reject( TC_LOADSAVE, "You must provide a work phone number for the primary contact.", null, null)
```

chapter 7

PolicyCenter rule reports

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

See also

- "Generating rule debugging information" on page 29
- *System Administration Guide*

About the Rule Repository report

To facilitate working with the Gosu rules, PolicyCenter provides a command prompt tool to generate a report describing all the existing 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

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.

Generate a Rule Repository Report

Procedure

1. Open a command prompt and navigate to the PolicyCenter installation directory.
2. Execute the following command:

```
gwb genRuleReport
```

Result

This command generates the following files:

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

About the Profiler Rule Execution report

The Guidewire Profiler provides information about the runtime performance of specific application code. It can also generate a report listing the 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.

See also

- "Understanding the Profiler Rule Execution Report" on page 50
- "Generate a Profiler Rule Execution Report" on page 50
- "Understanding the Profiler Rule Execution Report" on page 50
- *System Administration Guide*

Generate a Profiler Rule Execution Report

Procedure

1. Log into Guidewire PolicyCenter using an administrative account.
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 only.
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**.

Understanding the Profiler Rule Execution Report

After you enable the Web profiler and perform activity within PolicyCenter, the profiler **Profiler Analysis** screen displays zero, one, or multiple stack traces. The following list describes the meaning these 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 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**

Within the stack trace, it is possible to:

- Expand a stack trace completely by clicking **Expand All**.
- Collapse a stack trace completely by clicking **Collapse All**.
- Partially expand or collapse a stack trace by clicking the **+** (plus) or **–** (minus) next a stack node.

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

View rule information in the Profiler Chrono report

About this task

It is possible to use the Guidewire Profiler **Chrono** report to view frames filtered out by the **Rule Execution** report.

Procedure

1. Log into Guidewire PolicyCenter using an administrative account.
2. Access the Server Tools and click **Guidewire Profiler** on the menu at the left-hand side of the screen.
3. Generate a rule execution report.
4. Make a note of the following information from the rule exception report:
   - Name of the session
   - Name of the stack
   - Time offset
5. Select **Guidewire Profiler→Profiler Analysis→Web**.
6. Under **Profiler Result**, select **Chrono** from the **View Type** drop-down list.
7. Select the desired session, for example, **2016/05/05 09:23:25 web profiler session**.
8. Select the desired stack, for example, **DesktopActivities -> AssignActivity**.
9. As you select a stack, the **Profiler** page shows additional information about that stack at the bottom of the page.
10. Expand the **Time** node as needed to select the time offset that is of interest to you.

See also

- "Generate a Profiler Rule Execution Report" on page 50
- *System Administration Guide*

part 2

Advanced topics

chapter 8

Assignment in PolicyCenter

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

Understanding assignment in Guidewire PolicyCenter

At its core, the concept of assignment in Guidewire PolicyCenter is basically equivalent to ownership. The user to whom you assign an activity is the user who owns that activity, and who, therefore, has primary responsibility for it. Ownership of a `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 usually made to a user and a group, as a pair (group, user). However, the assignment of a user is independent of the group. In other words, you can assign an entity to a user who is not a member of the specified group.

IMPORTANT In the base configuration, PolicyCenter provides assignment methods that take only a user and that assign the entity to the default group for that user. Guidewire deprecates these types of assignment methods. Do not use them. Guidewire recommends that you rework any existing assignment methods that only take a user into assignment methods that take both a user and a group.

See also

- "Primary and secondary assignment" on page 56

Assignment Entity Persistence

PolicyCenter persists an assignment anytime that you persist the entity being assigned. Therefore, if you invoke the assignment methods on an entity from Gosu code, you must persist that entity. For example, you can persist an entity as part of a page commit in the PolicyCenter interface.

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.

Guidewire supports assigning `Activity` entities only to queues. Guidewire deprecates the `assignToQueue` methods on `Assignable` in favor of the `assignActivityToQueue` method. This makes the restriction more clear.

If you assign an activity to a queue, you cannot simultaneously assign it to a user as well.

See also

- “Queue assignment” on page 62

Primary and secondary assignment

Guidewire PolicyCenter distinguishes between two different types of assignment:

Assignment	Description
Primary	Also known as user-based assignment, primary assignment assigns an entity to a single owner only. The entity must implement either the `Assignable` delegate or the `PCAssignable` delegate, or both.
Secondary	Also known as role-based assignment, secondary assignment assigns an object 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. The entity must implement the `RoleAssignments` array.

In the PolicyCenter base configuration, only the `Activity` and `UserRoleAssignment` entities implement primary assignment. All other assignable entities use secondary assignment.

See also

- “Primary (user-based) assignment entities” on page 56
- “Secondary (role-based) assignment entities” on page 56

Primary (user-based) assignment entities

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

To be available for user-based or primary assignment, an entity must implement one or both of the following delegates:

- `Assignable` delegate
- `CPCAssignable` delegate

In the PolicyCenter base configuration, the following objects implement the required delegates:

- `Activity`
- `UserRoleAssignment`

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 entities

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. As 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.)

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 and 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 and 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.

Secondary assignment and round-robin assignment

Secondary assignment uses the assignment owner to retrieve the round-robin state. What this means is that different secondary assignments on the same assignment owner can end up using the same round-robin state, and they can affect each other.

In general, if you use different search criteria for different secondary assignments, you do not encounter this problem as the search criteria is most likely different. However, if you want to want to make absolutely sure that different secondary assignments follow different round-robin states, then you need to extend the search criteria. In

this case, add a flag column and set it to a different value for each different kind of secondary assignment. See “Round-robin assignment” on page 66 for more information on this type of assignment.

Assignment within the Assignment rule sets

All assignment rules in the **Assignment** folder use the Assignment engine. In working within the context of the assignment rules, use the following assignment properties:

Assignment type	Property
Primary	`entity.CurrentAssignment` This property returns the current assignment, regardless of whether you attempt to perform primary or secondary assignment. If you attempt to use it for secondary assignment, then the property returns the same object as the `CurrentRoleAssignment` property.
Secondary	`entity.CurrentRoleAssignment` This property returns `null` if you attempt to use it for primary assignment.

Secondary assignment and the Assignment rules

`UserRoleAssignment` entities share the rule set of their primary entities. Thus, PolicyCenter assigns the `UserRoleAssignment` entities for an activity using the Activity Assignment rule sets, and so on. It is for this reason that assignment statements in the rules always use the `CurrentAssignment` formulation, such as the following:

```
Activity.CurrentAssignment.assignUserAndDefaultGroup( user )
```

It is important for rules to use this syntax, because it allows PolicyCenter to use the rules (in this case) for both activities and activity-related `UserRoleAssignment` entities.

See also

- “Primary and secondary assignment” on page 56
- “Assignment outside the Assignment rule sets” on page 58

Assignment outside the Assignment rule sets

It is possible to assign an entity outside of the assignment rules, which use the PolicyCenter Assignment engine. For example, you can assign an entity in arbitrary Gosu code, outside of the assignment rules.

If you assign an entity outside the Assignment engine, then you do not need to use `entity.currentAssignment` to retrieve the current assignment. 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)
```

See also

- “Assignment within the Assignment rule sets” on page 58

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.

Role assignment and Submission Jobs

Assigning submission jobs entails involves the following:

- Setting the producer for the submission job
- Setting the underwriter for the submission job

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.)

How PolicyCenter sets the producer during assignment

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:
   a. Check if the `CurrentUser` is an Producer for that `ProducerCode`. If so, preferentially choose that `CurrentUser`.
   b. 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 the `superuser` user and log a warning that the method performed a default assignment only.

How PolicyCenter sets the underwriter during assignment

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`.
   a. Check if the `CurrentUser` is an underwriter for that branch. If so, preferentially choose that `CurrentUser`.
   b. 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 the `superuser` user and log a warning that the method performed a default assignment only.

Role assignment and 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
- Policy Change
- Reinstatement
- Renewal
- Rewrite

Audit Job Assignment

An Audit job 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 Job Assignment

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 **Related To** 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`

Activity Job 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

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`.

Assigning activities 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 anytime 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

Anytime that the assignment status changes on an assignment, PolicyCenter creates an assignment event. The following events can trigger an assignment change event:

- `AssignmentAdded`
- `AssignmentChanged`
- `AssignmentRemoved`

The following list describes these events.

Old status	New status	Event	Code
Unassigned	Unassigned	None	None
Unassigned	Assigned	`AssignmentAdded`	`Assignable.ASSIGNMENTADDED_EVENT`

Old status	New status	Event	Code
Assigned	Assigned	`AssignmentChanged`	`Assignable.ASSIGNMENTCHANGED_EVENT`
Assigned	Unassigned	`AssignmentRemoved`	`Assignable.ASSIGNMENTREMOVED_EVENT`

Note: A change can trigger the `AssignmentChanged` event for any number of reasons. It can happen, for example, if a field such as the assigned user, the assigned group, or the date changes.

Assignment method reference

Guidewire divides the assignment methods into the general categories described in the following topics:

- “Queue assignment” on page 62
- “Immediate assignment” on page 63
- “Condition-based assignment” on page 63
- “Round-robin assignment” on page 66
- “Dynamic assignment” on page 66

Note: For the latest information and description on assignment methods, consult the Gosudoc. You can also place the Studio cursor within a method signature and press `Ctrl+Q`.

IMPORTANT Guidewire deprecates assignment method signatures that do not have a `Group` parameter. Studio indicates this status by marking through the method signature in Gosu code and Studio code completion does not display deprecated methods. You can, however, see them in listed in the full list of assignment methods (with Studio again indicating their deprecated status). Do not use a deprecated assignment method. If your existing Gosu code contains deprecated methods, Guidewire recommends that you rework your code to use non-deprecated methods.

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 only supports assigning Activity entities to a Queue.

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

See also

- “Assignment Queues” on page 55

assignActivityToQueue

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

Use this method to assign this activity to the specified queue. The activity entity then remains in the queue until someone or something reassigns it to a specific user. It is possible to assign an activity in the queue manually (by the group supervisor, for example) or for an individual to choose the activity from the queue.

To use this method, you need to define an `AssignableQueue` object.

Defining an AssignableQueue object by using the group name

You can use the name of the group to retrieve a queue attached to that group.

```
Activity.AssignedGroup.getQueue(queueName)     //Returns an AssignableQueue object
Activity.AssignedGroup.AssignableQueues        //Returns an array of 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. Use the second method if you know the name of the queue.

```
var queue = Activity.AssignedGroup.AssignableQueues[0]
var queue = Activity.AssignedGroup.getQueue( "QueueName" )
```

You can then use the returned `AssignableQueue` object to assign an activity to that queue.

```
Activity.CurrentAssignment.assignActivityToQueue( queue, group )
```

Immediate assignment

Immediate assignment methods assign an object directly to the specified user or group.

assignUserAndDefaultGroup

```
boolean assignUserAndDefaultGroup(user)
```

This method assigns the assignable entity to the specified user, selecting a default group. The default group is generally the first group in the set of groups to which the user belongs. In general, use this method if a user only belongs to a single group, or if the assigned group really does not matter.

It is possible that the assigned group can affect visibility and permissions. Therefore, Guidewire recommends that use this method advisedly. For example, you might want to use this method only under the following circumstances:

- The users belong to only a single group.
- The assigned group has no security implications.

The following example assigns an Activity to the current user and does not need to specify a group.

```
Activity.CurrentAssignment.AssignUserAndDefaultGroup(User.util.CurrentUser)
```

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 note:

- PolicyCenter ties the round-robin sequence to the set of criteria, not to the set of users. Thus, using the same set of restrictions to find a set of users re-uses the same round-robin sequence. However, two different sets of restrictions can result in *distinct* round-robin sequences, even if the set of resulting users is the same.
- PolicyCenter does not use this kind of round-robin assignment with the load factors maintained in the PolicyCenter **Administration** screen. Those load factors are meaningful only within a single group, and condition-based assignment can span multiple groups.

assignByUserAttributes

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

This method assigns an assignable item to the user who best matches the set of user attribute constraints defined in the `attributeBasedAssignmentCriteria` parameter.

If no user matches the criteria specified by the method, the method assigns the item to the item owner. For example, if the method cannot determine a user from the supplied criteria, it assigns an activity to the owner of the policy associated with the activity.

The AttributeBasedAssignmentCriteria object contains two fields:

Group	If set, restricts the search to the indicated group. This can be null.
AttributeCriteria	An array of AttributeCriteriaElement entities.

The AttributeCriteriaElement entities represent the conditions to be met. If more than one AttributeCriteriaElement entity is present, the method attempts to assign the assignable entity to those users who satisfy all of them. In other words, the method performs a Boolean AND operation on the restrictions.

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 assignByUserAttribute group parameters

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 the following purposes:

1. The assignment method maintains separate round-robin states for the search criteria within each group. This is so that PolicyCenter can use the method for group-specific assignment rotations.
2. If the method selects a user, it uses the supplied group to determine the best group for the assignment:
   - If the user is as a member of multiple subgroups under the supplied group, the method assigns the object to the closest group in the hierarchy to the supplied group.
   - If the user is a member of multiple groups at the same level of the group hierarchy, the method assigns the object to one of these groups randomly.
   - If the user is not a member of any subgroup under the supplied group, the method assigns the object to the supplied group.
   - If the supplied group value is null, the method assigns the object to the first group it finds of which the user is a member. The user must belong to at least one group for this assignment to succeed.

Assign by attribute example

At times, it is important that you assign a particular policy to a specific user, such as one who speaks French or one who has some sort of additional qualification. It is possible that these specially qualified users exist across an organization, rather than concentrated in a single group.

The following example searches for all of `User` entities who have an `AttributeType` of language and `Attribute` value of French.

```
var attributeBasedAssignmentCriteria = new AttributeBasedAssignmentCriteria()
var frenchSpeaker= new AttributeCriteriaElement()
frenchSpeaker.AttributeType = UserAttributeType.TC_LANGUAGE
frenchSpeaker.AttributeField = "Name"
frenchSpeaker.AttributeValue = "French"
attributeBasedAssignmentCriteria.addToAttributeCriteria( frenchSpeaker )
activity.CurrentAssignment.assignByUserAttributes(attributeBasedAssignmentCriteria , false,
activity.CurrentAssignment.AssignedGroup )
```

assignUserByLocation

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

This method uses a location-based assigner to assign an assignable entity based on a given address. This is useful, for example, in the assignment of adjusters and accident appraisers, which is often done based on geographical territory ownership.

If no user matches the criteria specified by the method, the method assigns the item to the item owner. For example, if the method cannot determine a user from the supplied criteria, it assigns an activity to the owner of the policy associated with the activity.

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 assignment performs round-robin assignment through that set, ignoring any matches at a lower level. For example, suppose the method finds no users that match by zip, but a few that match by county. In this case, the method performs round-robin assignment through the users that match by county and it ignores any others that match by state.

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

The following example 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 */ ) )
```

assignUserByLocationAndAttributes

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

The `assigUserByLocationAndAttribute` 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 additional restrictions, such as the ability to handle large dollar amounts or foreign languages, for example.

If no user matches the criteria specified by the method, the method assigns the item to the item owner. For example, if the method cannot determine a user from the supplied criteria, it assigns an activity to the owner of the policy associated with the activity.

As with the `assignByUserAttribute` assignment method, if no user matches the criteria specified by the `attributeBasedAssignmentCriteria`, the method assigns the object to the owner of the object being assigned.

The following example searches for a French speaker that is closest to a given address.

```
var attributeBasedAssignmentCriteria = new AttributeBasedAssignmentCriteria()
var frenchSpeaker = new AttributeCriteriaElement()
frenchSpeaker.AttributeType = UserAttributeType.TC_LANGUAGE
frenchSpeaker.AttributeValue = "french"
attributeBasedAssignmentCriteria.addToAttributeCriteria( frenchSpeaker )
```

```
activity.CurrentAssignment.assignUserByLocationAndAttributes(Activity.Account.PrimaryLocation.Address,
        attributeBasedAssignmentCriteria , false, activity.CurrentAssignment.AssignedGroup)
```

Round-robin assignment

The round-robin algorithm rotates through a set of users, assigning work to each in sequence. It is important to understand that round-robin assignment is distinct from workload-based assignment. Round-robin assignment does not take the current number of entities assigned to any of the users into account.

You can use load factors (in some circumstances) to affect the frequency of assignment to one user or another. Suppose, for example, that person A has a load factor of 100 and person B has a load factor of 50. In this case, the round-robin algorithm selects person A for assignment twice as often as person B.

See also

- See “Secondary (role-based) assignment entities” on page 56 for a discussion on secondary assignment and round-robin states.

assignUserByRoundRobin

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

This method uses the round-robin user selection to choose the next user to receive the assignable from the current group or group tree. If the `includeSubGroups` parameter is `true`, the selector performs round-robin assignment not only through the direct children of the current group, but also through all of the parent group’s subgroups.

To give a concrete example, suppose that you have a set of policies that you want to assign to a member of a particular group. If you want all of the users within that group and all of its descendent groups to share the workload, then you set the `includeSubGroups` parameter to `true`.

The following example assigns an activity to the next user in a set of users in a group.

```
activity.CurrentAssignment.assignUserByRoundRobin( false, Activity.AssignedGroup )
```

Dynamic assignment

Dynamic assignment provides a generic hook for you to implement your own assignment logic, which you can use to perform automated assignment under more complex conditions. For example, you can use dynamic assignment to implement your own version of load balancing assignment.

There are two dynamic methods available, one for users and the other for groups. Both the user- and group-assignment methods are exactly parallel, with the only difference being in the names of the various methods and interfaces.

```
public boolean assignGroupDynamically(dynamicGroupAssignmentStrategy)
public boolean assignUserDynamically(dynamicUserAssignmentStrategy)
```

These methods take a single argument. Make this argument a class that implements one of the following interfaces:

```
DynamicUserAssignmentStrategy
DynamicGroupAssignmentStrategy
```

Dynamic 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. You must lock the rows in the database for these entities.
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 "Dynamic assignment flow" on page 66 and "Dynamic assignment flow" on page 66 previously. That method returns a `GroupUser` entity representing the user and group that you need to assign.
6. Commit the transaction, which results in the lock entities being updated and unlocked.

   Dynamic assignment is not complete after these steps. The interface methods allow for the failure of the commit operation by adding one last final step.
7. If the commit fails, roll back all changes made to the user information, if possible. If this is not possible, save the user name and reassign that user to the assignable item later, during a future save operation.

Dynamic assignment – required methods implementation

Any class that implements the `DynamicUserAssignmentStrategy` interface (or the Group version) must provide implementations of the following methods.

getCandidateUsers

Your implementation of the `getCandidateUsers` method must return the set of users to consider for assignment. (As elsewhere, the `Group` parameter establishes the root group to use to find the users under consideration. The Boolean `includeSubGroups` parameter indicates whether to include users belonging to descendant groups, or only those that are members of the parent group.)

getLocksForAssignable

The `getLocksForAssignable` method takes the set of users returned by `getCandidateUsers` and returns a set of entities that you must lock. By locked, Guidewire means that the current server node obtains the database rows corresponding to those entities, which must be persistent entities. Any other server nodes that needs to access these rows must wait until the assignment process finishes. Round-robin and dynamic assignment require this sort of locking to mandate that multiple nodes do not perform simultaneous assignments. This ensures that multiple nodes do not perform simultaneous assignments and assign multiple activities (for example) to the same person, instead of progressing through the set of candidates.

findUserToAssign

Your implementation of the `findUserToAssign` method must perform the actual assignment work, using the two sets of entities returned by the previous two methods. (That is, it takes a set of users and the set of entities for which you need to lock the database rows and performs that actual assignment.) This method must do the following:

- It makes any necessary state modifications (such as updating counters, and similar operations).
- It returns the `GroupUser` entity representing the selected User and Group.

Make any modifications to items such as load count, for example, to entities in the bundle of the assignable. This ensures that PolicyCenter commits the modifications at the same time as it commits the assignment change.

rollbackAssignment

Guidewire provides the final two API methods to deal with situations in which, after the assignment flow, some problem in the bundle commit blocks the assignment. This can happen, for example, if a validation rule caused a database rollback. However, at this point, PolicyCenter has already updated the locked objects and committed the objects to the database (as in "Dynamic assignment flow" on page 66 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 re-adjust 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 it is not possible to commit an assignment, your logic does one of the following:

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

Dynamic assignment – DynamicUserAssignmentStrategy implementation

As a very simple implementation of this API, Guidewire includes the following in the base configuration:

```
gw.api.LeastRecentlyModifiedAssignmentStrategy
```

The following code shows the implementation of the `LeastRecentlyModifiedAssignmentStrategy` class. This is a very simple application of the necessary concepts needed to create a working implementation. The class performs a very simple user selection, simply looking for the user that has gone the longest without modification.

Since the selection algorithm needs to inspect the user data to perform 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.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 = candidates as Set<User>
    var oldestModifiedUser = 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
  }
}
```

chapter 9

Rule-based validation

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

See also

- "Validation" on page 44
- *Configuration Guide*

Overview of rule-based validation

Gosu rules-based validation uses rules that you define through the Studio Rules editor. Rules-based validation works on non-policy objects only, objects that are not part of the policy graph.

For an entity to have 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:

- `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

- *Configuration Guide*

About the validation graph

During database commit, PolicyCenter performs validation on the following items:

- Any validatable entity that PolicyCenter updated or inserted into the validation graph.
- Any validatable entity that refers to an entity that PolicyCenter updated, inserted, or removed from the validation graph.

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 PolicyCenter inserted, updated, or removed from the validation graph.

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`.

Note: To actually trigger validation, you must set each link in the chain—`Address`, `ContactAddress`, and `Contact`—as triggering validation and set `Policy` as validatable.

PolicyCenter stores this path in reverse in the validation graph. Thus, if an address changes, PolicyCenter traverses 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`).

How PolicyCenter traverses 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 that you update an object linked to another 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 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 parents.

You can use the `triggersValidation` attribute on foreign keys and arrays that you add to custom subtypes, to custom entities, or to core entities by using extensions. The default value of this attribute is `false` if you do not set it specifically. If you want a new foreign key or array element of a validatable entity to trigger validation whenever its linked entities change, set `triggersValidation` to `true`.

How PolicyCenter works with owned arrays

In the PolicyCenter data model, an *owned array* is an array that is attached to a validatable entity and which has its `triggersValidation` attribute set to `true`. PolicyCenter treats owned arrays 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, PolicyCenter considers any change to the array as a change to the parent entity.

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 has its `triggersValidation` element set to `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 the B → C link has its `triggersValidation` element set to `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 the PolicyCenter installation `modules/configuration/config/metadata/entity` directory.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

Validatable entity	Foreign key entity	Array name
Activity		
Contact	Address	ContactAddresses
Group		
Organization		
ProducerCode		ProducerCodeRoles
Region		
User	Credential UserContact UserSettings	Attributes

Validation trigger 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	More information
`<array>`	`<array-override>`	*Configuration Guide*
`<foreignkey>`	`<foreignkey-override>`	*Configuration Guide*
`<onetoone>`	`<onetoone-override>`	*Configuration Guide*

See also

- See the *Configuration Guide* 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 table summarizes some of the common validation issues and how to mitigate each one.

Performance issue	Notes
Validating large amounts of 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 any object that a large number of other policies can reference. For example, suppose that you set the `Group` field on `Policy` to trigger validation. Editing one policy can then bring the entire application to a crawl as PolicyCenter validated hundreds or even thousands of policies. Setting this particular validation trigger would make editing policy objects nearly impossible.
Length of query paths	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 any time 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.
Links between top-level objects	It is legal to have top-level entities trigger validation on each other. This use of validation triggers, however, can unnecessarily increase the number of paths and the number of objects that PolicyCenter must validate on any particular commit.
Graph direction inconsistency	In general, Guidewire strongly recommends that you consistently order the validation graph to avoid problems. • PolicyCenter consider arrays and foreign keys that represent some sort of containment as candidates for triggering validation. • PolicyCenter does not consider foreign keys that you reference as arrays or that are merely there to indicate association as candidates for triggering validation.
Illegal links and arrays	Virtual foreign keys cannot trigger validation because PolicyCenter does not store them in the database. Attempting to use a virtual foreign key to trigger validation causes PolicyCenter to report an error at application startup. Similarly, it is not possible to set any array or link property defined at the application level (such as the `Driver` link on `Policy`) to trigger validation. PolicyCenter considers an array to be in the database if the foreign key that forms the array is in the database.

See also

- “How PolicyCenter traverses the validation graph” on page 72
- “View a text version of the validation graph” on page 76

View a text version of the validation graph

About this task

It is possible to view a text version of the PolicyCenter validation graph for aid in debugging validation issues.

Procedure

1. In PolicyCenter Studio **Project** window, expand **configuration→config→logging**.
2. Open file `log4j2.xml` for editing.
3. Add the following entry to this file:

```
###############################################################################
# PolicyCenter Policy Validation Graph
###############################################################################
log4j.category.com.guidewire.pl.system.bundle.validation.ValidationTriggerGraphImpl=DEB
UG, Console,
      PolicyValidationGraphLog
log4j.appender.PolicyValidationGraphLog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.PolicyValidationGraphLog.encoding=UTF-8
log4j.appender.PolicyValidationGraphLog.File=/tmp/gwlogs/PolicyValidationGraphLog.log
log4j.appender.PolicyValidationGraphLog.DatePattern=.yyyy-MM-dd
log4j.appender.PolicyValidationGraphLog.layout=org.apache.log4j.PatternLayout
log4j.appender.PolicyValidationGraphLog.layout.ConversionPattern=%-10.10X{server}
%-8.24X{userID}
      %d{ISO8601} %p %m%n
```

4. Redeploy logging configuration file and restart the PolicyCenter application server.

Result

This action causes the validation graph to print during application deployment both as text in the system console and to the following log file:

`/tmp/gwlogs/PolicyValidationGraphLog.log`