



Guidewire PolicyCenter™

Best Practices Guide

Release 10.1.2

© 2021 Guidewire Software, Inc.

For information about Guidewire trademarks, visit <http://guidewire.com/legal-notices>.

Guidewire Proprietary & Confidential — DO NOT DISTRIBUTE

Product Name: Guidewire PolicyCenter

Product Release: 10.1.2

Document Name: Best Practices Guide

Document Revision: 14-June-2021

Contents

Guidewire Documentation	6
About PolicyCenter documentation	6
Support	8
1 Overview of best practices	9
2 Data model best practices	11
Entity best practices	11
Observe entity declaration naming conventions	11
Add a suffix to entity extensions	12
Use singular words for field names except for arrays	12
Add ID as a suffix to column names for foreign keys	12
Typelist best practices	13
Observe typelist naming conventions	13
Add a suffix to new typelists and typecode extensions	13
Do not rely on implicit conversion of typecodes to strings	14
Script parameter best practices	14
Other data model best practices	14
Data model best practices checklist	14
3 User interface best practices	15
Page configuration best practices	15
Modify base PCF files whenever possible	15
Add a suffix to new PCF files to avoid name conflicts	15
Avoid using Gosu code in PCF files	15
Display keys best practices	16
Use display keys to display text	16
Use existing display keys whenever possible	16
Observe display key naming conventions	16
Add a suffix to new display keys to avoid name conflicts	16
Organize display keys by page configuration component	17
User interface performance best practices	17
Avoid repeated calculations of expensive widget values	17
Avoid expensive calculations of widget properties	18
Use application permission keys for visibility and editability	18
User interface best practices checklist	19
4 Rules best practices	21
Rules naming best practices	21
Observe rule naming conventions	21
Rule naming summary principles	22
Root rules naming conventions	22
Parent and child rules naming conventions	23
Observe operating system length restrictions on rule names	23
Get and display rule names in messages	24
Assign a dedicated rules librarian to manage rule names	24
Rules performance best practices	24
Purge unused and obsolete rules before upgrading	24
Rules best practices checklist	24

5 Gosu language best practices	27
Gosu naming and declaration best practices	27
Observe general Gosu naming conventions	27
Omit type specifications with variable initialization	28
Add a suffix to functions and classes to avoid name conflicts	28
Declare functions Private unless absolutely necessary	28
Use public properties instead of public variables	28
Do not declare static scope for mutable variables	29
Use extensions to add functions to entities	29
Match capitalization of types, keywords, and symbols	29
Gosu commenting best practices	30
Comment placement	30
Block comments	30
Javadoc comments	30
Single-line comments	30
Trailing comments	31
Using comment delimiters to disable code	31
Gosu coding best practices	31
Guidewire internal methods	32
Use whitespace effectively	32
Use parentheses effectively	32
Use curly braces effectively	33
Program defensively against conditions that can fail	33
Use case-insensitive comparisons	33
Check for null values	33
Allow default null-safe property access	34
Usage of explicit null-safe operators	34
Check boundary conditions	34
Use structured exception handling	34
Omit semicolons as statement delimiters	34
Observe null safety with equality operators	35
Rewrite comparison operators to avoid the equals method	35
Use typeis expressions for automatic downcasting	35
Observe loop control best practices	36
Implement conditional operators in loop conditions correctly	36
Interrupt loop execution as early as possible	36
Return from functions as early as possible	37
Use gw.api.util.DateUtil instead of java.util.Date	37
Gosu performance best practices	38
Consider the order of terms in compound expressions	38
Avoid repeated method calls within an algorithm	39
Remove constant variables and expressions from loops	39
Avoid doubly nested loop constructs	39
Pull up multiple performance intensive method calls	40
Be wary of dot notation with object access paths	41
Avoid code that incidentally queries the database	41
Use comparison methods to filter queries	42
Use comparison methods instead of the Where method	42
Use Count properties on query builder results and find queries	43
Use activity pattern codes instead of public IDs in comparisons	44
Create single plugin instance	44
Gosu template best practices	45
Gosu best practices checklist	45

6 Upgrade best practices	47
Upgradability best practices	47
Add minor changes directly to base files	47
Copy base files to add major changes	47
Copy base functions to make major changes	48
Switching from minor to major changes	48
Upgrade best practices checklist	48
7 Rating Management best practices	49
Rating Management component naming conventions best practices	49
Rate book edition naming conventions best practices	50
Rate routine step naming conventions best practices	51
Rating Management best practices checklist	51

Guidewire Documentation

About PolicyCenter documentation

The following table lists the documents in PolicyCenter documentation:

Document	Purpose
<i>InsuranceSuite Guide</i>	If you are new to Guidewire InsuranceSuite applications, read the <i>InsuranceSuite Guide</i> for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications.
<i>Application Guide</i>	If you are new to PolicyCenter or want to understand a feature, read the <i>Application Guide</i> . This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with PolicyCenter.
<i>Upgrade Guide</i>	Describes the overall upgrade process, and describes how to upgrade your configuration and database. The intended readers are implementation engineers who must merge base application changes into existing application extensions and integrations.
<i>Configuration Upgrade Tools Guide</i>	Describes the tools and functionality provided by the Guidewire InsuranceSuite Configuration Upgrade Tools. The intended readers are implementation engineers who must merge base application changes into existing application extensions and integrations. Visit the Guidewire Community to access the <i>Configuration Upgrade Tools Guide</i> , which is available for download, separately from the main documentation set, with the Configuration Upgrade Tools.
<i>Installation Guide</i>	Describes how to install PolicyCenter. The intended readers are everyone who installs the application for development or for production.
<i>System Administration Guide</i>	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.
<i>Configuration Guide</i>	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.
<i>PCF Format Reference</i>	Describes PolicyCenter PCF widgets and attributes. The intended readers are configuration engineers. See the <i>Configuration Guide</i> .
<i>Data Dictionary</i>	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.
<i>Security Dictionary</i>	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.
<i>Globalization Guide</i>	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.
<i>Rules Guide</i>	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.
<i>Guidewire Contact Management Guide</i>	Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are PolicyCenter implementation engineers and ContactManager administrators.
<i>Best Practices Guide</i>	A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers.
<i>Integration Guide</i>	Describes the integration architecture, concepts, and procedures for integrating PolicyCenter with external systems and extending application behavior with custom programming code. The intended

Document	Purpose
	readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java.
<i>REST API Client Guide</i>	Describes how to use the InsuranceSuite REST API Client to make outbound HTTP calls to internal or third-party REST services.
<i>Java API Reference</i>	Javadoc-style reference of PolicyCenter Java plugin interfaces, entity fields, and other utility classes. The intended readers are system architects and integration programmers.
<i>Gosu Reference Guide</i>	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
<i>Gosu API Reference</i>	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.
<i>ISBTF and GUnit Testing Guide</i>	Describes the tools and functionality provided by InsuranceSuite for testing application behavior during an initial implementation or an upgrade. The guide covers functionality related to Behavior Testing Framework, GUnit, and Gosu functionality designed specifically for application testing. There are two sets of intended readers: business analysts who will assist in writing tests that describe the desired application behavior; and technical developers who will write implementation code that executes the tests.
<i>Glossary</i>	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.
<i>Advanced Product Designer Guide</i>	Advanced Product Designer is a tool that helps you, a business user, design, simulate, and deploy an insurance product. The intended readers are business analysts who understand insurance products, business systems analysts who liaise between business analysts and IT, project managers, and IT who provides technical expertise in areas such as programming, testing, and databases.
<i>Product Model Guide</i>	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 <i>Product Designer Guide</i> .
<i>Product Designer Guide</i>	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.
<i>REST API Framework</i>	Describes the Guidewire InsuranceSuite framework that provides the means to define, implement, and publish REST API contracts. It also describes how the Guidewire REST framework interacts with JSON and Swagger objects. The intended readers are system architects and integration programmers who write web services code or plugin code in Gosu or Java.

Conventions in this document

Text style	Meaning	Examples
<i>italic</i>	Indicates a term that is being defined, added emphasis, and book titles. In monospace text, italics indicate a variable to be replaced.	A <i>destination</i> sends messages to an external system. Navigate to the PolicyCenter installation directory by running the following command: <pre>cd installDir</pre>
bold	Highlights important sections of code in examples.	<pre>for (i=0, i<someArray.length(), i++) { newArray[i] = someArray[i].getName() }</pre>
narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Click Submit .

Text style	Meaning	Examples
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.
<i>monospace italic</i>	Variable placeholder text within code examples, command examples, file paths, and URLs.	Run the startServer <i>server_name</i> command. Navigate to <code>http://server_name/index.html</code> .

Support

For assistance, visit the Guidewire Community.

Guidewire customers

<https://community.guidewire.com>

Guidewire partners

<https://partner.guidewire.com>

Overview of best practices

Best practices are recommended principles and practices that help ensure a successful project implementation. Following the best practices ensures quality, predictability, and security for InsuranceSuite applications. These recommendations focus on application development, and cover areas such as defining a data model, designing the user interface, and writing Gosu code.

Adhering to the best practices also helps facilitate a more seamless upgrade to future product releases or a transition to Guidewire Cloud. For more recommended best practices, see “Guidewire Cloud Standards” on the [Guidewire Documentation site](#).

Data model best practices

The PolicyCenter data model comprises metadata definitions of data entities that persist in the PolicyCenter application database. Metadata definition files let you define the tables, columns, and indexes in the relational database that supports your application. Typelist definitions let you define sets of allowed codes for specific typekey fields on entities.

Entity best practices

You can change the base data model of PolicyCenter to accommodate your business needs with *data model extensions*. Extensions let you add fields to existing data entities and add entirely new data entities to the data model of your PolicyCenter application. PolicyCenter uses the data model and your extensions to create and manage the tables, columns, and indexes in the relational database that supports your application.

As a best practice, Guidewire recommends that you edit the metadata definition files of the data model by using the Data Model Extensions editor in Guidewire Studio.

[See also](#)

- *Configuration Guide*

Observe entity declaration naming conventions

Entity declarations have three names: a metadata definition file name, an entity attribute name, and a `displayName` attribute name. Generally, entity declaration names begin with a capital letter. Medial capitals separate words in compound entity names.

Entity declaration names differ with respect to allowing a space to separate words in compound entity names and with respect to requiring a suffix. `displayName` attribute names permit spaces to separate words. Metadata definition file names and entity attribute names do not. Unlike the other entity declaration names, metadata definition file names require a three letter suffix. These suffixes include `.eti`, `.eix`, and `.etx`.

Metadata definition file names and entity attribute names are required programmatic names. They are for use in the Gosu programming language. These names prohibit spaces altogether. In addition, the base of a metadata definition file name must match the value of the corresponding entity attribute name.

While `displayName` attribute names are optional, Guidewire recommends that you use such names when two conditions are true. If you create a custom entity and the programmatic entity declaration name for the entity exceeds one word, use a `displayName` attribute name. For example, an entity with an entity attribute name of `Activity` does not require a `displayName` attribute name. However, consider a `displayName` attribute name when you have an entity attribute name such as `ContactAddress` or `AllEscalatedActivitiesClaimMetric`. In these

cases, use Contact Address and All Escalated Activities Claim Metric as respective `displayName` attribute names. The graphical user interface uses the `displayName` attribute name when displaying the corresponding entity.

Add a suffix to entity extensions

In future releases, Guidewire may add or change base entities. If you make your own changes to entities, there is the possibility of a naming conflict with one of these future updates. For example, you may add new entities, or new properties to existing entities. To avoid possible naming conflicts, Guidewire recommends that you append the suffix `_Ext` to your new entities and properties.

If you add a new entity, add `_Ext` to the endings of entity names so that Guidewire Studio and the *Data Dictionary* list them next to any entities that they extend. For example, `CreditHistory_Ext`. The property names in your new entity do not need the suffix `_Ext` because the name of the entity already has the `_Ext` suffix.

If you add new properties to an existing Guidewire entity, add the `_Ext` suffix to their names. Follow this recommendation for properties, and also for virtual properties and functions added through entity enhancements.

Note: Because PolicyCenter automatically prefixes extension table names with `pcx_`, if you run into limits on the length of the table name, you can consider removing the `_Ext` suffix from the table name.

As an example of property extensions to a base entity, the following sample metadata file extends the base Policy entity with an additional property (column) and an additional typekey.

```
<extension entityName="Policy">
  ...
  <column desc="Description of the column"
    name="MyCustomColumn_Ext"
    nullok="true"
    default="abc"
    type="varchar"
    <columnParam
      name="size"
      value="60" />
  </column>

  <typekey desc="Description of the typekey"
    name="MyCustomTypekey_Ext"
    typelist="myCustomTypeList_Ext"
    nullok="true" />
  ...
</extension>
```

Use singular words for field names except for arrays

Guidewire recommends that you name most fields with a singular word such as Phone or Age. However, because array fields reference a list of objects, Guidewire recommends that you name them with a plural word.

For example, `Policy.CurrentRoleAssignment` and `Policy.CorePolicyNumber` are single fields on a Policy entity, but `Policy.RiskNotes` is an array of multiple notes. Also, for arrays fields that are extensions, make the primary name plural and not the Ext prefix or suffix. For example, use `Ext_MedTreatments` or `MedTreatments_Ext`, and not `MedTreatment_Exts`.

Add ID as a suffix to column names for foreign keys

Guidewire recommends that you add ID as a suffix to the column names of foreign keys. By default, the column name of foreign keys have the same name as the foreign key names. Use the `columnName` attribute of `foreignkey` elements to override their default column names. For example:

```
<foreignkey
  columnName="PolicyID"
  ...
  name="Policy"/>
```

Adding the suffix ID to the column names of foreign keys helps database administrators identify columns in the database that Guidewire uses as foreign keys.

If you add a foreign key as an extension to a base entity, follow the best practice of adding a prefix or suffix to the name. For example:

```
<foreignkey
  columnName="Policy_ExtID"
  ...
  name="Policy_Ext"/>
```

Typelist best practices

A *typelist* represents a set of allowed values for specific fields on entities in the data model. A *typecode* represents an individual value within a typelist. A typecode comprises:

- A code that the database stores as a column value
- A name that the user interface displays
- A priority setting that drop-down lists use to order they typecode names that they display

As a best practice, Guidewire recommends that you edit typelist definitions by using the Typelist editor in Guidewire Studio.

See also

- *Configuration Guide*

Observe typelist naming conventions

The components of a typelist have the following naming conventions:

Typelist component	Naming conventions	Example
Typelist	Mixed case, with the first letter uppercase, and with the first letter of each internal word capitalized	ActivityCategory
Code	Only lowercase letters. Underscores (_) separate words.	approval_pending
Name	Each word in a name begins with a capital letter. Spaces separate words.	Approval Pending

Add a suffix to new typelists and typecode extensions

In future releases, Guidewire may add or change base typelists. If you make your own changes to typelists, there is the possibility of a naming conflict with one of these future updates. For example, you may add new typelists, or new typecodes to existing typelists. To avoid possible naming conflicts, Guidewire recommends that you append the suffix `_Ext` to your new typelists and typecodes.

If you add a new typelist, its typecode names do not need the suffix `_Ext` because the name of the typelist already has the `_Ext` suffix.

If you add new typecodes to an existing base Guidewire typelist, add the `_Ext` suffix to their names.

As an example of a new typelist, name one that represents types of medical procedures `MedicalProcedureType_Ext`. Name the typecodes in your new typelist without the suffix `_Ext`.

As an example of new typecodes in a base typelist, the following `AddressType` typelist has a new typecode for service entrances.

Code	Name	Description	Priority	Retired
-----	-----	-----	-----	-----
billing	Billing	Billing	-1	false
business	Business	Business	-1	false
home	Home	Home	-1	false
other	Other	Other	-1	false
service_entrance_Ext	Service Entrance	Service Entrance	-1	false

Do not rely on implicit conversion of typecodes to strings

Guidewire recommends that you do not send typecodes as arguments to display keys, but explicitly send the Name properties of typekey.

Script parameter best practices

The following are recommendations when working with script parameters:

- In future releases, Guidewire may add new script parameters. If you create your own script parameters, there is the possibility of a naming conflict with one of these future updates. To avoid possible naming conflicts, Guidewire recommends that you append the suffix `_Ext` to the names of your new script parameters.
- Create an enhancement property on script parameters with the same name as the script parameter itself, including the `_Ext` suffix. Create this enhancement property on a custom enhancement under your own package.

Other data model best practices

The following are additional recommendations when extending the data model:

- Always add an informative description for all new entities, properties, typelists, and typecodes.
- Use the defined text data types (for example, `shorttext`, `mediumtext`, and `longtext`) instead of `varchar`. You may choose to use `varchar` if a fixed width is known from or required by an external system.

Data model best practices checklist

Use the following checklist before you complete your data model configuration tasks to ensure that your data model follows Guidewire best practices.

Best practice to follow	Best practice was followed
"Observe entity declaration naming conventions" on page 11	☐
"Add a suffix to entity extensions" on page 12	☐
"Use singular words for field names except for arrays" on page 12	☐
"Add ID as a suffix to column names for foreign keys" on page 12	☐
"Observe typelist naming conventions" on page 13	☐
"Add a suffix to new typelists and typecode extensions" on page 13	☐

User interface best practices

PolicyCenter uses *page configuration format* (PCF) files to render the PolicyCenter application. PCF files contain metadata definitions of the navigation, visual components, and data sources of the user interface. Display keys provide the static text that visual components of the user interface display.

Page configuration best practices

You can change the user interface of the PolicyCenter application by adding, changing, and removing PCF files from the configuration of your PolicyCenter instance. As a best practice, Guidewire recommends that you edit PCF files by using the Page Configuration (PCF) editor in Guidewire Studio.

See also

- *Configuration Guide*

Modify base PCF files whenever possible

Modify the base configuration files wherever they can be modified. Create new files only when absolutely necessary.

Add a suffix to new PCF files to avoid name conflicts

Every page configuration file in your PolicyCenter instance must have a unique file name. The location of page configuration files within the folder structure of page configuration resources does not ensure uniqueness. To avoid future naming conflicts when Guidewire adds or changes base page configurations, Guidewire recommends that you append the suffix `_Ext` to the names of your page configuration files.

For example, name a new list view for contacts on a policy `PolicyContacts_ExtLV`.

Avoid using Gosu code in PCF files

Avoid including large blocks of Gosu code within the `<Code>` element in PCF files for the following reasons:

- Code in a particular PCF file is not reusable.
- You cannot debug code that is defined in a PCF file.
- Large amounts of code can complicate merging of PCF files during an upgrade.

Display keys best practices

A *display key* represents a single piece of user-viewable text. A display key comprises:

- A name to use in PCF files and Gosu code, where you want to display textual information
- A text value to display in place of the name, for each locale installed in your PolicyCenter instance

As a best practice, Guidewire recommends that you edit your display key definitions by using the Display Keys editor in Guidewire Studio.

See also

- *Configuration Guide*
- *Globalization Guide*

Use display keys to display text

Define display keys for any text that you display to users through the PolicyCenter user interface or in log messages. Do not use hard-coded `String` literals. Display keys help you localize your configuration of PolicyCenter with translations for specific languages.

Use existing display keys whenever possible

Before you create a new display key, search the existing display keys to find one with the text that you want. In Guidewire Studio, open the `display_LocaleCode.properties` file for the language you are using. Then, type the word or phrase you want. The Display Keys editor navigates to and highlights display keys that contain the text or phrase in their names or values.

Observe display key naming conventions

Generally, display key names begin with a capital letter. Internal capitalization separates words in compound display key names. For example:

```
ContactDetail
```

PolicyCenter represents display keys in a hierarchical name space. A period (.) separates display key names in the paths of the display key hierarchy. For example:

```
Validation.Contact.ContactDetail
```

Generally, you specify text values for display key names that are leaves on the display keys resource tree. Generally, you do not specify text values for display key names that are parts of the path to a leaf display key. In the preceding example, the display keys `Validation` and `Validation.Contact` have no text values, because they are parts of a display key path. The display key `ContactDetail` has a text value, “Contact Detail”, because it is a leaf display key with no child display keys beyond it.

Add a suffix to new display keys to avoid name conflicts

To avoid future naming conflicts when Guidewire adds or changes base display keys, append the suffix `_Ext` to your new display key names.

For example, your PolicyCenter instance has a branch of the display key hierarchy for text that relates to contact validation.

```
Validation.Contact.ContactDetail  
Validation.Contact.NewContact
```

You want to add a display key for the text “Delete Contact”. Add a new display key named `DeleteContact_Ext`.

```
Validation.Contact.ContactDetail  
Validation.Contact.DeleteContact_Ext  
Validation.Contact.NewContact
```

You can change the text for base display keys to change the text that the base configuration of the application displays. Guidewire recommends that you use the base configuration display keys for this purpose so the base configuration PCF files can just make use of the new values. If you add display keys with the suffix `_Ext` with the intention of using them in the base configuration, the base configuration PCF files must be altered to use them.

Organize display keys by page configuration component

Guidewire recommends that you organize display keys under paths specific to the page configuration component types and PCF file names where the text values of display keys appears. For example,

```
LV.Activity.Activities.DueDate
```

User interface performance best practices

The ways in which you configure the user interface of your PolicyCenter instance affects its performance. As performance best practices for user interface configuration, Guidewire recommends that you always do the following:

- “Avoid repeated calculations of expensive widget values” on page 17
- “Avoid expensive calculations of widget properties” on page 18
- “Use application permission keys for visibility and editability” on page 18

Avoid repeated calculations of expensive widget values

As a performance best practice, Guidewire recommends that you avoid repeated evaluation of performance intensive expressions for widget values. Depending on the needs of your application, performance intensive expressions may be unavoidable. However, you can improve overall performance of a page by choosing carefully where to specify the expression within the page configuration.

For example, the following value range expression has a performance problem. Evaluation of the expression requires two foreign key traversals and one array lookup. If the `PolicyPeriod` instance is not cached, PolicyCenter executes three relational database queries, which makes the evaluation even more expensive.

```
RangeInput
...
valueRange      |policyPeriod.Policy.Product.AllowedPolicyTerms
```

If a page with this range input widget has any `postOnChange` fields, PolicyCenter potentially evaluates the expression multiple times for each `postOnChange` edit that a user makes.

The following topics provide additional suggestions for handling expensive expressions:

- “Use page variables for expensive value expressions” on page 17
- “Use recalculate on refresh with expensive page variables cautiously” on page 18

Use page variables for expensive value expressions

Instead of specifying an expensive expression as the value for a widget, create a page variable and specify the expensive expression as the initial value. Then, specify the page variable as the value for the widget. Page variables are evaluated only during the construction of the page, not during the remaining lifetime of the page, regardless of `postOnChange` edits.

The following example suffers a performance problem, because it assigns an expensive expression to the value property of a widget.

```
Input: myInput
...
value      |someExpensiveMethod()
```

The following modified sample code improves performance. It assigns a performance intensive expression to a page variable and assigns the variable to the widget value.

```

Variables
...
initialValue |someExpensiveMethod()
name         |expensiveResult
-----
Input: myInput
...
value        |expensiveResult

```

Use recalculate on refresh with expensive page variables cautiously

PolicyCenter evaluates page variables only during the construction of the page, but sometimes you want PolicyCenter to evaluate a page variable in response to `postOnChange` edits. If so, you set the `recalculateOnRefresh` property of the page variable to `true`. If a page variable specifies an expensive expression for its `initialValue`, carefully consider whether your page really must recalculate the variable. If you set the `recalculateOnRefresh` property to `true`, PolicyCenter evaluates the expression at least once for every `postOnChange` edit to the page.

Although PolicyCenter evaluates page variable with `recalculateOnRefresh` set to `true` for each `postOnChange` edit, page variables can yield performance improvements compared to widget values. If several widgets use the same expression for their values, using a page variable reduces the number of evaluations by a factor equal to the number widgets that use it. For example, the `valueRange` of a range input used for drop-down lists in a list view column are evaluated at least once for each row.

Avoid expensive calculations of widget properties

Page configuration widget properties `editable`, `visible`, `available`, and `required`, may need to be evaluated in multiple contexts. If you have a complex or expensive expression in one of these properties, move the expression to a page variable. Otherwise, the expression is evaluated several times before PolicyCenter displays the page.

The following example suffers a performance problem. It assigns a performance intensive expression to the `visible` property of a widget.

```

Input: myInput
...
id          |myInput
...
visible     |activity.someExpensiveMethod()

```

The following modified sample code improves performance. It assigns a performance intensive expression to a page variable. PolicyCenter evaluates page variables only once before it displays a page, regardless how many contexts under which it evaluates widget properties on the page.

```

Variables
...
initialValue |activity.someExpensiveMethod()
name         |expensiveResult
-----
Input: myInput
...
id          |myInput
...
visible     |expensiveResult

```

Use application permission keys for visibility and editability

The `visible` and `editable` properties are evaluated many times during the lifetimes of locations and widgets. As a performance best practice, Guidewire recommends that you structure the page configuration of your user interface so application permission keys determine visibility and editability. This is the most common pattern for user access in business applications.

For example, use the following Gosu expression in the `visible` property of a **Renew** button on a panel that displays information about a policy period:

```
perm.PolicyPeriod.renew(aPolicy)
```

Application permission keys evaluate the current user against general system permissions and the access control lists of specific entity instances.

[See also](#)

- *System Administration Guide*

User interface best practices checklist

Use the following checklist before you complete your user interface configuration tasks to ensure that your user interface configuration follows Guidewire best practices.

Best Practice to Follow	Best Practice Was Followed
"Modify base PCF files whenever possible" on page 15	☐
"Add a suffix to new PCF files to avoid name conflicts" on page 15	☐
"Use display keys to display text" on page 16	☐
"Use existing display keys whenever possible" on page 16	☐
"Observe display key naming conventions" on page 16	☐
"Add a suffix to new display keys to avoid name conflicts" on page 16	☐
"Organize display keys by page configuration component" on page 17	☐
"Avoid repeated calculations of expensive widget values" on page 17	☐
"Use page variables for expensive value expressions" on page 17	☐
"Use recalculate on refresh with expensive page variables cautiously" on page 18	☐
"Use application permission keys for visibility and editability" on page 18	☐

Rules best practices

PolicyCenter rules comprise hierarchies of conditions and actions that implement complex business logic. As a best practice, Guidewire recommends that you edit rules by using the Rules editor in Guidewire Studio.

[See also](#)

- *Rules Guide*

Rules naming best practices

Guidewire recommends a number of rule naming best practices to help you identify and locate specific rules during configuration, testing, and production.

- “Observe rule naming conventions” on page 21
- “Observe operating system length restrictions on rule names” on page 23
- “Get and display rule names in messages” on page 24
- “Assign a dedicated rules librarian to manage rule names” on page 24

Observe rule naming conventions

Each rule name within a rule set must be unique. To help ensure uniqueness, Guidewire recommends that you follow the best practices naming conventions for rules described in this topic. In addition, these naming conventions help you quickly identify each rule within the complex hierarchy of rules in your Guidewire instance during testing and in production.

The basic format for a rule name has two parts:

Identifier- Description

Follow these conventions for *Identifier* and *Description*:

- Separate *Identifier* from *Description* with a space, followed by hyphen, followed by a space.
- Limit *Identifier* to eight alphanumeric characters.

IMPORTANT PolicyCenter truncates Identifier values that exceed eight characters if you include the actions.ShortRuleName property in rule actions to display rule names in messages that you log or display. PolicyCenter also truncates Identifier values that exceed eight characters in automatic log messages if you enable the RuleExecution logging category and set the server run mode to Debug.

- Begin *Identifier* with up to four capital letters to identify the rule set or parent rule of which the rule is a member.
- End *Identifier* with at least four numerals to identify the ordinal position of the rule within the hierarchy of rules in the set.
- For *Description* values, keep them simple, short, and consistent in their conventions.
- Limit the total length of rule name to 60 characters.

For example:

```
CV000100 - Future loss date
```

Rule naming summary principles

Remember these principles for rule names:

- Rule names are unique within a rule set.
- Rules numbers are sequential to mimic the order of rules in the fully expanded set.

The following example demonstrates these principles.

```
Claim Validation Rules
CV001000 - Future loss date
CV002000 - Policy expiration date after effective date
CV002500 - Not Set: Coverage in question
CV003000 - Injury
  CVI03100 - Workers Compensation
    CVIW3110 - Claimant exists
    CVIW3120 - Not Set: Injury description
  CVI03900 - Default
CV004000 - Expected recovery exceeds 100
```

Root rules naming conventions

Consider the following example rule set, Claim Validation Rules. The identifiers of rules in this set all begin with CV, a code to identify “Claim Validation” rules.

```
Claim Validation Rules
CV001000 - Future loss date
CV002000 - Policy expiration date after effective date
CV003000 - Injury
  CVI03100 - Workmen’s Compensation
  CVI03900 - Default
CV004000 - Expected recovery exceeds 100
```

The rule set contains four root rules, with identifiers CV001000, CV002000, CV003000, and CV004000. The numbers at the end of the identifiers, 1000, 2000, 3000 and 4000, are units of one thousand. This spread of numbers lets you add new root rules between existing ones without renumbering. You want identifier numbers for rules in a set to remain in sequential order to mimic the order of rules within the fully expanded set.

For example, you want to add a rule between CV002000 and CV003000. Assign the new rule the identifier CV002500.

```
Claim Validation Rules
CV001000 - Future loss date
CV002000 - Policy expiration date after effective date
CV002500 - Not Set: Coverage in question
CV003000 - Injury diagnosis validity dates
  CVI03100 - Workmen’s Compensation
  CVI03900 - Default
CV004000 - Expected recovery exceeds 100
```

Parent and child rules naming conventions

Many rule sets achieve their business function with simple rules in the root of the set. In preceding example, rules CV001000, CV002000, CV002500, and CV004000 are simple root rules. Frequently however, rule sets achieve their business function *only* with a hierarchy of parent and child rules. In the example, rule CV003000 is a parent rule with two child rules.

When you add child rules to a parent, follow these conventions:

- Expand the beginning code for the child rules with an additional letter to identify their parent.
- Assign each child rule an ending number that falls between the number of the parent and the sibling rule that follows the parent.
- Assign the children a spread of numbers so you can add more children later without renumbering.

In the preceding example, the identifiers for the child rules of CV003000 all begin with CVI, a code to identify “Claim Validation Injury” rules.

```
Claim Validation Rules
...
CV003000 - Injury
  CVI03100 - Workmen's Compensation
  CVI03900 - Default
CV004000 - Expected recovery exceeds 100
```

The spread of numbers for child rules of a root parent rule generally are units of one hundred. This spread of numbers lets you add new child rules between existing ones without renumbering. Most importantly, the numbers of child rules must fall between the numbers of their parent rule and the sibling rule that follows their parent. In this example, the numbers for child rules satisfy both conventions.

The parent and child naming convention applies to another, third level of children. For example, you want to add two new child rules to the rule CVI03100 - Workmen's Compensation. Begin the child identifiers with CVIW, a code to identify “Claim Validation Injury Workmen's Compensation” rules. At the third level of a rule set hierarchy, the spread of numbers for the child rules generally are units of ten.

```
Claim Validation Rules
...
CV003000 - Injury
  CVI03100 - Workmen's Compensation
    CVIW3110 - Claimant exists
    CVIW3120 - Not Set: Injury description
  CVI03900 - Default
CV004000 - Expected recovery exceeds 100
```

Observe operating system length restrictions on rule names

Guidewire stores rules in files within a directory structure that mimics the rule set category structure in Studio. The fully qualified path for a rule file can be quite long, depending on:

- The length of the path to your Guidewire installation directory
- The length of the path from your installation directory to the root of the rules directory, which is 34 characters:

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

- The depth of the hierarchy of rule categories, rule sets, rules, and parent/child rules
- The length of individual rule names

In addition, you must add four characters to the path for the name of each rule category, rule set, and parent rule.

You must understand the file system implications of rule names. Windows file systems have a file path limit of 255 characters. To avoid exceeding the limit, Guidewire recommends that you do the following:

- Avoid a long path to your Guidewire installation directory.
- Avoid deep hierarchies of rule set categories.
- Avoid hierarchies of parent and child rules within a rule set deeper than three levels.
- Avoid long names for rule set categories, rule sets, and rules.

Get and display rule names in messages

As a best practice, Guidewire recommends that you get and display rule names in messages. So, following the Guidewire best practices for rule names helps you identify specific rules in messages to users, in print statements for testing, and in log messages. The alphabetic beginning of a rule identifier helps you find the rule set or parent rule that contains the rule. The numeric ending helps you determine the order of a rule in a rule set or parent rule.

For example, you want to test a producer code validation rule, PCV01000 - No producer code roles. The identifier portion of the rule name begins with PCV. That identifies the rule as a member of the ProduceCodeValidationRules rule set. The identifier ends with 01000. That indicates that rule is near the beginning of the rule set. The rule has the following definition.

```
PCV01000 - No producer code roles
Rule Conditions:
producerCode.ProducerCodeRoles.Count == 0

Rule Actions:
producerCode.rejectField("ProducerCodeRoles", null, null, "loadsave",
    "A role is required for producer code" + producerCode.Code + ".")
)
```

As written, the rejection message that the rule action displays makes it difficult to determine exactly which rule caused an update to fail. To help identify the specific rule in the rejection message, use the `actions.getRule().DisplayName` property to include the identifier portion of the rule name in the message.

```
Rule Actions:
producerCode.rejectField("ProducerCodeRoles", null, null, "loadsave",
    "A role is required for producer code " + producerCode.Code +
    ". Rule: " + actions.getRule().DisplayName.substring(8)
)
```

By including the rule display name in the rule action, users see the following statement in the **Validation** window when the rule action executes.

```
A role is required for producer code ACME235. Rule: PCV01000
```

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

Assign a dedicated rules librarian to manage rule names

As a best practice, Guidewire recommends that you appoint someone in your organization to develop and enforce simple and consistent naming conventions for rule categories, rule sets, and rule names. This helps ensure that naming standards are followed to make sure that rule identifiers readily identify specific rules within the total catalog of rules in your Guidewire instance.

Rules performance best practices

Guidewire recommends performance best practices for rules to help you avoid known performance issues.

Purge unused and obsolete rules before upgrading

As a best practice, Guidewire recommends that you purge unused and obsolete rules from your PolicyCenter configuration. This improves the upgrade process because PolicyCenter does not spend time to evaluate inactive rules that are unused or obsolete.

Rules best practices checklist

Use the following checklist before you complete your rule configuration tasks to ensure that your rules follow Guidewire best practices.

Best Practice to Follow	Best Practice Was Followed
"Observe rule naming conventions" on page 21	☐
"Rule naming summary principles" on page 22	☐
"Root rules naming conventions" on page 22	☐
"Parent and child rules naming conventions" on page 23	☐
"Observe operating system length restrictions on rule names" on page 23	☐
"Get and display rule names in messages" on page 24	☐
"Assign a dedicated rules librarian to manage rule names" on page 24	☐
"Purge unused and obsolete rules before upgrading" on page 24	☐

Gosu language best practices

Gosu is a general-purpose programming language built on top of the Java Virtual Machine. PolicyCenter uses Gosu as its common programming language.

[See also](#)

- *Gosu Reference Guide*

Gosu naming and declaration best practices

Guidewire recommends a number of best practices for naming and declaring Gosu variables, functions, and classes.

- “Observe general Gosu naming conventions” on page 27
- “Omit type specifications with variable initialization” on page 28
- “Add a suffix to functions and classes to avoid name conflicts” on page 28
- “Declare functions Private unless absolutely necessary” on page 28
- “Use public properties instead of public variables” on page 28
- “Do not declare static scope for mutable variables” on page 29
- “Use extensions to add functions to entities” on page 29
- “Match capitalization of types, keywords, and symbols” on page 29

Observe general Gosu naming conventions

As a best practice, Guidewire recommends the following general naming conventions.

Language Element	Naming Conventions	Examples
Variable names	Name variables in mixed case, with the first letter lowercase and the first letter of each internal word capitalized. Name variables mnemonically so that someone reading your code can understand and easily remember what your variables represent. Do not use single letters such as “x” for variable names, except for short-lived variables, such as loop counts.	nextPolicyNumber firstName recordFound
Function names	Compose function names in verb form. Name functions in mixed case, with the first letter lowercase and the first letter of each internal word capitalized. Add the suffix <code>_Ext</code> to the ends of function names to avoid future naming conflicts if Guidewire adds or changes base functions.	getClaim_Ext() getWageLossExposure_Ext()

Language Element	Naming Conventions	Examples
Class name	Compose class names in noun form. Name classes in mixed case, with the first letter uppercase and the first letter of each internal word capitalized. Add the suffix <code>_Ext</code> to the ends of class names to avoid future naming conflicts if Guidewire adds or changes base classes.	<code>StringUtility_Ext</code> <code>MathUtility_Ext</code>

Omit type specifications with variable initialization

Type specifications in variable declarations are optional in Gosu if you initialize variables with value assignments. Whenever you initialize a variable, Gosu sets the type of the variable to the type of the value. As a best practice, Guidewire recommends that you always initialize variables and omit the type specification.

```
var amount = 125.00 // use an initialization value to set the type for a variable
```

```
var string = new java.lang.String("") // initialize to the empty string instead of null
```

Add a suffix to functions and classes to avoid name conflicts

In future releases, Guidewire may add or change base classes and functions. If you make changes to Guidewire packages, there is the possibility of a naming conflict with one of these future updates. For example, you may add enhancement methods or properties to existing Guidewire entities or classes. To avoid possible naming conflicts, Guidewire recommends that you append the suffix `_Ext` to your new functions and classes. For example, name a new function that calculates the days between two dates `calculateDaysApart_Ext(...)`.

You generally will not modify Guidewire classes or packages directly. Instead, create new code within your own package space. Code in your own package does not require a suffix, as it will not conflict with base Guidewire changes. Add a suffix only if there is a possibility of a naming conflict with Guidewire base code.

Declare functions Private unless absolutely necessary

As a best practice, Guidewire recommends that you declare functions as public only with good reason. The default access in Gosu is public. So, declare functions as private if you intend them only for use internally within a class or class extension. Always prefix private and protected class variables with an underscore character (`_`).

Use public properties instead of public variables

As a best practice, Guidewire recommends that you convert public variables to properties. Properties separate the interface of an object from the implementation of its storage and retrieval. Although Gosu supports public variables for compatibility with other languages, Guidewire strongly recommends public properties backed by private variables instead of public variables.

The following sample Gosu code declares a private variable within a class and exposes it as a public property by using the `as` keyword. This syntax makes automatic getter and setter property methods that the class instance variable backs.

```
private var _firstName : String as FirstName // Declare a public property as a private variable.
```

Avoid declaring public variables, as the following sample Gosu code does.

```
public var FirstName : String // Do not declare a public variable.
```

See also

- *Gosu Reference Guide*

Do not declare static scope for mutable variables

As a best practice, Guidewire recommends that you do not use `static` scope declaration for fields that an object modifies during its lifetime. Static fields have application scope, so all sessions in the Java Virtual Machine (JVM) share them. All user sessions see the modifications that made any user session makes to static properties.

For example, the following sample Gosu code is a bad example.

```
class VinIdentifier {
    static var myVector = new Vector() // All sessions share this static variable.

    static function myFunction(){
        myVector.add("new data") // Add data for the entire JVM, not just this session.
    }
}
```

Use extensions to add functions to entities

As a best practice, Guidewire recommends that you add functions that operate on entities as extensions to the existing entity type instead of as static functions on separate utility classes.

Implement functions that operate on single entity instances as extensions

If you want a new function that operates on single instances of an entity type, declare the new function in a separate class extension to that entity type.

For example, you want a new function to suspend a policy. Name your new function `suspend`. Do not declare the function as static with a `Policy` instance as its parameter. Instead, declare the function as an extension of the `Policy` class, so callers can invoke the method directly on a `Policy` instance. For example:

```
if policy.suspend() {
    // Do something to suspend the policy.
}
```

Package entity extensions for an entity type in a single package

Package all of your extensions for an entity type together in a package with the same name as the entity they extend. Do not place all of your entity extensions in a single package. Place all of your extension packages in a package folder that identifies your organization.

For example, place all of your extensions to the `Activity` entity type in a package named `com.CustomerName.activity`.

Match capitalization of types, keywords, and symbols

Access existing types exactly as they are declared, including correct capitalization. Use the Gosu editor's code completion feature to enter the names of types and properties correctly.

The following table lists conventions for capitalization of various Gosu language elements:

Language element	Standard capitalization	Example
Gosu keywords	Always specify Gosu keywords correctly, typically lowercase.	<code>if</code>
Type names, including class names	Uppercase first character	<code>DateUtil</code> <code>Claim</code>
Local variable names	Lowercase first character	<code>myClaim</code>
Property names	Uppercase first character	<code>CarColor</code>
Method names	Lowercase first character	<code>printReport</code>
Package names	Lowercase all letters in packages and subpackages	<code>com.mycompany.*</code>

Some entity and typelist APIs are case insensitive if they take `String` values for the name of an entity, a property, or a typecode. However, it is best write your code as if they are case sensitive.

See also

- *Gosu Reference Guide*

Gosu commenting best practices

As a best practice, Guidewire recommends a variety of comment usages and styles.

- “Comment placement” on page 30
- “Block comments” on page 30
- “Javadoc comments” on page 30
- “Single-line comments” on page 30
- “Trailing comments” on page 31
- “Using comment delimiters to disable code” on page 31

Comment placement

As a commenting best practice, always place block comments before every class and method that you write. Briefly describe the class or method in the block comment. For comments that you place within methods, use any of the commenting styles to help clarify the logic of your code.

Block comments

Block comments provide descriptions of libraries and functions. A block comment begins with a slash followed by an asterisk (`/*`). A block comment ends with an asterisk followed by a slash (`*/`). To improve readability, place an asterisk (`*`) at the beginnings of new lines within a block comment.

```
/*
 * This is a block comment.
 * Use block comments at the beginnings of files and before
 * classes and functions.
 */
```

Place block comments at the beginnings of files, classes, and functions. Optionally, place block comments within methods. Indent block comments within functions to the same level as the code that they describe.

Javadoc comments

Javadoc comments provide descriptions of classes and methods. A Javadoc comment begins with a slash followed by two asterisks (`/**`). A Javadoc comment ends with a single asterisk followed by a slash (`*/`).

```
/**
 * Describe method here--what it does and who calls it.
 * How to Call: provide an example of how to call this method
 * @param parameter description (for methods only)
 * @return return description (for methods only)
 * @see reference to any other methods,
 * the convention is
 * <class-name>#<method-name>
 */
```

Block comments that you format using Javadoc conventions allow automated Javadoc generation.

Single-line comments

Single-line comments provide descriptions of one or more statements within a function or method. A single-line comment begins with double slashes (`//`) as the first two characters two non-whitespace characters on a line.

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

If you cannot fit your comment on a single line, use a block comment, instead.

Trailing comments

Trailing comments provide very brief descriptions about specific lines of code. A trailing comment begins with double slashes (//) following the code that the comment describes. Separate the double slashes from the code with at least two spaces.

```
if (a == 2) {
    return true // Desired value of 'a'
}
else {
    return false // Unwanted value of 'a'
}
```

If you place two or more trailing comments on lines in a block of code, indent them all to a common alignment.

Using comment delimiters to disable code

Use a single-line comment delimiter (//) to comment out a complete or partial line of code. Use a pair of block comment delimiters (/*, */) to comment out a block of code, even if the block you want to comment out contains block comments.

Note: Do not use single-line comment delimiters (//) on consecutive lines to comment out multiple lines of code. Use block comment delimiters (/*, */), instead.

Use caution if you include slashes and asterisks within block comments to set off or divide parts of the comment. For example, do not use a slash followed by a line of asterisks as a dividing line within a block comment. In the following example, the compiler interprets the dividing line as the beginning of a nested comment (/*) but finds no corresponding closing block comment delimiter (*/).

```
/*
    The dividing line starts a nested block comment and causes an unclosed comment compiler error.
    //*****
*/
```

In the preceding example, compilation fails due to an unclosed comment. The following example avoids compilation errors by inserting a space between the slash and the first asterisk.

```
/*
    The dividing line does not start a nested block comment and causes no compiler error.
    // *****
*/
```

Single-line comment delimiters (//) are ignored in block comments.

Gosu coding best practices

Guidewire recommends a number of best practices for Gosu code.

- “Use whitespace effectively” on page 32
- “Use parentheses effectively” on page 32
- “Use curly braces effectively” on page 33
- “Program defensively against conditions that can fail” on page 33
- “Omit semicolons as statement delimiters” on page 34
- “Observe null safety with equality operators” on page 35
- “Use typeis expressions for automatic downcasting” on page 35

- “Observe loop control best practices” on page 36
- “Return from functions as early as possible” on page 37
- “Use `gw.api.util.DateUtil` instead of `java.util.Date`” on page 37

Guidewire internal methods

Some code packages contain Guidewire internal classes that are reserved for Guidewire use only. Gosu code written to configure PolicyCenter must never use an internal class nor call any method of an internal class for any reason. Future releases of PolicyCenter can change or remove an internal class without notification.

The following packages contain Guidewire internal classes.

- All packages in `com.guidewire.*`
- Any package whose name or location includes the word `internal`

Gosu configuration code can safely use classes defined in any `gw.*` package, except for those packages whose name or location includes the word `internal`.

Some Gosu classes are visible in Studio, but are not intended for use. You can distinguish these Gosu classes because they have no visibility annotations (neither `@Export` nor `@ReadOnly`) and they are not in a `gw` package. Do not use these methods in configuration. The methods are unsupported and may change or be withdrawn without notice.

Use whitespace effectively

Guidewire recommends the following best practices for effective use of whitespace:

- Add spaces around operators.

```
premium = Rate + (minLimit - reductionFactor) // proper form
```

```
premium=Rate+(minLimit-reductionFactor) // improper form
```

- Add no spaces between parentheses and operands.

```
((a + b) / (c - d)) // proper form
```

```
( ( a + b ) / ( c - d ) ) // improper form
```

- Indent logical blocks of code by two spaces only.
- Add a blank line after code blocks.
- Add two blank lines after methods, even the last method in a class.

Use parentheses effectively

As a best practice, Guidewire recommends that you always use parentheses to make explicit the operator order of precedence in computational expressions. Otherwise, your computations can be harder to read and more likely to contain mistakes.

Example

In the following example, the two expressions produce the same results, but the first expression uses parentheses to make the standard operator order clear:

```
value = rate + (limit * 10.5) + (deductible / autoGrade) - 15 // clear order of precedence
```

```
value = rate + limit * 10.5 + deductible / autoGrade - 15 // same result, but poor form
```

Example

In the following example, using parentheses in the second Boolean expression ensures that it is read correctly:

```
a == b or (c == d and e == f)    // clear order of precedence

a == b or c == d and e == f      // same result, but poor form
```

Use curly braces effectively

Guidewire recommends the following best practices for effective use of curly braces:

- Surround every logical block, even single-statement blocks, with curly braces ({}).
- Put the opening curly braces ({} on the same line that starts the block.
- Put the closing curly brace (}) on the line after the last statement in the block, aligned with the starting column of the block.

```
if(premium <= 1000) {           // Put opening curly brace on line that starts the block.
    print("The premium is " + premium) // Surround even single-line blocks with curly braces.
}                               // Put closing curly brace on line after last statement.
```

Program defensively against conditions that can fail

As a best practice, Guidewire recommends that you program defensively. Always assume that conditions might fail. Follow these recommendations to avoid common but easily missed programming flaws, such as potential null pointer exceptions and out-of-bounds exceptions.

Use case-insensitive comparisons

Use the `equalsIgnoreCase` method on the string literal to compare it with the value of a variable. Case mismatch can cause comparisons to fail unintentionally.

```
if("Excluded".equalsIgnoreCase(str)) { // proper comparison method
    print("They match.")
}

if(string.equals("Excluded")) { // improper comparison method
    print("They do NOT match.")
}
```

Check for null values

If your code cannot handle a variable that contains non-null values, check the variable for null before you access properties on the variable.

```
function formatName (aUser : User) {
    if (aUser != null) { // Check for null to avoid a null pointer exception later in the code.
        print(aUser.DisplayName)
    }
    else {
        print("No user")
    }
}
```

Also, check variables for null before you call methods on variables:

```
function addGroup (aUser : User, : aGroupUser : GroupUser) {
    if (aUser != null) { // Check for null to avoid a null pointer if you call methods on the variable.
        aUser.addToGroup(aGroupUser)
    }
}
```

Consider the following issues if you do not check variables for null:

- Accessing a property on a variable may use null-safe property access, which can cause null pointer exceptions in later code that handles only non-null values.
- Calling a method on a variable risks a null pointer exception.

Allow default null-safe property access

If your code can handle a variable that contains null values, you can rely on null-safe access of simple properties in most cases. With null-safe property access, the entire expression evaluates to null if the left-side of a period operator is null.

For example, the following sample Gosu code returns null if the user object passed in by the caller is null. Default null-safe property access prevents a null pointer exception in the return statement.

```
function formatName (aUser : User) : String {
    return aUser.DisplayName // Default null-safe access returns null if the passed-in user is null.
}
```

Usage of explicit null-safe operators

If your code does not need to perform additional actions for an object path expression that evaluates to null, use the explicit Gosu null safe operators:

- ?. – Null-safe access to methods
- ?[] – Null-safe access to arrays

Note that access to properties is always null-safe.

The following sample Gosu code uses the explicit null-safe operator ?. to check for null to avoid a null pointer exception while calling a method. If either aPerson or aPerson.PrimaryAddress is null, Gosu does not call the method hashCode and the entire object path expression aPerson.PrimaryAddress?.hashCode() evaluates to 0, which is the int value for null.

```
function formatName (aPerson : Person) : int {
    // An explicit null-safe operator returns null if aPerson or PrimaryAddress is null.

    return aPerson.PrimaryAddress?.hashCode()
}
```

The following sample Gosu code uses the explicit null-safe operator ?[] to check for null to avoid a null pointer exception while accessing an array. If strings[] is null, the entire expression strings?[index] evaluates to null. However, the null-safe operator does not avoid out-of-bounds exceptions if strings[] is non-null and the value of index exceeds the array length. For example, if the array is empty rather than null, an out-of-bounds exception occurs.

```
function getOneString (strings : String[], index : int) : String {
    return = strings?[index] // An explicit null-safe operator evaluates to null if an array is null.
}
```

See also

- *Gosu Reference Guide*

Check boundary conditions

In for loops, check for boundary conditions ahead of the loop. Entering a loop if the boundary condition is satisfied already can cause null pointer exceptions and out-of-bounds exceptions.

Use structured exception handling

Use try/catch blocks wherever required and possible. If you want to catch multiple exceptions, handle them in a hierarchy from low-level, specific exceptions to high-level, generic exceptions.

Omit semicolons as statement delimiters

Semicolons as statement delimiters are optional in Gosu. As a best practice, Guidewire recommends that you omit semicolons. They are unnecessary in almost all cases, and your Gosu code looks cleaner and easier to read without them.

```
// Omit semicolons with statements on separate lines.
print(x)
print(y)
```

Gosu requires semicolons only if you place multiple statements on a single line. As a best practice, Guidewire generally recommends against placing multiple statements on a single line. Exceptions include simple statement lists declared in-line within Gosu blocks.

```
// Include semicolons with multiple statements on a single line.
var adder = \ x : Number, y : Number -> { print("I added!"); return x + y; }
```

Observe null safety with equality operators

The equality and inequality comparison operators `==` and `!=` are null safe if one side or the other evaluates to `null`. Operators that are null safe do not throw null pointer exceptions. As a best practice, Guidewire recommends that you use these comparison operators instead of the `equals` method on objects.

```
if (variable1 == variable2) {           // Comparison operators are null safe.
    print("The variables are equal.")
} else {
    print("The variables are NOT equal.")
}
```

Rewrite comparison operators to avoid the equals method

Do not write Gosu code that uses the `equals` method because it is not type safe.

```
if (activitySubject.equals(row.Name.text) { // This expression is not null safe.
    ...
}
```

As a best practice, Guidewire recommends that you rewrite your Gosu code with comparison operators instead of `equals` methods to make your code type safe and easier to read.

```
if (activitySubject == row.Name.text) { // This expression is null safe and easier to read.
    ...
}
```

Use typeis expressions for automatic downcasting

As a best practice, Guidewire recommends that you use `typeis` expressions to compare the type of an object with a given subtype. After a `typeis` expression, Gosu automatically downcasts subsequent references to the object if the object is of the type or a subtype of the original `typeis` expression. Use `typeis` expressions for automatic downcasting to improve the readability of your code by avoiding redundant and unnecessary casts.

The automatic downcasting of `typeis` expressions is particularly valuable for `if` statements and similar Gosu flow of control structures. Within the code block of an `if` statement, you can omit explicitly casting the object as its subtype. Gosu confirms that the object is the more specific subtype and considers it be the subtype within the `if` code block.

The following sample Gosu code shows a pattern for how to use a `typeis` expression with an `if` statement.

```
var variableName : TypeName           // Declare a variable as a high-level type.

if (variableName typeis SubtypeName) { // Downcast the variable to one of its subtypes.
    ...                               // Use the variable as its subtype without explicit casting.
}
```

The following sample Gosu code follows the pattern and declares a variable as an `Object`. The `if` condition downcasts the variable to its more specific subtype, `String`.

```
var x : Object = "nice" // Declare a String variable as type Object.
var strlen = 0

if (x typeis String) { // Downcast the Object variable to its subtype String.
```

```
    strlen = x.length      // Use the variable as a String without explicit casting.
}
```

Because Gosu propagates the downcasting from the `if` condition into the `if` code block, the expression `x.length` is valid. The `length` property is on `String`, not `Object`.

The following sample Gosu code is equivalent to the preceding example, but it redundantly casts the variable as a `String` within the `if` code block.

```
var x : Object = "nice"      // Declare a String variable as type Object.
var strlen = 0

if (x typeis String) {      // Downcast the Object variable to its subtype String.
    strlen = (x as String).length // Explicit casting as String is redundant and unnecessary.
}
```

See also

- *Gosu Reference Guide*

Observe loop control best practices

Gosu provides `for()`, `while()`, and `do...while()` statements for loop control. Guidewire recommends a few best practices for your loop control logic:

- “Implement conditional operators in loop conditions correctly” on page 36
- “Interrupt loop execution as early as possible” on page 36

Implement conditional operators in loop conditions correctly

As a best practice, Guidewire recommends that you verify the conditional operators in your conditional expressions to be certain that you fully satisfy the requirements for your loop control logic. For example, `<`, `>`, or `=` might need to be `<=`, `>=`, or `!=`.

Interrupt loop execution as early as possible

As a best practice, Guidewire recommends that you interrupt loop execution as early as possible with `continue` or `break` commands.

Use `break` to break out of loop iteration altogether

The `break` command stops execution of the loop altogether, and program execution proceeds immediately to the code that follows the loop.

The following sample Gosu code breaks out of the loop altogether on the fourth iteration, when `i` equals 4.

```
for (i in 1..10) {
    if (i == 4) {
        break      // Break out of the loop on the fourth iteration.
    }

    print("The number is " + i)
}

print("Stopped printing numbers")
```

The preceding sample Gosu code produces the following output.

```
The number is 1
The number is 2
The number is 3
Stopped printing numbers
```

Notice that the loop stops executing on the fourth iteration, when `i` equals 4.

Use a `continue` statement to continue immediately with the next loop iteration

The `continue` statement stops execution of the current iteration, and the loop continues with its next iteration.

The following sample Gosu code interrupts the fourth iteration, when *i* equals 4, but the loop continues executing through all remaining iterations.

```
for (i in 1..10) {
  if (i == 4) {
    continue    // End the fourth iteration here.
  }

  print("The number is " + i)
}
```

The preceding sample code produces the following output.

```
The number is 1
The number is 2
The number is 3
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is 10
```

Notice that the loop continues through all nine iterations, but it interrupts the fourth iteration, when *i* equals 4.

Return from functions as early as possible

As a best practice, Guidewire recommends that functions return as early as possible to avoid unnecessary processing.

The following sample Gosu code is inefficient. The function uses a variable unnecessarily, and it does not return a result as soon as it detects the affirmative condition.

```
public function foundThree() : boolean {
  var threeFound = 0           // A numeric variable is unnecessary to return a boolean result.

  for (x in 5) {
    if (x == 3) {
      threeFound = threeFound + 1 // The loop keeps iterating after third one is found.
    }
  }

  return threeFound >= 1       // The function returns long after third one is found.
}
```

The following modified sample code is more efficient. The function returns a result as soon as it detects the affirmative condition.

```
public function foundThree() : boolean {
  for (x in 5) {
    if (x == 3) {
      return true                // The function returns as soon as the third one is found.
    }
  }

  return false
}
```

Use `gw.api.util.DateUtil` instead of `java.util.Date`

As a best practice, Guidewire recommends that you use the utility functions of `gw.api.util.DateUtil` instead of `java.util.Date` in working with dates. Each class provides a set of utility functions that operate on date and time data. Guidewire recommends using the `DateUtil` methods, however, as the utility functions in `DateUtil` derive their time from the PolicyCenter database time. In contrast, the `java.util.Date` functions derive their time from PolicyCenter application time.

IMPORTANT If you intend to do time-based testing, any dates affected by your implementation of the `ITestingClock` plugin must use the `DateUtil` utility functions.

See also

- *Integration Guide*

Gosu performance best practices

The ways in which you write your Gosu code affects compile-time and run-time performance. As best practices for improving the performance of your code, Guidewire recommends that you always do the following:

-
- “Consider the order of terms in compound expressions” on page 38
- “Avoid repeated method calls within an algorithm” on page 39
- “Remove constant variables and expressions from loops” on page 39
- “Avoid doubly nested loop constructs” on page 39
- “Pull up multiple performance intensive method calls” on page 40
- “Be wary of dot notation with object access paths” on page 41
- “Avoid code that incidentally queries the database” on page 41
- “Use comparison methods to filter queries” on page 42
- “Use comparison methods instead of the `Where` method” on page 42
- “Use `Count` properties on query builder results and find queries” on page 43
- “Use activity pattern codes instead of public IDs in comparisons” on page 44
- “Create single plugin instance” on page 44

Consider the order of terms in compound expressions

As a performance best practice, Guidewire recommends that you carefully consider the order of comparisons in compound expressions that use the `and` and `or` operators. Runtime evaluation of compound expressions that use `and` proceed from left to right until a condition fails. Runtime evaluation of compound expressions that use `or` proceed from left to right until a condition succeeds. The order in which you place individual conditions can improve or degrade evaluation performance of compound expressions.

With `and` expressions, place terms likely to fail earlier

Whenever you use the `and` operator, place the condition that is most likely to fail or the least performance intensive earliest in the compound expression. Use the following formula to help you determine which condition to place first, based on the condition with the lowest value.

$$(100 - failurePercentage) * (performanceCost)$$

For example, you have a condition that you expect to fail 99% of the time, with an estimated performance cost of 10,000 per evaluation. You have another condition that you expect to fail only 1% of the time, with an estimated performance cost of 100 per evaluation. According to the formula, place the second condition earliest because it has the lowest score.

$$\begin{aligned} (100 - 99) * 10,000 &= 10,000 \\ (100 - 1) * 100 &= 9,999 \end{aligned}$$

You rarely have accurate figures for the failure percentages or performance costs of specific condition. Use the formula to develop an educated guess about which condition to place earliest. In general, give preference to less performance intensive condition. If the performance costs are roughly equal, give preference to condition with a higher percentage of likely failures.

With or expressions, place terms likely to pass earlier

When you use the or operator, place the condition that is most likely to succeed earliest in compound expressions.

Avoid repeated method calls within an algorithm

Calling a method repeatedly to obtain a value often results in poor performance. As a performance best practice, Guidewire recommends that you save the value from the first method call in a local variable. Then, use the local variable to repeatedly test the value.

The following sample Gosu code suffers a performance problem. It calls a performance intensive method twice to test which value the method returns.

```
if (policy.expensiveMethod() == "first possibility") { // first expensive call
    // do something
}

else if (policy.expensiveMethod() == "second possibility") { // second expensive call
    // do something else
}
```

The following modified sample code improves performance. It calls a performance intensive method once and saves the value in a local variable. It then uses the variable twice to test which value the method returns.

```
var expensiveValue = policy.expensiveMethod() // Save the value of an expensive call.

if (expensiveValue == "first possibility") { // first reference to expensive result
    // do something
}

else if (expensiveValue == "second possibility") { // second reference to expensive result
    // do something else
}
```

Remove constant variables and expressions from loops

As a performance best practice, Guidewire recommends that you do not include unnecessary variables, especially ones that hold objects and entity instances, within loops.

The following sample Gosu code suffers a performance problem. The loop includes an object access expression, `period.Active`, which remains unchanged throughout all iterations of the loop.

```
var period : PolicyPeriod

for (x in 5) {
    if (x == 3 and period.Active) { // Evaluate a constant expression redundantly within a loop.
        print("x == 3 on active period")
    }
}
```

In the preceding example, Gosu evaluates the expression `period.Active` at least twice unnecessarily. The following modified sample code improves performance.

```
var period : PolicyPeriod

if (period.Active) { // Evaluate a constant expression only once before a loop.
    for (x in 5) {
        if (x == 3) {
            print("x == 3 on active period")
        }
    }
}
```

Avoid doubly nested loop constructs

Nesting a loop construct inside another often produces inefficient code or code that does not produce correct results. As a performance best practice, Guidewire recommends that you avoid any doubly nested loop constructs.

The following sample Gosu code attempts to find duplicate values in an array by using a for loop nested inside another. The code is inefficient because it loops through the array five times, once for each member of the array. It produces inappropriate results because it reports the duplicate value twice.

```
var array = new int[]{1, 2, 3, 4, 3} // An array with a duplicated value

for (y in array index m) {          // Loop through the array

    for (z in array index n) {      // Nested loop through the array on each
                                    // member in the outer loop

        if (m != n and y == z) {    // If current members in outer and inner
                                    // loops differ and member values are equal
            print("duplicate value: " + z) // Print a duplicate value in the array
        }
    }
}
```

The preceding sample code produces the following output.

```
duplicate value: 3
duplicate value: 3
```

The following sample Gosu code is a better solution. The code is more efficient because it loops through the array once only. It produces appropriate results because it reports the duplicate value once only.

```
var array = new int[]{1, 2, 3, 4, 3} // An array with a duplicated value
var hashSet = new java.util.HashSet() // Declare an empty hash set, which prohibits
                                       // duplicate values

for (y in array)                      // Loop through the array
    if (!hashSet.add(y)) {            // If array value cannot be added
                                       // to the hash set
        print("duplicate value: " + y) // Print a duplicate value in the array
    }
```

The preceding sample code produces the following output.

```
duplicate value: 3
```

Pull up multiple performance intensive method calls

As a performance best practice, Guidewire recommends a technique called *pulling up*. With the pulling up technique, you examine your existing code to uncover performance intensive method calls that occur in multiple, lower-level methods. If you identify such a method call, pull it up into the higher level-method, so you call it only once. Cache the result in a local variable. Then, call the lower-level methods, and pass the result that you cached down to the lower-level methods as a context variable.

The following sample Gosu code suffers a performance problem. It pushes an expensive method call down to the lower-level routines, so the code repeats the expensive call three times.

```
function computeSomething() { // Performance suffers with an expensive call pushed down.

    computeA()
    computeB()
    computeC()
}

function computeA() {
    var expensiveResult = expensiveCall() // Make the expensive call once.
    //do A stuff on expensiveResult
}

function computeB() {
    var expensiveResult = expensiveCall() // Make the same expensive call twice.
    //do B stuff on expensiveResult
}

function computeC() {
    var expensiveResult = expensiveCall() // Make the same expensive call three times.
```

```
//do C stuff on expensiveResult
}
```

The following modified sample code improves performance. It pulls the expensive method call up to the main routine, which calls it once. Then, it passes the cached result down to the lower-level routines, as a context variable.

```
function computeSomething() { // Performance improves with an expensive call pulled up.

    var expensiveResult = expensiveCall() // Make the expensive call only once.

    computeA(expensiveResult)
    computeB(expensiveResult)
    computeC(expensiveResult)
}

function computeA(expensiveResult : ExpensiveResult) { // Use the pushed down result.
    //do A stuff on expensiveResult
}

function computeB(expensiveResult : ExpensiveResult) { // Use the pushed down result.
    //do B stuff on expensiveResult
}

function computeC(expensiveResult : ExpensiveResult) { // Use the pushed down result.
    //do C stuff on expensiveResult
}
```

Be wary of dot notation with object access paths

As a performance best practice, Guidewire recommends that you be aware of the performance impact of dot notation to access instance arrays on object access paths. You can write an object access path quickly, but your code at runtime can run extremely slowly.

The following sample Gosu code suffers a performance problem. It acquires an array of interest types for all additional interests for all vehicles on the Personal Auto line of business.

```
var personalAutoLine : PersonalAutoLine
var personalAutoAddresses = personalAutoLine.Vehicles*.AdditionalInterests*.AdditionalInterestType
```

Most likely this was not what the developer intended. Determine the most efficient means of acquiring just the data that you need. For example, rewrite the preceding example to use a query builder expression that fetches a more focused set of interest types from the application database.

See also

- *Integration Guide*

Avoid code that incidentally queries the database

As a performance best practice, Guidewire recommends that you avoid object property access or method calls that potentially query the relational database.

Accessing entity arrays does not incidentally query the database

The following sample Gosu code accesses an array of Vehicle instances as entity array.

```
policyLine.Vehicles // Accessing an entity array does not query the database.
```

Accessing the entity array does not incidentally query the relational database. The application database caches them whenever it loads a parent entity from the relational database.

Accessing finder arrays incidentally queries the database

The following sample Gosu code accesses an array of Policy entities by using a Finder method on a Policy instance.

```
policy.Finder.findLocalPoliciesForAccount(account) // Accessing a finder array queries the database.
```

Calling a Finder method does incidentally query the relational database. However, the application database does not cache finder arrays. Only your code keeps the array in memory.

To avoid repeated, redundant calls that incidentally query the database, Guidewire recommends as a best practice that you cache the results once in a local array variable. Then, pass the local array variable to lower level routines to operate on the same results. This design approach is an example of *pulling up*.

See also

- “Pull up multiple performance intensive method calls” on page 40

Use comparison methods to filter queries

The relational database that supports PolicyCenter filters the results of a query much more efficiently than your own Gosu code, because it avoids loading unnecessary data into the application server. As a performance best practice, Guidewire recommends that you filter your queries with comparisons methods rather than filter the results with your own code.

The following sample Gosu code suffers a performance problem. It inadvertently loads most of the claims, along with their policies, from the relational database. Then, the code iterates the loaded claims and their policies and process only those few that match a specific policy. In other words, the code loads an excessive amount of data unnecessarily and takes an excessive amount of time to search for a few instances to process.

```
var targetPolicy : Policy
var claimQuery = Query.make(Claim) // Performance suffers because the query loads all claims.

for (claim in claimQuery.select()) {
  if (claim.Policy == targetPolicy) { // Local Gosu code filters claims.
    // Do something on claims of targeted policies.
    ...
  }
}
```

The following modified sample Gosu code improves performance. It finds only relevant policies to process with the compare method:

```
var targetPolicy : Policy
var claimQuery = Query.make(Claim) // Performance improves because the query loads few claims.

claimQuery.compare("policy", Equals, targetPolicy) // Query comparison method filters claims.

for (claim.Policy in query.select()) {
  // Do something on claims of targeted policies.
}
```

Use comparison methods instead of the Where method

As a performance best practice, Guidewire recommends that you always use comparison methods on query objects made with the query builder APIs. Never convert a query builder result object to a collection and then use the where method on the collection to specify the criteria for the query result. PolicyCenter applies comparison methods whenever a query executes, and the database returns only qualifying entity instances to the application. Without any comparison methods, converting a query result to a collection loads all instances of the primary entity type into the application cache. The where method on the collection then creates a second collection with the qualifying instances.

Comparison methods example

The following sample Gosu code queries the database for addresses in the city of Chicago by using a compare method on the query object. The select object returns only Chicago addresses from the database. The database executes the query at the time the code calls the iterator method.

```
uses gw.api.database.Query

var queryObject = Query.make(Address) // Create a query object.

queryObject.compare(Address#City, Equals, "Chicago") // Apply qualifying, where-clause, criteria.
```

```

var selectObject = queryObject.select()           // Convert the query object to a select object.

var resultIterator = selectObject.iterator()      // Convert the select object to an iterator object,
                                                // which causes the query to be executed
                                                // in the database.

for (address in resultIterator) { // Iterate the the qualifying addresses in result object.
    print (address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
}

```

The preceding sample code performs efficiently, because the database selects the Chicago addresses. The sample code also uses the application cache efficiently, because it loads only the Chicago addresses into application memory.

The Where method example

In contrast to a comparison method on a query object, the `where` method on a collection performs less efficiently and is highly inefficient in its use of application memory. For example, the following sample Gosu code queries the database for all addresses because it omits comparison methods on the query object. The code then converts the result object, with all addresses, to a collection. Finally, the code calls the `where` method on the collection to create a second collection with only addresses in the city of Chicago.

```

uses gw.api.database.Query

var resultContainer = Query.make(Address).select().toCollection().where( // Fetch all addresses into
    \ elt -> elt.City == "Chicago"                                       // a collection and make
)                                                                        // another collection with
                                                                    // qualifying addresses.

for (address in resultContainer) { // Iterate the second container with the qualifying addresses.
    print (address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
}

```

Returning all addresses to the application uses the cache inefficiently by loading it with unwanted addresses. Converting the result to a collection uses the application heap inefficiently by loading the collection with unwanted addressees. Calling the `where` method on the collection to select only addresses in Chicago performs the selection much less efficiently than the database.

Use Count properties on query builder results and find queries

As a performance best practice, Guidewire recommends that you obtain counts of items fetched from the application database by using the `Count` properties on query builder result objects. The same recommendation applies to find expression query objects. Do *not* iterate result or query objects to obtain a count.

IMPORTANT Guidewire recommends the query builder APIs instead of find expressions to fetch items from the application database whenever possible, especially for new code. For more information, see the *Integration Guide*.

Use Empty properties if you want to know whether anything was found

If you want to know only whether a result or query object fetched anything from the application database, use the `Empty` property instead of the `Count` property. The value of the `Empty` property returns to your Gosu code faster, because the evaluation stops as soon as it counts at least one fetched item. In contrast, the value of the `Count` property returns to your Gosu only after counting all fetched items.

The following sample Gosu code uses the `Count` property on a query builder API result object.

```

uses gw.api.database.Query

var policyPeriodQuery = Query.make(PolicyPeriod) // Find all policy periods.
var result = policyPeriodQuery.select()

if (result.Empty) { // Does the result fetch anything?
    print ("Nothing found.")
}

```

```

}
else {
    print ("Got some!")
}

```

The following sample Gosu code uses the Count property on a find expression query object.

```

var policyPeriodQuery = find(p in PolicyPeriod) // Find all policy periods.

if (policyPeriodQuery.Empty) { // Did the query fetch anything?
    print ("Nothing found.")
}
else {
    print ("Got some!")
}

```

Use Count properties if you want the number found

The following sample Gosu code uses the Count property on a query builder API result object.

```

uses gw.api.database.Query

var policyPeriodQuery = Query.make(PolicyPeriod) // Find all policy periods.
var result = policyPeriodQuery.select()
print("Number of policy periods: " + result.Count)

```

The following sample Gosu code uses the Count property on a find expression query object.

```

var policyPeriodQuery = find(p in PolicyPeriod) // Find all policy periods.
print("Number of policy periods: " + policyPeriodQuery.Count)

```

Use activity pattern codes instead of public IDs in comparisons

As a performance best practice, Guidewire recommends that you always use activity pattern codes (`Activity.Pattern.Code`) in comparison expressions for conditional processing and in queries. Comparisons of activity patterns by codes frequently avoid database reads to evaluate the expression.

The following sample Gosu code suffers a performance problem. The comparison with `Activity.Pattern` by public ID `cc:12345` always requires a database read to evaluate the expression.

```

if (Activity.ActivityPattern == ActivityPattern("cc:12345")) { // Comparisons of activity pattern
                                                                // public IDs always require a
                                                                // database read.
}

```

The following sample Gosu code improves performance by comparing `Activity.Pattern.Code` with an activity pattern code in the conditional expression.

```

if (Activity.ActivityPattern.Code == "MyActivityPatternCode") { // Comparisons of activity pattern
                                                                // codes generally avoid a
                                                                // database read.
}

```

Never localize activity pattern codes. These codes are intended for use in Gosu expressions, not for display in error messages of the application user interface.

See also

- *Globalization Guide*

Create single plugin instance

Instantiate each plugin class as a Singleton that is created one time and accessible throughout the application.

```

var samplePlugin = gw.plugin.Plugins.get(com.acme.plugin.SamplePlugin)

```

Gosu template best practices

The Gosu compiler converts a Gosu template file into generated Java class files. The Java compiler has a maximum size of 65535 bytes for any class method. Sufficiently large Gosu templates can result in templates that fail at run time due to this JVM limitation.

If you have very large templates, break them into nested templates. For example, suppose you have a large template that generates three different sections of a large page. Create three additional templates that generate one part of the content. The original template could contain code that calls the other three templates. This design practice prevents bumping up against the size limit. Additionally, this style produces more readable and more manageable code.

See also

- *Gosu Reference Guide*

Gosu best practices checklist

Use the following checklist before you complete your Gosu coding tasks to ensure that your Gosu code follows Guidewire best practices.

Naming and declaration best practices

Best Practice to Follow	Best Practice Followed
"Observe general Gosu naming conventions" on page 27	☐
"Omit type specifications with variable initialization" on page 28	☐
"Add a suffix to functions and classes to avoid name conflicts" on page 28	☐
"Declare functions Private unless absolutely necessary" on page 28	☐
"Use public properties instead of public variables" on page 28	☐
"Do not declare static scope for mutable variables" on page 29	☐
"Use extensions to add functions to entities" on page 29	☐
"Match capitalization of types, keywords, and symbols" on page 29	☐

Commenting best practice

Best Practice to Follow	Best Practice Followed
"Comment placement" on page 30	☐
"Block comments" on page 30	☐
"Javadoc comments" on page 30	☐
"Single-line comments" on page 30	☐
"Trailing comments" on page 31	☐
"Using comment delimiters to disable code" on page 31	☐

Coding best practices

Best Practice to Follow	Best Practice Followed
"Use whitespace effectively" on page 32	☐

Best Practice to Follow	Best Practice Followed
"Use parentheses effectively" on page 32	☐
"Use curly braces effectively" on page 33	☐
"Program defensively against conditions that can fail" on page 33	☐
"Omit semicolons as statement delimiters" on page 34	☐
"Observe null safety with equality operators" on page 35	☐
"Use typeis expressions for automatic downcasting" on page 35	☐
"Observe loop control best practices" on page 36	☐
"Return from functions as early as possible" on page 37	☐

Performance best practices

Best Practice to Follow	Best Practice Followed
	☐
"Consider the order of terms in compound expressions" on page 38	☐
"Avoid repeated method calls within an algorithm" on page 39	☐
"Remove constant variables and expressions from loops" on page 39	☐
"Avoid doubly nested loop constructs" on page 39	☐
"Pull up multiple performance intensive method calls" on page 40	☐
"Be wary of dot notation with object access paths" on page 41	☐
"Avoid code that incidentally queries the database" on page 41	☐
"Use comparison methods to filter queries" on page 42	☐
"Use comparison methods instead of the Where method" on page 42	☐
"Use Count properties on query builder results and find queries" on page 43	☐
"Use activity pattern codes instead of public IDs in comparisons" on page 44	☐
"Create single plugin instance" on page 44	☐

Upgrade best practices

An upgrade of your PolicyCenter installation comprises automated and manual procedures. The ways in which you configure your PolicyCenter installation and the methods you use during an upgrade help determine the ease or difficulty of the procedures for future upgrades.

See also

- *Planning your upgrade* in the *Upgrade Guide*

Upgradability best practices

Guidewire recommends a number of best practices to help prepare your PolicyCenter installation for future upgrades.

- “Add minor changes directly to base files” on page 47
- “Copy base files to add major changes” on page 47
- “Copy base functions to make major changes” on page 48
- “Switching from minor to major changes” on page 48

Add minor changes directly to base files

Whenever you make only minor changes to a file, make them directly within the base file. If the file changes in a future release, you can accept or reject the changes during the upgrade. Your changes and the changes in the new release are visible side-by-side within your three-way merge tool while you merge the upgrade code manually.

Copy base files to add major changes

Whenever you make major changes to a file, make a copy of the file. Name the copy of file the same as the original, with the customer identifier inserted. For example, make a copy of `SomeBaseScreenDV.pcf` and give it the name `SomeBaseScreen_ExtDV.pcf`.

In the original file, add a comment at the top that states you copied the file to make major changes, and include the filename of the copy. For example:

```
<!-- This file was copied to SomeBaseScreen_ExtDV.pcf -->
```

If the file changes in a future release, you will notice that the file was copied. You then can decide whether to replicate the changes in the new file in your copy of the base version of the file. Especially if the change enhances the base file or fixes a defect, you may want to apply the same changes to your copy of the file.

Copy base functions to make major changes

Whenever you make major changes to a function, or method, defined in a Gosu class, make a copy of the function and place it in a customer class. Give the copy of the function the same name as the original, with the customer identifier as a suffix. For example, make a copy of `SomeBaseFunction` in a folder within your customer package, such as `com.Customername`. Name the copied function `SomeBaseFunction_Ext`.

In the original function, add a comment at the top that states you copied the function to make major changes. For example:

```
// -- This function was copied to SomeBaseFunction_Ext --
```

To confirm that you changed all existing code to use your copied function, temporarily rename the original function and then compile your entire project to check for compilation errors. After you remove all calls to the original function, consider commenting out the original function to prevent developers in the future from using it accidentally.

If the function changes in a future release, you will notice that the function was copied. You then can decide whether to replicate the changes in your copy of the base function. Especially if the change enhances the function or fixes a defect, you may want to apply the same changes to your copy of the function.

Switching from minor to major changes

About this task

After you make minor changes to a file or function, you might decide to make additional major changes to the same file or function. If you switch from making minor changes to making major changes, switch from the recommendations for minor changes to the recommendations for major changes.

Procedure

1. Rename the base file or function by including the customer identifier.
For example, make a copy of `SomeBaseScreenDV.pcf` and give it the name `SomeBaseScreen_ExtDV.pcf`.
2. Restore the original base file from `base.zip` or from your source code repository.
3. Add a comment to the top of the restored base file or function to state that the file or function was copied.

For example:

```
<!-- This file was copied to SomeBaseScreen_ExtDV.pcf -->
```

4. Make your additional major changes to the copy of the file or function.

Upgrade best practices checklist

Use the following checklist before you complete your configuration of the base PolicyCenter installation to help ease future upgrades.

Best Practice to Follow	Best Practice Was Followed
"Add minor changes directly to base files" on page 47	☐
"Copy base files to add major changes" on page 47	☐
"Copy base functions to make major changes" on page 48	☐
"Switching from minor to major changes" on page 48	☐

Rating Management best practices

Guidewire Rating Management provides a set of tools to manage and maintain rating in PolicyCenter. Guidewire recommends a number of best practices for Rating Management.

IMPORTANT To determine whether your Guidewire PolicyCenter license agreement includes Guidewire Rating Management, contact your Guidewire sales representative. Rating Management requires an additional license key. For instructions on obtaining and installing this key, contact your Guidewire support representative.

See also

- *Application Guide*
- *Configuration Guide*

Rating Management component naming conventions best practices

Guidewire recommends a number of naming best practices for Rating Management components. Naming conventions are intended to provide consistency, and make it easier to identify components by line of business and purpose.

Component names are not required to be unique, but the identifying code must be unique. For easy identification, Guidewire recommends that you use the same value for both component name and code.

Rating Management component	Naming convention	Example
Rate book	<i>PackageName + DescriptiveName</i>	PASstandardCA – Standard for Personal Auto in California
Rate routine	<i>PackageName + DescriptiveName + _rr</i>	PAAssignDriver_rr
Rate table	<i>PackageName + DescriptiveName + _VersionNumber</i>	PACoverageBaseRate PACoverageBaseRate_v2
Parameter set	<i>PackageName + DescriptiveName</i>	PAVehicleStandardSet

Package name

Follow these conventions for policy line package name:

- Use capital letters, such as CP or PA for Commercial Property or Personal Auto policy lines, respectively.
- If the rating component applies to all lines of business, use ALL.

Descriptive name

Follow these general naming conventions for *DescriptiveName*:

- Use mixed case with the first letter of each word capitalized.
- Do not use spaces.

Rate books

For *DescriptiveName*:

- If you have multiple rate books per policy line, then add underwriting company, jurisdiction, or offering codes to differentiate the books.
- It is not necessary to append the words book or rate book.

Rate routine

Append `_rr` to easily identify rate routine codes as distinct from rate table codes.

For *DescriptiveName*:

- For a rate routine that calculates premium for a specific coverage, use the coverage code.
For example, a rate routine that calculates premium for Comprehensive coverage (PAComprehensiveCov is the code) on a Personal Auto policy can be named: PAComprehensiveCov_rr.
- If a coverage has different rate routines that calculate premium for different types of risks, append the risk type to the rate routine names. To use fewer characters, you can abbreviate the risk type.
For example, a rate routine that calculates premium for Collision coverage for occasional drivers (abbreviate to OD) in a Personal Auto policy can be named: PACollisionOD_rr. The default rate routine for Collision coverage can be named: PACollision_rr.
- If a rate routine applies to multiple coverages, then create a descriptive name. The name can be based on the covered object. Add the word Coverages at the end.
For example, a rate routine that calculates premium for certain coverages with a covered object of PersonalVehicle can be named: PAPersonalVehicleCoverages_rr.

Rate table

In *DescriptiveName*, do not include the word table. When doing a rate table lookup in a rate routine, PolicyCenter displays **table:** before the name.

Rate tables do not have a version field. However, you can copy a rate table and create the equivalent of a new version by including a version number in the name and code.

If you plan on versioning, then use a versioning identifier as a suffix. This can be a version number or additional information that helps you identify the version, such as `_v2` or `_IS02015`. You can choose to add a version number beginning with the second version.

Rate book edition naming conventions best practices

The rate book edition groups a specific set of rate table versions. The rate book edition must be unique for that rate book. For the edition, you can use any sequence or numbering scheme that you prefer.

For example:

- A series of numbers starting at 1 to be consistent with other Rating Management numbering schemes, such as rate routine versions.
- Date in YYYYMMDD format such as 20130101.
- Filing identifier such as CA-2012-39A.

Rate routine step naming conventions best practices

Observe these naming conventions for the instruction and operand types used in rate routine steps.

- **Variables** – Follow naming conventions for Gosu variable names as described in “Gosu naming and declaration best practices” on page 27.
- **Data model** – Objects in the data model, including Cost objects follow naming conventions as described in “Observe entity declaration naming conventions” on page 11. If you add new objects, follow these entity naming conventions.
- **Functions** – Follow naming conventions for Gosu function names as described in “Gosu naming and declaration best practices” on page 27.

Rating Management best practices checklist

Use the following checklist before you complete your rating management configuration tasks to ensure that your implementation of rating management follows Guidewire best practices.

Best Practice to Follow	Best Practice Was Followed
“Rating Management component naming conventions best practices” on page 49	☐
“Rate book edition naming conventions best practices” on page 50	☐
“Rate routine step naming conventions best practices” on page 51	☐

