Guidewire PolicyCenter®

PolicyCenter Globalization Guide

RELEASE 8.0.3

Guidewire®
Deliver insurance your way™

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

Contents

Part IV
National Formats and Data Configuration

About PolicyCenter Documentation

The following table lists the documents in PolicyCenter documentation.

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

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

Conventions in This Document

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

Support

For assistance with this software release, contact Guidewire Customer Support on the Guidewire Resource Portal at `http://guidewire.custhelp.com`.

part I

Introduction

chapter 1

Understanding Globalization

Globalization in PolicyCenter is the set of features and configuration procedures that make PolicyCenter suitable for operation in a global environment.

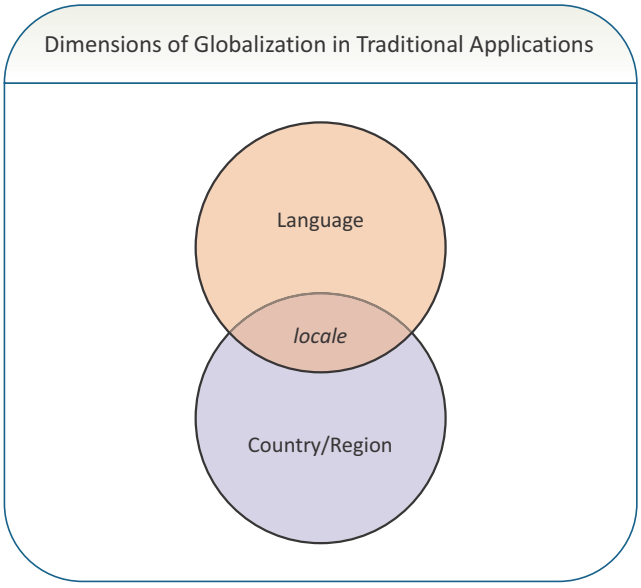
This topic includes:

Dimensions of Globalization

Traditionally, software solves the problem of operation in a global environment along two dimensions that intersect:

- **Language** – Writing system and words to use for text in the user interface
- **Country/region** – Formatting of dates, times, numbers, and monetary values that users enter and retrieve

The intersection of these two dimensions – language with country/region – is known as *locale*.



Traditionally, applications let you select from a pre-configured set of locales. Java embodies this globalization dichotomy in Java locale codes. A Java locale code combines an ISO 639-1 two-letter language code with an ISO 3166-1 two-letter country/region code.

For example, the Java locale code for U.S. English is `en-US`. A locale defines the language of text in the user interface as used in a specific country or region of the world. In addition, the locale defines the formats for dates, times, numbers, and monetary amounts as used in that same country or region.

Shortcomings of the Two Traditional Globalization Dimensions

In traditional applications, the two dimensions of globalization do not cover the following issues for software that operates in a global environment:

- Linguistic searching and sorting
- Phone number formats
- Address formats

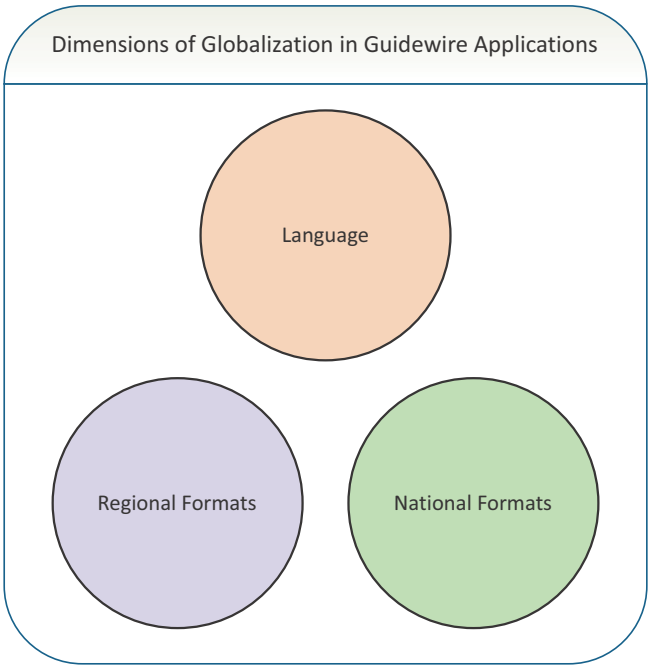
Furthermore, traditional applications enable users to select only pre-defined locales. For example, users typically cannot select French as the language, and, at the same time, select formats for dates, times, numbers, and monetary amounts used by convention in Great Britain.

Globalization Dimensions in Guidewire Applications

Guidewire applications overcome the shortcomings of the traditional model by providing three dimensions for operating in a global environment. These three dimensions are independent:

- **Language** – Writing system and words to use for text in the user interface, as well as for linguistic searching and sorting behavior.
- **Regional formats** – Formatting of dates, times, numbers, and monetary amounts that users enter and retrieve. Regional formats specify the visual layout of data that has no inherent association with specific countries, but for which formats vary by regional convention.

- **National formats** – Formatting of addresses and phone numbers. National formats specify the visual layout of data for which the country or region is inherent, and the format remains the same regardless of local convention.



In Guidewire applications, you can select the language to see in PolicyCenter independently of the regional formats in which you enter and retrieve dates, times, numbers, and monetary amounts.

Phone numbers and addresses in PolicyCenter, however, use national (country) formatting, set through application configuration. For example, if you enter the China country code +86 in a phone field, PolicyCenter displays the phone number by using Chinese formatting. If you enter France for the country in an address field, PolicyCenter shows address fields specific for France, including a CEDEX field.

Selecting Language and Regional Formats in PolicyCenter

In Guidewire PolicyCenter, each user can set the following:

- The language that PolicyCenter uses to display labels and drop-down menu choices
- The regional formats that PolicyCenter uses to enter and display dates, times, numbers, monetary amounts, and names.

You set your personal preferences for display language and for regional formats by using the Options menu at the top, right-hand side of the PolicyCenter screen. On that menu, click **International**, and then select one of the following:

- **Language**
- **Regional Formats**

To take advantage of international settings in the application, you must configure PolicyCenter with more than one region.

- PolicyCenter hides the **Language** submenu if only one language is installed.
- PolicyCenter hides the **Regional Formats** submenu if only one region is configured.
- PolicyCenter hides the **International** menu option entirely if a single language is installed and PolicyCenter is configured for a single region.

PolicyCenter indicates the current selections for **Language** and **Regional Formats** by placing a check mark to the left of each selected option.

Options for Language

In the base configuration, Guidewire has a single display language, English. To view another language in PolicyCenter, you must install a language pack and configure PolicyCenter for that language. If your installation has more than one language, you can select among them from the **Language** submenu. The `LanguageType` typelist defines the set of language choices that the menu displays.

If you do not select a display language from the **Language** submenu, PolicyCenter uses the default language. The configuration parameter `DefaultApplicationLanguage` specifies the default language. In the base configuration, the default language is `en_US`, U.S. English.

Options for Regional Formats

If your installation contains more than one configured region, you can select a regional format for that locale from the **Regional Formats** submenu. At the time you configure a region, you define regional formats for it.

Regional formats specify the visual layout of the following kinds of data:

- Date
- Time
- Number
- Monetary amounts
- Names of people and companies

The `LocaleType` typelist defines the names of regional formats that users can select from the **Regional Formats** menu. The base configuration defines the following locale types:

- Australia (English)
- Canada (English)
- Canada (French)
- France (French)
- Germany (German)
- Great Britain (English)
- Japan (Japanese)
- United States (English)

Unless you select a regional format from the **Regional Formats** menu, PolicyCenter uses the regional formats of the default region. The configuration parameter `DefaultApplicationLocale` specifies the default region. In the base configuration, the default region is `en_US`, United States (English). If you select your preference for region from the **Regional Formats** menu, you can later use the default region again only by selecting it from the **Regional Formats** menu.

See also

- “About Display Languages” on page 21
- “Working with Regional Formats” on page 79

Configuration Files Used for Globalization

You use Guidewire Studio to edit the configuration files for globalization. The following list describes the configuration files and how to navigate to them in the **Project** window.

Location in Project window	Configuration file	Description
configuration → **config** → **Extensions** → **Typelist**	`LanguageType.ttx`	List of defined languages
	`Currency.ttx`	Contains currency code and similar information for the supported currencies
	`PhoneCountryCode.ttx`	Phone country codes
	`State.ttx`	Address configuration
	`StateAbbreviations.ttx`	Enables determining country of duplicate state abbreviations.
	`Jurisdiction.ttx`	Used by a number of PCF files to display a list of states or provinces, as well as jurisdictions that are not states or provinces.
	`Country.ttx`	Country codes that reflect the Unicode Common Locale Data Repository (CLDR) country names
configuration → **config** → **Metadata** → **Typelist**	`LocaleType.tti`	List of defined locales
configuration → **config**	`config.xml`	Configuration parameters related to localization. The localization-related configuration parameters include, among others: • `DefaultApplicationLocale` • `DefaultApplicationLanguage` • `DefaultCountryCode` • `DefaultPhoneCountryCode` Each Guidewire application instance contains a single copy of `config.xml`.
configuration → **config** → **Localizations**	`collations.xml`	Specialized rules for database sort and search for various languages and database types.

Location in Project window	Configuration file	Description
configuration → **config** → **Localizations** → *LocalizationFolder*	`localization.xml`	Currency formatting information for use with single currency rendering mode only. Other locale settings like date and numeric separators. Studio contains multiple copies of this file, one for each locale,
	`display.properties`	Application display keys for a specific language. Each region must have a separate `display.properties` file. It is the presence of multiple `display.properties` files that alerts PolicyCenter to the fact that multiple locales exist. The `display.properties` file (as with all property files) is a standard Java property file with the following format: `display_key_name = value.`
	`typelist.properties`	Application typecode names and descriptions for a specific language.
	typelistName`.sort`	Sort order for typecodes in a typelist for a specific language.
	`gosu.display.properties`	Contains Gosu error messages. Studio displays these messages if it encounters a Gosu error condition. You can translate these error messages into the languages of your choice.
	`language.xml`	Collation and input field label width configuration for a specific language.
	`productmodel.display.properties`	Display keys for the fields used in the PolicyCenter product model. The text strings in this file are in the language specified by the name of the containing folder.
configuration → **config** → **currencies**	`currency.xml`	Regional format overrides for monetary amounts in installations with multiple currencies.
configuration → **config** → **datatypes**	*dataType*`.dti`	Data-type declarations for column types in the data model.
configuration → **config** → **fieldvalidators**	`datatypes.xml`	Default values for `precision` and `scale` values in the default currency.
	`fieldvalidators.xml`	Format information for fields such as currencies, phone numbers, and ID fields, and other fields that need validation and input masks.
configuration → **config** → **fieldvalidators** → *LocalizationFolder*	`fieldvalidators.xml`	Field validator definition overrides by country.
configuration → **config** → **geodata**	*LocaleCode*`-locations.txt`	Mapping between postal codes and cities for a country.

Location in Project window	Configuration file	Description
configuration → **config** → **geodata** → *LocalizationFolder*	address-config.xml	Address configuration by country.
	country.xml	
	zone-config.xml	Zone information for a specific region. Zones are address components used for address autofill and region creation.
configuration → **config** → **phone**	nanpa.properties	Area codes as defined by the North American Numbering Plan Administration (NANPA). These area codes apply to North American countries other than the United States.
	PhoneNumberMetaData.xml	Area codes and validation rules for international phone numbers. See the comments at the beginning of the file for more information.
	PhoneNumberAlternateFormats.xml	Additional area codes and validation rules for international phone numbers. See the comments at the beginning of the file for more information.

Globalization Configuration Parameters in config.xml

Some configuration parameters in `config.xml` relate to general globalization features, such as display languages, regional formats, national formats, territorial data, and currencies.

Configuration Parameters for General Globalization Features

The following parameters in `config.xml` relate to the general globalization features of PolicyCenter. For more information about these parameters, see “Globalization Parameters” on page 57 in the *Configuration Guide*

Parameter name	Sets...
AlwaysShowPhoneWidgetRegionCode	Whether the phone number widget in PolicyCenter displays a selector for phone region codes.
DefaultApplicationCurrency	Currency to use by default for new monetary amounts or whenever the use does not specify a currency for an amount.
DefaultApplicationLangauge	Language that the PolicyCenter shows by default for field labels and other string resources, unless the user selects a different personal preference for language.
DefaultApplicationLocale	Locale for regional formats in the application, unless the user selects a different personal preference for regional formats.
DefaultCountryCode	Country code to use for new addresses by default or if a user does not specify a country code for an address explicitly.
DefaultNANPACountryCode	Default country code for region 1 phone numbers. NANPA stands for North American Numbering Plan Administration.
DefaultPhoneCountryCode	Country code to use for new phone numbers by default or if a user does not specify the country code during phone number entry.
DefaultRoundingMode	Default rounding mode for monetary amount calculations.
MulticurrencyDisplayMode	Whether PolicyCenter displays a currency selector for monetary amounts. In single currency installations, there is no need for a currency selector because all amounts are in the default currency.

part II

Language Configuration

chapter 2

Working with Languages

PolicyCenter enables you to configure support for multiple display languages. Display languages specify the writing system and words to use for text in the user interface, as well as linguistic searching and sorting behavior. Generally, you configure PolicyCenter for display languages by installing language packs from Guidewire.

This topic includes:

About Display Languages

The default display language in PolicyCenter is United States English. To display languages other than U.S. English, you must do the following:

- **Install additional display languages** – Install additional language packs from Guidewire by using the language pack installer. Language packs contain localized screen and field labels and other string resources for a specific language.
- **Select the default display language for the application** – Set one of the installed languages as the default display language for PolicyCenter. Users see the application in the default language at the time that they log in. A user can select any installed language as their preferred display language.

Important:

PolicyCenter enables you to configure display languages of your choice manually. However, Guidewire does not provide language configuration support for either of the following configurations:

- Configuring PolicyCenter with a language for which Guidewire does not provide a language pack.

- Configuring PolicyCenter with a language for which Guidewire provides a language pack without installing the Guidewire language pack for that language.

Note: If you manually configure your own language and have questions, contact your Guidewire Professional Services representative.

See also

- "Installing Display Languages" on page 23
- "Setting the Default Display Language" on page 33

About Language Hierarchies

You can configure PolicyCenter with language hierarchies in which one display language inherits localized values for display keys and other string resources from another language. Languages lower in a hierarchy contain only localized display keys and other string resources that differ from the localized values in the higher language. For example, you could set up Spanish as a root language, with branch languages for the variants of Spanish spoken in Spain and in Mexico.

Configuring The Root Language in a Language Hierarchy

In a language hierarchy, you configure the root language in the **Localizations** folder. You create a localization subfolder and name it by using a lower-case ISO language code that has only two letters. For example, in the Studio **Project** window, the localization folder for the root English language would be **configuration** → **config** → **Localizations** → **en**.

In the case of a Spanish language hierarchy, you start by creating an **es** folder in the **Localizations** folder to configure the root Spanish language. In the **es** folder, provide values for all the PolicyCenter display keys and other string resources localized to your version of the root Spanish language. The values that you provide might be equivalent to the values for Spanish spoken in Spain. Or, the values might be a form of international Spanish, suitable for native speakers throughout the Hispanic world.

Configuring Variant Branch Languages in a Language Hierarchy

In a language hierarchy, you configure the variant branch languages in the **Localizations** folder with localization folders that you name by using Java locale codes. A Java locale code is a language-country pair. It specifies the language by using a lower-case, two-letter ISO language code and it specifies the country by using an upper-case two-letter ISO country code.

In the case of a Spanish language hierarchy, after you establish an **es** folder for the root Spanish language, create localization folders for variants of the Spanish language. Create an **es_ES** localization folder for Spanish spoken in Spain and an **es_MX** localization folder for Mexican Spanish. For example, viewed in the Studio Project window, the localization folders for the variants of English spoken in Spain and Mexico would be the following:

- **configuration** → **config** → **Localizations** → **es_ES**
- **configuration** → **config** → **Localizations** → **es_MX**

The display keys and typecodes defined in these folders override settings in the **es** folder, or add to them, if the equivalent language is selected.

The **es_ES** localization folder would most likely hold very few display keys or other string resources. There are few variations between Spanish spoken in Spain and the **es** localization folder for root Spanish. In contrast, the **es_MX** localization folder would most likely hold many more display keys and other resource strings than the **es_ES** localization folder. Compared to root Spanish, Mexican Spanish has many more linguistic variations than does the Spanish spoken in Spain.

Configuring a Root English Language for English-speaking Countries

Creating an English folder to support British English requires more than just using **en** as the base folder because the English locale in the base configuration is **en_US**. In this case, Guidewire recommends that you do the following:

1. Create an **en** folder and copy the **en_US** property files into that folder.
2. Create an **en_GB** folder, with the localized property files that contain only the specific keys and values for use in Great Britain.
3. Create additional folders for any other variants of English you want to provide. Add the localized properties files that contain only the specific keys and values for use in those countries.

For example, viewed in the Studio **Project** window, the localization folders for the variants of English spoken in Great Britain, Australia, and New Zealand would be the following:

- **configuration** → **config** → **Localizations** → **en_GB**
- **configuration** → **config** → **Localizations** → **en_AU**
- **configuration** → **config** → **Localizations** → **en_NZ**

See also

- “Installing Display Languages” on page 23
- “Upgrading Display Languages” on page 28
- “Activity Logging by the Language Pack Installer” on page 27

Installing Display Languages

To use a display language other than U.S. English in PolicyCenter, you must install a language pack from Guidewire. To install a Guidewire language pack, you must use the language pack installer that Guidewire provides with PolicyCenter. See “About the Language Pack Installer” on page 24.

The language pack installer performs the following actions when it installs a new language pack:

- Verifies the integrity of the files in the language pack before installing them
- Copies the files to their proper locations in the Guidewire home directory
- Modifies the files in place as necessary
- Verifies whether the language pack was properly installed in your configuration of PolicyCenter.

After you install a language pack, the installation is permanent. Language packs cannot be removed.

This topic includes:

- “About the Language Pack Installer” on page 24
- “Installing a Language Pack by Using the Language Pack Installer” on page 24
- “Installed Language Pack Files in the File System” on page 24
- “Installed Language Pack Files in Studio” on page 25
- “Activity Logging by the Language Pack Installer” on page 27

See also

- “Setting the Default Display Language” on page 33
- “Selecting a Personal Language Preference” on page 34
- “Upgrading Display Languages” on page 28

About the Language Pack Installer

The language pack installer is a command line tool that enables you to install or upgrade installed display languages.

Note: This installer is also described in the release notes file for each Guidewire language pack.

The syntax for the command is:

```
gwpc install-localized-module -Dmodule.file=fileName.zip -Dinstall.type=installType
```

The command requires you to specify the following parameters.

Command Parameter	Description
`Dmodule.file`	The name of the ZIP file that contains the language pack. Be certain to include the `.zip` extension. The file must have been saved in the root of the PolicyCenter home directory.
`Dinstall.type`	The type of operation for the language pack installer to perform: • `install` – Install a language pack for a language that is not currently installed.

Installing a Language Pack by Using the Language Pack Installer

Use the language pack installer to install a language pack for a new language. You run the command to run the installer by opening a command prompt in the *PolicyCenter*`/bin` directory.

To install a language pack

1. Stop the PolicyCenter application server and Guidewire Studio, if either is running.
2. Copy the ZIP file that contains the language pack to the root of the PolicyCenter home directory.
3. At a command line prompt, navigate to:
   PolicyCenter`/bin`
4. Enter the following command:
   ```
   gwpc install-localized-module -Dmodule.file=fileName.zip -Dinstall.type=install
   ```
   The command requires you to specify the parameters described previously in “About the Language Pack Installer” on page 24.

   The language pack installer records all file operations that it executes in the command line window and in log files.
5. Restart the application server, and restart Studio if needed.
6. In PolicyCenter, the language that you installed is now a choice from the Options menu.

Installed Language Pack Files in the File System

After you install a language pack, the PolicyCenter `modules` directory contains the following:

- A copy of the language pack ZIP file. Do not delete this file. For example:
  `pc-lang-fr.zip`
- A language module folder with the same name as the language pack ZIP file. For example:
  `pc-lang-fr`
- An archive file that contains copies of all the files that the language pack installation modified during the installation process. You can delete this file. For example:
  `pc-lang-fr_config_archive-8.0.2.62.zip`

- A `base.zip` file

  Note: After you complete the language pack installation, you can delete the source language pack ZIP file that you saved in the root application directory. You can also delete the archive zip file in the `modules` directory. Do not delete any other files that the language pack installed in the `modules` directory. For example, do not delete the `base.zip` file or the installed source language ZIP file, such as `pc-lang-fr.zip` for a French language installation, from the `modules` directory.

If you restart Guidewire Studio, you can see the new module in the **Project** window. The module is in a folder with the same name as the ZIP file for the language pack. For example, if you install a language pack named `pc-lang-zh-cn`, the installer creates the following folder for it:

```
modules/pc-lang-zh-cn
```

The new folder is visible at the top level of the **Project** window in Studio, at the same level as **configuration**. The files installed by a language pack can vary.

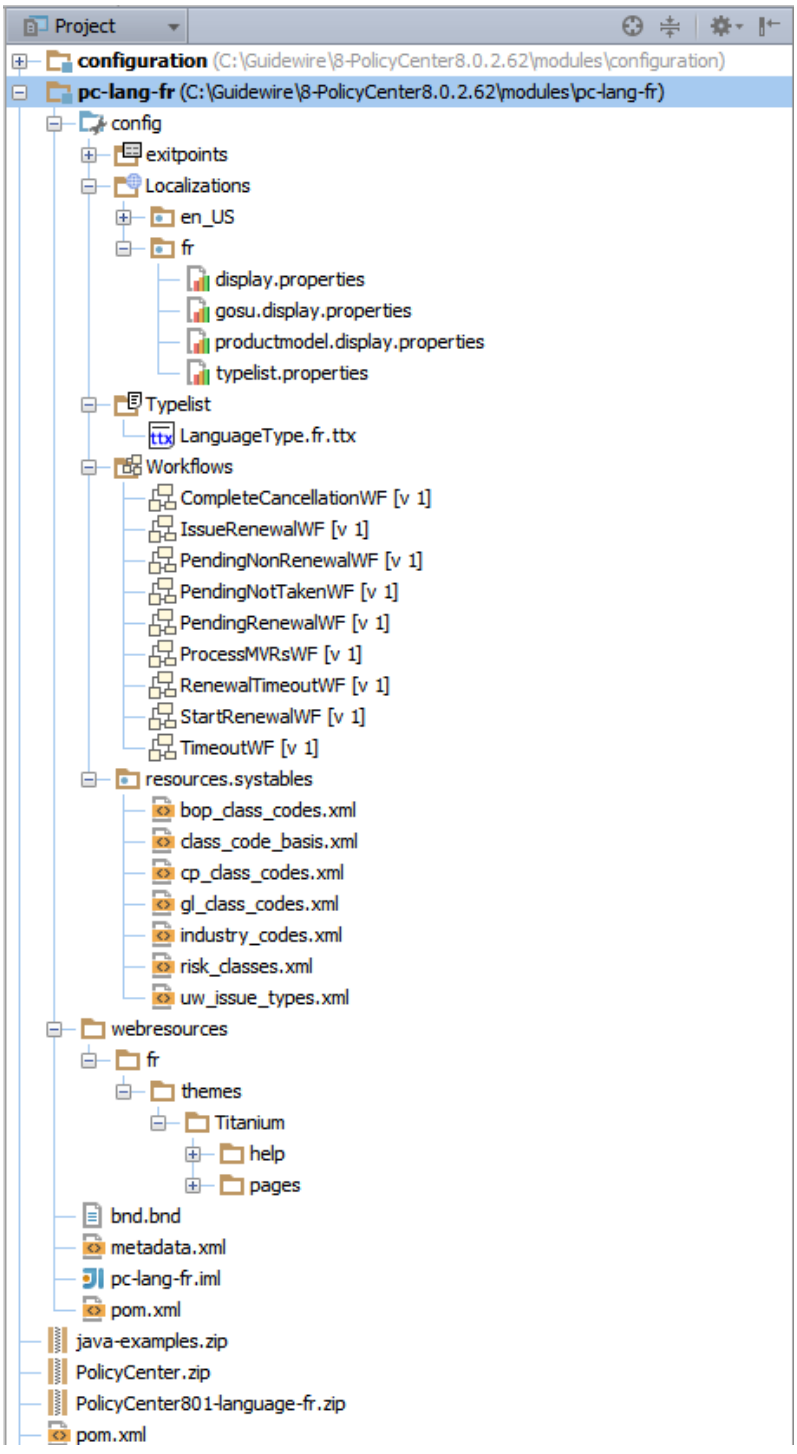
See “Installed Language Pack Files in Studio” on page 25.

Installed Language Pack Files in Studio

After installing a language pack, if you open Guidewire Studio, you see the new language files in the **Project** window. They are stored in a folder that uses the name of the language pack. The folder is at the same level as the **configuration** folder.

IMPORTANT Do not edit the files in any installed language folder. If you want to make changes to the translations provided in these files, do so in the equivalent localization folders in **configuration** → **config** → **Localizations**.

The following figure shows the files visible in the **Project** window of Studio after you install a French language pack in PolicyCenter:



See also

- “Setting the Default Display Language” on page 33
- “Selecting Language and Regional Formats in PolicyCenter” on page 13

Activity Logging by the Language Pack Installer

Every time that the language pack installer runs, it writes messages about its activities to the command console. The activity messages have the following formats:

```
1:18:07,913 INFO INSTALL Localized Module
11:18:11,137 INFO Result:FINISHED
11:18:11,138 INFO
11:18:11,138 INFO Change Details:
11:18:11,138 INFO
11:18:11,138 INFO PREINSTALL STEP
11:18:11,138 INFO Validate Localized Module Checksum
11:18:11,138 INFO
11:18:11,138 INFO PREINSTALL STEP
11:18:11,138 INFO Validate Installation Environment and Version
11:18:11,139 INFO
11:18:11,139 INFO PREINSTALL STEP
11:18:11,139 INFO Validate Clean Environment without Previous Installation
11:18:11,139 INFO
11:18:11,139 INFO PREINSTALL STEP
11:18:11,139 INFO Ensure no typelist conflict
11:18:11,139 INFO
11:18:11,139 INFO PREINSTALL STEP
11:18:11,139 INFO Archive Current Module
11:18:11,139 INFO
11:18:11,140 INFO PREINSTALL STEP
11:18:11,140 INFO Archive Configuration Files
11:18:11,140 INFO ARCHIVE, source: C:/tmp/localized-module-test/.idea/modules.xml, target:
   pc-lang-es_config_archive-8.0.2.dev.201306181134.zip
11:18:11,140 INFO ARCHIVE, source: configuration/pom.xml, target:
   pc-lang-es_config_archive-8.0.2.dev.201306181134.zip
11:18:11,140 INFO ARCHIVE, source: C:/tmp/localized-module-test/pom.xml, target:
   pc-lang-es_config_archive-8.0.2.dev.201306181134.zip
11:18:11,140 INFO
11:18:11,141 INFO INSTALL STEP
11:18:11,141 INFO Extract Module to the Application
11:18:11,141 INFO EXTRACT, source: C:/tmp/pc-lang-es.zip/pc-lang-es, target: pc-lang-es
11:18:11,141 INFO
...
```

The first line of the log output indicates the type of installer activity in progress, which, in this example, is `INSTALL`. Valid installer activity types are:

- `INSTALL`
- `UPGRADE`

The second line of the log output indicates whether the installer was successful and whether the specified activity succeeded or failed. A value of `FINISH` indicates that the installer process completed successfully.

The following example further illustrates logging activity:

```
15:42:47,425 INFO UPGRADE STEP
15:42:47,426 INFO DisplayKey Merger
15:42:47,426 INFO DELETE_IN_MODULE, target: pc-lang-es/config/locale/es_PE/gosu.display.properties
...
```

In the previous example log:

- Line 1 indicates the type of step, which, in this case, is `UPGRADE`.
- Line 2 lists the name of the current installer step.
- Additional lines list file operations that the installer performs on this `UPGRADE` step.

In the following example, the log shows that there was an issue during the upgrade process due to a denial of file access. The log indicates that there is one file that was not accessible to the installer. As a consequence of the file access denial, the installer generates an error. The installer then restores any modified files to their original version before the upgrade started to preserve data integrity.

```
15:52:07,780 INFO UPGRADE Localized Module
15:52:17,434 INFO Result:FAILED
15:52:18,441 ERROR There was an error. The changes have been reverted
15:52:18,441 INFO
15:52:18,441 INFO Change Details:
15:52:18,441 INFO
15:52:18,441 INFO PREINSTALL STEP
15:52:18,441 INFO Validate Installation Environment
```

```
15:52:18,442 INFO
15:52:18,442 INFO PREINSTALL STEP
15:52:18,442 INFO Validate Base Module Present
15:52:18,442 INFO
15:52:18,442 INFO PREINSTALL STEP
15:52:18,442 INFO Validate Module Already Installed
15:52:18,442 INFO
15:52:18,442 INFO PREINSTALL STEP
15:52:18,442 INFO Archive Previous Module
15:52:18,442 INFO ARCHIVE, source: pc-lang-es, target: pc-lang-es-module-archive.zip
15:52:18,443 INFO
15:52:18,443 INFO PREINSTALL STEP
15:52:18,443 INFO Delete Staging Directory
15:52:18,443 INFO DELETE, target: C:/tmp/upgrade-staging
15:52:18,443 INFO
15:52:18,443 INFO UPGRADE STEP
15:52:18,443 INFO Set up the Staging Directory
15:52:18,443 INFO COPY_TO_STAGING, source: pc-lang-es, target: C:/tmp/upgrade-staging
15:52:18,444 INFO
15:52:18,444 INFO UPGRADE STEP
15:52:18,444 INFO Localized Config File Upgrade
15:52:18,444 INFO COPY_TO_MODULE, source: C:/tmp/pc-lang-es-upgrade.zip/pc-lang-es/config/locale
  /es_ES/newGuidewireSetting.properties, target: pc-lang-es/config/locale/es_ES
  /newGuidewireSetting.properties
15:52:18,444 INFO COPY_TO_STAGING, source: C:/tmp/pc-lang-es-upgrade.zip/pc-lang-es/config/locale
  /es_ES/newGuidewireSetting.properties, target: C:/tmp/upgrade-staging/config/locale/es_ES
  /newGuidewireSetting.properties
15:52:18,445 INFO COPY_TO_STAGING, source: C:/tmp/pc-lang-es-upgrade.zip/pc-lang-es/config/extensions
  /typelist/Currency_es.ttx, target: C:/tmp/upgrade-staging/config/extensions/typelist
  /Currency_es.ttx
15:52:18,445 INFO COPY_TO_MODULE, source: C:/tmp/pc-lang-es-upgrade.zip/pc-lang-es/config/locale
   /es_PE/newConfigFile.xml, target: pc-lang-es/config/locale/es_PE/newConfigFile.xml
15:52:18,445 INFO COPY_TO_STAGING, source: C:/tmp/pc-lang-es-upgrade.zip/pc-lang-es/config/locale
  /es_PE/newConfigFile.xml, target: C:/tmp/upgrade-staging/config/locale/es_PE/newConfigFile.xml
15:52:18,446 INFO COPY_TO_STAGING, source: C:/tmp/pc-lang-es-upgrade.zip/pc-lang-es/config/locale
  /es_ES/display.properties, target: C:/tmp/upgrade-staging/config/locale/es_ES
  /display.properties
15:52:18,446 INFO COPY_TO_STAGING, source: C:/tmp/pc-lang-es-upgrade.zip/pc-lang-es/config/locale
  /es_ES/localization.xml, target: C:/tmp/upgrade-staging/config/locale/es_ES/localization.xml
15:52:18,447 INFO DELETE_IN_STAGING, target: C:/tmp/upgrade-staging/config/locale/es_MX
  /localization.xml
15:52:18,447 INFO COPY_TO_STAGING, source: C:/tmp/pc-lang-es-upgrade.zip/pc-lang-es/config/extensions
  /typelist/LanguageType_es.ttx, target: C:/tmp/upgrade-staging/config/extensions/typelist
  /LanguageType_es.ttx
15:52:18,447 ERROR display.properties:
  C:\tmp\localized-module-upgrade-test\modules\pc-lang-es\config\locale\es_ES\display.properties
  (Access is denied)
15:52:18,447 INFO
15:52:18,448 INFO REVERT STEP
15:52:18,448 INFO Delete Staging Directory
15:52:18,448 INFO DELETE, target: C:/tmp/upgrade-staging
15:52:18,448 INFO
15:52:18,448 INFO REVERT STEP
15:52:18,449 INFO Restore Archive Module
15:52:18,449 INFO RESTORE, source: pc-lang-es-module-archive.zip, target: pc-lang-es
15:52:18,449 INFO
```

Upgrading Display Languages

The process for upgrading display languages depends on whether you are upgrading from a previous major release or upgrading between 8.0 maintenance releases.

The overall upgrade process is covered in the *PolicyCenter Upgrade Guide*. This topic covers only the language pack aspects of upgrade.

This topic includes:

Upgrading Language Packs Between PolicyCenter Major Releases

This topic applies if you are upgrading from an earlier major release of PolicyCenter, such as 7.0 or 6.0, to the current release. If you are upgrading a maintenance release of PolicyCenter, see "Upgrading Language Packs Between 8.0.x Maintenance Releases" on page 31

IMPORTANT If you have made changes to files in an installed language module, installing a new language module overwrites those files, causing you to lose those changes. To preserve your changes, back up your modified files and merge in your changes after you install the new language pack.

1. Manually uninstall the currently installed language pack. Depending on your existing application version, use one of the following uninstall procedures:
   - "Uninstalling a Pre-7.0 Language Pack" on page 30
   - "Uninstalling a 7.0 Language Pack" on page 30
2. Upgrade your Guidewire installation to the current version of PolicyCenter 8.0. See the *PolicyCenter Upgrade Guide*.
3. Before you restart the application server after the system upgrade, set the `DefaultApplicationLanguage` parameter in `config.xml` to `en_US`.

   The system upgrade process removes all languages other than English from the `LangugeType` typelist. In addition, if you have removed any region pack, then you also need to set parameter `DefaultApplicationLocale` to `en_US` as well. See:
   - "Setting the Default Display Language" on page 33
   - "Setting the Default Application Locale for Regional Formats" on page 84
4. If you have restarted the application server after the system upgrade, then stop the application server before beginning the language pack upgrade process.
5. For the language pack upgrade, point your application server to an empty database by setting the database configuration in `database-config.xml`.

   This step is necessary because you cannot change certain configuration parameters after you start the application server, which populates the database. Therefore, you must start from an empty database. Do not restart the application server until these instructions say to do so.
6. Install the language pack. See "Installing a Language Pack by Using the Language Pack Installer" on page 24.
7. If your previous installation contained either or both of the following localized elements, use the following instructions for that element:

   Typelist

   In 7.0.x releases, Guidewire released language packs that contained multiple regional variants of a language in the same language pack. In 8.0, each language pack contains the main root variant for that language only. For example, in 8.0 releases, the French language module uses `fr` as the language identifier instead of the `fr_CA` and `fr_FR` designations used in 7.x releases.

   If you are upgrading from a pre-8.0 installation, you need to maintain your `LanguageType` typecodes properly to achieve backward compatibility. Thus, if your 8.0 installation contains a `LanguageType.xx.ttx` typelist in your language module, in which `xx` is the two-letter ISO language code, do the following:

   a. Remove the `LanguageType.xx.ttx` typelist in your 8.0 installation from your language module.

   b. Add the pre-8.0 language typecode to the 8.0 typelist `LanguageType.ttx` in `configuration/config/Extensions/Typelist`.

   Workflow

If there are any localized `workflow.xml` files in `configuration/config/Workflows`, your 8.0 installation contains localized workflow. In this case, you need to change the 8.0 language folder name to match the language name used in pre-8.0 installation configuration. Renaming the 8.0 language folder causes the Guidewire application to change the locale code in `workflow.xml` in the 8.0 language module to match that of the pre-8.0 configuration. The application makes this change the first time that you open any localized workflows—any `workflow.xml` file—in Guidewire Studio.

8. Install any needed region pack accelerator, as described in the release notes for the region pack accelerator.

   Set configuration parameters `DefaultApplicationLanguage` and `DefaultApplicationLocale` to the language and region settings that match your default language and locale. See:

   - “Setting the Default Display Language” on page 33
   - “Setting the Default Application Locale for Regional Formats” on page 84

9. Point your application server to the database you intend to use for production. You set your production database in `database-config.xml`.
10. Set the parameters `DefaultApplicationLanguage` and `DefaultApplicationLocale` as needed for the language and locale you want to see when the application opens.
11. Start the application. This action permanently sets the default language and locale in the database.

Uninstalling a Pre-7.0 Language Pack

You can safely ignore or remove any existing pre-7.0 language pack installation, because the 8.0 language pack installer does not write to the same file space. However, if you have modified your language pack content from the base configuration, do not remove it. After installing the 8.0 language pack, you can manually merge your changes from the previous version.

Uninstalling a 7.0 Language Pack

If your Guidewire installation includes a 7.0 language module, you must uninstall that language module before you upgrade your Guidewire application.

Note: If you have modified your 7.0 language pack content, leave it in place so you can merge your changes later.

After you complete the upgrade, you can install an 8.0 language module.

1. In your 7.0 installation, open the following file:

   ```
   modules/configuration/module.xml
   ```

2. Change the value of `<dependency>` for the installed language module to `pc-gunit`. For example:

   ```
   <?xml version="1.0"?>
     <module name="configuration">
       <dependency>pc-gunit</dependency>
       <src>gsrc</src>
       <src>gtest</src>
   </module>
   ```

3. Save your altered version of `module.xml`.

IMPORTANT After you uninstall the 7.0 language module, the Guidewire 7.0 application server will not start.

Upgrading Language Packs Between 8.0.x Maintenance Releases

This topic applies if you are upgrading from one 8.0.x maintenance release to another, such as from PolicyCenter 8.0.1 to the current release. This upgrade procedure does not apply if you are upgrading from a major release prior to 8.0 that has language packs installed. For those upgrade instructions, see "Upgrading Language Packs Between PolicyCenter Major Releases" on page 29.

IMPORTANT You must either remove or back up and remove any existing language packs before you upgrade your Guidewire application. If you have made changes to files in an installed language module, back up your modified files and merge in your changes after you install the new language pack.

Step 1: Remove any Existing Language and Region Modules

Before you can upgrade your existing Emerald installation, you must manually remove all installed language modules and any installed region module.

1. In your file system, open `PolicyCenter/pom.xml` in an editor.
2. Remove all `<module>` subelements from the `<modules>` element that refer either to a language or a region. These elements have the following form:

   ```
   <module>modules/pc-lang-...</module>
   <module>modules/pc-region-...</module>
   ```

   For example, if the French language pack had been installed, you would delete the following element:

   ```
   <module>modules/pc-lang-fr</module>
   ```

3. In your file system, open `PolicyCenter/.idea/modules.xml` for editing.
4. Remove all `<module>` subelements for all localized modules from the `<modules>` element. These elements have the following form:

   ```
   <module filepath="$PROJECT_DIR$/modules/pc-lang-.../pc-lang-....iml"
           fileurl="file://$PROJECT_DIR$/modules/pc-lang-.../pc-lang-....iml"/>
   <module filepath="$PROJECT_DIR$/modules/pc-region-.../pc-region-....iml"
           fileurl="file://$PROJECT_DIR$/modules/pc-region-.../pc-region-....iml"/>
   ```

   For example, if the French language pack had been installed, you would delete the following element:

   ```
   <module fileurl="file://$PROJECT_DIR$/modules/pc-lang-fr/pc-lang-fr.iml"
           filepath="$PROJECT_DIR$/modules/pc-lang-fr/pc-lang-fr.iml" />
   ```

5. In your file system, open `PolicyCenter/modules`, and then remove all localization folders of the following form:

   ```
   pc-lang-...
   pc-region-...
   ```

   For example, if the French language pack had been installed, you would delete the following folder in `PolicyCenter/modules`:

   ```
   pc-lang-fr
   ```

6. Open Guidewire Studio.
7. In the **Project** window, navigate to configuration → `pom.xml` and then double-click `pom.xml` to open it in the editor.
8. Remove all localized module `<dependency>` subelements from the `<dependencies>` element. These elements have the following form:

   ```
   <dependency>
     <groupId>com.guidewire.pc.appbuild</groupId>
     <artifactId>pc-region-...</artifactId>
     <version>1.0-SNAPSHOT</version>
   </dependency>
   <dependency>
     <groupId>com.guidewire.pc.appbuild</groupId>
     <artifactId>pc-lang-...</artifactId>
   ```

```
  <version>1.0-SNAPSHOT</version>
</dependency>
```

For example, if the French language pack had been installed, you would delete the following element:

```
<dependency>
  <groupId>com.guidewire.pc.appbuild</groupId>
  <artifactId>pc-lang-fr</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

9. In the **Project** window, navigate to **configuration** → `configuration.iml` and then double-click `configuration.iml` to open it in the editor.

10. Remove the `<orderEntry>` elements for all localized modules. These elements have the following form:

```
<orderEntry type="module" module-name="pc-region-..."/>
<orderEntry type="module" module-name="pc-lang-..."/>
```

For example, if the French language pack had been installed, you would delete the following element:

```
<orderEntry type="module" module-name="pc-lang-fr" />
```

11. It is possible that the typecode representing the language pack to uninstall exists in the configuration module in one of the following typelists (with xx being any custom label):

```
LanguageType.ttx
LanguageType.xx.ttx
```

If this is the case, then you need to remove the typecode from the typelist before installing the new language pack. For example, suppose that the typecode `fr`, for the French language, existed in one of these typelists. If you are installing a new French language pack, then you need to remove the `fr` typecode from the configuration-level typelist before installing the new language pack. To remove this typecode from `LanguageType.ttx`:

a. Close Guidewire Studio.

b. In your file system, open `PolicyCenter/modules/configuration/config/extensions/typelist/LanguageType.ttx` for editing.

c. Remove the following element:

```
<typecode
  code="fr"
  desc="French (FR)"
  name="French (FR)"
  priority="1"/>
```

If you do not remove the typecode from the configuration-level typelist, then the language pack installation process generates the following error:

```
---------------------
INFO Ensure no typelist conflict
ERROR The typecode fr has already been defined in the system...
```

Step 2: Upgrade Your 8.0.x Application Installation

1. Follow the steps in the *PolicyCenter Upgrade Guide* to upgrade your configurations from the base release to the target release. See “Upgrading from 8.0.x” on page 25 in the *Upgrade Guide*.

IMPORTANT If the instructions say to set the `upgrader.steps.class` property to a value, the value to set it to is `com.guidewire.tools.upgrade2.emerald.EmeraldToEmeraldConfigUpgraderStepList`.

---

2. After the upgrade, set the `DefaultApplicationLanguage` parameter in `config.xml` is to `en_US` before you install any language packs. The application upgrade process removes all languages other than English from the `LangugeType` typelist. If you removed any region when preparing for the upgrade, then you also need to set the parameter `DefaultApplicationLocale` to `en_US`.

See also

- “Setting the Default Display Language” on page 33
- “Setting the Default Application Locale for Regional Formats” on page 84

Step 3: Install the New 8.0.x Language Pack

Use the installation instructions in the release notes for your language pack or region accelerator to install these items. See also “Installing Display Languages” on page 23.

1. Make sure your application is pointing to an empty database.
2. Install the new 8.0.x language pack.
3. Install any needed region accelerator as described in the release notes for the language pack.
4. Set `DefaultApplicationLanguage` and `DefaultApplicationLocale` to the values you want them to have for production. See:
   - “Setting the Default Display Language” on page 33
   - “Setting the Default Application Locale for Regional Formats” on page 84
5. Point your application to the database you plan to use for production.

After completing these steps, in most cases you can start the new version of the server with the language pack installed.

Additional Steps for Other Modified files

In some cases, Guidewire modifies files other than `*.display.properties` files between different versions of language or region packs. The changed files can include system table files, help files, workflow, Gosu, and PCF files. If you have modified any of these files in the configuration module and want to use the upgraded version of the file from the new language pack, do the following:

1. Locate the files that have changed by comparing the files in the old and new language pack.
2. Restore any files that you modified or customized back to the base configuration version.

Note: Performing this procedure will overwrite any custom changes that you have made to these files.

The following list shows the files other than the `*.display.properties` files that the language pack can potentially provide:

```
PolicyCenter/modules/configuration/...
  config/resources/bop_class_codes.xml
  config/resources/class_code_basis.xml
  config/resources/cp_class_codes.xml
  config/resources/gl_class_codes.xml
  config/resources/industry_codes.xml
  config/resources/risk_classes.xml
  config/resources/uw_issue_types.xml
  config/import/chargepatterns.xml
  config/web/pcf/exitpoints/Help.pcf
  config/workflow/*.xml
  gsrc/gw/sampledata/SampleQuestions.gs
  gsrc/gw/sampledata/SampleExchangeRates.gs
  gsrc/gw/api/address/AddressFormatter.gs
  gsrc/gw/api/name/NameFormatter.gs
  webresources/{language}/themes/Titanium/pages/ServerFailedToStart.html
  webresources/{language}/themes/Titanium/pages/ServerDown.html
  webresources/themes/Titanium/help/shortcuts.html
  webresources/themes/Titanium/help/index.html
```

Setting the Default Display Language

You must install the language that you want to be your application default and set `DefaultApplicationLanguage` in `config.xml` before you start your PolicyCenter server for the first time.

IMPORTANT You can install additional display languages later, but you cannot change the default application language.

The value that you set for `DefaultApplicationLanguage` must be a typecode in the `LanguageType` typelist. If you set the value of parameter `DefaultApplicationLanguage` to a value that does not exist as a `LanguageType` typecode, the application server refuses to start. The language pack installer adds a typecode for the installed language automatically to the `LanguageType` typelist.

See also

- "Globalization Parameters" on page 57 in the *Configuration Guide*

Selecting a Personal Language Preference

Users of PolicyCenter can choose a preferred display language from the Options menu by navigating to **International** → **Languages**. Language choices are available only for installed languages. A user's language preference overrides the default application language that you set system-wide with parameter `DefaultApplicationLanguage` in `config.xml`. A user's choice for preferred display language persists between logging out and logging in again.

See also

- "Setting the Default Display Language" on page 33
- "Selecting Language and Regional Formats in PolicyCenter" on page 13

chapter 3

Localized Printing

Generating PDF documents in languages other than U.S. English typically requires additional configuration of your system and of Guidewire PolicyCenter.

This topic includes:

Printing Specialized Character Sets and Fonts

PDF document generation in a specialized character set for languages other than U.S. English typically requires additional configuration. Adobe PDF (Portable Document Format) provides a set of fonts that are always available to all PDF viewers. This set of fonts includes the Courier, Helvetica, and Times font families and several symbolic type fonts. In some cases, however, it is possible that you need to download and install specialized font families to handle specific languages, such as Japanese.

Guidewire does not provide fonts for use with Guidewire products. Any fonts that you use are part of the operating system platform as provided by a specific vendor. It is the operating system vendor that defines how you can use a specific font and under what circumstances. If you have questions about the acceptable use of a specific font, contact the operating system vendor that provided the font.

In particular:

- Guidewire assumes that appropriate fonts are made available for document printing and other features of the Guidewire applications.
- Guidewire expects that document fonts are provided and supported by the operating system vendor.
- Guidewire cannot and does not guarantee any of the fonts supplied as part of an operating system platform.

If you intend to print in a language not supported by one of the standard Adobe PDF fonts:

1. Install the required font.
2. Download and install the Apache Formatting Objects Processor (FOP).

Apache FOP is a print formatter of XML objects intended primarily for generating PDF output.

3. Configure Apache FOP and Guidewire PolicyCenter for the font that you installed.

See also

- For additional information on the Apache Formatting Objects Processor, refer the following:
  `http://xmlgraphics.apache.org/fop/trunk/fonts.html`
- For Japanese fonts, refer to the following:
  `http://connie.slackware.com/~alien/slackbuilds/sazanami-fonts-ttf/pkg/12.0/`
- For information on configuring Apache FOP and PolicyCenter for specific fonts, see:
  - “Localized Printing in a Windows Environment” on page 36
  - “Localized Printing in a Linux Environment” on page 38

Localized Printing in a Windows Environment

Suppose that you want to print PDF documents in Russian.The Russian language uses the Cyrillic character set. The default font for PDF generation does not support the Cyrillic character set. Therefore, you need to customize the Apache Formatting Objects Processor (FOP) application so that it uses fonts that do support the Cyrillic character set.

Fortunately, the generic Microsoft Windows Arial TrueType font family (normal, bold, italic, bold-italic) does support Cyrillic. If you work in a Windows environment, you can simply use the Arial TrueType font. To obtain a font that supports a particular language requirement but which is not currently installed as part of your operating system, contact your operating system vendor.

The following example describes how to configure Apache (FOP) and Guidewire PolicyCenter to print documents in Russian by using Cyrillic characters in a Windows environment.

This example assumes the following:

- Apache FOP exists on your machine.
- The `fop.jar` is on the class path.
- The Arial fonts exist in `C:\WINDOWS\Fonts`.
- The fonts are TrueType fonts.

The process for non-TTF FOP supported fonts is slightly different. See the Apache FOP documentation for more information.

The example also assumes the following:

- You have a working Guidewire PolicyCenter application.
- You have installed the proper language pack for your Guidewire application.

Step 1: Download and Install Apache FOP on Windows

Download Apache FOP from the following Apache web site:

`http://xmlgraphics.apache.org/fop/download.html`

Step 2: Configure TTFReader

After you download and install Apache FOP, do the following:

1. Make a copy of `fop.bat` in the root installation directory and rename it `ttfreader.bat`.
2. Open `ttfreader.bat` and change the last line to read:

```
"%JAVACMD%" %JAVAOPTS% %LOGCHOICE% %LOGLEVEL% -cp "%LOCALCLASSPATH%"
        org.apache.fop.fonts.apps.TTFReader %FOP_CMD_LINE_ARGS%
```

   Essentially, you are changing `org.apache.fop.cli.Main` to `org.apache.fop.fonts.apps.TTFReader`. You need to do this step so that the code to generate the font metrics works correctly.

Step 3: Generate FOP Font Metrics Files

You must generate font metrics for FOP to enable it use a font.

To generate the font metrics:

1. Create the following directory:

```
C:\fopconfig
```

2. Run the following commands, which activate the Apache FOP TTFReader:

```
ttfreader.bat -enc ansi C:\WINDOWS\Fonts\arial.ttf   C:\fopconfig\arial.xml
ttfreader.bat -enc ansi C:\WINDOWS\Fonts\ariali.ttf  C:\fopconfig\ariali.xml
ttfreader.bat -enc ansi C:\WINDOWS\Fonts\arialbi.ttf C:\fopconfig\arialbi.xml
ttfreader.bat -enc ansi C:\WINDOWS\Fonts\arialbd.ttf C:\fopconfig\arialbd.xml
```

   Running these commands generates metrics files for the Arial font family and stores those metrics in the `C:\fopconfig` directory. Do not proceed until you see the following files in the `fopconfig` directory:

```
arial.xml
arialbd.xml
arialbi.xml
ariali.xml
```

Step 4: Register the Fonts with Apache FOP

You must register the fonts that you installed with Apache FOP. The following example registers the Arial font family with Apache FOP.

To create a user configuration file for FOP, perform the following steps:

1. Copy the following file into the directory `C:\fopconfig\`:

```
C:\fop_install\conf\fop.xconf
```

2. Open `fop.xconf` in an XML editor and find the `<fonts>` section. It looks similar to the following:

```
<fonts>
  ...
```

3. Enter the following font information in the appropriate place.

```
<fonts>
  <font metrics-url="c:\\fopconfig\\arial.xml" kerning="yes" embed-url="arial.ttf">
    <font-triplet name="Cyrillic" style="normal" weight="normal"/>
  </font>
  <font metrics-file="c:\\fopconfig\\arialbd.xml" kerning="yes" embed-url="arialbd.ttf">
    <font-triplet name="Cyrillic" style="normal" weight="bold"/>
  </font>
  <font metrics-url="c:\\fopconfig\\ariali.xml" kerning="yes" embed-url="ariali.ttf">
    <font-triplet name="Cyrillic" style="italic" weight="normal"/>
  </font>
  <font metrics-file="c:\\fopconfig\\arialbi.xml" kerning="yes" embed-url="arialbi.ttf">
    <font-triplet name="Cyrillic" style="italic" weight="bold"/>
  </font>
</fonts>
...
```

   If you do not want to embed the font in the PDF document, then do not include the `embed-url` attribute.

Step 5: Register FOP Configuration and Font Family with PolicyCenter

You must register your Apache FOP configuration file and font family with PolicyCenter. The following example registers the Cyrillic font family with PolicyCenter.

1. Open PolicyCenter Studio and press `Ctrl+Shift+N`, and then search for `config.xml`.
2. Open `config.xml` and set the following parameters:

Parameter	Description	Example value
`PrintFontFamilyName`	The name of the font family for the custom fonts as defined in your FOP configuration file.	Cyrillic
`PrintFOPUserConfigFile`	The fully qualified path to a valid FOP configuration file.	C:\fopconfig\fop.xconf

3. Stop and start the PolicyCenter server so that these changes can take effect.

Step 6: Test Your Configuration

After you perform the listed configuration steps, test that you are able to create and print a PDF that uses the correct font. To test the Apache FOP configuration, you must have a PolicyCenter implementation that supports the Russian locale.

Localized Printing in a Linux Environment

The following example illustrates how to configure Guidewire PolicyCenter and the Apache Formatting Objects Processor (FOP) to print Japanese characters in a Linux environment. This example includes the following steps:

The example also assumes the following:

- You have a working Guidewire PolicyCenter application.
- You have installed the proper language pack for your Guidewire application.

Before Starting

Guidewire recommends that you use a package manager to manage the download and installation of the necessary application files and packages on Linux. One such package manager is `yum`, which works with the following Linux distributions, among others:

- Fedora
- CentOS-5
- Red Hat Enterprise Linux 5 or higher

In any case, chose a package manager that works with your particular Linux distribution.

Step 1: Download and Install the Required Fonts

If it does not already exist in your operating system, you must obtain and install a font that supports the language in which you want to print.

1. Obtain a font from your operating system vendor that supports your particular language requirement.

   To print Japanese characters, for example, you need to install a font that supports Japanese characters. The following are examples of fonts that support the printing of Japanese characters:

   - IPA Gothic
   - Sanazami

2. This step depends on which package manager you are using:
   - If you are using the `yum` package manager, substitute the actual font name for *`FONT-NAME`* in the `yum install` command. For example:

     ```
     yum clean all
     yum install [FONT-NAME]
     ```

   - If you are using a package manager other than `yum`, use the install commands specific to your particular Linux distribution.

See also

- "Printing Specialized Character Sets and Fonts" on page 35

Step 2: Download and Install Apache FOP on Linux

To install Apache FOP in a Linux environment:

1. Download Apache FOP from the following Apache web site:

   ```
   http://xmlgraphics.apache.org/fop/download.html
   ```

2. Unpack the ZIP file into the desired directory, using your version of FOP in place of `fop-0.95`:

   ```
   mkdir /usr/local/fop-0.95
   cd /usr/local/fop-0.95
   cp /tmp/fop-0.95-bin.zip .
   unzip fop-0.95-bin.zip
   ```

3. Perform the following test to be certain that Apache FOP is working correctly:

   ```
   ./fop -fo examples/fo/basic/readme.fo -awt
   ```

Step 3: Configure the Font

Enter commands such as the following to generate the font configuration file. Use commands that are specific to your font. The following example is specific to the IPA Gothic font family and FOP version 0.95:

```
cd /usr/local/fop-0.95
cp /usr/share/fonts/ipa-gothic/ipag.ttf .
java -cp build\fop.jar:lib\avalon-framework-4.2.0.jar:lib
        \commons-logging-1.0.4.jar:lib\xmlgraphics-commons-1.3.1.jar:lib
        \commons-io-1.3.1.jar org.apache.fop.fonts.apps.TTFReader -ttcname
        "IPA Gothic" ipag.ttf ipag.xml
```

Step 4: Register the Font with Apache FOP

You must register the font that you installed with Apache FOP. The following example registers the IPA Gothic font family with Apache FOP version 0.95.

1. Open `fop.xconf` for editing. To use the `vi` editor, enter the following at a command prompt:

   ```
   vi conf/fop.xconf
   ```

2. Add the following lines to `fop-xconf` in the `<fonts>` section:

   Note: Use the version of FOP that you installed in place of `fop-0.95`.

```
<font metrics-url="/usr/local/fop-0.95/ipag.xml" kerning="yes"
        embed-url="/usr/local/fop-0.95/ipag.ttf">
  <font-triplet name="IPAGothic" style="normal" weight="normal"/>
  <font-triplet name="IPAGothic" style="normal" weight="bold"/>
</font>
```

Step 5: Register FOP Configuration and Font Family with PolicyCenter

You must register your Apache FOP configuration file and font family with PolicyCenter.

1. Open PolicyCenter Studio and search for `config.xml`.
2. Find the following parameters in `config.xml` and set them accordingly. The following example registers the IPA Gothic font family with PolicyCenter.

Parameter	Description	Example value
`PrintFontFamilyName`	The name of the font family for the custom fonts as defined in your FOP configuration file.	IPA Gothic
`PrintFOPUserConfigFile`	The fully qualified path to a valid FOP configuration file.	`/usr/local/fop-0.95/fopconfig/` **Note:** Use the version of FOP that you installed in place of `fop-0.95`.

3. Stop and start the PolicyCenter server so that these changes take affect.

Step 6: Test Your Configuration

After you perform the listed configuration steps, Guidewire recommends that you test that you are able to create and print a PDF file that uses the correct font. To test the Apache FOP configuration, you must have a PolicyCenter implementation that supports the Japanese locale.

chapter 4

Localizing PolicyCenter String Resources

This topic describes how to localize the following string resources that PolicyCenter displays in the application user interface:

- **Display keys** – Strings to display as field and screen labels and interactive error messages
- **Typecodes** – Strings to display as choices in drop-down lists
- **Gosu error messages** – Strings to display as Gosu error and warning messages in Studio
- **Workflow step names** – Strings to display as individual step names in workflow processes
- **Product model string resources** – Strings to display in screens that use product models

  Note: Ruleset names and descriptions are not localized as strings. See “Localizing Rule Set Names and Descriptions” on page 56.

This topic includes:

See also

About String Resources

PolicyCenter uses string resources for the following:

- **Display Keys** – Strings to display as field and screen labels and interactive error messages
- **Typecodes** – Strings to display as choices in drop-down lists
- **Workflow Step Names** – Strings to display as individual step names in workflow processes
- **Product model string resources** – Strings to display in screens that use product models

You can extract these string resources from PolicyCenter and localize them separately from other application resources.

This topic includes:

See also

- "Localizing Product Model String Resources" on page 51

Display Keys

PolicyCenter stores as *key/value* pairs the United States English string resources from which it generates field and screen labels and interactive error messages in the user interface. Guidewire calls these particular key/value pairs *display keys*. You specify the key/value pair of a display key in standard Java properties syntax. For example:

```
Admin.Workload.WorkloadClassification.General = General
```

PolicyCenter stores the key/value pairs for display keys in a file called `display.properties`. In the base configuration, PolicyCenter contains a single copy of `display.properties`, with string resources in United States English. In Guidewire Studio, you can navigate to this `display.properties` file in the **Project** window as follows:

configuration → **config** → **Localizations** → **en_US**

If you install a language pack for a language other than U.S. English, the installer adds a version of `display.properties` localized for that language.

See also

- "Localizing Display Keys" on page 45.
- "Installing Display Languages" on page 23.

Typecodes

PolicyCenter stores as *key/value* pairs the U.S. English string resources from which it displays choices and choice descriptions in drop-down lists in the user interface. Guidewire calls these particular key/value pairs *typecodes*. You specify the key/value pairs for the name and description of a typecode in standard Java properties syntax. For example:

```
TypeKey.CoverageType.CPBldgCov = Building Coverage
TypeKeyDescription.CoverageType.CPBldgCov = Building Coverage
```

PolicyCenter stores the key/value pairs for typecodes in a file called `typelist.properties`. In the base configuration, PolicyCenter contains a single copy of `typelist.properties`, with string resources in U.S. English. In Guidewire Studio, you can navigate to the `typelist.properties` file in the **Project** window, as follows:

configuration → **config** → **Localizations** → **en_US**

If you install a language pack for a language other than U.S. English, the installer adds a version of `typelist.properties` localized for that language.

See also

- "Localizing Typecodes" on page 47
- "Installing Display Languages" on page 23

Workflow Step Names

In PolicyCenter, it is possible to provide localized versions for the names of individual steps in a workflow process. It is also possible to set a specific language and set of regional formats on each workflow.

See also

- "Localizing Guidewire Workflow" on page 65
- "Localizing Workflow Step Names" on page 66

Exporting and Importing String Resources

PolicyCenter enables you to export some string resources to an external file, including the following:

- **Display Keys** – Strings to display as field and screen labels and interactive error messages
- **Typecodes** – Strings to display as choices in drop-down lists
- **Workflow Step Names** – Strings to display as individual step names in workflow processes

By exporting and importing string resources, you can make all your translations directly in a single file. PolicyCenter provides separate commands for exporting and importing string resources.

Command	Related topic
`gwpc export-l10ns [-Dexport.file] [-Dexport.locale]`	"Exporting Localized String Resources with the Command Line Tool" on page 43
`gwpc import-l10ns [-Dimport.file] [-Dimport.locale]`	"Importing Localized String Resources with the Command Line Tool" on page 44

The commands provide parameters that you can use to specify the locations of the export and import files and a language-specific set of string resources to export and import. The export and import files are in the format of Java property files.

See also

- For information on using Java property files in PolicyCenter, see "Properties Files" on page 397 in the *Gosu Reference Guide*.

Exporting Localized String Resources with the Command Line Tool

Guidewire provides a command line tool to manually export certain string resources from PolicyCenter. The command exports the following strings resources as name/value pairs:

- Display keys
- Typecodes
- Workflow step names

The export file organizes the strings into translated and non-translated groups. The command provides parameters that enable you to specify the location of the export file and a language-specific set of string resources to export.

To run the export tool

1. Ensure that the application server is running.
2. Navigate to your application installation `bin` directory, for example:
   ```
   PolicyCenter/bin
   ```
3. Run the following command:
   ```
   gwpc export-l10ns -Dexport.file=targetFile -Dexport.locale=localizationFolder
   ```

-Dexport.file Command Parameter

Command line parameter `-Dexport.file` specifies *`targetFile`*, the name of the file in which PolicyCenter saves the exported resource strings. You must add the file extension to the file name. By default, PolicyCenter puts the export file in the root of the installation directory. You specify the directory as follows:

- To leave the export file in the same location, enter only the name of the file to export.
- To save the file in a different location, enter either an absolute path or a relative path to the file from the root of the installation directory.

-Dexport.locale Command Parameter

Command line parameter `-Dexport.locale` specifies *`localizationFolder`*, the target localization folder for the translated strings. The localization folder name must match a PolicyCenter language type that exists in the `LanguageType` typelist. For example: `fr_FR` or `ja_JP`.

See also

"Importing Localized String Resources with the Command Line Tool" on page 44

Importing Localized String Resources with the Command Line Tool

PolicyCenter provides a command-line tool to import localized string resources that you previously exported. The command imports the following string resources as key/value pairs:

- Display keys
- Typecode names and descriptions
- Workflow step names

The command provides parameters that enable you to specify the location of the import file and a language-specific set of string resources to import.

To run the import tool

1. Ensure that the application server is running.
2. Navigate to your application installation `bin` directory, for example:
   ```
   PolicyCenter/bin
   ```
3. Run the following command:
   ```
   gwpc import-l10ns -Dimport.file=sourceFile -Dimport.locale=localizationFolder
   ```

-Dimport.file Command Parameter

Command line parameter `-Dimport.file` specifies *`sourceFile`*, the file that contains the translated resource strings. It must be in the same format as a file exported from PolicyCenter. By default, PolicyCenter puts the export file in the root of the installation directory. You can set the directory as follows:

- To leave the import translation file in the same location, you need enter only the name of the file to import.
- To move the translation file to a different location, you must enter an absolute or relative path to the file from the root of the installation directory.

-Dimport.locale Command Parameter

Command line parameter `-Dexport.locale` specifies *`localizationFolder`*, the name of the localization folder into which to save the translated strings. The localization folder name must match a PolicyCenter language type that exists in the `LanguateType` typelist, such as `fr_FR` or `ja_JP`.

See also

"Exporting Localized String Resources with the Command Line Tool" on page 43

Localizing Display Keys

PolicyCenter initializes the display key system as it scans all localization nodes in all modules for the display key property files `display.properties`.

In the base configuration, PolicyCenter provides a single `display.properties` file. In the Studio **Project** window, this file is located in:

configuration → **config** → **Localizations** → **en_US**

The **Localizations** folder contains multiple folders with localization names such as **de_DE**, each of which can also contain, after configuration, additional property files. In addition, any language pack that you install also contains property files.

It is possible to provide translated display keys in either of the following ways:

Translation technique	Related topic
Using the Studio **Display Keys** editor	"Localizing Display Keys by Using the Display Key Editor" on page 46
Using the display key import and export tools	"Exporting and Importing String Resources" on page 43

Note: Many of the display keys you localize are string values that PolicyCenter displays as labels or messages. There are no restrictions on the translated text for these types of display keys. You can also localize QuickJump commands in `display.properties`. When you localize a QuickJump command, there must not be any spaces in the translated text. For example, the base configuration display key `Web.QuickJump.RunBatchProcess = RunBatchProcess` must be translated into a term with no spaces. See also:
"Adding a QuickJump Navigation Command" on page 120 in the *Configuration Guide*
"QuickJump" on page 53 in the *Application Guide*

See also

- For more information on display keys and string resources in general, see "Display Keys" on page 42.
- For information on the use of property files in Guidewire PolicyCenter, see "Properties Files" on page 397 in the *Gosu Reference Guide*.

PolicyCenter and the Master List of Display Keys

Using the localization node property files, on startup, PolicyCenter generates a master list of display keys for use in the user interface. For each property file, PolicyCenter loads the display keys and adds each display key to the master list under the following circumstances:

1. The master list does not already contain the display key.
2. The master list already contains the display key, but, the display key in the master list has a different number of arguments than the display key to add. If this is the case, then PolicyCenter logs a warning message noting that it found a display key value with different arguments in different regions. For example:

```
Configuration Display key found with different argument lists across locales: Validator.Phone
```

As PolicyCenter creates the master display key list, it scans the application localization folders in the following order for copies of file `display.properties`:

- The application default localization folder, as set by configuration parameter `DefaultApplicationLanguage`
- All other localization folders configured for use by the server
- The Guidewire default localization folder (`en_US`)
- All remaining localization folders

After PolicyCenter creates the master list of display keys, the application checks the display keys for the default region against the master list. PolicyCenter then logs as errors any display keys that are in the master list but missing from the default application region. For example:

```
ERROR Default application locale (en_US) missing display key: Example.Display.Key
```

Because the error message returns the display key name, you can use that name to generate a display key value in the correct localization folder.

Localizing Display Keys by Using the Display Key Editor

It is possible to enter a localized version of a display key directly in the Studio editor. To access the editor, first, in Studio, navigate to **configuration** → **config** → **Localizations**. Expand the node for the target language and open the `display.properties` file in that folder. You can also press `Ctrl+Shift+N` to find a property file for a specific region.

It is not necessary to use Studio to localize display keys. If you have a large number of translated strings to enter, you can use the export and import commands. With these commands, you export the strings, translate them, and import the translated strings into Studio. See “Exporting and Importing String Resources” on page 43.

Identifying Missing Display Keys

Guidewire provides a display key difference tool that does the following:

- Compares each language configured on the server against the master display key list.
- Generates a file that contains a list of any missing keys.

To generate a display key difference report, run the following command from the application `bin` directory:

```
gwpc displaykey-diff
```

The `displaykey-diff` tool creates a new build directory under the application root directory, for example:

```
PolicyCenter/build
```

If the tool detects that an installed language has missing display keys:

- The tool creates a subdirectory for that language using the localization code for that language to name the subdirectory.
- The tool populates that subdirectory with a `display.properties` file containing the list of missing keys.

Each `display.properties` file contains a list of display keys that are in the master list but not in that localization node. The format of the file is exactly the same as the display key configuration files. For example, the following code illustrates the contents of the file for `en_US`:

```
#
# Missing display keys for locale
# en_US
#
Web.QA.I18N.ContactDetail.AddressDetail.City = Suburb
Web.QA.I18N.ContactDetail.AddressDetail.ZipCode = Postcode
```

Note: PolicyCenter does not generate a `display.properties` file for a region that does not have any missing display keys.

Working with Display Keys for Later Translation

It is possible to create a display key in a specific localization folder that is not actually localized yet. This display key is simply a placeholder string for a display key that you intend to localize at some point. If you create one of these *to-be-translated* display keys, then Guidewire recommends that you add a suffix of [TRANSLATE] to each display key that you create as a placeholder. For example:

Actions [TRANSLATE]

The suffix can be any string that is meaningful. Use the same string in all cases to make it easy to find the placeholder display keys. Using a string such as [TRANSLATE] makes it easy to see the string in the PolicyCenter interface. It also makes it easy for users to understand that the display key has not yet been translated.

It is important to use the same tag for all the placeholder strings so that you do not miss any during a search.

Localizing Typecodes

You can provide localized typecodes for a typelist in the following ways:

- **Using the gwpc export and import commands** – Use these commands to localize all typecodes by editing a single text file. See “Localizing Typecode Names by Using the gwpc Export and Import Commands” on page 47.
- **Using the Typelist Localization editor** – Enter localized values for individual typecodes directly through the Typelist editor. See “Localizing Typecodes by Using the Typelist Localization Editor” on page 48.
- **Editing the `typelist.properties` file in a localization folder** – Navigate to a localization folder and open the `typelist.properties` file so that you can edit it.

You can also specify a sort order for each typelist, by language. See “Setting Localized Sort Orders for Localized Typecodes” on page 49.

There is also Gosu syntax for accessing typecodes that you need to be aware of. See “Accessing Localized Typekeys from Gosu” on page 50.

Localizing Typecode Names by Using the gwpc Export and Import Commands

You can use the `gwpc` import and export commands to create a single file of localizable strings, including the names of typecodes for all typelists.

The export command has the following syntax:

```
gwpc export-l10ns -Dexport.file=targetFile -Dexport.locale=localizationFolder
```

There is a full description of this command at “Exporting Localized String Resources with the Command Line Tool” on page 43.

The exported file has all the typecodes in it, which cannot be guaranteed for any given `typelist.properties` file. The typecodes are separated into translated typecodes and untranslated typecodes, which makes it easier to see which ones you need to translate. The translated typecodes are extracted from the existing `typelist.properties` file in that localization folder, if there is one.

Each typecode begins with the keyword `TypeKey`, followed by the typelist name, and then the typecode.

For example, the following code example shows some of the typecodes in the untranslated and translated sections of an exported file:

```
#untranslated keys
TypeKey.AccidentType.01=Contact with chemicals
TypeKey.AccidentType.02=Contact with hot objects or substances
TypeKey.AccidentType.03=Contact with temperature extremes
TypeKey.AccidentType.04=Contact with fire or flame
...
#already translated
...
TypeKey.CoverageType.BABobtailLiabCov=Bobtail Liability
TypeKey.CoverageType.BACollisionCov=Collision
TypeKey.CoverageType.BACollisionLimited_MAMI=Collision - Limited Coverage
TypeKey.CoverageType.BAComprehensiveCov=Comprehensive
TypeKey.CoverageType.BADealerLimitLiabCov=Auto Dealer Limited Liability
...
```

The untranslated section shows typecodes of the `AccidentType` typelist. The translated section shows typecodes of the `CoverageType` typelist, which were defined in the `typelist.properties` file in the `en_US` localization folder at the time the file was exported. Because this file was exported from the `en_US` localization folder, all typecode names are in U.S. English.

After you make your translations, import the translated file into the appropriate localization folder.

The import command has the following syntax:

```
gwpc import-l10ns -Dimport.file=sourceFile -Dimport.locale=localizationFolder
```

There is a full description of this command at “Importing Localized String Resources with the Command Line Tool” on page 44.

All strings in the file that you import, whether translated or not, are saved in property files in that folder. PolicyCenter extracts the typecode strings and updates the `typelist.properties` file for that folder.

Note: The imported file can also contain strings other than typecodes. Those strings are also extracted and saved in their appropriate property files, such as `display.properties`.

Localizing Typecodes by Using the Typelist Localization Editor

Note: You can localize typecodes in the Typelist Localization editor only for languages that you configured in PolicyCenter. See “Installing Display Languages” on page 23.

You can use the Typelist Localization editor to localize typecodes. This technique is labor intensive, but can be useful if you want to localize just a few typecodes. For example, you have created some extension typecodes that you want to localize. When you edit typecodes in the Typelist Localization editor, PolicyCenter adds them to the associated `typelist.properties` file.

To localize typecodes by using the Typelist Localization editor in Studio

1. Type `Ctrl+Shift+N` in the Studio **Project** window and enter the name of the typelist that you want to localize.
2. Select a typecode for which you want to provide a localized version.
3. Click the **Localization** link in the bottom right corner of the Typelist editor.
4. Find the group with the language code for which you want to provide localized strings.
5. Enter localized strings for any or all of the following:
   - **name** – The natural language name associated with this typecode.

- **desc** – The description of this typecode.

For example, you installed the French language pack, so you see the **fr** language code. Defining the French language version of the **name** field for a typecode updates the `typelist.properties` file in **configuration** → **config** → **Localizations** → **fr**. If necessary, PolicyCenter creates a file in this location and then adds the entry.

Editing the typelist.properties file

The `typelist.properties` file in a localization folder provides localized definitions for typecodes for the language represented by the folder. This file does not necessarily have definitions for all typecodes. You might edit this file, for example, to override or add to the translations in the `typelist.properties` file saved by the language pack installer.

As you use either of the other techniques to update typecode localizations, this file is also updated.

- If you want to ensure that all typecodes are localized for a language, use export and import. See "Localizing Typecode Names by Using the gwpc Export and Import Commands" on page 47.
- If you want to edit just a few typecodes, you can use the Typelist Localization editor. See "Localizing Typecodes by Using the Typelist Localization Editor" on page 48

To edit a typelist.properties file

1. Open the Studio **Project** window.
2. Navigate to the localization folder and double-click the file to open it.

   For example, navigate to **configuration** → **config** → **Localizations** → **en_US** and double-click `typelist.properties`.

   Note: If there is no `typelist.properties` file in the folder, you can copy one in or right-click the folder and create a new file.
3. Edit the file, and then save it when you are finished.

Setting Localized Sort Orders for Localized Typecodes

You can set the sort order for typecodes in a typelist for specific languages in Guidewire Studio. Create a file that has the name of the typelist with a file extension of `.sort`. Then save the file in the localization folder for the language to which the sort order applies, such as **configuration** → **config** → **Localizations** → **en_US** for U.S. English. PolicyCenter stores the sort order information by language in the typelist table.

In the file, list the typecode display names in the appropriate language, one per line, in the order in which you want them to appear. Lines that are not typecode display names or are duplicate typecode display names are reported as errors in the log file. You can also add comments on separate lines. Each comment line must start with two slash characters, //.

A typical use for a `.sort` file is to support Japanese with other languages on the same server. For example, you might want to provide a sort order for Japanese prefectures, which customarily are in order from north to south—Hokkaido, Aomori, Iwate, so on. Typically, instead of using `.sort` files, typecode order is determined by the priority defined in the typelist and by the sort order of the typecode display name. If a `.sort` file is present for a language, then the typekey priority is not used to determine sort order for that language. See "Determining the Order of Typekeys" on page 164.

In the base configuration, the `State` typelist uses typekey priority to make the Japanese prefectures appear in the north-to-south order. Hokkaido has priority 1, Aomori has priority 2, Iwate has priority 3, and so forth. However, English-speaking non-Japanese users might expect to see the prefectures listed in alphabetic order. You can get this result by adding a `State.sort` file to the `ja_JP` localization folder at **configuration** → **config** → **Localizations** → **ja_JP**. In that file, list the prefectures in Japanese and in north-to-south order. Then do either of the following:

- Remove the priorities for the prefectures from `State.ttx`
- Create an empty `State.sort` file in the en_US localization folder

The following example is not a likely one and is included only to demonstrate the elements of the file. It applies to typecodes from the `State` typelist. The file causes some U.S. western states to be listed in the order specified, and first in any U.S English list of states in the PolicyCenter user interface. The other lines in the file do not affect the list of states because they start with //. The file is named `State.sort` and is intended for U.S. English, and therefore is saved in the folder **configuration** → **config** → **Localizations** → **en_US**:

```
// sort order definition for some U.S. western states
Washington
Oregon
California
Nevada
Utah
Arizona
// Other states will be listed in
// alphabetical order after Arizona.
```

Any typecodes in the typelist that are not in the `.sort` file are listed after the typecode display names listed in the `.sort` file. These typecodes are ordered according to the sort order specified in the `language.xml` file for that language.

PolicyCenter does not provide any sort order files in the base configuration. You must put any `.sort` file that you create in the appropriate localization folder.

IMPORTANT Any change that you make to a typelist sort order file triggers a database upgrade.

Accessing Localized Typekeys from Gosu

Gosu provides three `String` representations that you can use for typekeys.

Typekey property	Description
typelist.typekey.Code	String that represents the typecode
typelist.typekey.DisplayName	Localized language version of the entity name
typelist.typekey.UnlocalizedName	Name listed in the data model

For example, to extract localized information about a typekey, you can use the following:

```
var displayString = myTypekey.DisplayName
```

The following code is a more concrete example.

```
print(AddressType.TC_BUSINESS.DisplayName)
```

It is important to understand that the display key reference acts more as a function call rather than as a value. If the language setting for the user changes, then the display key value changes as well. However, the value stored in `displayString` does not automatically change as the language changes.

Localizing Gosu Error Messages

Similar to the key/value paris stored in `display.properties`, PolicyCenter stores the string resources defining Gosu errors in the file `gosu.display.properties`. These strings display in Guidewire Studio if there is an error in Gosu code or if code is being compiled and an error occurs. Do not add your own strings to this file.

In the base configuration, Guidewire provides only a single U.S. English version of this file. In Studio, you can see this file at the following location in the **Project** window:

configuration → **config** → **Localizations** → **en_US**

File `gosu.display.properties` provides the key/value pairs like the following ones:

```
ARRAY = Array
BEAN = Bean
BOOLEAN = Boolean
```

```
DATETIME = DateTime
FUNCTION = Function
IDENTIFIER = Identifier
METATYPENAME = Type
NULLTYPENAME = Null
NUMERIC = Number
OBJECT_LITERAL = ObjectLiteral
STRING = String
MSG_BAD_IDENTIFIER_NAME = Could not resolve symbol for : {0}
...
```

If you install a new language pack, that language pack contains a translated version of file `gosu.display.properties` in the target language.

See also

- "Localizing Display Keys" on page 45
- "Localizing Typecodes" on page 47

Localizing Product Model String Resources

Note: You manage most product model functionality through Guidewire Product Designer.

Guidewire PolicyCenter uses many of the data fields in the PolicyCenter data model to display information in the PolicyCenter interface. These data fields are string resources that symbolically represent business data configuration used in various places in Guidewire PolicyCenter. These data fields are the same key/value pairs used in other application property files.

Guidewire stores these string resources in file `productmodel.display.properties`. By translating the contents of this file, it is possible to provide translated versions of the following:

- All product model Name and Description fields
- All question Name and Description fields
- All question Text and Failure message fields

In the base configuration, Guidewire provides only a single U.S. English version of this file, in the following location in the Studio **Project** window:

configuration → **config** → **Localizations** → **en_US**

Every additional Guidewire language pack that you install also installs another version of `productmodel.display.properties` with the string resources in the target language.

See "Localizing PolicyCenter String Resources" on page 41 for details of working with display string property files.

This topic includes:

- "Translating Product Model Strings in Product Designer" on page 51
- "Localizing Coverage Term Options" on page 52

Translating Product Model Strings in Product Designer

To facilitate translating product model strings in Product Designer, use the **Display Key Values by Locale** dialog box. You access this dialog box through the **Translate** icon that appears at the end of selected product model fields. In this dialog, you can add a translated version of the string label for any defined language in PolicyCenter.

Product Designer adds any change that you make to the product model field labels to the active change list. Product Designer pushes the active change list back to the application configuration files after you commit your change. See "Working with Change Lists" on page 18 in the *Product Designer Guide* for a discussion of how change lists work in Product Designer.

For you to see the field label change in PolicyCenter, you must manually push the change list back to PolicyCenter.

Localizing Coverage Term Options

Coverages in the PolicyCenter product model can have *coverage terms*. A coverage term is a statement or a value that defines the extent or limit of the coverage. A *coverage term option* is one of a set of coverage terms pertaining to a specific coverage. Coverage term options can be textual or numeric, but are always stored as strings. You can localize coverage term options in Product Designer.

Textual coverage terms options are descriptions that help classify a limitation or extent of the coverage. Examples of textual coverage term options include:

- Class 1 Employees
- Class 2 Employees
- Alaska Attorney Fees Limit

Numeric coverage term options include single values and packages. A *package* is a set of values selected as a single item. Examples of numeric coverage term options include:

- 10,000
- 250K
- $20,000/$50,000/$10,000

The string value that users see after selecting a numeric coverage term option is stored in the **Description** of the option. The actual numeric value represented by this description is stored in the **Value** of the option. For example:

- Description = “$12,500”
- Value = 12500

Because they are stored as strings, numeric coverage term option display values often include separators, decimal points, and currency symbols. And because they are stored as strings, these separators, decimal points, and currency symbols are not influenced by regional settings. Furthermore, the sample coverage term option values that Guidewire provides in the PolicyCenter base configuration are potentially suitable only for a North American deployment.

Therefore, to localize numeric coverage term options, you must:

- Choose coverage term option values that are suitable for your target region.
- Localize any currency symbols, separators, or decimal points to suit your target region.

chapter 5

Working with a Localized Studio

This topic describes how to view the Guidewire Studio editors in a language other then English.

This topic includes:

Localization in the Guidewire Studio User Interface

Guidewire Studio is built on JetBrains IntelliJ IDEA. JetBrains does not support localization of IntelliJ. Guidewire does provide, however, limited localization support for the Guidewire plugins that comprise Guidewire Studio in IntelliJ. Guidewire provides translations for the Studio editors for a limited number of languages only.

To view the Guidewire Studio editors in a different language, you must install both the following:

- A Guidewire application language pack for the target language
- A Guidewire Studio language pack for the target language

Installing an application language pack provides translations for application resources such as field labels in the user interface. You also see some of these translated elements in the Studio PCF editor. Additionally, the application language packs provide translation for Gosu error messages in Studio.

In particular:

- A Guidewire language pack installs a `gosu.display.properties` file that provides translated Gosu warning and error messages in Guidewire Studio.

- A Guidewire language pack installs `display.properties` and `typelist.properties` files that provide translations of strings in the PolicyCenter user interface. These language pack files also affect PCF files that you open in the PCF editor. In those files, you see translated strings for all the PCF elements that use display keys, such as labels, buttons, and so on.

Installing a Studio language pack provides translations for some of the labels and controls in the Guidewire editors. Not all labels and controls can be translated due to the limitations of IntelliJ. Therefore, it is possible to see a mixture of English and non-English labels and controls in the Studio user interface.

See also

- "Specifying a Language for Studio" on page 54
- "Installing Display Languages" on page 23

Specifying a Language for Studio

You can specify a display language for Guidewire Studio either from the command line or in the **Settings** dialog after Studio starts up. As described in the previous topic, specifying a display language affects only certain aspects of the Guidewire Studio menus and windows.

This topic includes:

- "Specifying a Language for Studio on the Command Line" on page 54
- "Specifying a Language for Studio in the Settings Dialog" on page 55

Specifying a Language for Studio on the Command Line

When you start Guidewire Studio from the command line, you can use two command-line options to specify the language and locale. You must use both options.

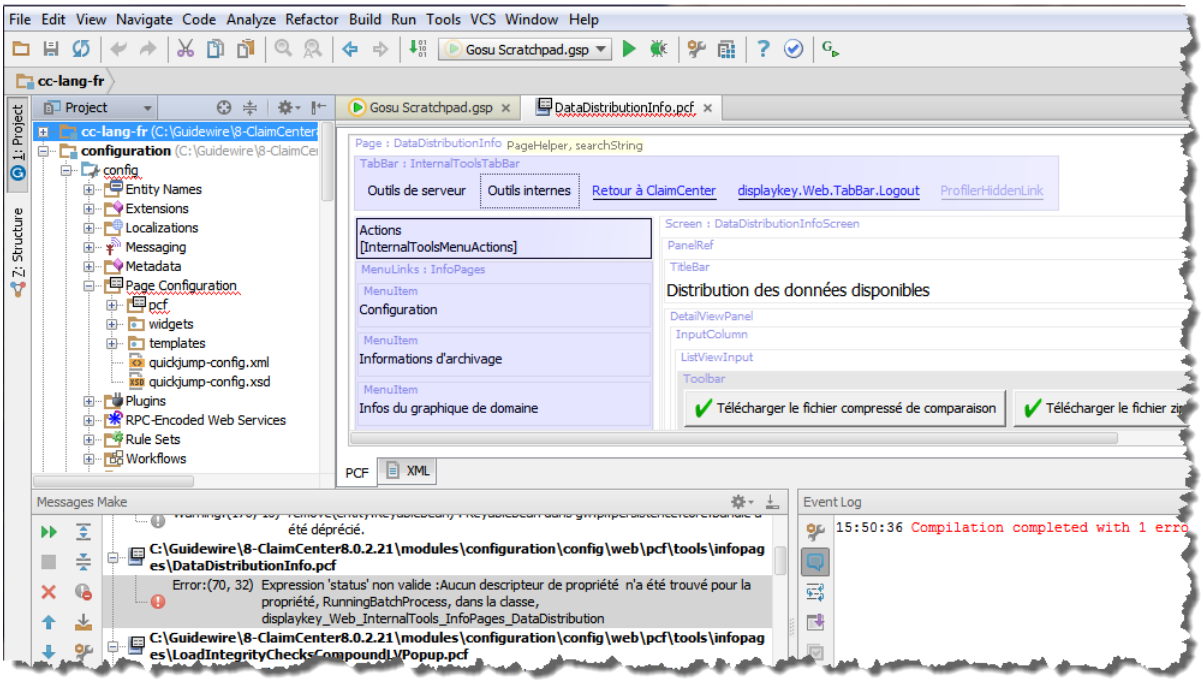
> **Note:** You must have installed the language pack for that language, and, if there is a language pack for the Studio editors, you must have installed it as well.

The following example shows how to specify French as the language for Studio.

To specify French as the language to use in PolicyCenter Studio

1. Open a command line in *PolicyCenter*`/bin`, and then enter the following command:
   ```
   gwpc studio -Dstudio.language=fr -Dstudio.country=FR
   ```
2. In Studio, navigate to **File** → **Settings** → **Appearance**.
3. Change the **Theme** setting to a value that works for your operating system.
   - If you are on Microsoft Windows, change the theme to **Windows**.
   - If you are on Linux, change the theme to **Nimbus**.
4. You see Gosu error and warning messages in French. If you open a PCF file in the editor, its labels, button names, and other strings that use display keys in are in French. All other Studio messages, labels, menus, and so on continue to be in English.

The following figure shows ClaimCenter Studio with translated Gosu messages and PCF fields:



Specifying a Language for Studio in the Settings Dialog

In Guidewire Studio, you can change to a different display language if you have installed the Guidewire Studio language pack and the corresponding Guidewire application language pack for that language.

To select a display language for Guidewire Studio

1. In Studio, navigate to **File** → **Settings** → **Guidewire Studio**.
2. At the top of the **Guidewire Studio** settings page under **Language Settings**, click **Language**.
3. In the drop-down list that opens, choose a language.

   Note: The list of languages is determined by the application language packs that are installed. If you choose a language for which there is no Studio language pack installed, you see translated strings for that language only in the PCF editor and Gosu messages. The menus, tree nodes, and so on will all be shown in English. If there is a Studio language pack installed for the language that you choose, you also see translated menu items, tree nodes, and so on.

See also

- “Localization in the Guidewire Studio User Interface” on page 53
- “Installing Display Languages” on page 23

Viewing Unicode Characters in Studio

To configure Guidewire Studio to display Unicode characters:

1. In Studio, navigate to **File** → **Settings** → **Appearance**.
2. Change the **Theme** setting to a value that works for your operating system.
   - If you are on Microsoft Windows, change the theme to Windows.
   - If you are on Linux, change the theme to Nimbus.

3. Click **OK**.

You can define and view Unicode characters for the following elements in Studio:

- Database column descriptions, but not names
- Gosu identifiers, such as class names, method names, and variable names
- Gosu comments
- Typekey/Typecode names and descriptions, but not codes
- XSD element and attribute names
- Web service method names

You cannot define Unicode values for the following elements visible in Studio:

- Database column names, which are limited to a-z, A-Z, 0-9, and underscore
- PCF file names or the ID property
- Typekey and Typecode codes

Note: Guidewire does not supply fonts for your system. You must download the fonts for the languages that you want to use in Guidewire Studio. Microsoft Windows has downloadable support for many languages. For example, if you install support for Korean, Chinese, or Japanese, the downloaded files include unicode character support for the characters used by those languages. Linux might require special font downloads. For example, for Japanese fonts, see:

`http://connie.slackware.com/~alien/slackbuilds/sazanami-fonts-ttf/pkg/12.0/`

Localizing Rule Set Names and Descriptions

In Studio, it is possible to show rule names, rule set names, and rule set descriptions in a language other than English. To display a rule name or a rule set name and description in another language, you can translate these items in the definition file for that rule or rule set.

In PolicyCenter Studio, to access the rule sets, navigate in **Project** window to:

configuration → **config** → **Rule Sets**

In the application directory structure, Guidewire stores rule-related files in the following location:

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

The following list describes the various rule file types and what you can localize in each file type.

File type	Rule type	Translatable unit	Example
`.grs`	Rule set	Rule set name	`@gw.rules.RuleName("Translated rule set name")`
		Rule set description	`@gw.rules.RuleSetDescription("Translated description")`
`.gr`	Rule	Rule name	`@gw.rules.RuleName("Translated rule name")`

To modify these files, open them in a text editor in your system directory structure and not in Studio.

You can view only a single language translation of a rule set name or description in Studio. You cannot provide multiple translations at once, as you can with string translations.

Setting the IME Mode for Field Inputs

IME mode controls the state of an input method editor (IME) for entering Japanese or Chinese language characters. PolicyCenter provides three states for `imeMode`, which you set at the field level in Guidewire Studio. The three possible states are:

State	Description
Active	• Japanese – Turns on the last entry type that you selected. For example, Hiragana or full-width Katakana,. • Chinese – Turns on pinyin entry
Inactive	• Japanese – Turns off Roman entry • Chinese – Turns on Roman entry
Not selected	Does nothing – Leaves the currently-selected IME mode as set

You set the `imeMode` at the field level on a PCF file. For a rōmaji field in Japanese, for example, you might set `imeMode` to `inactive`. Then, on the next field, one that needs Kanji entry, you would set `imeMode` to `active`, and so on for the various fields in the PCF file. In general, you set `imeMode` on text input fields and possibly drop-down fields such as typelists and range input fields.

It is important to understand that PolicyCenter does not actually set the actual IME mode. PolicyCenter merely turns IME on or off for each field. PolicyCenter cannot dictate that a certain field must contain Zenkaku Katakana and a different field must contain Hiragana. The choice of input conversion style is left to the user.

IME and Text Entry

Guidewire applications support Unicode characters by default in the base configuration. However, certain PCF entry fields do not support using IME to enter Unicode characters. Many of the PolicyCenter application screens capture keystrokes as a user enters data. This capture mechanism does not work properly if you use IME to input data directly into input fields.

In particular, if you use IME to input data instead of keystrokes, the following entry field types do not process character data properly:

- Text entry fields that implement `inputMask` functionality.
- Text entry fields that implement `maxChars` functionality, especially `TextArea Input` fields.

  Note: Unlike `TextBox` fields, there is no built-in browser support for `TextArea Input` fields.

Setting a Language for a Block of Gosu Code

It is possible to set a specific language in any Gosu code block by wrapping the Gosu code in any of the following methods.

```
gw.api.util.LocaleUtil.runAsCurrentLanguage(alternateLanguage, \ -> { code } )

gw.api.util.LocaleUtil.runAsCurrentLocaleAndLanguage(
    alternateLocale,
    alternateLanguage,
    \ -> { code } )
```

Note: The second method sets both region and language at the same time. This topic covers only setting the language.

A typical use of this feature is to override the current language that the Gosu code block uses by default. The default current language is specified by the current user or the `DefaultApplicationLanguage` parameter in `config.xml`. If neither is set, PolicyCenter uses the browser language setting.

The parameters for the methods are:

Parameter	Description
`alternateLocale`	An object of type `ILocale` that represents a regional format from the `LocaleType` typelist. Specify a `GWLocale` object for this parameter.
`alternateLanguage`	An object of type `ILocale` that represents a language from the `LanguageType` typelist. Specify a `GWLanguage` object for this parameter.
`\ -> { code }`	A Gosu block as a `GWRunnable` object—the Gosu code to run in a different locale or language

Specifying an ILocale Object for a Language Type

To run a block of Gosu code with a specified language, you must use a language object of type `GWLangauge` for the first parameter to `runAsCurrentLangauge`. You can specify the object in a number of ways:

- Use the `gw.api.util.LocaleUtil.toLanguage` method to provide a `GWLanguage` object that corresponds to a Gosu typecode in the `LanguageType` typelist. You can specify the parameter to this method by using typecode syntax, such as `LanguageType.TC_EN_US`. The typecode has to be defined in the `LanguageType` typelist.
- You can specify a typecode of the correct type directly, without using the `toLanguage` method. For example, for U.S. English, use `gw.i18n.ILocale.LANGUAGE_EN_US`. This syntax requires that the language typecode `en_US` be defined the `LanguageType` typelist.
- You can specify the first parameter by using a method on `LocaleUtil` that can get the current or default language. For example, `getCurrentLanguage` and `getDefaultLanguage`.

You can add a typecode to the `LanguageType` typelist. If you add a new typecode to this typelist, you must restart Studio to be able to use it in `gw.i18n.ILocale`. For a similar example, see “To add a Java locale code to the LocaleType typelist” on page 80.

The following example Gosu code sets U.S. English as the language for a display string, overriding the current language:

```
uses gw.api.util.LocaleUtil

// Run a block of Gosu code that prints the display string in U.S. English
LocaleUtil.runAsCurrentLanguage(
    LocaleUtil.toLanguage(LanguageType.TC_EN_US),
    \-> {print(displaykey.Activity)})
```

See also

- “Setting Regional Formats for a Block of Gosu Code” on page 84
- “Gosu Blocks” on page 233 in the *Gosu Reference Guide*

Additional Useful Methods on gw.api.util.LocaleUtil

In addition to `runAsCurrentLanguage`, `runAsCurrentLocaleAndLanguage`, and `runAsCurrentLocale`, the class `gw.api.util.LocaleUtil` provides a number of other methods useful in working with localization, including the following:

Method	Description
`canSwitchLanguage`	Returns boolean `true` if the current user is assigned to a role that has the `usereditlang` permission, which allows the user to switch to a different language.
`canSwitchLocale`	Returns boolean `true` if the current user is assigned to a role that has the `usereditlang` permission, which allows the user to switch to a different locale.
`getAllLanguages`	Returns a list of the typecodes of all languages defined in the `LanguageType` typelist.
`getAllLocales`	Returns a list of the typecodes of all locales defined in the `LocaleType` typelist.

Method	Description
getCurrentLanguage	Returns the effective language for the thread as a GWLanguage object. This language can be, in order of priority, the temporary setting, such as by runAsCurrentLanguage, the user setting, or the system setting.
getCurrentLocale	Returns the effective locale for the thread as a GWLocale object. This locale can be, in order of priority, the temporary setting, such as by runAsCurrentLocale, the user setting, or the system setting.
getCurrentLanguageType	Calls getCurrentLanguage and returns a LanguageType typecode.
getCurrentLocaleType	Calls getCurrentLocale and returns a LocaleType typecode.
getDefaultLanguage	The system setting for the language, set in configuration parameter DefaultApplicationLanguage.
getDefaultLocale	The system setting for the locale, set in configuration parameter DefaultApplicationLocale.
getLanguageLabel	Returns the localized display name for the language as a String. For example, if the current language is English and the method parameter is LanguageType.TC_DE_DE, the method returns "German". However, if the current language is German, the method returns "Deutsch".
getLocaleLabel	Returns the localized display name for the locale as a String. For example, if the current language is English and the method parameter is LocaleType.TC_DE_DE, the method returns "German (Germany)". However, if the current language is German, the method returns "Deutsch (Deutschland)".
toLanguage	Converts a LanguageType typekey to a GWLanguage object and returns it as an ILocale object.
toLocale	Converts a LocaleType typekey to a GWLocale object and returns it as an ILocale object.

Setting Default Width for Input Field Labels

In the base configuration, the default width for an input field label in PCF files is 150 pixels. Depending on the display language, the default width of your input field labels might need to be wider or narrower. You can change the default width by setting the <InputLabelWidth> subelement of the <GWLanguage> XML element in the language.xml file for the appropriate localization folder. To access the localization folders, in Studio navigate in the **Project** window to **configuration** → **config** → **Localizations**.

Typically, you do not need to change the default value of this setting. If you do make a change, you must check all your pages to ensure that this change did not introduce any layout issues. You can confirm the label width by using the Inspect Element feature of your web browser.

You can set this width as appropriate in a language.xml file in any localization folder, such as the default en_US folder. The setting then applies to that language. For example, if you download the Japanese language pack, you can open the language.xml file in the ja localization folder. Then set a value, such as 175 pixels, for the <InputLabelWidth> element. For example:

```
<GWLanguage code="ja" name="Japanese" typecode="ja">
  <LinguisticSearchCollation strength="primary"/>
  <SortCollation strength="primary"/>
  <InputLabelWidth width="175"/>
</GWLanguage>
```

See also

- For more information on installing language packs, see “Installing Display Languages” on page 23.
- For more information on language.xml, see “Searching and Sorting in Configured Languages” on page 158.

chapter 6

Localizing Administration Data

You can localize shared administration data through the use of localized database columns. For example, PolicyCenter uses activity patterns to create new activities. A localized column enables users to see activity names in their preferred languages.

This topic includes:

Understanding Administration Data

Guidewire refers to certain types of application data to as *administration data*, or by the shortened term *admin data*. For example, activity patterns are administration data. For selected fields in administration data, PolicyCenter stores localized—translated—values directly in the application database.

You enter translations for admin data directly in PolicyCenter, in screens that you can open on the **Administration** tab. If you have configured PolicyCenter for multiple languages, for entities with localization tables, you see a **Localization** list view that is visible at the bottom of a screen. This list view has a row for each enabled language in the application and has fields for each element on that screen that you can localize.

Localized Columns in Entities

To accommodate localized values for shared administration data or any entity data, you can specify that a column contains localized values in the database. To configure an entity to store localized values for a column, in Guidewire Studio, add a `localization` element as a child of the `column` element for the entity.

For example, in Studio open the entity in the editor. Then right-click the column you want to localize and choose **Add new** → **localization**.

Adding a `localization` element to a column triggers the creation of a localization table during the next database upgrade. A *localization table* stores localized values for a column for every language other than the default application language. The column itself stores values for the default application language.

The `localization` element requires that you specify a `tableName` attribute, which is the name of the localization table. This name has length restrictions and a special format. For an example, see "Localization Tables in the Database" on page 63.

Localization Attributes

Guidewire provides several attributes on the `localization` element on `column` that affect the use of the element. The following list describes each attribute:

Attribute	Type	Description
`nullok`	`Boolean`	This attribute is required. Set it to the same value as the `nullok` attribute for the column to which you are adding the table. If you set this attribute to `false`, PolicyCenter flags missing entries that it finds during a database consistency check, but it can start up with these missing entries. If PolicyCenter is configured with multiple languages: • PolicyCenter stores the values for the default application language in the main database table of the entity. • PolicyCenter stores the values for additional languages in a separate localization table. During a consistency check, PolicyCenter flags entries in the main database table for the default language if corresponding entries for additional languages cannot be found in the localization table. Entries flagged as missing additional languages are warnings only. A missing language value does not prevent the server from starting. **Note:** If only one language is configured, PolicyCenter does not run the consistency check.
`tableName`	`String`	The name of the localization table. Use the following format for this name: *`mainEntityNameAbbrevation_columnNameAbbreviation`*`_l10n` **IMPORTANT:** The table name must be no longer than 16 characters. If the name exceeds this length, the application server will not start.
`extractable`	`Boolean`	Default value is `false`. If you set this attribute to `true`, PolicyCenter adds the localization table to the archive for the entity. See "The Extractable Delegate" on page 253 in the *Configuration Guide*.
`overlapTable`	`Boolean`	Default value is `false`. Overlap tables are tables in which individual table rows can exist either in the domain graph or as part of reference data, but not both. The database table itself exists both in the domain graph and as reference data. If you set this attribute to `true`, PolicyCenter marks the localization table as an overlap table. See "The OverlapTable Delegate" on page 253 in the *Configuration Guide*.
`unique`	`Boolean`	Default value is `false`. If you set this attribute to `true`, PolicyCenter enforces that for each language the values are unique and there are no duplicates. If the entity is of type `effdated` or `effdatedbranch`, do not set `unique` to `true`. See "Data Entities and the Application Database" on page 155 in the *Configuration Guide*.

See also

- "Checking Database Consistency" on page 40 in the *System Administration Guide*

Localization Element Example

In the base configuration, the `ActivityPattern` entity's `Description` column is configured for localization. The `Description` column can store localized values for each configured language.

To see the localization element definition on the Description column

1. In Studio, navigate in the **Project** window to **configuration** → **config** → **Metadata** and double-click `Activity.eix` to open it in the editor.
2. Expand the `Description` column, and then click its `localization` element.
3. This element has the following property values:

Property	Value
`nullok`	`true`
`tablename`	`actpat_desc_l10n`
`extractable`	`false`
`overlapTable`	`false`
`unique`	`false`

Localization Tables in the Database

PolicyCenter stores the localized values for columns that have a `localization` element in separate localization tables in the database. PolicyCenter generates localization tables automatically. Guidewire recommends that you use the following format for the table name attribute.

`mainEntityNameAbbrevation_columnNameAbbreviation``_l10n`

IMPORTANT The length of the table name must not exceed 16 characters.

For example, the localization table name for the `Subject` column of the `ActivityPattern` entity uses the abbreviation `actpat` for the main entity and `sbj` for the column name:

`actpat_sbj``_l10n`

Localization tables have the following columns:

- `Owner` – An integer that represents an ID of the owner
- `Language` – A typekey to the `LanguageType` typelist
- `Value` – A column of type `String`

Localization tables contain localized values for configured languages other than the default application language. The localized column itself contains values for the default application language.

System Table Localization

System tables are database tables that support business logic in PolicyCenter lines of business. Developers who use Guidewire Studio define system tables with needed columns as entities in the data model. Business analysts who use Product Designer can then examine, edit, and enter values for the system tables columns.

System tables provide additional metadata beyond the capacity of typelists. System tables typically provide storage for information that must be maintained periodically by non-developers. Examples include:

- Class codes
- Territory codes
- Industry codes
- Reason codes
- Reference dates
- Rate factors

- Underwriting companies

See also

- For complete information, see "System Tables" on page 75 in the *Product Model Guide*.

The Product Designer System Table Editor

In Product Designer, you can examine, edit, and enter values for system table columns. In addition, you can enter localized values for localized system tables columns.

In Product Designer, the **System Table** page displays content for localized columns in the language that you select in the **User Settings** page. If a localized column has a value for that language, the column displays it in the **System Table** page. If a localized column does not have a value for that language, the column is empty. To examine, enter, or edit values for other languages, click the Translate button to display the **Display Key Values by Locale** dialog box.

See also

- To learn how to configure a column as a localized column, see "Localized Columns in Entities" on page 61.
- To learn how the database stores localized values, see "Localization Tables in the Database" on page 63.

What You Can Localize

The default configuration of PolicyCenter provides localized columns in the following system tables.

System table file	System table entity	Localized column
bop_class_codes.xml	BOPClassCode	• Classification
class_code_basis.xml	ClassCodeBasis	• Name • Description
cp_class_codes.xml	CPClassCode	• Classification
gl_class_codes.xml	GLClassCode	• Classification
industry_codes.xml	IndustryCode	• Classification
risk_classes.xml	RiskClass	• Description
territory_codes.xml	DBTerritory	• Description
wc_class_codes.xml	WCClassCode	• Classification • ShortDesc

chapter 7

Localizing Guidewire Workflow

This topic discusses localization as it relates to Guidewire workflow.

This topic includes:

Localizing PolicyCenter Workflow

At the start of the execution of a workflow, PolicyCenter evaluates the language and locale set for the workflow. PolicyCenter then uses that language for notes, documents, templates, and similar items associated with the workflow. The language and locale that the workflow uses depend on the settings of the user that executes the workflow code.

> **Note:** See "Localizing Templates" on page 69 for information on localizing application documents, notes, and emails.

It is possible to set the workflow region and the workflow language independently of the default application language and region in Studio. To set either workflow language or region, open the workflow in the Studio Workflow editor. Navigate in the **Project** window to **configuration** → **config** → **Workflows** and double-click the workflow node to open it in the Studio Workflow editor. Then click the background area in the workflow layout view. This action opens the **Properties** area at the bottom of the workflow area. In this **Properties** area, you can enter one of the following:

- A fixed name for the language or locale
- A Gosu expression that evaluates to a valid type for the language or locale

For example, to set a specific workflow locale, use one of the following:

Type	Gosu
Fixed string	`gw.i18n.ILocale.FR_FR`
Variable expression	`gw.api.util.LocaleUtil.toLocale(PolicyPeriod.Policy.Account.PrimaryLanguage)` `gw.api.util.LocaleUtil.toLocale(Group.Supervisor.Contact.PrimaryLanguage)`

Localizing Workflow Step Names

You can provide translated versions of workflow step names. PolicyCenter displays translated step names in the following locations:

- **Workflow summary** – On the **Find Workflows** search screen, the workflow summary shows the last completed workflow step. Administrative accounts only can access this screen.
- **Workflow log** – On the **Workflow Detail** screen, the log shows all workflow steps.

Additionally, Guidewire Studio can show translated step names in the Workflow editor it you specify the translated language when you start Studio.

There are two techniques you can use to translate workflow step names:

- To translate a small number of names, see "Translating Workflow Step Names in Studio" on page 66
- To translate a large number of names, see "Exporting Workflow Steps Names as String Resources for Translation" on page 67

See also

- "Selecting Language and Regional Formats in PolicyCenter" on page 13
- "Specifying a Language for Studio" on page 54

Translating Workflow Step Names in Studio

You can translate workflow step names directly in the Studio Workflow editor. This technique is useful if you want to translate a small number of names.

To enter a translated workflow step name in Studio

1. Install a Guidewire language pack for each language you want to translate. See "Installing Display Languages" on page 23.
2. Navigate in the **Project** window to **configuration** → **config** → **Localizations**. In each localization folder for the languages you want to translate, there must be a `display.properties` file. If there is no file, you can add one. The file does not have to have entries in it.

   For example, you install the French language pack. The language pack installer creates a new folder for the language pack, `pc-lang-fr`, at the same level in the **Project** window as the `config` folder. The language pack installer saves a `display.properties` file in that folder, in **pc-lang-fr** → **config** → **Localizations** → **fr**. In addition, you must add a `display.properties` file to **configuration** → **config** → **Localizations** → **fr_FR**. This second file can be empty.
3. Navigate in the **Project** window to **configuration** → **config** → **Workflows** and double-click the workflow node to open it in the Studio Workflow editor.
4. Click the workflow step that you want to localize.
5. In the **Properties** window at the bottom of the screen, click the **Step Name Localizations** tab.
6. Under the localization code for the target language, enter the translated step name.

7. You can restart Studio and, on the command line, specify one of the languages for which you entered a translation. Then you can see the translated workflow step name in the Workflow editor. You also see the translated name in the PolicyCenter user interface if that language is the system default or the user selection.

See also

- "Specifying a Language for Studio" on page 54

Exporting Workflow Steps Names as String Resources for Translation

If you have more than a few workflow step names to translate, it can be more convenient to export them, translate them in the exported file, and then import them.

To export workflow step names as string resources for translation

1. Export the PolicyCenter string resources for a particular locale by using the following `gwpc` command syntax:
   ```
   gwpc export-l10ns -Dexport.file=targetFile -Dexport.locale=localizationFolder
   ```
   In the command, you must provide a target file name and specify from which localization folder to export the string resources.
2. Find the workflow by name. For example:
   In PolicyCenter, find `Workflow.ProcessMVRsWF` and keep searching until you see an entry that ends in `.Step`. For example, `Workflow.ProcessMVRsWF.1.BeforeOrder.Step`.

   Note: The display key names for workflow items might contain a box character between the workflow name and the version number. A box character represents a character that your system is not configured to display, in this case the character *one dot leader*, UTF-16 hexadecimal code 0x2024. The previous examples represent that character as a period.
3. Translate the workflow step names into the target language.
4. Import the translated strings resources back into PolicyCenter by using the following `gwpc` command syntax:
   ```
   gwpc import-l10ns -Dimport.file=sourceFile -Dimport.locale=localizationFolder
   ```
   In the command, you must provide a source file name and specify into which localization folder to import the string resources.

See also

- For command parameters, "Exporting and Importing String Resources" on page 43.
- To understand their format and syntax, "Properties Files" on page 397 in the *Gosu Reference Guide*.

Creating a Locale-Specific Workflow SubFlow

You can create a child workflow, or subflow, in Gosu by using the following methods on `Workflow`. Each method handles the locale of the subflow differently.

Method	Description
`createSubFlow`	Creates a child subflow synchronously, meaning that PolicyCenter starts the subflow immediately upon method invocation. The new subflow automatically uses the default application locale, not the locale of the parent workflow. Thus, if you set the locale of the parent workflow to be different from the default application locale, the subflow does not inherit that locale.

Method	Description
`createSubFlowAsynchronously`	Creates a child subflow asynchronously, meaning that PolicyCenter does not start the subflow until it finishes executing all code in the Gosu initialization block. Again, the subflow uses the default application locale, not the locale set for the workflow itself. Because PolicyCenter executes all the Gosu code in the block before starting the subflow, it is possible to set the locale of the subflow before the workflow starts.
`instantiateSubFlow`	Creates a child subflow, but does not start it. You can modify the subflow that the method returns before you start the subflow.

To create a subflow that inherits the locale of the parent workflow

1. Define a workflow that has the `LanguageType` property. See "Creating New Workflows" on page 391 in the *Configuration Guide* for information on how to create a new workflow with a `LanguageType` property.
2. Set the locale for this subflow so that it uses your desired language. See "Localizing PolicyCenter Workflow" on page 65 for details.
3. Instantiate the subflow by using the `instantiateSubFlow` method rather than the `createSubFlow` method.
4. Set the `LanguageType` property on the instantiated subflow to the locale of the parent workflow.
5. Start the subflow by using one of the workflow start methods described at "Instantiating a Workflow" on page 395 in the *Configuration Guide*.

See also

"Workflow Subflows" on page 400 in the *Configuration Guide*

Localizing Gosu Code in a Workflow Step

It is possible to localize the language used for Gosu code that you add to any workflow step, such as in an **Enter Script** block. To do so, wrap the Gosu code in the following method.

```
gw.api.util.LocaleUtil.runAsCurrentLanguage(alternateLanguage, \ -> { code } )
```

In the code, set the value of `alternateLanguage` to the language to use for the Gosu code block. For example:

```
gw.api.util.LocaleUtil.runAsCurrentLanguage(gw.i18n.ILocale.LANGUAGE_FR_FR, \ -> { code } )
```

Other wrapper methods useful for localization include the following:

- `runAsCurrentLocale`
- `runAsCurrentLocaleAndLanguage`

For more information on these methods, see:

- "Setting a Language for a Block of Gosu Code" on page 57
- "Setting Regional Formats for a Block of Gosu Code" on page 84

chapter 8

Localizing Templates

This topic describes application localization as it applies to documents, emails, and notes.

This topic includes:

About Templates

In the base configuration, Guidewire provides a number of template-related definition files for notes, emails, and documents. You can navigate in the Studio **Project** window to the following folders containing these files:

- **configuration** → **config** → **resources** → **doctemplates**
- **configuration** → **config** → **resources** → **emailtemplates**
- **configuration** → **config** → **resources** → **notetemplates**

Each folder contains two files for each resource type, the template file and a descriptor file. The two files have the same names but different file extensions. PolicyCenter uses the files to define a document, an email, or a note. The following table describes these files.

File extension	Description	Example

`.rtf`
`.htm`
`.pdf`
`.xml`
`.xls`
	Template files of various types. Template files contain the actual content of the document, email, or note.	`CreateEmailSent.gosu.htm`

`.descriptor`	Template descriptor file in XML format. This file contains the template metadata, such as: • `name` • `subject` This file can also contain symbol definitions for context objects that PolicyCenter substitutes into the template content file in creating the final document.	`CreateEmailSent.gosu.htm.descriptor`

See also

For general information on templates and how to create them and use them, see:

- "Gosu Templates" on page 351 in the *Gosu Reference Guide*
- "Data Extraction Integration" on page 607 in the *Integration Guide*

Creating Localized Documents, Emails, and Notes

Creating localized versions of document, email, and note templates mainly involves:

- Creating locale-specific folders in the correct location.
- Populating each folder with translated versions of the required document, email, or note templates and descriptor files.

The following steps describe the process.

- Step 1: Create Locale-Specific Folders
- Step 2: Copy Template Content Files
- Step 3: Localize Template Descriptor Files
- Step 4: Localize Template Files
- Step 5: Localize Documents, Emails, and Notes in PolicyCenter

Notes:

- In PolicyCenter, the default locale for a document, note, or email template is the configured default locale for the application.
- Any time you add a file to a Studio-managed file folder, you must stop and restart Studio so that it recognizes the change.
- Guidewire does not provide the ability to localize Velocity templates.

Step 1: Create Locale-Specific Folders

In PolicyCenter Studio, you can see the locations of the unlocalized PolicyCenter document, email, and note templates by navigating in the **Project** window to the following folders:

- **configuration** → **config** → **resources** → **doctemplates**
- **configuration** → **config** → **resources** → **emailtemplates**

- **configuration** → **config** → **resources** → **notetemplates**

To create your own localized versions of the template files:

1. Open Studio and navigate in the **Project** window to the following folder:

   configuration → **config** → **resources** → **doctemplates**

2. Right-click the **doctemplates** folder and choose **New** → **Package**.

3. Enter the name of your locale-specific folder and click OK.

   For example, if you are creating a French locale and want to use French-language document templates, enter `fr_FR` for the name.

4. If you also want to localize the email and note templates, repeat the previous steps for the following folders:
   - **configuration** → **config** → **resources** → **emailtemplates**
   - **configuration** → **config** → **resources** → **notetemplates**

After you complete this task, you see the locale-specific folders in the **Project** window, along with all the non-localized templates. For example:

- **configuration** → **config** → **resources** → **doctemplates** → **fr_FR**
- **configuration** → **config** → **resources** → **emailtemplates** → **fr_FR**
- **configuration** → **config** → **resources** → **notetemplates** → **fr_FR**

Step 2: Copy Template Content Files

After you set up the template locale folders, copy the template files from the main directory into the locale subfolders.

- For documents, you copy only the template content files, not the descriptor files. For example, you might copy the following files from **doctemplates** to **doctemplates/fr_FR**:

```
CreateEmailSent.gosu.htm
PolicyQuote.gosu.rtf
```

- For email and notes, you copy both the template content files and the template descriptor files. For example, you might copy the following files from **emailtemplates** to **emailtemplates/fr_FR**:

```
NeedXYZ.gosu
NeedXYZ.gosu.descriptor
GotXYZForPolicy.gosu
GotXYZForPolicy.gosu.descriptor
ActivityActionPlan.gosu
ActivityActionPlan.gosu.descriptor
```

Step 3: Localize Template Descriptor Files

After you copy the template files to a template locale folder, you create localized versions of the template descriptor files.

It is important to understand that localizing template descriptor files serves a different purpose from that of localizing (translating) template content files. For example:

- Localizing the subject context object in an email template descriptor file enables a PolicyCenter user to see the subject line of that email template in the localized language in PolicyCenter.
- Localizing the content of an email template enables the recipient of that email to see its contents in the localized language.

The manner in which you create localized document template descriptor files is different from the process for creating localized email and note template descriptor files. See the following topics for details:

- Localizing Document Descriptor Files
- Localizing Email and Note Descriptor Files

Localizing Document Descriptor Files

The document template descriptor files remain in the main **doctemplates** folder, and you edit them there. Descriptor files are XML-based files that conform to the specification defined in file `document-template.xsd`. You can view these files in the Studio **Project** window at:

configuration → **config** → **resources** → **doctemplates**

The descriptor file defines context objects, among other items. *Context objects* are values that PolicyCenter inserts into the document template to replace defined symbols. For example, PolicyCenter replaces `<%=Subject%>` in the document template with the value defined for the symbol `Subject` in the descriptor file.

For example, in the base configuration, PolicyCenter provides an XML definition for the `AccountEmailSent` template descriptor associated with the `AccountEmailSent` document content template. The descriptor file defines a context object for the `Subject` symbol. You can define as many context objects and associated symbols as you need. You can add elements that localize the template for any languages supported by your system.

```
<?xml version="1.0" encoding="UTF-8"?>
  <DocumentTemplateDescriptor
    id="AccountEmailSent.gosu.htm"
    name="Gosu Sample Account Email Sent Record"
    description="Record of an email being sent"
    ...
    keywords="CA, email">

    <DescriptorLocalization language="someLanguage" name="localizedName"
           description="localizedDescription" />
    ...

    <ContextObject name="Subject" type="string">
      <DefaultObjectValue></DefaultObjectValue>
      <ContextObjectLocalization language="someLanguage" display-name="localizedName" />
      ...
    </ContextObject>
  ...

</DocumentTemplateDescriptor>
```

Localizing a template descriptor file requires that you localize a number of items in the file. The following list describes some of the main items that to localize in a descriptor file:

Element	Attribute	Description
`<DocumentTemplateDescriptor>`	• `keywords`	Localize the keywords associated with this template to facilitate the search for this template in the PolicyCenter search screen.
`<DescriptorLocalization>`	• `language` • `name` • `description`	Subelement of `<DocumentTemplateDescriptor>` – Enter a valid `GWLanguage` value for `language`, such as `gw.i18n.ILocale.LANGUAGE_EN_US`, which uses the language typecode `en_US`. The language typecode must be defined in the `LanguageType` typelist. See also, “Obtaining an ILocale Object for a Locale Type” on page 85. You can also localize the name and description of this template as it appears in PolicyCenter.
`<ContextObjectLocalization>`	• `language` • `display-name`	Subelement of `<ContextObject>` – Enter a valid `GWLanguage` value for `language`. See the previous description for more information. You can also localize the name of this template as it appears PolicyCenter.

To localize a document template descriptor file, add the appropriate `<DescriptorLocalization>` and `<ContextObjectLocalization>` subelements to the file.

IMPORTANT There is only one copy of each document template descriptor file. Do not create additional copies in locale folders. Instead, add localization elements to the descriptor files in the `doctemplates` folder.

Localizing Email and Note Descriptor Files

To localize email and note descriptor files, you put a copy of each descriptor file in the correct locale folder and localize the following attributes in that file.

Element	Attribute	Description
`<emailtemplate-descriptor>` `<notetemplate-descriptor>`	• `keywords` • `subject`	Localize the keywords associated with this template to facilitate the search for this template in the PolicyCenter search screen. Also, localize the subject of this template to show that localized value in PolicyCenter.

For example, to localize an email or note template descriptor file for French (France), first copy the descriptor file to an `fr_FR` folder. Then localize any keywords that you want and the subject tag for the template.

Step 4: Localize Template Files

After copying the template content files to your locale folder, as described in “Step 2: Copy Template Content Files” on page 71, you then need to translate them. Unless you want to create a new template, the simplest procedure is to do the following:

- Open the copied base language content template.
- Translate it into the language of your choice.
- Save the translated file in the locale-specific configuration folder.

Step 5: Localize Documents, Emails, and Notes in PolicyCenter

After you create localized versions of your templates, you can then use these templates in PolicyCenter to create a language-specific version of a document, an email, or a note.

Note: In PolicyCenter, you can select the unlocalized templates that are in the default directory. PolicyCenter displays these templates with no language specified. If you select one, however, PolicyCenter makes the **Language** field in the **New Document** worksheet editable.

To create a localized document

1. In PolicyCenter, open a policy and navigate to **Actions** → **New Document** → **Create a new document from a template**.

   This action opens the **New Document** worksheet at the bottom of the screen.

2. In the **New Document** worksheet, click the search icon in the **Document Template** field.

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

3. In the search screen that opens, set the **Language** field to your language and set the other search fields as needed. If a document template for your language exists, PolicyCenter displays it in **Search Results**.

4. Click **Select**. PolicyCenter returns to the **New Document** worksheet with the selected localized template.

5. Complete the rest of the worksheet fields as necessary. You can enter text in your chosen language in the appropriate fields to further localize the document.

To create a localized email

1. In PolicyCenter, open a policy and choose **Actions** → **New Email** to open the **New Email** worksheet at the bottom of the screen.
2. In the **Email** worksheet, you can either enter text in your chosen language or click **Use Template** to open the template selection worksheet.
3. If you click **Use Template**:
   a. In the search screen that opens, set the **Language** field to your chosen language and set the other search fields as necessary. If an email template for the language exists, PolicyCenter displays it in **Search Results**.
   b. Click **Select**. PolicyCenter returns to the **New Email** worksheet with the selected localized template.
4. Complete the rest of the worksheet fields as needed. You can enter text in your chosen language in appropriate fields to further localize the document.

To create a localized note

1. In PolicyCenter, open a policy and choose **Actions** → **New Note** to open the **New Note** worksheet at the bottom of the screen.
2. If you click **Use Template**:
   a. In the search screen that opens, set the **Language** field to your chosen language and set the other search fields as necessary. If an email template for the language exists, PolicyCenter displays it in **Search Results**.
   b. Click **Select**. PolicyCenter returns to the **New Note** worksheet with the selected localized template.
3. Complete the rest of the worksheet fields as needed. You can enter text in your chosen language in appropriate fields to further localize the document.

Document Localization Support

PolicyCenter provides a number of useful methods for working with document localization in the following APIs. You can use methods of classes that implement the interfaces described in the following topics to search for document templates and generate documents by using a specific locale:

- “IDocumentTemplateDescriptor Interface Methods” on page 75
- “IDocumentTemplateSource Plugin Interface Methods” on page 75
- “IDocumentTemplateSerializer Plugin Interface Methods” on page 75

See also

- “Document Creation” on page 103 in the *Rules Guide* for general information on APIs used in document creation.
- “Document Management” on page 191 in the *Integration Guide* for integration-related issues in document creation.

IDocumentTemplateDescriptor Interface Methods

If you need to customize the PolicyCenter interface, Guidewire provides the following methods on the `IDocumentTemplateDescriptor` interface for working with locales:

Method	Return type	Returns
`getLanguage()`	`String`	Language to use to create the document from this template descriptor. A return value of `null` indicates the default language for the application. PolicyCenter returns the language that matches the directory that contains the `.descriptor` file that the user selected in the PolicyCenter interface.
`getName(locale)`	`String`	Localized name from the document descriptor file for a given `locale`. If a translation does not exist, the method returns `null`.

You can also use the following property on `IDocumentTemplateDescriptor` to retrieve the locale:

```
gw.plugin.document.IDocumentTemplateDescriptor.locale
```

See also

- "Localizing Document Descriptor Files" on page 72

IDocumentTemplateSource Plugin Interface Methods

The `IDocumentTemplateSource` plugin provides the following useful getter methods.

Method	Return type	Returns
`getDocumentTemplate(date, valuesToMatch, maxResults)`	`IDocumentTemplateDescriptor[]`	Array of `IDocumentTemplateDescriptor` objects. If you need to perform a search based on the locale of a document template, you can specify this in the `valuesToMatch` argument.
`getDocumentTemplate(templateId, locale)`	`IDocumentTemplateDescriptor`	Document template instance corresponding to the specified template ID and locale.
`getTemplateAsStream(templateId, locale)`	`InputStream`	One of the following: • Returns an `InputStream` for reading the specified template • Returns `null` if PolicyCenter cannot find the template • Returns `null` if the caller does not have adequate privileges to retrieve the template

IDocumentTemplateSerializer Plugin Interface Methods

The `IDocumentTemplateSource` plugin provides the following method that you can use to retrieve a localized version of a template.

Method	Return type	Returns
`localize(locale, descriptor)`	`IDocumentTemplateDescriptor`	Localized instance of the specified template

part III

Regional Format Configuration

chapter 9

Working with Regional Formats

You can configure support for multiple regional formats in PolicyCenter. Regional formats specify how to format items like dates, times, numbers, and monetary amounts for use in the user interface. Regional formats specify the visual format of data, not the database representation of that data.

This topic includes:

Configuring Regional Formats

In PolicyCenter, you can either use the regional formats defined in the International Components for Unicode (ICU) Library, or you can override those formats by defining formats in `localization.xml`. You save this file in the localization folder for the region it defines. If a localization folder does not have a `localization.xml` file, PolicyCenter uses the ICU Library. You configure these formatting options in Guidewire Studio.

This topic includes:

About the International Components for Unicode (ICU) Library

The International Components for Unicode (ICU) library is an open source project that provides support for Unicode and software globalization. PolicyCenter includes this library.

The ICU library, `icu4j`, attempts to maintain API compatibility with the standard Java JDK. However, for most features, the ICU library provides significant performance improvements and a richer feature set than the Java JDK. The core of the ICU library is the Common Locale Data Repository (CLDR). The CLDR repository is a comprehensive repository of locale data.

See also

- For a feature comparison between the ICU library and the Oracle JDK, see `http://site.icu-project.org/`.
- For comparisons of text collation performance, see `http://site.icu-project.org/charts/collation-icu4j-sun`.
- For more information about the CLDR, see `http://cldr.unicode.org/`.

Locale Codes, localization.xml, and the ICU Library

A localization folder does not have to contain a `localization.xml` file. If you do add a `localization.xml` file, you need to define only the locale settings that you want to override the ICU library defaults. For any locale settings that you have not defined, PolicyCenter uses the ICU library defaults, as follows:

- PolicyCenter uses the ICU library for regional formats for dates, times, numbers, and monetary amounts.
- PolicyCenter uses the ICU library for the Japanese Imperial Calendar.

Java Locale Codes and the ICU Library

ICU uses Java locale codes to identify specific regional settings. For PolicyCenter to be able to access the ICU library for a region, the locale code for that region must be defined in the `LocaleType` typelist. In the base configuration, PolicyCenter provides the following set of Java locale codes in the `LocaleType` typelist:

```
en_US United States (English)
en_GB Great Britain (English)
en_CA Canada (English)
en_AU Australia (English)
fr_CA Canada (French)
fr_FR France (French)
de_DE Germany (German)
ja_JP Japan (Japanese)
```

The Java locale codes defined in this typelist are used by PolicyCenter as follows:

- To define the locale codes that you can use to set the default application locale, as described in “Setting the Default Application Locale for Regional Formats” on page 84.
- To access the regional formats for a locale supplied by the ICU library. See “About the International Components for Unicode (ICU) Library” on page 79.

Adding a Locale Code to the LocaleType Typelist

You can add additional Java locale codes to the `LocaleType` typelist.

To add a Java locale code to the LocaleType typelist

1. In PolicyCenter Studio, navigate in the **Project** window to **configuration** → **config** → **Metadata** → **Typelist**.
2. Right-click `LocaleType.tti` and choose **New** → **Typelist Extension**.
3. Click OK in the dialog box to create `LocaleType.ttx` in the `extensions/typelist` folder.

   The Typelist editor opens so you can edit the new `LocaleType.ttx` file.

   Note: You need to perform these steps only the first time you add a new Java locale code. To add subsequent locale codes, just navigate to **configuration** → **config** → **Extensions** → **Typelist** and double-click `LocaleType.ttx`.
4. Right-click the typelist element and choose **Add new** → **typecode**.
5. For **code**, enter the Java locale code.

   For example, enter the Java locale code `nl_NL` to configure regional formats used in the Netherlands.
6. For **name** and **description**, by convention, specify the country followed by the language in parentheses.

   For example, enter `Netherlands (Dutch)` for both the name and the description.

Configuring a localization.xml File

A localization folder can contain a `localization.xml` file, which defines regional formats. If you open Guidewire Studio and navigate in the **Project** window to **configuration** → **config** → **Localizations**, you can see the localization folders provided in the base configuration of PolicyCenter. These localization folders have Java locale codes for names, such as:

- **de_DE**
- **en_US**
- **fr_FR**
- **ja_JP**

In the base configuration, the `en_US`, `fr_FR`, and `ja_JP` localization folders have `localization.xml` files that define regional formats customary to each region. For example:

- In `en_US`, the file contains configuration information on date, time, number, and currency formats for use in the United States.
- In `ja_JP`, the file contains configuration information on how to format address information for Japan. It also contains configuration information on the Japanese Imperial Calendar in use in Japan.

  Note: In the base configuration, Guidewire uses the ICU library defaults for certain regions, such as `de_DE`. For those regions, there is no `localization.xml` file. See “International Components for Unicode (ICU) Library” on page 22.

You can add a new localization folder to **Localizations**. To do so, use the contextual right-click menu to create a new folder that uses a Java locale code defined in the `LocaleType` typelist. If there are multiple copies of a localization folder, then the contents of the folder in the main **configuration** module override any files installed by a language pack.

You can use an existing `localization.xml` file as a starting point if you want to create a new `localization.xml` file to override the ICU library settings for a region. You define attributes and subelements of the `<GWLocale>` element in a `localization.xml` file, as described in the following topic.

<GWLocale> XML Element

A `localization.xml` file contains a single `<GWLocale>` element in which you define the settings for a region. If you do not define an element or attribute, PolicyCenter uses the ICU library setting. The `<GWLocale>` element can take the following attributes and subelements.

<GWLocale> Attributes

- `code` – The region identifier, typically the same as the Java locale code defined in `LocaleType`. For example, `"en_US"`.
- `name` – A `String` value for the name of the locale, which the application can display to the user in its screens. For example, `"United States (English)"`.
- `firstDayOfWeek` – Defines the first day of the week for the region. Value is an integer representing a day of the week, starting with `"1"` for Sunday, `"2"` for Monday, and so on.

  If not defined, the base configuration uses the default ICU library setting for the region:

  - **Sunday** – `en_AU`, `en_CA`, `en_US`, `fr_CA`, `ja_JP`
  - **Monday** – `de_DE`, `en_GB`, `fr_FR`
- `typecode` – The corresponding regional typecode defined in the `LocaleType` typelist. For example, `"en_US"`.
- `defaultCalendar` – `"Gregorian"` or `"JapaneseImperial"`. The default value is `"Gregorian"`.
- `enableJapaneseCalendar` – A `Boolean` value, `true` or `false`, that determines whether or not to enable the Japanese calendar for this region. See “Working with the Japanese Imperial Calendar” on page 99.

<GWLocale> Subelements

- `CurrencyFormat` – Currency format pattern for the region, used in single currency display mode.

  Attributes are:

  - `negativePattern, positivePattern` – Define how PolicyCenter displays positive and negative monetary amounts. PolicyCenter displays and formats the numeric value in place of the pound sign (`#`) in the pattern. PolicyCenter displays all other characters in the pattern as they are without modification.

    For example, the positive pattern `$#` displays the numeric value 32 as `$32.00`. The negative pattern `($#)` displays the numeric value -5 as `($5.00)`.

  - `zeroValue` – Defines how PolicyCenter displays zero amounts. For example, the zero value pattern can be `0` (zero) or `-` (dash). If the numeric value of a monetary amount is null, PolicyCenter displays the amount as empty or blank. The monetary amount must be 0.00 for the zero value pattern to be used.

- `DateFormat` – Patterns for the date formatter and parser. Attributes are:

  - `long` – The long date format pattern. For example, `"E, MMM d, yyyy"`.
  - `medium` – The medium date format pattern. For example, `"MMM d, yyyy"`.
  - `short` – The short date format pattern. For example, `"MM/dd/yyyy"`.

  See “<DateFormat>, <TimeFormat>” on page 83.

- `FlexibleDateFormat` – Date format patterns in addition to short, medium, and long. This element is used only by ClaimCenter. Attributes are:

  - `year-month-day` – Format that contains year, month, and day. For example, `"M/d/y"` or `"y-MM-dd"`.
  - `year-month` – Format that contains year and month. For example, `"MMM y"` or `"y MMM"`.
  - `month-day` – Format that contains month and day. For example, `"MMM d"` or `"MM-dd"`.

- `JapaneseImperialDateFormat` – Japanese imperial calendar settings. See “Working with the Japanese Imperial Calendar” on page 99.

  Attributes are:

  - `long` – Long Japanese imperial date format.
  - `medium` – Medium Japanese imperial date format.
  - `short` – Short Japanese imperial date format.
  - `yearSymbol` – Japanese year symbol.

- `NumberFormat` – Number formatter and parser configurations. Determines how PolicyCenter displays numbers. Attributes are:

  - `decimalSymbol` – Symbol used to separate a whole number from its decimal fraction, usually a period or a comma. For example, `"."`.
  - `negativeEntryPattern` – Format for entering negative numeric values. For example, `"(#)"`.
  - `thousandsSymbol` – Symbol used to separate groups of three digits to the left of the decimal mark, usually a period or a comma. For example, `","`.

- `TimeFormat` – Time formatter and parser configuration. Attributes are:

  - `long` – Long time format pattern. For example, `"h:mm:ss a z"`.
  - `medium` – Medium time format pattern. For example, `"hh:mm:ss a"`.
  - `short` – Short time format pattern. For example, `"h:mm a"`.

  See “<DateFormat>, <TimeFormat>” on page 83.

- `NameFormat` – Formatting of names by the PCF file `GlobalContactNameInputSet` or `GlobalPersonNameInputSet`. Attributes are:

  - `PCFMode` – Mode of the PCF file to use. The mode must exist. In the base configuration, there are two modes, `default` and `Japan`. The default value in the base configuration is `default`.
  - `textFormatMode` – How to format the text. In the base configuration, `Japan` and `France` are possible values.

  - `visibleFields` – Fields the user can see in the PCF file `GlobalContactNameInputSet` or `GlobalPersonNameInputSet`. For example, for France, the visible fields are `"Prefix,FirstName,MiddleName,Particle,LastName,Suffix,Name"`.
- `CalendarWidget` – Defines the year-month pattern used by the calendar widget. Attribute is:
  - `yearMonth` – For example, for `ja_JP`, the value is `"yyyyMM"`.

<DateFormat>, <TimeFormat>

You can specify any of the following attribute values for the `DateFormat` and `TimeFormat` elements:

- `short`
- `medium`
- `long`

For example (for the `en_us` locale):

```
<DateFormat short="MM/dd/yyyy"
            medium="MMM d, yyyy"
            long="E, MMM d, yyyy" />

<TimeFormat short="hh:mm aa"
            medium="hh:mm aa"
            long="hh:mm aa"/>
```

In general, you can use any of the date and time patterns supported by the Java class `SimpleDateFormat` for the `medium` and `long` formats. However, Guidewire maps the `short` format to the date picker widget, which does not support arbitrary date formats. For a description of these patterns, refer to the following web sites:

```
http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html
http://cldr.unicode.org/translation/date-time-patterns
```

PolicyCenter generally uses the `short` form to recognize dates entered by the user. It generally uses the other forms to display dates and times.

Using the Short Format

Define patterns for the `short` date and time definitions that result only in a fixed-length output that matches the pattern length.

> **Note:** Time and date inputs in PCF files do not accept variable length format patterns, such as MMM. Therefore, for these fields, use only format patterns that result in fixed-length output. If you do use a variable-length input pattern, it is coerced during input into the short form.

In general, dates have three components, year, month, and day, each defined by a specific pattern. Each pattern provides a fixed-width output. For example, the following patterns all provide a fixed-width output. In this case, each output contains the same number of characters as the format pattern.

Format	Pattern	Output
`year`	`yyyy`	4 digit output, fixed
`month`	`MM`	2 digit output, fixed
`day`	`dd`	2 digit output, fixed

A Pattern that Does Not Work

The following list describes an incorrect, non-fixed-width pattern:

Format	Pattern	Output
`year`	`yyyy`	4 digit output, fixed
`month`	`MMM`	variable length output
`day`	`dd`	2 digit output, fixed

The pattern `MMM` does not work as there are languages in which the abbreviated month string is not three characters in length. Attempting to use this pattern with a language in which there are abbreviated month strings that are not three characters in length can cause a validation error.

Setting the Default Application Locale for Regional Formats

The default application locale determines the regional formats for users who have not chosen a personal preference. You must set a value for the default application locale, even if you configure PolicyCenter with a single region as the choice for regional formats. You set the default application locale in Guidewire Studio by editing the configuration parameter `DefaultApplicationLocale` in the file `config.xml`. In the base configuration, the default application locale is set to `en_US`.

The following example sets the default application locale to `France (French)`, which sets the default choice for regional formats.

```
<param name="DefaultApplicationLocale" value="fr_FR" />
```

IMPORTANT The value for `DefaultApplicationLocale` must match a typecode in the `LocaleType` typelist. If you set the value of parameter `DefaultApplicationLocale` to a value that does not exist as a `LocaleType` typecode, then the application server refuses to start.

See also

- “Locale Codes, localization.xml, and the ICU Library” on page 80

Setting Regional Formats for a Block of Gosu Code

The class `gw.api.util.LocaleUtil` provides the following methods to enable you to run a block of Gosu code with a specific set of regional formats:

```
gw.api.util.LocaleUtil.runAsCurrentLocale(alternateLocale, \ -> { GosuCode } )

gw.api.util.LocaleUtil.runAsCurrentLocaleAndLanguage(
    alternateLocale,
    alternateLanguage,
    \ -> { code } )
```

The second method sets both region and language at the same time.

Running a block of Gosu code in this way enables PolicyCenter to format dates, times, numbers, and names of people appropriately for a specific region. Otherwise, the block of Gosu code uses the regional formats specified by the current region. The current region is either specified by the current user or, if not, by the `DefaultApplicationLocale` parameter in `config.xml`.

The parameters that the methods can take are:

Parameter	Description
`alternateLocale`	An object of type `ILocale` that represents a regional format from the `LocaleType` typelist. Specify a `GWLocale` object for this parameter.
`alternateLanguage`	An object of type `ILocale` that represents a language from the `LanguageType` typelist. Specify a `GWLanguage` object for this parameter.
`\ -> { code }`	A Gosu block as a `GWRunnable` object—the Gosu code to run with different regional formats or a different language

Obtaining an ILocale Object for a Locale Type

To run a block of Gosu code with a specified set of regional formats, you must specify a locale object of type `ILocale`. You must specify the subtype `GWLocale` or `GWLanguage` as appropriate for the parameter `alternateLocale` or `alternateLanguage`.

- For the parameter `alternateLocale`, use the `gw.api.util.LocaleUtil.toLocale` method to provide an `ILocale` object that corresponds to a Gosu typecode in the `LocaleType` typelist. The object is actually of type `GWLocale`, which implements `ILocale`. You can specify the object directly by using typecode syntax. For example `LocaleType.TC_EN_US`. The typecode has to be defined in the `LocaleType` typelist.
- For the parameter `alternateLanguage`, use the `gw.api.util.LocaleUtil.toLanguage` method to provide an `ILocale` object that corresponds to a Gosu typecode in the `LanguageType` typelist. The object is actually of type `GWLanguage`, which implements `ILocale`. You can specify the object directly by using typecode syntax. For example `LanguageType.TC_EN_US`. The typecode has to be defined in the `LanguageType` typelist.
- You can specify a typecode of the correct type directly, without using the `toLocale` or `toLanguage` method. Use the following syntax:
  - `GWLocale` – `gw.i18n.ILocale.FR_FR`
  - `GWLanguage` – `gw.i18n.ILocale.LANGUAGE_FR_FR`
- You can specify the first parameter by using a method on `LocaleUtil` that can get the current or default locale or language, as appropriate. For example, `getDefaultLocale` or `getDefaultLanguage`.

You can add a typecode to the `LocaleType` or `LanguageType` typelist. If you add a new typecode to one of these typelists, you must restart Studio to be able to use it in `gw.i18n.ILocale`. For example, see “To add a Java locale code to the LocaleType typelist” on page 80.

The following example Gosu code formats today’s date by using the default application regional formats, overriding the any region that the user might have specified. The code uses `runAsCurrentLocale` to run a block of code that prints today’s date formatted according the default application region’s date format. The first parameter to this method is a call to `getDefaultLocale` to obtain a `GWLocale` object that represents the application’s default regional formats.

```
uses gw.api.util.LocaleUtil
uses gw.api.util.DateUtil

// Run a block of Gosu code that prints the current date
// in the application's default regional format
LocaleUtil.runAsCurrentLocale(getDefaultLocale(),
    \-> {print(DateUtil.currentDate().format("long"))}
)
```

See also

- “Setting a Language for a Block of Gosu Code” on page 57
- “Gosu Blocks” on page 233 in the *Gosu Reference Guide*

chapter 10

Configuring Name Information

Name formats shown in PolicyCenter can vary by region. For example, for `Person` contact types, as opposed to `Company` types, a contact detail view can show the following name fields:

- United States – Prefix, First Name, Last Name, Suffix
- France – Prefix, First Name, Particle, Last Name, Suffix
- Japan – Last Name (phonetic), First Name (phonetic), Last Name, First Name

  Note: For Japanese names, the labels shown on the screen map to the following columns:

  - Last Name (phonetic) maps to `LastName`.
  - First Name (phonetic) maps to `FirstName`.
  - Last Name maps to `LastNameKanji`.
  - First Name maps to `FirstNameKanji`.

You can customize this feature, as described in this topic.

Note: ContactManager supports configuration of names for globalization with features that are similar to those of PolicyCenter. Information for ContactManager name configuration is included in this topic where it differs from PolicyCenter name configuration.

This topic includes:

Names in PolicyCenter

In PolicyCenter, a name is part of contact information, such as a policy or account owner. PolicyCenter provides modal PCF files that support name formats that vary by region.

See also

- "PolicyCenter Contacts" on page 373 in the *Application Guide*

Read-only and Editable Names

PolicyCenter can display name information as read-only text or as editable text entry fields:

- PolicyCenter displays a read-only name as a display name on a single line. PolicyCenter uses the `localization.xml` file for the current region and the `NameFormatter` class to determine which name fields to use.
- PolicyCenter displays editable names as a set of editable text fields in which you can add, modify, or delete information. PolicyCenter uses the `localization.xml` file and the current region to determine the name fields to show.

Name Owners

PolicyCenter uses a `NameOwner` object to control the display and editing of names. The `NameOwner` object identifies the object that contains a particular name. The `NameOwner` determines how read-only values are formatted. For editable names, the `NameOwner` determines which PCF mode to use and which fields to show. You can define different name owners depending on your requirements.

See also

- "Understanding Global Names" on page 88
- "Modal PCF Files and Name Configuration" on page 93

Understanding Global Names

The information that you see for a name depends on:

- The current region for the name.
- Whether PolicyCenter displays the name as text entry fields or a display name, as described at "Read-only and Editable Names" on page 88.

This topic includes:

- "Localization XML Files Overview" on page 88
- "Modal Name PCF Files Overview" on page 89
- "NameFormatter Class Overview" on page 89

Localization XML Files Overview

The `localization.xml` files define settings that control the appearance of name information in PolicyCenter. PolicyCenter stores each `localization.xml` file in its own individual localization folder under the folder **Localizations**. For example, the `localization.xml` file for the name-related information in Japan is in the following location in Studio:

configuration → **config** → **Localizations** → **ja_JP**

See also

- "Configuring the Localization XML File for Names" on page 89.

Modal Name PCF Files Overview

The current region determines which name fields in the database are visible in the PolicyCenter user interface for a name. The page configuration files `GlobalContactNameInputSet` and `GlobalPersonNameInputSet` have modal versions that make different sets of name fields visible based on the current region. These PCF files also control the order in which PolicyCenter displays name fields.

A single mode of these PCF files can be shared between multiple regions. You can add new fields to the existing PCF files. If you need a different set or order of name fields to display for a region, you can add a new modal PCF file for the region. You can use a copy of one of the existing modal PCF files as a starting point for the new file.

To determine which modal PCF file is used for a region, you set the `PCFMode` attribute in the `localization.xml` file for that region. See "PCFMode Attribute of the NameFormat Element" on page 90.

See also

- For more information on `GlobalContactNameInputSet` and `GlobalPersonNameInputSet`, see "Modal PCF Files and Name Configuration" on page 93.
- For more information on modal PCF files, see "Working with Shared or Included Files" on page 291 in the *Configuration Guide*.

NameFormatter Class Overview

The `NameFormatter` class is used to convert `Contact` names (or `ABContact` names) to localized strings. PolicyCenter displays read-only name information in various screens and uses class `gw.name.NameFormatter` to format these read-only strings.

See also

- "NameFormatter Class" on page 94

Configuring Name Data and Fields for a Region

You use region-specific `localization.xml` files to configure the data that PolicyCenter uses to display names for specific regions.

> **Note:** For read-only name display, the order is determined by code in the `NameFormatter` class. For editable name fields, the order is determined by the order of the fields in the PCF mode for the region.

If you add a new name format for a region or you add a new `Contact` (or `ABContact`) name property, you must also configure the files that support read-only names. See "Additional Region and Name Configurations" on page 91.

This topic includes:

- "Configuring the Localization XML File for Names" on page 89
- "Additional Region and Name Configurations" on page 91

Configuring the Localization XML File for Names

PolicyCenter stores `localization.xml` files in region-specific localization folders under the **Localizations** folder, which you can access in Guidewire Studio. For example, the `localization.xml` file for Japan is stored in the following folder:

configuration → config → Localizations → ja_JP

The file `localization.xml` defines the following name-related attributes in the `NameFormat` element:

Attribute	Description	Related topic
`PCFMode`	Which mode of the appropriate PCF name file to use	"PCFMode Attribute of the NameFormat Element" on page 90
`textFormatMode`	Specifies the region to be used by `NameFormatter` to format a read-only name.	"Text Format Mode Attribute of the NameFormat Element" on page 90
`visibleFields`	Which name fields to display	"Visible Fields Attribute of the NameFormat Element" on page 91

PCFMode Attribute of the NameFormat Element

The attribute `PCFMode` of the `localization.xml` file determines which modal version of the name input PCF files PolicyCenter uses for specific countries. These PCF files are `GlobalContactNameInputSet` and `GlobalPersonNameInputSet`.

For example, for Japan, `localzation.xml` file specifies the PCF mode as follows:

```
PCFMode="Japan"
```

If the user chooses Japan as the region, PolicyCenter uses one of the following PCF files as appropriate to display name information:

- `GlobalContactNameInputSet.Japan`
- `GlobalPersonNameInputSet.Japan`

If a `localization.xml` file does not define the `PCFMode` attribute, or there is no `localization.xml` file for a region, PolicyCenter uses the default modal version of the name PCF file. In the base configuration, the default version is suitable for many countries.

`PCFMode` values are defined in the class `NameLocaleSettings`. If you want to add a PCF mode, you must define it in this class. The default base configuration definition is:

```
private static final var validPCFModes : Set<String> = { "default", "Japan", "", null }
```

See also

- "Modal PCF Files and Name Configuration" on page 93

Text Format Mode Attribute of the NameFormat Element

The attribute `textFormatMode` specifies the region name that the `NameFormatter` class uses to format a read-only name. The region names are defined in the `internalFormat` method of that class. The default value of this attribute is `default`.

Valid values for `textFormatMode` are defined in the class `NameLocaleSettings`. If you want to add a text format mode, you must define it in this class. The default base configuration definition is:

```
private static final var validTextFormatModes : Set<String> =
    { "default", "France", "Japan", "", null }
```

See also

- "NameFormatter Class" on page 94

Visible Fields Attribute of the NameFormat Element

The attribute `visibleFields` defines the set of name fields that can be visible for this region. For example:

France	`visibleFields="Prefix,FirstName,MiddleName,Particle,LastName,Suffix,Name"`
Japan	`visibleFields="LastName,FirstName,LastNameKanji,FirstNameKanji,Name,NameKanji"`

The field order and the actual fields that can be shown are specified in the PCF files or the `NameFormatter` class. The order of the fields defined in `visibleFields` does not matter, and listing a field in this attribute simply enables it to be shown if it is specified.

The fields listed in this attribute must be defined in the class `NameOwnerFieldId`. By convention, the names used in `visibleFields` match field names defined in `Contact` or a subentity of `Contact`. In ContactManager, the names match field names defined in `ABContact` or a subentity of `ABContact`. Names are case-sensitive. If the values are not valid, you get error messages in the log when the data is first used. The error messages are defined in `gw.api.name.NameLocaleSettings.init`.

If a `localization.xml` file does not have `visibleFields` defined, PolicyCenter uses the entire set of name fields defined in the PCF file.

See also

- "NameOwnerFieldId Class" on page 96

Additional Region and Name Configurations

You can add a new name format for a region, and you can extend Contact or a Contact subtype with a new name column. These changes require that you update various files and classes.

Note: If you have ContactManager installed, you must make corresponding changes in ContactManager.

Adding a Name Format for a Region

If you want to add an additional name format for a region, you must configure a `localization.xml` file, as described in "Localization XML Files Overview" on page 88. Additionally, you must:

- Define modal name input PCF files that list the input fields if the current files do not have that sequence of fields. See "Modal PCF Files and Name Configuration" on page 93.
- Add a new `if` statement to the `NameFormatter.internalFormat` method to handle the formatting of the name string for the new region. See "NameFormatter Class" on page 94.

Extending Contact with a New Name Column

If you extend the `Contact` entity or a subtype of `Contact` to add a new name column, you must make the following changes in PolicyCenter.

Note: In ContactManager, you would be extending the `ABContact` entity or a subtype of `ABContact` to add a new name column

Make the following changes:

1. Add the new field to the `Contact` or the `Contact` subtype and ensure that the integration files are correctly set up.

   See "Extending the Contact Data Model" on page 137 in the *Contact Management Guide*.

2. Update the class `NameOwnerFieldID` to add the new field.

Add a variable for the new column to this class. Also, add the variable as needed to any of the following existing constants that list name fields in this class:

```
ALL_PCF_FIELDS
ALL_CONTACT_PCF_FIELDS
REQUIRED_NAME_FIELDS
DISPLAY_NAME_FIELDS
FIRST_LAST_FIELDS
HIDDEN_FOR_SEARCH
```

See “NameOwnerFieldId Class” on page 96.

3. Add the new field to one of the following interfaces:
   - For a non-person contact, add the field to `ContactNameFields`.
   - For a person contact, add the field to `PersonNameFields`.
4. Compile the project to see which name delegate classes you need to update.

   For name delegates that take their values from a contact or person, you will see error messages identifying those classes. For example, you will need to add the field to either `ContactNameDelegate` or `PersonNameDelegate`, depending on whether you added it to `ContactNameFields` or `PersonNameFields`.

   Note: In ContactManager, these classes are `ABContactNameDelegate` and `ABPersonNameDelegate`.

   For the remaining name delegates, decide if the new name column needs to be added to the underlying entity.
   - For example, there are several non-persistent entities used for searching. If the new column will not be used for searching, create a getter for the field that returns `null` and a setter for the field that throws an exception. For example, `MiddleName` is not a search column in the base configuration. For an example, see the implementation of `MiddleName` in `gw.api.name.PLContactSearchNameDelegate`.
   - For persistent entities, you can add a new column to match the column you added to the `Contact` or `Contact` subtype. Then you create a getter and setter for the field in the associated name delegate. If you do not add the column to the persistent entity, create a getter for the field that returns `null` and a setter for the field that throws an exception.

   For more information on configuring search for contacts, see “Overview of Contact Search” on page 87 in the *Contact Management Guide*.
5. If necessary, add a `localization.xml` file for the region and then add the new field to it. Also, update the existing `localization.xml` files with the field. See “Configuring the Localization XML File for Names” on page 89.
6. Add the field to the appropriate modal contact input set files, `GlobalContactNameInputSet` or `GlobalPersonNameInputSet`.

   Add new fields as needed to these PCF files. For example, if the new name column applies only to persons, change only the `GlobalPersonNameInputSet` modal files as needed. You might not have to change the file for all modes. Configure your new fields by following the pattern of the existing fields in the file.

   See “Modal PCF Files and Name Configuration” on page 93.
7. Use the new field in the `NameFormatter` class.

   In the `internalFormat` method, add a call to `append` for the new field. Add the `append` call to the `if` statement for the `mode` value that matches the region for the new field. Follow the pattern used for the other regions and fields. The `fieldid` values are defined in `gw.api.name.NameOwnerFieldId`. See “NameFormatter Class” on page 94.
8. Add the new field as needed to the entity display names that use the `NameFormatter` class.

   For example, if the new name field applies only to persons, you would not add it to `Company.en` or `Place.en`, but you would add it to `Contact.en`. In the Entity Names editor, add the field both to the list of fields in the table at the top and to the code in the lower text entry area. For example `:NewFieldName = newFieldName`.

   See “Using the Entity Names Editor” on page 125 in the *Configuration Guide*.

To find the display names, press Ctrl+Shift+F and search for NameFormatter with **Scope** set to **Whole project** and **File mask** set to *.en.

- The following list shows the display names that use NameFormatter in the base configuration of PolicyCenter:

```
CommercialDriver.en
Company.en
Contact.en
        Place.en
PolicyContactRole.en
```

- The following list shows the display names that use NameFormatter in the base configuration of ContactManager:

```
ABCompany.en
ABContact.en
ABPlace.en
```

Modal PCF Files and Name Configuration

PolicyCenter and ContactManager use one of two modal PCF files for names:

- GlobalContactNameInputSet – Used for Company names. In ContactManager, used for ABCompany names.
- GlobalPersonNameInputSet – Used for Person names. In ContactManager, used for ABPerson names.

Modal versions of the global name PCF files determine the available fields and their order for specific countries.

In the base configuration, PolicyCenter provides the following modal versions of the global name input set files.

Modal Name PCF Files	Region
• GlobalContactNameInputSet.default • GlobalPersonNameInputSet.default	• Australia • Canada • France • Germany • Great Britain • United States
• GlobalContactNameInputSet.Japan • GlobalPersonNameInputSet.Japan	• Japan

To see these PCF files, navigate in the Guidewire Studio **Project** window to **configuration** → **config** → **Page Configuration** → **pcf** → **name**.

Mapping Regions to Modes

A region maps to a mode through its associated localization folder's localization.xml settings. If there is no localization file for the region, the default mode is used.

Controlling Field Properties

Each modal PCF file uses an implementation of the Gosu interface NameOwner to get the fields values for the following field properties:

```
available
editable
label
required
value
```

Gosu Name Formatter

PolicyCenter uses the Gosu class NameFormatter to format read-only name display fields. You can extend NameFormatter to handle additional regions. See "NameFormatter Class" on page 94.

Name Owners

NameOwner is the interface for a helper object that is passed to the modal PCF files GlobalPersonNameInputSet and GlobalContactNameInputSet. The helper object provides a way to set and get a single name on the enclosing entity. Typically you extend NameOwnerBase, which implements NameOwner.

While NameOwner provides methods for setting a field to be required or visible, generally it is better not to modify or override methods such as isVisible in NameOwnerBase. For most purposes, specifying the set, required, or hidden fields is more easily controlled by overriding property getters such as RequiredFields and HiddenFields declared in NameOwner. You override these property getters with values such as NameOwnerFieldId.REQUIRED_NAME_FIELDS or NameOwnerFieldId.NO_FIELDS.

Following are some properties on NameOwner:

Property	Description
ContactName	Sets (or retrieves) a single contact name on the enclosing entity, Contact. For example, the Name field of GlobalContactNameInputSet.default has the following value: nameOwner.ContactName.Name
PersonName	Sets (or retrieves) a single person name on the enclosing entity, Person. For example, the First Name field of GlobalPersonNameInputSet.default has the following value: nameOwner.PersonName.FirstName
HiddenFields	Set of name fields that ClaimCenter hides (does not show) in the application interface.
RequiredFields	Set of name fields for which the user must supply a value.

In the base configuration, PolicyCenter provides the following classes that implement NameOwner or extend a class that implements NameOwner:

```
NameOwnerBase
AffinityGroupNameOwner
BasicNameOwner
ContactNameNoSummaryOwner
ContactNameOwner
GroupNameOwner
MVRDriverNameOwner
NewAffinityGroupNameOwner
OrganizationNameOwner
PLPersonNameSearchOwner
RequiredBasicNameOwner
UserSearchNameOwner
```

In the base configuration, ContactManager provides the following classes that implement NameOwner or extend a class that implements NameOwner:

```
NameOwnerBase
ABUserNameOwner
ContactNameOwner
PLPersonNameSearchOwner
SearchNameOwner
UserSearchNameOwner
```

NameFormatter Class

PolicyCenter displays read-only name information in various screens. It uses class gw.name.NameFormatter to manage and format these read-only strings.

The NameFormatter class is used to convert Contact names to strings for globalization. If you change, add, or delete name columns of the Contact entity or its subentities, you must also update this class.

Note: ContactManager uses NameFormatter to convert names used in ABContact and subentities of ABContact to strings for globalization

Additionally, if you add a new region definition, you might need to update the `internalFormat` method of this class, which uses fields defined in the `NameOwnerFieldId` class.

The class contains two versions of the `format` method with different signatures.

- The following version of the `format` method formats a name as text and includes all fields that are used for the current region. This case is the common one. This method has the following signature:

```
format(name : ContactNameFields, delimiter : String)
  : String { _filter = \ fieldId : NameOwnerFieldId -> { return true }
```

- The following version of the `format` method formats a name as text and includes only the specified set of fields from the current region. This method has the following signature:

```
format(name : ContactNameFields, delimiter : String, fields : Set<NameOwnerFieldId>)
  : String { _filter = \ fieldId : NameOwnerFieldId -> { return fields.contains(fieldId) }
```

The method parameters have the following meanings:

Parameter	Description
`name`	The `Contact` name to format.
`delimiter`	The delimiter that separates elements of the name, typically `" "`.
`fields`	The set of `Contact` fields to include in the name.

The following PolicyCenter modal PCF file uses `NameFormatter`:

- `GlobalPersonNameInputSet` sets the following `Value` for the `Name` field:

```
new gw.api.name.NameFormatter().format(nameOwner.PersonName, " ")
```

The following PolicyCenter Gosu class and Gosu enhancements call `NameFormatter`:

```
gw.contact.AbstractContactResult.gs
gw.plugin.contact.impl.ContactEnhancement.gsx
gw.plugin.contact.impl.ContactKanjiEnhancement.gsx
```

The following PolicyCenter display names call `NameFormatter` to return `Contact` names:

- `displaynames/CommercialDriver.en` returns:

```
new NameFormatter().format(person, " ", NameOwnerFieldId.DISPLAY_NAME_FIELDS)
```

- `displaynames/Company.en` returns:

```
new NameFormatter().format(contact, " ")
```

- `displaynames/Contact.en` returns one of the following, based on `Contact` subtype:

```
new NameFormatter().format(person, " ", NameOwnerFieldId.DISPLAY_NAME_FIELDS)
new NameFormatter().format(contact, " ")
```

- `displaynames/Place.en` returns:

```
new NameFormatter().format(contact, " ")
```

- `displaynames/PolicyContactRole.en` returns one of the following, based on `Contact` subtype:

```
new NameFormatter().format(person, " ", NameOwnerFieldId.DISPLAY_NAME_FIELDS)
new NameFormatter().format(contact, " ")
```

The following ContactManager modal PCF file uses `NameFormatter`:

- `GlobalPersonNameInputSet` sets the following `Value` for the `Name` field:

```
new gw.api.name.NameFormatter().format(nameOwner.PersonName, " ")
```

The following ContactManager display names call `NameFormatter` to return `ABContact` names:

- `displaynames/ABCompany.en` returns:

  ```
  new NameFormatter().format(contact, " ")
  ```

- `displaynames/ABContact.en` returns one of the following, based on `ABContact` subtype:

  ```
  new NameFormatter().format(person, " ", NameOwnerFieldId.DISPLAY_NAME_FIELDS)
  new NameFormatter().format(contact, " ")
  ```

- `displaynames/ABPlace.en` returns:

  ```
  new NameFormatter().format(contact, " ")
  ```

See also

- "NameOwnerFieldId Class" on page 96
- "Additional Region and Name Configurations" on page 91

NameOwnerFieldId Class

Guidewire provides a `gw.api.name.NameOwnerFieldId` class that provides type safety for `Name` entity fields. If you extend the `Contact` entity or the `ABContact` entity with a new name column, you must add the new column to this class as a new constant.

In the base configuration, `NameOwnerFieldId` provides the following constants that represent name fields:

```
PREFIX
FIRSTNAME
MIDDLENAME
PARTICLE
LASTNAME
SUFFIX
FIRSTNAMEKANJI
LASTNAMEKANJI
NP_NAME
NAMEKANJI
```

For example, you add the new `Contact` name column `MyNameColumn` to the `NameOwnerFieldId` class with the following code:

```
public static final var MYNAMECOLUMN :
    NameOwnerFieldId = new NameOwnerFieldId("MyNameColumn")
```

The class `NameOwnerFieldId` also defines a set of constants that use these name field ID constants:

```
public final static var ALL_PCF_FIELDS : Set<NameOwnerFieldId> =
  { PREFIX, FIRSTNAME, MIDDLENAME, PARTICLE, LASTNAME, SUFFIX, FIRSTNAMEKANJI,
    LASTNAMEKANJI, NP_NAME, NAMEKANJI }.freeze()

/** Fields used for non-Person names */
public final static var ALL_CONTACT_PCF_FIELDS : Set<NameOwnerFieldId> =
  { NP_NAME, NAMEKANJI }.freeze()

/** Required fields (union of fields for both Persons and non-Persons) */
public final static var REQUIRED_NAME_FIELDS : Set<NameOwnerFieldId> =
  { LASTNAME, NP_NAME }.freeze()

/** Fields shown in display names */
public static final var DISPLAY_NAME_FIELDS : Set<NameOwnerFieldId> = {
  FIRSTNAME, PARTICLE, LASTNAME, SUFFIX }.freeze()

/** Fields for simple name */
public final static var FIRST_LAST_FIELDS : Set<NameOwnerFieldId> =
  { FIRSTNAME, LASTNAME }.freeze()

public final static var HIDDEN_FOR_SEARCH : Set<NameOwnerFieldId> =
  { PREFIX, MIDDLENAME, PARTICLE, SUFFIX }.freeze()
```

You can add a new `Contact` name column to one of these constants by using its constant name, such as `MYNAMECOLUMN`. For example:

```
public final static var ALL_PCF_FIELDS : Set<NameOwnerFieldId> =
  { PREFIX, FIRSTNAME, MIDDLENAME, MYNAMECOLUMN, PARTICLE, LASTNAME, SUFFIX, FIRSTNAMEKANJI,
    LASTNAMEKANJI, NP_NAME, NAMEKANJI }.freeze()
```

chapter 11

Working with Kanji Fields

This topic discusses the Kanji fields provided in various entities and how to enable indexes to improve searching for them.

Kanji fields are available in various entities, such as `Contact` and its subtypes and `Address`. Additionally, Kanji fields are supported by contact search entities and classes.

- For information on Kanji fields used in names, see "Configuring Name Information" on page 87.
- For information on Kanji fields used in addresses, see "Understanding Global Addresses" on page 132.

This topic includes:

Enabling Indexes for Kanji Fields

Indexes are used to improve search performance and, where needed, to provide uniqueness. For Kanji fields, there are a number of indexes provided as commented out code in the base configuration that must be enabled if your installation uses Kanji fields. If your installation does not use the Kanji fields provided in the base configuration, do not enable these indexes.

You must enable Kanji indexes in both PolicyCenter and ContactManager.

To enable Kanji indexes in PolicyCenter

Note: You will need to restart PolicyCenter and ContactManager after making these changes.

Navigate to the following files in your file system and open them in an editor, not in Studio. Search for a comment that starts with `KanjiIndexDefinition`.

Note: While you cannot make these changes in Guidewire Studio, you can use Studio to find all the files with commented-out Kanji indexes in them. In Studio, press `Ctrl+Shift+F`, and then search for the string `KanjiIndexDefinition` in the entire project.

Uncomment the following Kanji index definitions:

- `PolicyCenter/modules/configuration/config/extensions/entity/Contact.Global.etx`.

Uncomment the index named `CompanyNameK1`.

- `PolicyCenter/modules/configuration/config/extensions/entity/Group.Global.etx`.
  - Uncomment the index named `group1K`.
  - Comment out the index named `group1`.

To enable Kanji indexes in ContactManager

Note: You will need to restart Contact Manager after making these changes.

Navigate to the following files in your file system and open them in an editor, not in Studio. Search for a comment that starts with `KanjiIndexDefinition`.

Note: While you cannot make these changes in ContactManager Studio, you can use Studio to find all the files with commented-out Kanji indexes in them. In Studio, press `Ctrl+Shift+F`, and then search for the term `KanjiIndexDefinition` in the whole project.

Uncomment the following Kanji index definitions:

- `ContactManager/modules/configuration/config/extensions/entity/Contact.Global.etx`.

  Uncomment the index named `CompanyNameK1`.
- `ContactManager/modules/configuration/config/extensions/entity/Group.Global.etx`.

  1. Uncomment the index named `group1K`.
  2. Comment out the index named `group1`.
- `ContactManager/modules/configuration/config/extensions/entity/ABPerson.etx`.

  Uncomment the indexes named `CFindDupeNameK`, `ABPersonKWFNCSK`, and `ABPersonKWFNPCK`.
- `ContactManager/modules/configuration/config/extensions/entity/Address.etx`.

  Uncomment the indexes named `address2K` and `address6K`.

chapter 12

Working with the Japanese Imperial Calendar

This topic discusses the Japanese Imperial Calendar and how to configure Guidewire PolicyCenter to access and display Japanese Imperial Calendar dates correctly.

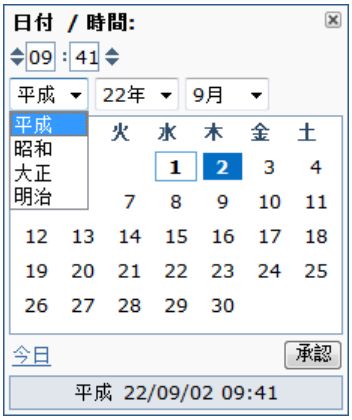
This topic includes:

The Japanese Imperial Calendar Date Widget

PolicyCenter provides a Japanese Imperial calendar date widget that you can display in the user interface. You must enable this feature through configuration. After you enable it, you can specify the default calendar type for the following:

- An entity property
- A Java or Gosu property

The following graphic illustrates the date picker for the Japanese Imperial calendar:



The date widget displays only the four most recent Imperial eras. They are:

- Heisei
- Showa
- Taisho
- Meiji

Note: Guidewire does not support the use of an input mask for Japanese Imperial calendar input. Enter date input by using the date picker. If you use a copy-and-paste method to enter a date, then you must paste an exact matching string into the date field.

Configuring Japanese Dates

You configure PolicyCenter display of Japanese Imperial calendar dates by specifying the `<JapaneseImperialDateFormat>` element in `localization.xml`. The `<JapaneseImperialDateFormat>` element is similar to the existing Gregorian `<DateFormat>` element. You define the following attributes:

Date format	Description
`long` `medium`	PolicyCenter uses only the `long` and `medium` date formats to display Japanese dates. You cannot use these fields to format user input data.
`short`	PolicyCenter uses the `short` date format to format user date input. Any date input format pattern that you choose must be compatible with the fixed width of the input mask. Guidewire recommends that you use `short="G yy/MM/dd"`.
`yearSymbol`	PolicyCenter uses the Japanese `yearSymbol` as a signal to render the Japanese Imperial calendar date picker.

The following graphic shows the base configuration Japanese <GWLocale> definition, which is in localization.xml in the localization folder ja_JP. This definition includes the <JapaneseImperialDateFormat> element.

Setting the Japanese Imperial Calendar as the Default for a Region

PolicyCenter provides a way to set a default calendar for a region through the defaultCalendar attribute of the <GWLocale> element. Additionally, for the Japanese Imperial calendar, you must also set the enableJapaneseCalendar of the <GWLocale> element to true. These attributes determine whether or not to enable the Japanese calendar for this region.

To set the Japanese Imperial calendar as the default calendar for Japan:

1. In the Guidewire Studio **Project** window, navigate to **configuration** → **config** → **ja_JP** and double click localization.xml to open it in the editor.
2. Add the defaultCalendar attribute to the <GWLocale> element, with the value set to JapaneseImperial.
3. Add the enableJapaneseCalendar attribute to the <GWLocale> element, with the value set to true.

```
<GWLocale
  code="ja_JP"
  name="Japanese"
  typecode="ja_JP"
  defaultCalendar="JapaneseImperial"
  enableJapaneseCalendar="true">
```

IMPORTANT Guidewire uses the *International Components for Unicode* (ICU) open source library to format dates according to the Japanese Imperial calendar. The ICU library specifies formats according to the historical Imperial calendar. Contemporary changes to the calendar, such as the start of a new era, require an updated ICU library. If such an event occurs, contact Guidewire support for details on how to upgrade to a newer version of the ICU library.

See also

- “Working with Regional Formats” on page 79

Setting Fields to Display the Japanese Imperial Calendar

You can set a field of the PolicyCenter user interface either to always display the Japanese Imperial Calendar or to conditionally display the calendar.

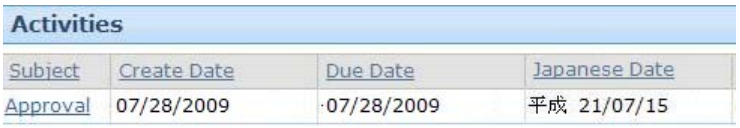
Always Displaying the Japanese Imperial Calendar

You can set a field of an entity to always display the Japanese Imperial calendar when it is used the PolicyCenter user interface.

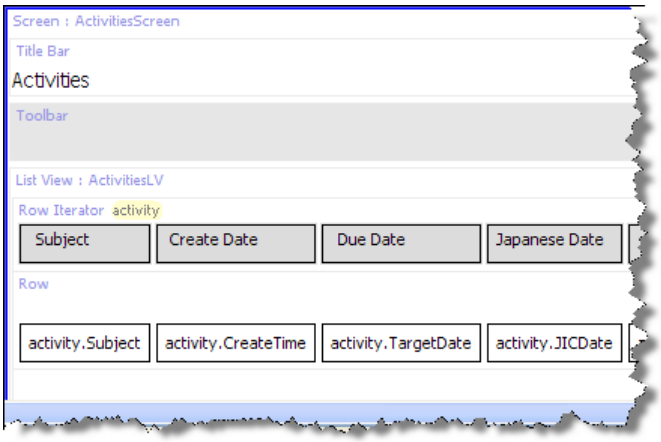
Suppose, for example, that you want to add an additional field to the `Activity` object that uses the `japaneseimperialdate` data type.

1. Navigate in the Studio **Project** window to **configuration** → **config** → **Extensions** → **Entity,** and then double-click `Activity.etx` to open it in the editor.
2. Right-click an element on the left, such as **entity (extension)** and choose **Add new** → **column.**
3. Select the new column and enter the following values for the following attributes:
   - **name** – `JICDate`
   - **type** – `japaneseimperialdate`
   - **nullok** – `true`

To be useful, you need to use this field to display a value in the Japanese Imperial calendar format. For example, in the **Activities** screen, you can add a **Japanese Date** field.



In the `Activities` PCF file, it looks similar to the following:



IMPORTANT You must configure your installation for the Japanese locale in order to display a date in Japanese Imperial calendar format. Otherwise, regardless of the calendar setting, you see only Gregorian dates.

Conditionally Displaying the Japanese Imperial Calendar

You might want to treat a field as either a Gregorian date or a Japanese Imperial calendar date depending on certain factors. In this case, you can use a datatype annotation that causes PolicyCenter to display certain fields in

different formats in different situations. For example, as described in "Working with Japanese Imperial Dates in Gosu" on page 104, suppose that you want to do the following:

- If the user's locale is `ja_JP` and `Policy.PolicyType != "CALI"`, then PolicyCenter is to display these fields by using the Gregorian calendar.
- If the user's locale is `ja_JP` and `Policy.PolicyType == "CALI"`, then PolicyCenter is to display these fields by using the Japanese calendar.

PolicyCenter provides several different ways to accomplish this date formatting. For example, you can:

- Annotate an entity field (database column)
- Annotate a Gosu property

To annotate an entity field

1. Configure the data type for the entity field in an extension file. For example, add a column with the following attributes:
   - `name` – `JICDate`
   - `type` – `japaneseimperialdate`
   - `nullok` – `true`

   See "Setting Fields to Display the Japanese Imperial Calendar" on page 102.
2. Define a `PresentationHandler` class for the data type. For example:

```
class JapaneseImperialDateDataTypeDef implements IDataTypeDef {

  construct() { }

  override property get PresentationHandler() : IDataTypePresentationHandler {
    return new JapaneseImperialCalendarPresentationHandler()
  }
  ...
}
```

   If the class that you create implements an interface, in this case, `IDataTypeDef`, then you need to provide implementations for all methods and properties in the interface. If you place your cursor in the class definition line and press `Alt+Enter`, Studio prompts you with the required method and property bodies to automatically insert. You then need to define these properties and methods to suit your business needs.

   For an example of how to implement a presentation handler class, see "Defining a New Tax Identification Number Data Type" on page 241 in the *Configuration Guide*.
3. Implement the date presentation handler for the data type. You can make the implementation logic dynamic based on the entity context. For example:

```
class JapaneseImperialCalendarPresentationHandler implements IDatePresentationHandler {

  construct() { }

  override function getCalendar( ctx: Object, prop: IPropertyInfo ) : DisplayCalendar {
    /** If the "ctx" object is a policy, and the policy is "cali" type
      * and the "prop" is the effectivedate field ...
      **/
    if (...)
      return JAPANESEIMPERIAL }
    else
      return GREGORIAN
  }
  ...
}
```

To annotate a Gosu property

1. Annotate the data type for the Gosu property. For example:

```
@DataType("japaneseimperialdate")
property get JICDate() : Date {
  return _bean.DateField1
}
```

2. Define a `PresentationHandler` class for the data type in a similar fashion to that described previously in step 2 of "To annotate an entity field" on page 103.
3. Implement the date presentation handler class for the data type in a similar fashion to that described previously in step 3 of "To annotate an entity field" on page 103.

See Also

- For an example of how to implement the `IDataTypeDef` interface, see "Defining a New Tax Identification Number Data Type" on page 241 in the *Configuration Guide*. Specifically, see "Step 2: Implement the IDataTypeDef Interface" on page 242 in the *Configuration Guide*.
- "Working with Japanese Imperial Dates in Gosu" on page 104

Working with Japanese Imperial Dates in Gosu

PolicyCenter enables you to create an object extension of type `japaneseimperialdate` based on `java.util.Date`. You can use this data type to do the following:

- To set a calendar field value so that it always displays values in Japanese Imperial calendar style.
- To set a calendar field value so that it displays values in Japanese Imperial calendar style depending on a data field. This option requires additional configuration.

For example, consider the following two fields:

- `Policy.EffectiveDate`
- `Policy.ExpirationDate`

Now consider the following cases:

Locale	Policy.PolicyType	Description
`en_US`	–	If `Policy.PolicyType` is `null` (unspecified), then PolicyCenter displays all date fields in the Gregorian calendar in all situations
`ja_JP`	–	If `Policy.PolicyType` is `null` (unspecified), then PolicyCenter displays all date fields in the Japanese Imperial calendar in all situations
`en_US`	==”CALI”	If `Policy.PolicyType = “CALI”`, then PolicyCenter displays `Policy.EffectiveDate` and `Policy.ExpirationDate` in the Japanese Imperial calendar and all other date fields in the Gregorian calendar.
`ja_JP`	!=”CALI”	If `Policy.PolicyType != “CALI”`, then PolicyCenter displays `Policy.EffectiveDate` and `Policy.ExpirationDate` in the Gregorian calendar and all other date fields in the Japanese Imperial calendar.

Note: *CALI* (*Compulsory Auto Liability Insurance*) is a type of automobile policy in Japan that every driver must carry by law. It covers only injury to other parties, basically mandating that every driver protect others around them but not themselves, just like auto liability coverage in the United States. A driver who wants personal coverage must purchase another policy, known as a Voluntary policy.

These cases result in the following behavior:

- In the first two cases, the calendar to use in formatting `EffectiveDate` and `ExpirationDate` is dependent on the locale of the user only. For this, you merely need to set the `defaultCalendar` attribute on `<GWLocale>` in `localization.xml` to the correct value.
- In the last two cases, PolicyCenter displays these fields depending on additional conditions. For this, you need to provide additional configuration. See "Conditionally Displaying the Japanese Imperial Calendar" on page 102.

See Also

- "Setting the Japanese Imperial Calendar as the Default for a Region" on page 101

- “Setting Fields to Display the Japanese Imperial Calendar” on page 102
- “Conditionally Displaying the Japanese Imperial Calendar” on page 102

part IV

National Formats and Data Configuration

chapter 13

Configuring Currencies

You can configure PolicyCenter with multiple currencies. With multiple currencies, you can specify monetary amounts in different currencies in a single instance of PolicyCenter.

This topic includes:

PolicyCenter Base Configuration Currencies

In the base configuration, Guidewire provides support for the following global currencies:

- `AUD` – Australian dollar
- `CAD` – Canadian dollar
- `EUR` – European Union euro
- `GBP` – British pound
- `JPY` – Japanese yen
- `RUB` – Russian ruble
- `USD` – U.S. dollar

You use the following files in working with a currency, depending on whether you configure your installation for single currency or multicurrency mode:

Studio	Single currency rendering mode	Multicurrency rendering mode
`Currency` typelist	Provides currency code and similar information for the supported currency. • In a new installation, remove all currency typecodes except that of the default application currency. • In an installation that you are upgrading, retire all Currency typecodes except that of the default application currency.	Contains information on all supported currencies, such as currency codes and similar information.
`currency.xml`	In single currency mode, do not use this file to define currency formats. Remove all currency formatting from this file. Instead, use file `localization.xml` to format currency information.	Studio supports multiple copies of `currency.xml`, one for each supported currency, in each currency folder. For example: **configuration** → **config** → **currencies** → **aud** → `currency.xml`
`datatypes.xml`	Use to set the following for the application currency—the default application currency: • `precision` • `scale` • `appscale`	Use to set the following for the default currency only: • `precision` • `scale` • `appscale` Use individual `currency.xml` files to set the `storageScale` attribute on the `<CurrencyType>` element for each defined currency.
`localization.xml`	Contains currency formatting information for use with single currency rendering mode only. Studio supports multiple copies of this file, one for each localization folder. For example: **configuration** → **config** → **Localizations** → **en_US**	In multicurrency mode, do not use this file to define currency formats. Remove all currency formatting from `localization.xml`.

In addition, in working with currency and monetary amounts, you must set the following configuration parameters in `config.xml`:

- `DefaultApplicationCurrency` – Default application currency for currency and monetary amounts
- `DefaultRoundingMode` – Default rounding mode for currency and monetary amounts
- `MulticurrencyDisplayMode` – Currency display mode in the application for selecting currencies

IMPORTANT If you integrate the core applications in Guidewire InsuranceSuite, you must set the values of `DefaultApplicationCurrency` and `MulticurrencyDisplayMode` to be the same in each application.

See also

- "Setting the Default Application Currency" on page 114
- "Choosing a Rounding Mode" on page 115
- "Setting the Currency Display Mode" on page 115

Working with Currency Typecodes

The typecodes in the `Currency` typelist determine which currencies you can use to enter monetary amounts in PolicyCenter. The base configuration of the `Currency` typelist defines the following currencies:

- U.S. Dollar
- Euro

- United Kingdom Pound
- Canadian Dollar
- Australian Dollar
- Russian Ruble
- Japanese Yen

For your configuration, add or retire currencies in the `Currency` typelist to suit your needs. The typelist must include at least one active currency.

Working with Currency Typecodes in the Currency Typelist

Depending on your installation type, you need to add, remove (delete), or retire existing typecodes from the `Currency` typelist.

Installation type	Action
Single currency install	Remove all Currency typecodes except for the default application Currency typecode.
Multiple currency install	Add or delete Currency typecodes as representative for your installation.
Upgrading previous install	Retire all Currency typecodes that you do not intend to use in the upgraded installation.

To add a currency to the Currency typelist

1. In Studio, open the `Currency.ttx` typelist either by:
   - Searching for it with `Ctrl+Shift+N`.
   - Navigating in the **Project** window to **configuration** → **config** → **Extensions** → **Typelist** and double-clicking `Currency.ttx`.
2. In the toolbar, click the Add icon ✚.

   Studio adds a row for the typecode and selects it automatically.
3. Enter the following properties for the new currency.

code	Specify the lowercase version of the three-letter ISO 4217 currency code, for example, `nzd`.
name	Specify the uppercase version of the three-letter ISO 4217 currency code, for example, `NZD`.
desc	Specify the country or jurisdiction that issues the currency, followed by the name of the currency, for example, `New Zealand Dollar`.

Monetary Amounts in the Data Model and in Gosu

In the base configuration, Guidewire defines multiple data types to handle monetary amounts. You use the monetary data types to store and display monetary amounts in Guidewire PolicyCenter.

In the base configuration, Guidewire defines the `<monetaryamount>` subelement in metadata definition files for fields that handle monetary amounts. Use the monetary amount data type to store and display monetary amounts within Guidewire PolicyCenter.

The `<monetaryamount>` subelement is a compound data type that stores its values in two separate database columns. One of the columns is `<money>` column type, and the other is a typekey field to the `currency` typelist.

IMPORTANT Do not use the `currencyamount` column type in your data model. Do not use the corresponding `CurrencyAmount` Gosu or Java type. Use the `<monetaryamount>` subelement instead.

Monetary Amount Data Type

The money data types store and show monetary amounts in the system. PolicyCenter assumes that all monetary amounts in a money data type column are in the default application currency, as set by configuration parameter DefaultApplicationCurrency. The base configuration has the following money data types.

Money data type	Description
money	Permits positive, negative, and zero values.
nonnegativemoney	Permits positive and zero values only.
positivemoney	Permits positive values only.

The money data types use the following specialized attributes, which you set in datatypes.xml for the default application currency:

- precision
- scale
- appscale

These settings that you make for these parameters affect all money columns, which are a component of the <monetaryamount> subelement.

Precision and Scale of Monetary Amounts

With money columns, precision controls the total number of digits and scale controls the number of digits to the right of the decimal point. For example, if precision=5 and scale=2, then the maximum value for monetary amounts is 999.99 and the minimum value is -999.99.

Some factors to consider in choosing the scale value include globalization issues and specific business requirements. For example, to track monetary amounts down to 1/100 of the currency unit, then set scale to 4. The default value for scale is 2.

IMPORTANT The MulticurrencyDisplayMode parameter setting is permanent. After you change the value of MultiCurrencyDisplayMode to MULTIPLE and then start the server, you cannot change the value back to SINGLE again. The MultiCurrencyDisplayMode parameter is in config.xml.

Guidewire recommends that you set the scale attribute to the maximum number of decimal position that you would possibly need in the future. Setting scale initially to a large value enables you to later add currencies for which the number of decimal digits is larger than your initial configuration.

You set precision and scale for the default application currency in the <MoneyDataType> element in file datatypes.xml, for example:

```
<MoneyDataType precision="18" scale="2" validator="Money"/>
```

The precision and scale attributes control how PolicyCenter displays monetary amounts in the user interface and how PolicyCenter stores monetary amounts in the database.

See also

- "The Data Types Configuration File" on page 236 in the *Configuration Guide*

Storage Scale in Multicurrency Installations

Installations that contain multiple currency definitions frequently need to set the number of decimal digits to store in the database differently for monetary amounts in different currencies. To set specific storage values for individual currencies, use the `storageScale` attribute on the `<CurrencyType>` element in file `currency.xml`. In installations that support the display of multiple currencies, there is a `currency.xml` file for every defined currency.

Attribute `storageScale` sets the number of decimal places to store in the database for monetary amounts in that currency. For example, in the base configuration:

File `currency.xml` for the Australian dollar defines the following `storageScale`:

```
<CurrencyType code="aud" desc="Australian Dollar" storageScale="2">
```

File `currency.xml` for the Japanese yen shows the following `storageScale`:

```
<CurrencyType code="jpy" desc="Japanese Yen" storageScale="0">
```

Application Scale

The `appscale` attribute is a rarely-used optional data type attribute. Guidewire provides the `appscale` attribute to facilitate upgrades of single currency configurations to configurations with multiple currencies. The value of `appscale` must be less than the value of `scale`.

Setting a value for the `appscale` attribute enables a single currency configuration to operate with a scale in the user interface that is appropriate to a particular currency. At the same time, monetary amounts stored in the database have a different and larger scale. The `appscale` value becomes the effective scale of `BigDecimal` values that PolicyCenter saves to and loads from monetary columns in the database.

For example, suppose that you implement a single currency configuration of PolicyCenter for the Japanese Yen, which typically has a scale of zero. You intend to convert to a multicurrency configuration in the future. In the conversion, you intend to add the U.S. Dollar, which typically has a scale of two.

In this example, you initially set `appscale=0` and `scale=2`. Thus in the interface, `BigDecimal` values for `money` and `monetaryamount` columns have a scale of zero but are stored in the database with a scale of two. Later, during your conversion to a configuration with multiple currencies, you removed the `appscale` attribute from `datatypes.xml`. In its place, you use the `storageScale` attributes in the `currency.xml` configuration files for each currency to specify the scale each currency.

Otherwise, you must re-create monetary columns during an upgrade to have a scale value of two. The Guidewire upgrade tool does not support changes in the scale of monetary columns.

The following rules and restrictions apply to the `money` data type.

- Set the value for `appscale` to the largest number of decimal positions that the currency you currently use requires. This value sets the scale for monetary amounts in the interface.
- Set the value for `scale` to the largest number of decimal positions that the currencies you plan to use in the future require. This value sets the scale for monetary amounts in the database.
- The value for `appscale` must be less than the value for `scale`.
- If you do not set a value for `appscale`, then PolicyCenter uses the value for `scale` in the user interface and in the database.

Guidewire does not use the `appscale` attribute in the base configuration. If you want to use this attribute, then you must add this attribute to the `<MoneyDataType>` element in `datatypes.xml`. Use the `appscale` attribute only in single currency mode. In multicurrency mode, use the `storageScale` attribute in the `currency.xml` file for each currency.

MonetaryAmount Data Type in Gosu

When programming in Gosu, the `MonetaryAmount` data type is provided to store monetary values in one of the supported currencies. Like other numeric values, `MonetaryAmount` objects can perform standard mathematical calculations.

A `MonetaryAmount` value is initialized when it is constructed. Two constructors are supported.

```
MonetaryAmount( BigDecimal amount, Currency currency )
MonetaryAmount( String value )
```

The default currency can be retrieved by calling the `gw.api.util.CurrencyUtil.getDefaultCurrency` method. The method returns a `Currency` object specifying the setting of the `config.xml` parameter `DefaultApplicationCurrency`. For other useful `CurrencyUtil` methods, refer to the *Gosu API Reference*.

The format of the `value` string is "`<Amount> <Currency_Code>`" where a space character separates the two parameters. An example is shown below.

```
MonetaryAmount    myBalance( "123.45 USD" )
```

A `MonetaryAmount` object can also be created from a `BigDecimal` object by calling the `BigDecimal` extension methods `ofCurrency` or `ofDefaultCurrency`. The `ofCurrency` method accepts a `Currency` type argument. The `ofDefaultCurrency` method uses the default currency specified in `config.xml`. Example invocations are shown below.

```
var   baseBigD         = new java.math.BigDecimal( "123.45" )
var   monetaryAmount01 = baseBigD.ofCurrency( Currency.TC_USD )
var   monetaryAmount02 = baseBigD.ofDefaultCurrency()
```

For additional information about the `MonetaryAmount` data type, refer to the *Gosu API Reference*.

Currency-related Configuration Parameters

The following list describes the configuration parameters that you must set in file `config.xml` that relate to currency.

Configuration parameter	Description	See...
`DefaultApplicationCurrency`	Default currency for the application	"Setting the Default Application Currency" on page 114
`DefaultRoundingMode`	Default rounding mode for money and currency amount calculations	"Choosing a Rounding Mode" on page 115
`MulticurrencyDisplayMode`	Determines whether PolicyCenter displays currency selectors for monetary values	"Setting the Currency Display Mode" on page 115

IMPORTANT If you integrate the core applications in Guidewire InsuranceSuite, you must set the values of `DefaultApplicationCurrency` and `MulticurrencyDisplayMode` to be the same in each application.

See also

- "Globalization Parameters" on page 57 in the *Configuration Guide*

Setting the Default Application Currency

You must set configuration parameter `DefaultApplicationCurrency` to a typecode in the `Currency` typelist. You must set this configuration parameter even if you configure PolicyCenter with a single currency.

You set the default application currency in file `config.xml`, for example:

```
<param name="DefaultApplicationCurrency" value="usd"/>
```

Guidewire applications that share currency values must have the same `DefaultApplicationCurrency` setting in their respective `config.xml` files.

The base PolicyCenter configuration sets the default application currency to the U.S. dollar.

IMPORTANT The `DefaultApplicationCurrency` parameter setting is permanent. After you set the parameter and then start the server, you cannot change the value.

Choosing a Rounding Mode

There are a number of factors to consider as you select a rounding mode. If you are creating a multicurrency transaction, PolicyCenter converts the `TransactionAmount` that you enter in the transaction currency to the claim currency. PolicyCenter then stores this amount as the `ClaimAmount`. The rounding mode that you stipulate and which PolicyCenter uses in this calculation can result in unexpected behavior at a later time as PolicyCenter makes Payments against the Available Reserves.

By default, PolicyCenter enforces a constraint that demands that the payments in the claim currency not exceed the reserves in the claim currency. If the selected rounding modes result in rounding upwards for particular payment amounts, and downwards for particular reserve amounts, then it is possible to violate this rule. This can occur even though the result does not exceed the Available Reserves if you consider the Transaction Amounts alone.

Therefore, Guidewire recommends that you make a careful selection of the rounding modes for the payments and the reserves. For example, you do not want ever to have the sum of the rounded claim-currency payments exceed the sum of the rounded claim-currency reserves. The default rounding modes for Payments and Reserves of `Down` and `Up`, respectively, achieve this invariant. Other combinations are possible as well.

Guidewire recommends that you choose `HALF_UP` or `HALF_EVEN` for both Payment and Reserve rounding modes in the following cases:

1. A downstream system, such as your General Ledger accounting system, expects a particular rounding mode to be used while determining the conversion.
2. You consider using `HALF_UP` or `HALF_EVEN` rounding to be more correct, and are comfortable with the possible side effect listed in the previous discussion.

See also

- “DefaultRoundingMode” on page 58 in the *Configuration Guide*

Setting the Currency Display Mode

You use configuration parameter `MulticurrencyDisplayMode` in `config.xml` to set the currency display mode for your instance of PolicyCenter. You must set a currency display mode for PolicyCenter. Set the parameter to one of the following values:

- `SINGLE`
- `MULTIPLE`

Setting the currency display mode to SINGLE disables currency selectors for entering monetary amounts in the user interface. If you configure PolicyCenter with one currency only, then set MulticurrencyDisplayMode to SINGLE. If you configure PolicyCenter with multiple currencies, then set MulticurrencyDisplayMode to MULTIPLE.

IMPORTANT The MulticurrencyDisplayMode parameter setting is permanent. After you change the value of MultiCurrencyDisplayMode to MULTIPLE and then start the server, you cannot change the value back to SINGLE again. The MultiCurrencyDisplayMode parameter is in config.xml.

The value that you set for MulticurrencyDisplayMode influences where you specify certain configuration settings that influence the display and storage of monetary amounts.

Specifying Regional Formats for a Currency in Single Currency Mode

Currency display mode	Decimal digits to show...
SINGLE	PolicyCenter uses the value of the appscale attribute on the money data type in datatypes.xml to determine the number of decimal digits to show in the interface. If you do not set a value for appscale, then PolicyCenter uses the scale value to determine the number of decimal digits to show. **Note** The appscale attribute does not exist in the base configuration. You need to add this attribute to the <MoneyDataType> element in the datatypes.xml file.
MULTIPLE	PolicyCenter uses the value of the storageScale attribute on <CurrencyType> in the individual currency.xml files to determine the number of decimal digits to show in the interface for that currency. The appscale attribute is useful in multicurrency configurations for the money data types. However, with currencyamount data types, the storageScale attribute supersedes the appscale attribute on the <MoneyDataType> element in datatypes.xml. Make the currency storageScale value less than or equal to the appscale value and make the appscale value less than or equal to the scale value. The following must always be true. storageScale <= appScale <= scale

Configuring PolicyCenter to use single currency display mode means that all monetary amounts in the system are in the same currency. PolicyCenter uses the localization.xml file in the localization folder corresponding to the locale selected by the user to format the monetary amount. If there is no localization.xml file corresponding to the chosen locale, PolicyCenter uses the default locale's localization.xml file.

If you open Guidewire Studio and navigate in the **Project** window to **configuration** → **config** → **Localizations**, you can see the localization folders that are available. In the base configuration, the en_US, fr_FR, and ja_JP localization folders have localization.xml files that define regional formats customary to each region. Only the localization.xml file in the en_US folder has a <CurrencyFormat> definition because the base configuration is set to single currency mode and uses US dollars as the default currency.

To set regional formats for a currency, open, or if necessary create, the localization.xml file for that region and set the following <CurrencyFormat> attributes:

- negativePattern, positivePattern – Define how PolicyCenter displays positive and negative monetary amounts. PolicyCenter displays and formats the numeric value in place of the pound sign (#) in the pattern. PolicyCenter displays all other characters in the pattern as they are without modification.

  For example, the positive pattern $# displays the numeric value 32 as $32.00. The negative pattern ($#) displays the numeric value -5 as ($5.00).
- zeroValue – Defines how PolicyCenter displays zero amounts. For example, the zero value pattern can be 0 (zero) or - (dash). If the numeric value of a monetary amount is null, PolicyCenter displays the amount as empty or blank. The monetary amount must be 0.00 for the zero value pattern to be used.

For example:

```
<Localization xmlns="http://guidewire.com/localization">
  <GWLocale code="en_US" name="English (US)" typecode="en_US">
    ...
    <CurrencyFormat negativePattern="($#)" positivePattern="$#" zeroValue="-"/>
  </GWLocale>
</Localization>
```

Because all monetary values are in the default currency, any `<CurrencyFormat>` definitions must specify the monetary format for that one currency. It is possible to set slightly different formatting based on local custom, but all monetary formatting must be for the one default currency.

IMPORTANT Guidewire strongly recommends that you set configuration parameter `DefaultApplicationCurrency` to its correct value if you set `MulticurrencyDisplayMode` to `SINGLE`. Doing so ensures the correctness of the data in the database if you upgrade to multicurrency at a later time.

See also

“Configuring a localization.xml File” on page 81

chapter 14

Configuring Geographic Data

This topic describes how to configure the typelists `Jurisdiction`, `Country`, and `State`, and configuration file `zone-config.xml` for the territorial data in your instance of PolicyCenter.

This topic includes:

Working with Country Typecodes

In the base configuration, Guidewire provides the country typelist `Country.ttx`, which defines a standard set of countries. The country information includes the ISO country code and the English language country name. The ISO country code is the two-character uppercase ISO 3166 code for a country or region. To view a list of countries codes, refer to the following web site:

```
http://www.iso.org/iso/country_codes/iso_3166_code_lists /country_names_and_code_elements.htm
```

Individual country definitions can provide additional country information as well. For example, the `Country.ttx` typecode definition for `VA`, Vatican City, indicates that the currency in use is the euro, code `eur`.

Configuring Country Address Formats

The `country.xml` files in the country folders define settings that control the appearance of address information for each country in PolicyCenter. In Guidewire Studio, to access a country's `country.xml` file, you can navigate in the **Project** window to **configuration** → **config** → **geodata** → *countryCode*. For example, the `country.xml` file for the address-related information in Australia is in the following location in Studio:

configuration → config → geodata → AU

In the base configuration, Guidewire provides `country.xml` definition files for the following countries:

- `AU` – Australia
- `CA` – Canada
- `DE` – Germany
- `FR` – France
- `GB` – Great Britain
- `JP` – Japan
- `US` – United States

You use `country.xml` to set address-related configuration for a single country. In this file, you can set the following attributes on `<Country>`.

> **Note:** For attributes that take display key values, PolicyCenter uses the `display.properties` file defined in the associated localization folder, such as **Localization → ja_JP** for Japan. If there is no `display.properties` file in the localization folder for that region, PolicyCenter uses the `display.properties` file defined in **Localization → en_US**. See "Localizing Display Keys" on page 45.

Attributes	Sets...
`PCFMode`	PCF mode for `GlobalAddressInputSet.pcf`. In the base configuration, Guidewire provides three modes for showing address information. The base configuration modes are: • `BigToSmall` – Used to format addresses for Japan • `PostCodeBeforeCity` – Used to format addresses for France and Germany • `default` – Used to format all other addresses The default value is `default`. You can add new PCF modes and use them in file `country.xml`.
`postalCodeDisplayKey`	Name of the display key to use for the postal code label in PolicyCenter. For example: • In the United States, `US` – `Web.AddressBook.AddressInputSet.ZIP` • In Japan, `JP` – `Web.AddressBook.AddressInputSet.Postcode` The default value is `Web.AddressBook.AddressInputSet.PostalCode`.
`cityDisplayKey`	Name of the display key to use for the city label for a country. For example: • In Great Britain, `GB`, – `Web.AddressBook.AddressInputSet.TownCity` The default value is `Web.AddressBook.AddressInputSet.City`.
`stateDisplayKey`	Name of the display key to use for the label designating the major administrative subdivisions in a country. For example: • In the United States, `US` – `Web.AddressBook.AddressInputSet.State` • In Canada, `CA` – `Web.AddressBook.AddressInputSet.Province` • In Japan, `JP` – `Web.AddressBook.AddressInputSet.Prefecture` The default value is `Web.AddressBook.AddressInputSet.State`.
`visibleFields`	Comma-separated list of address-related fields that you want to be visible in PolicyCenter for this country. For example, for `AU`, the values are: • `Country` • `AddressLine1` • `AddressLine2` • `AddressLine3` • `City` • `State` • `PostalCode` Any address field that you designate as visible in this file must correspond to the constants defined in `AddressOwnerFieldId.gs`.

See also

- "Configuring Address Data and Field Order for a Country" on page 135

- "Address Modes in Page Configuration" on page 138

Setting the Default Application Country

You set the default application country in `config.xml`, in configuration parameter `DefaultCountryCode`. The country code must be a valid ISO country code that exists as a typecode in the `Country` typelist. See the following web site for a list of valid ISO country codes:

```
http://www.iso.org/iso/english_country_names_and_code_elements
```

See "Globalization Parameters" on page 57 in the *Configuration Guide* for more information on this configuration parameter.

Configuring Jurisdiction Information

Guidewire applications divide jurisdictions into several areas:

- **National jurisdictions** – Japan or France, for example
- **State or province jurisdictions** – United States and Canada, for example
- **Other jurisdictions** – Other local regulatory jurisdictions at a level below the country level, such as Berlin, Germany

In the base configuration, Guidewire provides a `Jurisdiction` typelist that contains a set of pre-defined jurisdictions. This typelist is used by a number of PCF files to display a list of states or provinces, as well as jurisdictions that are not states or provinces.

Many PolicyCenter fields that appear to reference a state actually reference a jurisdiction instead. For example:

- `TaxLocation.State`
- `TaxLocationSearchCriteria.State`
- `TerritoryLookupCriteria.State`

Configuring State Information

PolicyCenter provides a `State` typelist to represent states, prefectures, or provinces. In the base configuration, the typelist provides typecodes for the following countries: Australia (AU), Canada (CA), Germany (DE), Japan (JP), and the United States (US). The typecodes in this typelist have `Name` values that are full, unabbreviated names, such as Washington for the US typecode `WA` and Western Australia for the AU typecode `AU_WA`.

Abbreviations corresponding to `State` typecodes are defined in the typelist `StateAbbreviation`. This typelist provides a set of typecodes with `Name` values that are abbreviations for states or provinces. In the base configuration, `StateAbbreviation` provides typecodes for the following countries: Australia (AU), Canada (CA), Germany (DE), and the United States (US).

Note: Japan (JP) does not use abbreviations for its prefecture names.

For countries that use state abbreviations, such as AU and US, there is a one-to-one relationship between the country's typecodes for `State` and `StateAbbreviation`. This relationship supports distinguishing between abbreviations that have the same value in more than one country. For example, the abbreviation WA is used both in the United States for the state of Washington and in Australia for the state of Western Australia.

For the countries defined in the `StateAbbreviation` typelist, you can use methods on the `State` and `StateAbbreviation` typelists to get abbreviations and state names, as described in the following topics.

State Typelist Abbreviation Methods

The following methods used by the `State` typelist are defined in `gw.entity.GWStateEnhancement`.

State.getAbbreviation

- **Parameters** – none
- **Return value** – `typekey.StateAbbreviation`
- **Comments** – Returns the abbreviation typekey for the state as defined in the `StateAbbreviation` typelist. This value can vary based on the current language setting. Returns `null` if no abbreviation is defined, such as for Japanese prefectures.

State.getState

- **Parameters** – `country:Country, anAbbreviation:String`
- **Return value** – `typekey.State`
- **Comments** – Look up a `State` by using the country typecode and the country-dependent, localized state abbreviation. For example, if the language is U.S. English:
  - `getState(TC_AU, "WA")` returns Western Australia.
  - `getState(TC_US, "WA")` returns Washington.

State.getStateAbbreviation

- **Parameters** – `country:Country, anAbbreviation:String`
- **Return value** – typekey.`StateAbbreviation`
- **Comments** – Looks up a `StateAbbreviation` by using the country and the country-dependent, localized state abbreviation. For example, if the language is U.S. English:
  - `getStateAbbreviation(TC_AU, "WA")` returns the `StateAbbreviation` typekey for Western Australia.
  - `getStateAbbreviation(TC_US, "WA")` returns the `StateAbbreviation` typekey for Washington.

StateAbbreviation Typelist Abbreviation Method

There is one method used by the `StateAbbreviation` typelist that is defined in `gw.entity.GWStateAbbreviationEnhancement`:

`StateAbbreviation.getState`

- **Parameters** – none
- **Return value** – `typekey.State`
- **Comments** – Returns the `State` typekey associated with the current `StateAbbreviation` typelist.

Configuring Zone Information

PolicyCenter uses `zone-config.xml` files to define one or more zones. A *zone* is a combination of a country and zero or more address elements, such as a city or state in the United States or a province in Canada. You can configure zones to apply to any area in a single country. In the United States, for example, you typically define zones by state, city, county, and ZIP code. There is a separate `zone-config.xml` file for each country defined in the base configuration.

PolicyCenter uses zones for the following:

- Region and address auto-fill
- Business week and holiday definition by zone

To use zones in your application, you must import zone files, either files that you have created or files that you have purchased from a vendor. For example, your company might have bought a dataset from a company like GreatData. You then use this dataset to plan your zone configuration. For more information, see “Importing Address Zone Data” on page 128

> **Note:** PolicyCenter ships with sample address data for Australia, Canada, Canada with FSA, Germany, France, Japan, and the United States of America. This data is for use in testing and is not suitable for a production environment.

In planning, consider what parts of an address you need to auto-fill and the format of the ZIP or postal codes. Also, note if any of the values are not unique for a country. For example, the country has two cities with the same name.

While most countries have a similar format for addressing, there can be differences. For example, France uses a five-digit postal code plus a city. Additionally, France has a small locality format that includes more information.

You can define multiple zone types for a country. For example, you can divide your country into zones by county, state, and city. Most cities are in a single zone, such as a state in the United States or a province in Canada.

Zone types are defined as typecodes in the `ZoneTypes` typelist. You use these typecodes as elements in the `zone-config.xml` file. You can add your own zone types to this typelist.

A zone type can be defined to refer to another zone types by using a link. PolicyCenter uses links to look up a zone based on another zone. An example is the relationship between the ZIP code and the state for US zones. Your configuration could link ZIP code to state. For example, given the ZIP code of 94404, the application could auto-fill the corresponding state, California.

You do not necessarily need to define all the `Zone` subelements for a zone. For example, you can define a zone as the entire country, such as the following definition for all of Australia:

```
<Zones countryCode="AU"/>
```

This topic includes:

Location of Zone Configuration Files

PolicyCenter stores the `zone-config.xml` files in the `geodata` folder, each in its own individual country folder. For example, the `zone-config.xml` file for the address-related information in Australia is in the following location in the Studio **Project** window:

configuration → **config** → **geodata** → **AU**

In the base configuration, Guidewire defines zone hierarchies for the following geographical locations:

- `AU` – Australia
- `CA` – Canada
- `DE` – Germany
- `FR` – France
- `GB` – Great Britain
- `JP` – Japan
- `US` – Unites States of America

In a `zone-config.xml` file, you define:

- The links between the country’s zones, the *zone hierarchy*

- How PolicyCenter uses those links to extract the value of a zone from address data
- How PolicyCenter imports data from zone data files

Working with Zone Configuration Files

A `zone-config.xml` files contains the following XML elements:

- `<Zones>`
- `<Zone>`
- `<ZoneCode>`
- `<Links>`
- `<AddressZoneValue>`

For example, in the base configuration, PolicyCenter defines the zone hierarchy for the United States in the following file:

configuration → config → geodata → US → zone-config.xml

This file uses the following elements:

```
<Zones countryCode="US">
  <Zone code="zip" fileColumn="1" granularity="1" regionMatchOrder="1" unique="true">
    <AddressZoneValue>Address.PostalCode.substring(0,5)</AddressZoneValue>
    <Links>
      <Link toZone="city"/>
    </Links>
  </Zone>

  <Zone code="state" fileColumn="2" granularity="4" regionMatchOrder="3">
    <AddressZoneValue>
      Address.State.Abbreviation.DisplayName
    </AddressZoneValue>
    <Links>
      <Link toZone="zip" lookupOrder="1"/>
      <Link toZone="county"/>
      <Link toZone="city"/>
    </Links>
  </Zone>

  <Zone code="city" fileColumn="3" granularity="2">
    <ZoneCode>state + ":" + city</ZoneCode>
    <AddressZoneValue>
      Address.State.Abbreviation.DisplayName + ":" + Address.City
    </AddressZoneValue>
    <Links>
      <Link toZone="zip"/>
    </Links>
  </Zone>

  <Zone code="county" fileColumn="4" granularity="3" regionMatchOrder="2">
    <ZoneCode>state + ":" + county</ZoneCode>
    <AddressZoneValue>
      Address.State.Abbreviation.DisplayName + ":" + Address.County
    </AddressZoneValue>
    <Links>
      <Link toZone="zip" lookupOrder="1"/>
      <Link toZone="city" lookupOrder="2"/>
    </Links>
  </Zone>
</Zones>
```

This topic includes:

See also

- "Zone Import Command" on page 169 in the *System Administration Guide*.

<Zones>

This element defines the largest region, a country. The `zone-config.xml` files contains a single `<Zones>` element representing the zones in a specific country. Each `<Zones>` element contains one or more `<Zone>` elements.

The `<Zones>` element takes the following attributes:

Attribute	Description
`countryCode`	*Required.* Defines the country to which this zone configuration applies.
`dataFile`	For Guidewire internal use only. Do not set this attribute.

<Zone>

The `<Zone>` subelement of `<Zones>` defines a zone type. The zone type must exist in the `ZoneType` typelist. The `Zone` element takes the following attributes, all optional except for the `code` attribute:

Attribute	Description
`code`	Sets the zone type, which must be a valid value defined in the `ZoneType` typelist. You also use this value as a symbol in `<ZoneCode>` expressions to represent the data extracted from the data import file based on the column specified in the `fileColumn` attribute.

`fileColumn`	*Optional.* Specifies the column in the import data file from which to extract the zone data. The numeric value of `fileColumn` indicates the ordered number of the column in the data file. For example, `fileColumn="4"` specifies the fourth column in the data file.
	
	A `<Zone>` element without a `fileColumn` attribute can contain an expression that derives a value from the other zone values. For example, in the base configuration, Guidewire defines the following `fsa` zone in the Canadian `<Zones>` element:
	`<Zone code="fsa" regionMatchOrder="1" granularity="1">`
	`<ZoneCode> postalcode.substring(0,3) </ZoneCode>`
	`<AddressZoneValue> Address.PostalCode.substring(0,3) </AddressZoneValue>`
	`</Zone>`
	
	Both the `<ZoneCode>` and `<AddressZoneValue>` elements extract data from the actual address data by parsing the data into substrings.
	
	Notes:
	• Specify at least one `<Zone>` element with a `fileColumn` attribute. If you do not specify at least one `fileColumn` attribute, then PolicyCenter does not import data from the address zone data file.
	• Import address zone data upon first installing Guidewire PolicyCenter, and then at infrequent intervals thereafter as you update zone definitions.

`granularity`	Sets size levels for each defined zone. The smallest geographical region is 1. The next larger geographical region is 2, and so on. The sequence of numbers must be continuous. PolicyCenter uses this value with holidays and business weeks.
`orgZone`	For Guidewire internal use only. Do not set this attribute.

Attribute	Description
regionMatchOrder	Controls the order in which PolicyCenter uses these zones in matching algorithms. PolicyCenter uses this attribute as it matches users to a zone for location-based or proximity-based assignment. For example, in the base configuration for the United States, Guidewire defines the following <Zone> attributes for a county: <Zone code="county" fileColumn="4" regionMatchOrder="2" granularity="3" > ... </Zone> Setting the regionMatchOrder to 2 means that PolicyCenter matches county data second, after another zone, while matching a user to a location. The county value also: • Appears in the fourth column of the data file. • Is third in granularity, one size less than a state—granularity 4—and one size more than a city—granularity 2.
unique	*Optional.* Specifies whether this zone data is unique. For example, in the United States, a county data value by itself does not guarantee uniqueness across all states. There is a county with the name of Union in seventeen states and a county with the name of Adams in twelve states. To differentiate zones that can be identical—not unique—use a <ZoneCode> element to construct a zone expression that uniquely identifies that zone data. For example, you can combine the county name with the state name to make a unique identifier by defining the following <ZoneCode> subelement of <Zone>: <ZoneCode> state + ":" + county </ZoneCode> See "<ZoneCode>" on page 126.

<ZoneCode>

The <ZoneCode> element is a subelement of <Zone>. It is possible for zone information to not be unique, meaning that the zone import data column contains two or more identical values. In this case you need to use <ZoneCode> to define an expression that uniquely identifies the individual zone.

For example, in the United States, it is possible for multiple states to have a city with the same name, such as Portland, Oregon and Portland, Maine. To uniquely identify the city, you associate a particular state with the city. To make this association, create an expression that prepends the state import data value to the city import data value to obtain a unique city-state code. For example:

```
<Zone code="city" fileColumn="3" granularity="2">
  <ZoneCode>
    state + ":" + city
  </ZoneCode>
  ...
</Zone>
```

This expression instructs PolicyCenter to concatenate the State value with the County value, separated by a colon delimiter. The values you use to construct the expression must be valid <Zone> code values, other than constants, that are defined in a <Zones> element for this country.

See also

- "<Zone>" on page 125

<Links>

The <Links> element is a subelement of <Zone>. This element defines one or more <Link> elements, each of which defines a connection—a link—between one zone type and another. For example, in the base configuration,

Guidewire provides a <Link> element that defines a link between a county and a ZIP code in the United States. You can also use a link to define a lookup order to speed up searches.

Note: An address-config.xml file can define auto-fill elements that use these links. See "Configuring Autofill and Autocompletion in address-config.xml" on page 140.

The <Link> element has the following attributes:

Attributes	Description
lookupOrder	*Optional.* Tells the application the order in which to apply this value while performing a lookup. Specifying a lookup order can increase performance somewhat. If you do not specify a lookupOrder value, PolicyCenter uses the order that appears in the file.
toZone	*Required.* Defines a relationship to another zone for PolicyCenter to use if the <Zone> code value is not available for lookup.

For example, the following code defines a relationship between one zone type, county, and several other zone types. PolicyCenter uses these other zone types to look up an address if the address does not contain a county value.

```
<Zone code="county" fileColumn="4"  regionMatchOrder="2">
  ...
  <Links>
    <Link toZone="zip" lookupOrder="1"/>
    <Link toZone="city" lookupOrder="2"/>
  </Links>
</Zone>
```

This code has the following meaning:

- The first <Link> definition, <Link toZone="zip" lookupOrder="1"/>, creates a link from county to zip. If PolicyCenter cannot look up the address by its county value, then PolicyCenter attempts to look up the address first by zip value.
- The second <Link> definition, <Link toZone="city" lookupOrder="2"/>, creates a link from county to city. If PolicyCenter cannot look up the address by its county value or its zip value, it looks up the address by its city value.

See also

- "<Zone>" on page 125

<AddressZoneValue>

The <AddressZoneValue> element is a subelement of <Zone>. This optional element uses an expression to define how to extract the zone code from an Address entity. Use entity dot notation, such as Address.PostalCode, to define a value for this element. These expressions can be subsets of an element or a concatenation of elements.

PolicyCenter extracts a value from the address data that uses this element and matches the value against zone data in the database. For example, in the base configuration, PolicyCenter defines a <Zone> element of type postalcode for Canada. It looks similar to the following:

```
<Zone code="postalcode" fileColumn="1" unique="true">
    <AddressZoneValue>
      Address.PostalCode
    </AddressZoneValue>
</Zone>
```

Given this definition, PolicyCenter uses the value of the PostalCode field on the Address entity as the value of the postalcode zone.

Note: Guidewire recommends that you set this value even if there is a property defined on the Address entity that has the same name as the Zone name.

See also

- "<Zone>" on page 125

Importing Address Zone Data

You can import address zone data when you first install the product and at infrequent intervals thereafter as you receive data updates. You use the `zone_import` command to import zone data into staging tables, and then you use the `table_import` command to import the staging table data. For full instructions, see "Zone Import Command" on page 169 in the *System Administration Guide*.

In the base configuration, Guidewire provides the following sample zone data files in **configuration** → **config** → **geodata**:

- `AU-Locations.txt` – Australia
- `CA-Locations.txt` – Canada
- `DE-Locations.txt` – Germany
- `FR-Locations.txt` – France
- `GB-Locations.txt` – Great Britain
- `JP-Locations.txt` – Japan
- `US-Locations.txt` – Unites States of America

Guidewire provides these files for testing purposes to support auto-fill and auto-complete when users enter addresses. You must import one of these files to use it in testing those features. This data is provided on an as-is basis. The provided zone data files are not complete and might not include recent changes.

The formatting of individual data items in these files might not conform to your internal standards or the standards of third-party vendors that you use. For example, the names of streets and cities are formatted with mixed case letters, but your standards may require all upper case letters.

The `US-Locations.txt` file contains information that does not conform to United States Postal Service (USPS) standards for bulk mailings. You can edit this file to conform to your particular address standards, and then import that version of the file.

Guidewire does not provide complete and up-to-date zone data files that you can import with the `zone_import` command. You must create these files or obtain them from vendors, like GreatData.

Adding Basic Zone Types

You can define basic zone types for your country configuration in the `ZoneType` typelist.

The base configuration comes with a set of typecodes for Australia, Canada, France, Germany, Japan, and the United States. The base PolicyCenter configuration contains zones for:

- country
- city
- city kanji
- county
- state
- prefecture
- province
- postal code
- ZIP code
- FSA (Forward Sortation Area, first three letters of a Canadian postal code)
- post code area

- post code region

To edit the typelist:

1. If necessary, start PolicyCenter Studio as follows:

   At a command prompt, navigate to the `PolicyCenter\bin` directory and enter:

   ```
   gwpc studio
   ```

2. In the **Project** window, navigate to **configuration** → **config** → **Extensions** → **Typelist** and then double-click `ZoneType.ttx` to edit the typelist.

chapter 15

Configuring Address Information

Address data and formats in Guidewire PolicyCenter vary by country. For example, PolicyCenter shows a ZIP code or postal code field for an address depending on the country of the address. You can customize this feature, as described in this topic.

This topic includes:

Addresses in PolicyCenter

In PolicyCenter, an address represents a street or mailing address. An address can also contain geographic location information for use in proximity searches.

Guidewire PolicyCenter uses address information for the following business entities:

- Contacts
- Account locations
- Policy locations
- Policy addresses
- Producer addresses

PolicyCenter bases organizations and users on contacts. Therefore, these objects use addresses as well. PolicyCenter also uses address-related information for the following:

- Territory codes
- Tax locations
- WC excluded workplaces

However, these last items are largely specific to the United States and you can generally ignore them if you are not using the `en_US` region.

Read-only and Editable Addresses

PolicyCenter displays address information as read-only text or as editable text entry fields:“Understanding Global Addresses” on page 132

- PolicyCenter displays read-only addresses as read-only text on multiple lines. PolicyCenter uses the `country.xml` file for the current country and the `AddressFormatter` class to determine which address fields to show.
- PolicyCenter displays editable addresses as a set of editable text fields in which you can add, modify, or delete information. PolicyCenter uses the `country.xml` file for the current country to determine the address fields to show.

PolicyCenter displays a range selector—a drop-down list—in the address screen to enable you to select from an array of appropriate addresses. If you cannot edit an address, the address fields are dimmed. The drop-down list can also display a **New** command that you can use to create a new address.

Address Owners

PolicyCenter builds the address view around the concept of an address owner. The address owner identifies the object that owns a particular address. Additionally, the address owner controls how the address widget looks in the user interface and ensures that PolicyCenter saves the address properly. You can define different address owners depending on your requirements.

See also

- “AddressFormatter Class” on page 133
- “Address Modes in Page Configuration” on page 138
- “Address Owners” on page 139
- “AddressOwnerFieldId Class” on page 139

Understanding Global Addresses

The information that you see for an address depends on:

- The country of the address.
- Whether PolicyCenter displays the address as text entry fields or read-only text, as described at “Read-only and Editable Addresses” on page 132.

This topic includes:

Country XML Files Overview

The `country.xml` files define settings that control the appearance of address information in PolicyCenter. In Guidewire Studio, you can see that there is a `country.xml` file for each defined country. PolicyCenter stores the `country.xml` files in the `geodata` folder, each in its own individual country folder. For example, the `country.xml` file for the address-related information in Australia is in the following location in Studio:

configuration → **config** → **geodata** → **AU**

See "Configuring Address Data and Field Order for a Country" on page 135.

Modal Address PCF Files Overview

In general, the country of an address determines which address fields in the database are visible in the PolicyCenter user interface for that address. The page configuration file `GlobalAddressInputSet` has modal versions that make different sets of address fields visible. The modal versions of `GlobalAddressInputSet` also control the order in which PolicyCenter displays address fields.

You can configure the modal versions of PCF `GlobalAddressInputSet` to provide a modal version for each country for which you want to support address editing. However, in practice, the addresses of different countries follow a small number of patterns in terms of components of an address and their order of presentation. Components of an address include the street name, the house number, the city, and country. Some countries have additional address components, such as prefecture in Japan, state in the United States, and province in Canada.

In the base configuration, Guidewire provides the following modal versions of the PCF file `GlobalAddressInputSet`:

- `GlobalAddressInputSet.default` – Used for Australia, Canada, Great Britain, and the United States.
- `GlobalAddressInputSet.BigToSmall` – Used for Japan.
- `GlobalAddressInputSet.PostCodeBeforeCity` – Used for France and Germany.

To determine which modal PCF file is used for a country, you set the `PCFmode` attribute in the `country.xml` file for that country. See "PCFMode Attribute of the Country XML File" on page 136.

See also

- For more information on `GlobalAddressInputSet`, see "Address Modes in Page Configuration" on page 138.
- For more information on modal PCF files, see "Working with Shared or Included Files" on page 291 in the *Configuration Guide*.

AddressFormatter Class

The `AddressFormatter` class is used to covert addresses to localized strings for display as read-only address information. If you change, add, or delete columns of the `Address` entity, you must also update this class. Additionally, if you add a new country definition, you might need to update the switch statement in the `internalFormat` method of this class.

There are two types of addresses in PolicyCenter:

- Synchronized
- Unsynchronized

Unsynchronized addresses apply to policy location addresses and policy addresses. An address is unsynchronized if the following are all true:

- The address is part of a completed job (a promoted branch).
- The associated account location address has changed since the job was completed.

Unsynchronized addresses are read-only. There are several ways to handle this situation:

- Expand the unsynchronized address definition in the PCF to apply to the entire address. Currently, PolicyCenter does not treat the street address portion of the PCF as either synchronized or unsynchronized.
- Format the display string using the `gw.address.AddressFormatter` class.

Displaying Read-only Address Information

Guidewire PolicyCenter displays read-only address information in several different places. PolicyCenter displays read-only address information as a string in which the address values are separated by commas or by line-feeds. PolicyCenter uses class `gw.address.AddressFormatter` to manage and format read-only strings.

The `AddressFormatter` class consists of two parts:

- The class contains variables for all the address columns. You can extend the class to add new variables if you extend the `Address` entity with new columns.
- The class contains two versions of the `format` method with different signatures.
  - The following version of the `format` method formats an address as text and by default includes all fields. The `IncludeCountry` and the `IncludeCounty` properties of this class can hide the `Country` and `County` fields. This method has the following signature:

    ```
    format(address : AddressFillable, delimiter : String) : String
    ```
  - The following version of the `format` method formats an address as text and includes only the specified set of fields. This method has the following signature:

    ```
    format(address : AddressFillable, delimiter : String,
           fields : Set<AddressOwnerFieldId>) : String
    ```

  The method parameters have the following meanings:

Parameter	Description
`address`	The address to format.
`delimiter`	Use this delimiter to separate the various string elements. If the delimiter is a comma, then the method also adds a space after the comma.
`fields`	The set of fields to include in the address.

PolicyCenter calls class `AddressFormatter` from the following places:

- From `entity.Address.DisplayName`. Use the Entity Names editor to modify the address string definition.
- From the `Address.addressString` method, defined in enhancement `AddressEnhancement`
- From the `PolicyLocation.addressString` method, defined in enhancement `PolicyLocationEnhancement`
- From the `PolicyAddress.addressString` method, defined in enhancement `PolicyAddressEnhancement`

Each of these uses creates a new `AddressFormatter` instance, populates the variables and then returns `AddressFormatter.addressString`.

In particular:

- `Address.DisplayName` always passes the same parameters. The other three uses just pass through the three parameters that they receive. This method always return a comma-delimited address string. However, this method is simpler to use as it requires no additional parameters.
- `Address.addressString` is the more general form of usage with an address. This is especially true if the delimiter value is something other than a comma.
- `PolicyLocation.addressString` and `PolicyAddress.addressString` work in the same manner as the `Address.addressString` method, except that they pass in the internally enhanced values for the address values. Other code in these enhancements determines whether to use the associated address values, or the internal values.

See also

- “Additional Country and Address Configurations” on page 137

Addresses and States or Jurisdictions

The country of an address controls the label used for the state or province field of an address through the `stateDisplayKey` setting in `country.xml` for that country. The `Jurisdiction` and `State` typelists have definitions for states, provinces, and other jurisdictions, each of which can be filtered.

Some examples:

- For Japan, PolicyCenter displays Kanji address fields.
- For Canada, PolicyCenter displays the label **Province** for this field.

See also

- “Configuring the Country XML File” on page 136
- “Configuring Jurisdiction Information” on page 121

Address Configuration Files

The following configuration files play a role in address configuration.

Studio location	File	Description
configuration → **config** → **Extensions** → **Typelist**	`State.ttx`	Used for addresses and locations.
	`StateAbbreviation.ttx`	Abbreviations used for states, provinces, and jurisdictions.
	`Jurisdiction.ttx`	Jurisdictions that regulate insurance and licensing. Similar to the `State` typelist.
	`Country.ttx`	Definitions of country codes for countries and regions.
configuration → **config** → **geodata** → *`CountryCode`*	`address-config.xml` – Defines formats to use for address autofill and input masks for postal codes.	Each country code has its own settings for these three files
	`country.xml` – See “Configuring the Country XML File” on page 136	
	`zone-config.xml` – See “Configuring Zone Information” on page 122	
configuration → **config** → **geodata**	*`CountryCode`*`-locations.txt`	Mappings between postal codes and cities for a country

Configuring Address Data and Field Order for a Country

You use country-specific `country.xml` files to configure the data and order that PolicyCenter uses to display addresses for specific countries. A country maps to an address mode by using the settings in `country.xml`.

If you add a new address format for a country or you add a new `Address` property, you must configure the files that support read-only addresses. See “Additional Country and Address Configurations” on page 137.

This topic includes:

- “Configuring the Country XML File” on page 136
- “Additional Country and Address Configurations” on page 137

Configuring the Country XML File

PolicyCenter stores `country.xml` files in country-specific folders under the `geodata` folder, which you can access in Guidewire Studio. For example, the `country.xml` file for Japan is stored in the following folder:

configuration → **config** → **geodata** → **JP**

The file `country.xml` defines the following address-related attributes:

Attribute	Description	Related topic
`visibleFields`	Which address fields to display	"Visible Fields Attribute of the Country XML File" on page 136
`PCFMode`	Order in which to display address fields	"PCFMode Attribute of the Country XML File" on page 136
`postalCodeDisplayKey`	Label for the postal code field	"Postal Code Display Key Attribute of the Country XML File" on page 137
`stateDisplayKey`	Label for state or province field	"State Display Key Attribute of the Country XML File" on page 137

Visible Fields Attribute of the Country XML File

The attribute `visibleFields` defines the set of address fields that are visible for this country. For example:

France	`visibleFields="Country,AddressLine1,AddressLine2,AddressLine3,PostalCode,City, CEDEX,CEDEXBureau"`
Japan	`visibleFields="Country,PostalCode,State,City,CityKanji,AddressLine1,AddressLine1Kanji, AddressLine2,AddressLine2Kanji"`
United States	`visibleFields="Country,AddressLine1,AddressLine2,AddressLine3,City,State,PostalCode, County"`

The order of the fields does not matter. The fields must be defined in the class `AddressOwnerFieldId`.

See "AddressOwnerFieldId Class" on page 139.

If a `country.xml` file does not have `visibleFields` defined, PolicyCenter uses the set of address fields defined for the United States.

PCFMode Attribute of the Country XML File

The attribute `PCFMode` of the `country.xml` file determines which modal version of the address PCF file that PolicyCenter uses for specific countries. For example, country-specific `country.xml` files individually specify the PCF mode for the following countries:

France	`PCFMode="PostCodeBeforeCity"`
Japan	`PCFMode="BigToSmall"`

If a `country.xml` file does not define the `PCFMode` attribute, PolicyCenter uses the default modal version of the address PCF file. In the base configuration, the default version is generally suitable for English-speaking countries.

See also

- "Address Modes in Page Configuration" on page 138

Postal Code Display Key Attribute of the Country XML File

The attribute `postalCodeDisplayKey` sets the display key to use as the label for the postal code of an address. For example, PolicyCenter uses the following display keys to label the postal code field for the following countries:

Japan	`postalCodeDisplayKey="Web.AddressBook.AddressInputSet.Postcode"`
United States	`postalCodeDisplayKey="Web.AddressBook.AddressInputSet.ZIP"`

If a `country.xml` file does not define `postalCodeDisplayKey`, PolicyCenter uses the value `"Postcode"`.

State Display Key Attribute of the Country XML File

The attribute `stateDisplayKey` sets the display key to use as the label for state or province of an address. For example, PolicyCenter uses the following display keys to label the state or province field for the following countries:

Japan	`stateDisplayKey="Web.AddressBook.AddressInputSet.Prefecture"`
United States	`stateDisplayKey="Web.AddressBook.AddressInputSet.State"`

If a `country.xml` file does not have `stateDisplayKey` defined, PolicyCenter uses the value of state display key for the United States.

Additional Country and Address Configurations

If you add a new country, in addition to the configurations described previously for the `country.xml` file, you must also add the country to a method of the `AddressFormatter` class. Add a new case option to the switch statement of the `AddressFormatter.internalFormat` method to handle the formatting of the address string for the new country.

If you extend the `Address` entity to add a new column, you must also incorporate this column into the following:

Class, enhancement, or configuration file	Task
`gw.address.AddressFormatter`	Add a new `case` option to the `switch` statement of the `AddressFormatter.internalFormat` method to handle the formatting of the address string for the new country.
`Address.en`	Modify the definition of the display name through the Entity Names editor. See “Using the Entity Names Editor” on page 125 in the *Configuration Guide* for details.
`AddressOwnerFieldID.gs`	Add a variable for the new `Address` column to this class. See “AddressOwnerFieldId Class” on page 139.
`gw.policylocation.PolicyLocationEnhancement`	1. Modify the `PolicyLocation.addressString` method as needed and populate any added column before calling the `AddressFormatter` class. 2. Add the `get` and `set` values for the new internal address columns.
`gw.policyaddress.PolicyAddressEnhancement`	1. Modify the `PolicyAddress.addressString` method as needed and populate any added column before calling the `AddressFormatter` class. 2. Add the `get` and `set` values for the new internal address columns. 3. Add the new columns to the `copyToNewAddress` and `copyFromAddress` functions.

See also

- “AddressFormatter Class” on page 133
- “Example: Adding a Country with a New Address Field” on page 144

Address Modes in Page Configuration

PolicyCenter uses a modal PCF file for all addresses, `GlobalAddressInputSet`. Modal versions of the global address PCF file determine the order of fields in addresses of specific countries.

In the base configuration, PolicyCenter provides the following modal versions of `GlobalAddressInputSet`.

Modal Address PCF File	Country
`GlobalAddressInputSet.BigToSmall`	• Japan
`GlobalAddressInputSet.PostCodeBeforeCity`	• France • Germany
`GlobalAddressInputSet.default`	• Australia • Canada • Great Britain • United States

To see the `GlobalAddressInputSet` PCF files, navigate in the Guidewire Studio **Project** window to **configuration** → **config** → **Page Configuration** → **pcf** → **address**.

Mapping Countries to Modes

A country maps to a mode through the settings in `country.xml`.

Controlling Field Properties

Each modal PCF file uses an implementation of the Gosu interface `AddressOwner` to control the following field properties:

- `available`
- `editable`
- `required`
- `visible`

Gosu Address Formatter

PolicyCenter uses Gosu class `AddressFormatter` to format the address display fields. You can extend `AddressFormatter` to handle additional countries.

Address Owners

`AddressOwner` is the interface for a helper object that is passed to the `GlobalAddressInputSet` PCF file. The helper object provides a way to set and get a single address on the enclosing entity. It also provides methods that you can use to set a field as required or visible, for example. Following are some properties on `AddressOwner`.

Property	Description
`Address`	Sets (or retrieves) a single address on the enclosing entity. For example, you can use this to set or get the primary address for a `Contact`. ClaimCenter automatically creates a new `Address` object if you use a Gosu expression of the form: `owner.Address.State = someState`
`HiddenFields`	Set of address fields that ClaimCenter hides (does not show) in the application interface.
`RequiredFields`	Set of address fields for which the user must supply a value.

In the base configuration, PolicyCenter provides the following classes that implement `AddressOwner` or extend a class that implements `AddressOwner`:

```
AddressOwnerBase
OptionalSelectedCountryAddressOwner
AccountAddressSearchOwner
AddressCountryCityStatePostalCodeOwner
ContactResultAddressSearchOwner
PolicyInfoAddressOwner
```

AddressOwnerFieldId Class

Guidewire provides a `gw.api.address.AddressOwnerFieldId` class that provides type safety for `Address` entity fields. If you extend the `Address` entity with a new column, you must add the new column to this class as a new constant.

In the base configuration, `AddressOwnerFieldId` provides the following constants that represent address fields:

```
ADDRESSLINE1
ADDRESSLINE2
ADDRESSLINE1KANJI
ADDRESSLINE2KANJI
ADDRESSLINE3
ADDRESSTYPE
CEDEX
CEDEXBUREAU
CITY
CITYKANJI
COUNTRY
COUNTY
DESCRIPTION
POSTALCODE
STATE
VALIDUNTIL
```

The class `AddressOwnerFieldId` also defines a set of constants that use these address field ID constants. Some examples:

```
public final static var ALL_PCF_FIELDS : Set<AddressOwnerFieldId> =
  { ADDRESSLINE1, ADDRESSLINE2, ADDRESSLINE3, CITY, COUNTY, STATE, POSTALCODE, COUNTRY,
    ADDRESSLINE1KANJI, ADDRESSLINE2KANJI, CITYKANJI, CEDEX, CEDEXBUREAU }.freeze()


public final static var CITY_STATE_ZIP : Set<AddressOwnerFieldId> =
    { CITY, STATE, POSTALCODE, CEDEX, CEDEXBUREAU }.freeze()


public final static var HIDDEN_FOR_SEARCH : Set<AddressOwnerFieldId> =
  { ADDRESSLINE1, ADDRESSLINE2, ADDRESSLINE3, COUNTY, ADDRESSLINE1KANJI,
    ADDRESSLINE2KANJI, CEDEX, CEDEXBUREAU }.freeze()
```

Address Autocompletion and Autofill

PolicyCenter supports automatic fill-in and completion of address information.

Address Autofill

The *autofill* feature enables you to enter a value in one address field in a Guidewire application and have the application fill in other address fields automatically.

For example, if configured, entering a postal code causes PolicyCenter to fill in the city and state or province fields automatically. For example, the user enters a postal code of 99501 for a United States address. PolicyCenter sets the city to Anchorage, the state to Alaska, and the county to Anchorage.

To trigger autofill, the user must enter a value and then navigate away from the initial address field, such as the postal code field. It is also possible to trigger this functionality by using the autofill icon next to certain address fields. There must also be a unique match, such as state and city to postal code.

Address Autocompletion

The *autocomplete* feature enables you to enter the first few characters of a field and see a drop-down list showing the matching values. The drop-down list displays possible completions of the entered characters based on the values in other address fields.

For example, suppose that a user sets the **State** value in a United States address to CA (California). The user then moves to the **City** field and enters `Pa` as the start of a city name. If configured, PolicyCenter opens a drop-down list that shows names of cities in California that start with `Pa`, such as:

- Pacifica
- Pacoima
- Palm Springs
- Palo Alto
- Panorama City
- Pasadena
- Paso Robles

The user can then select one of the items on the drop-down list to populate the address field.

This topic includes:

Configuring Autofill and Autocompletion in address-config.xml

To configure the fields that support address autofill and autocompletion for a country, configure that country's `address-config.xml` file.

> **Note:** The `address-config.xml` file uses zones defined in `zone-config.xml`. See "Working with Zone Configuration Files" on page 124.

PolicyCenter stores `address-config.xml` files in country-specific folders under the `geodata` folder, which you can access in Guidewire Studio. For example, the `address-config.xml` file for Japan is stored in the following folder:

configuration → **config** → **geodata** → **JP**

The following default configuration for the United States, in **configuration** → **config** → **geodata** → **US**, shows elements that you can use in this file:

```
<AddressDef name="US">
  <Match>
    <Field name="Country" value="US"/>
  </Match>
  <Fields>
    <Field name="City" zonecode="city">
      <AutoFillFromZone code="zip"/>
      <AutoFillFromZone code="state"/>
    </Field>
    <Field name="County" zonecode="county">
      <AutoFillFromZone code="zip"/>
      <AutoFillFromZone code="city"/>
      <AutoFillFromZone code="state"/>
    </Field>
    <Field name="State" zonecode="state">
      <AutoFillFromZone code="zip"/>
      <AutoFillFromZone code="city"/>
    </Field>
    <Field name="PostalCode" zonecode="zip">
      <AutoFillFromZone code="city"/>
      <AutoFillFromZone code="state"/>
      <ValidatorDef
         value="[0-9]{5}(-[0-9]{4})?"
         description="Validator.PostalCode.US"
         input-mask="#####-####"/>
    </Field>
  </Fields>
</AddressDef>
```

The following table describes the elements in the `address-config.xml` file.

`AddressDef`	The name of the address format. This element takes a `name` attribute and an optional `priority` attribute. By convention, the `name` is the country code. A country can have more than one address format—for example, it is possible that different regions have different formats. The `priority` attribute specifies which address definition has priority if several of them match. The highest priority is 1, the next lower one is 2, and so on.
`Match`	Each `AddressDef` must contain one `Match` element. PolicyCenter matches only on the country. The `Match` element contains a `Field` subelement that defines the `name` and `value` attributes that the application uses to determine which definition applies.
`Fields`	Each `AddressDef` contains a single `Fields` element that contains a list of the address `Field` elements.
`Field`	Specifies an address field. Each field takes a name that must match a property on the `Address` entity. The `Field` element can appear in the `Match` element or the `Fields` element. In the `Match` element, the `Field` element has a required `name` attribute and an optional `value` attribute, which is the `code` value from the `Country` typelist. It has no child elements. In the `Fields` element, the `Field` element has a required `name` attribute and optional `zonecode` and `autoCompleteTriggerChars` attributes. It can also have the child elements `AutoFillFromZone` and `ValidatorDef`. • The `value` attribute is a valid value for the `Field`. This value is usually a code value from a typelist, such as a state typecode from the typelist for a `Field` with the `name` set to `State`. • The `zonecode` attribute corresponds to a `Zone` code configured for the given country in `zone-config.xml`. This value links the `Address` configuration to the `Zone` configuration. For information on zone configuration, see “Working with Zone Configuration Files” on page 124. • The `autoCompleteTriggerChars` attribute specifies the number of characters to enter before the application triggers autocomplete. The default value is 0 (zero).
`AutoFillFromZone`	Specifies a field that is examined for autofill and autocompletion of the field. PolicyCenter uses the zone `code` to look up the field `value`. See “Working with Zone Configuration Files” on page 124.
`ValidatorDef`	Contains information for validating the field in the optional attributes `value`, `description`, and `input-mask`. **Note:** Do not define field validators in `address-config.xml`. Guidewire recommends that you define them in `fieldvalidators.xml`. See “Field Validators” on page 259 in the *Configuration Guide*.

The previous address-config.xml example defines a Match on the country with the match value as US. Additionally, it defines four fields:

- City
- County
- State
- PostalCode

Each field defines one or more AutoFillFromZone elements. Looking at the County, you can see that the application fills the County from the zip, the city, and the state. Each AutoFillFromZone entry must correspond to a <Link> definition in zone-config.xml, as described in "Working with Zone Configuration Files" on page 124. That file specifies that autofill can use ZIP code and city for the look-up operation.

As PolicyCenter loads address-config.xml, it validates the configuration. PolicyCenter verifies that every Field element, regardless of whether it is defined in Match or Fields, corresponds to a field on the Address entity. Then, PolicyCenter verifies that each AutoFillFromZone element references a zone in zone-config.xml.

See also

- "Automatic Address Completion and Fill-in Plugin" on page 253 in the *Integration Guide*

Configuring Autofill and Autocompletion in a PCF File

You can configure autofill and autocomplete after you have configured zone mapping and addressing and have imported your zone data. The main classes used for address autofill and autocomplete in PCF files are the following classes:

- gw.api.contact.AddressAutocompleteHandler provides methods for getting matching values from the database.
- gw.api.contact.AddressAutocompleteUtil supplies methods for automatically completing an address field.

You can integrate these components into a PCF file to:

- Complete a field automatically after you type in a few characters. For example, if you enter 941 for the ZIP code, the autocomplete list shows 94110, 94111, 94112, and so forth.
- Fill other address fields after you complete a field and tab out of it. For example, after you enter the ZIP code 94115, autofill can fill in the city, county, and state as San Francisco, San Francisco, and California respectively. This level of autofill happens only if there is a unique match between the fields. For example, with San Francisco as the city and California as the state, autofilling the ZIP code would not work because San Francisco has multiple ZIP codes. This matching is done in the default autocompletion and autofill plugin, matching between the fields and the zone definitions.
- Force an override of already filled fields. By default, if a field already has a value, PolicyCenter does not overwrite it. You can force it to override the field value with zone data by using a method on AddressAutocompleteUtil to autofill the address.

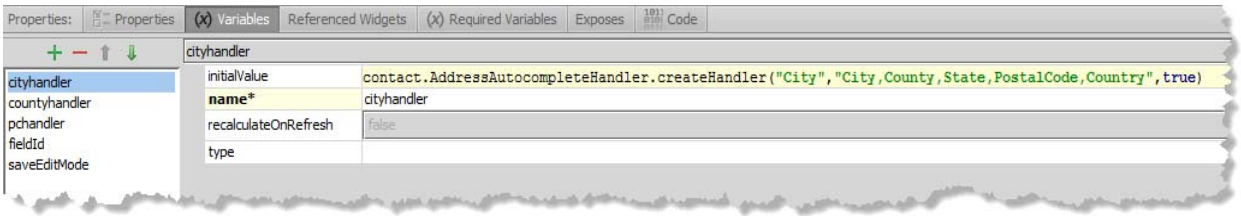
There are two PCF widgets that you can use to support autofill: the AddressAutoFillRange and the AddressAutoFillInput widgets. Both widgets provide a small icon for autofilling the fields on demand.

Adding Autocomplete

To use the address autocomplete feature, first create an AddressAutocompleteHandler instance in the appropriate PCF file. You create the instance by calling the factory method gw.api.contact.AddressAutocompleteHandler.createHandler with the following arguments:

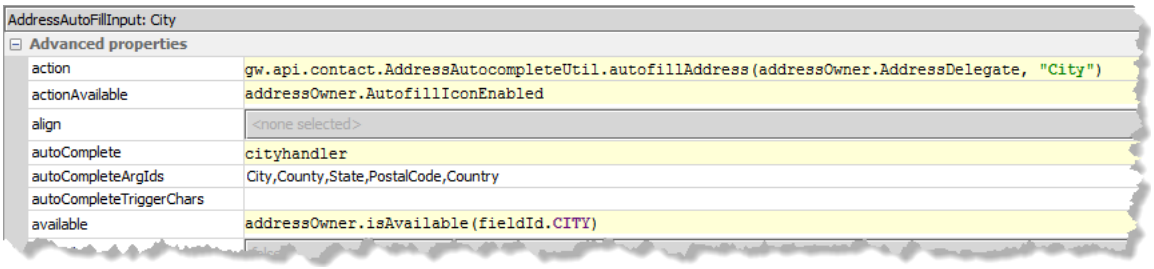
- A String identifying the field to complete
- A comma-separated list of the reference fields, field names of the Address entity, that the application passes to the handler
- A Boolean value that specifies whether PolicyCenter waits for a key press before fetching suggestions

You edit the PCF file in Guidewire Studio and add the factory method in the **Variables** tab of the PCF widget. The call to the method is in the `initialValue` field. The following figure shows an example of constructing an `AddressAutocompleteHandler` called `cityhandler` in the `GlobalAddressInputSet.default.pcf` file:



From the method call, you can see that this handler operates on the `City` field. The handler requires the following set of reference fields: `City`, `County`, `State`, `PostalCode`, and `Country`.

Next, edit the widget that uses the handler. Specify which handler is responsible for autofill in the `autoComplete` property and the arguments the handler expects in the `autoCompleteArgIds` field.



This example uses an `AddressAutoFillInput` widget for the city field. This widget displays an autofill icon beside the field that acts as a cue. You do not have to use this specific type of widget for this purpose.

Adding Autofill

Setting `PostOnChange` enables you to change one field in a Guidewire user interface based on a change in another field. You configure `PostOnChange` by setting the `PostOnChange` element of an input widget.

The following figure depicts the use of a `PostOnChange` element in the `City` input field:

The `onChange` property uses the `AddressAutocompleteUtil.autofillAddress` method to get the `City` value.

PolicyCenter fills in the field value only if there is a unique match. If there is no unique match, the autocomplete configuration on the same field forces the application to show a narrowed list of possible matches.

Address Automatic Completion and Autofill Plugin

The address automatic completion plugin template `IAddressAutocompletePlugin` declares methods that support automatic completion and fill-in for PolicyCenter. The base configuration of PolicyCenter provides the Java implementation class `DefaultAddressAutocompletePlugin`. This class supports the use of `address-config.xml` and `zone-config.xml` described earlier in this topic. If you want to implement your own autofill and autocompletion configuration, such as one that does not use `zone-config.xml`, you can create a class that implements `IAddressAutocompletePlugin`. Alternatively, you can extend `DefaultAddressAutocompletePlugin` and override specific methods.

Two methods that you might override are:

- `autofillAddress(address : AddressFillable, triggerFieldName : String, doOverride : boolean)`

Autofills the address if there is a unique match.

- `getStates(country : Country): States[]`

  Gets the states for the given country.

Example: Adding a Country with a New Address Field

This example shows how to add a new New Zealand address field named Suburb to PolicyCenter and ContactManager. The example configures classes, entities, and configuration files that support address fields so the new field can be used in the user interface and be available for autofill and autocompletion.

Basic Configuration of the Suburb Field

The region being added is New Zealand, and the new address field is `Suburb`. This example is not a full example showing how to perform a complete localization configuration, but rather focuses on configuring the new address field to work in ContactManager and PolicyCenter. When configured, if you edit a contact's address and set the country to New Zealand, you will be able to use the Suburb field.

Note: Except where the instructions say explicitly to work only in ContactManager or only in PolicyCenter, do everything in both ContactManager and PolicyCenter.

To perform the basic configuration in Studio

1. Add a `Suburb` field to the `GlobalAddress` entity.
   a. Navigate to **configuration** → **config** → **Extensions** → **Entity** and double-click `GlobalAddress.etx` to open it in the editor.
   b. Right-click the top element on the left, the **delegate(extension)** `GlobalAddress`, and choose **Add new** → **column**.
   c. Enter the following values on the right:
   - `name` – `Suburb`
   - `type` – `varchar`
   - `nullok` – `true`
   - `desc` – `Address field for countries that have both a suburb and a city field`

   d. On the left, right-click the new `Suburb` column and choose **Add new** → **columnParam**.
   e. Enter the following values on the right:
   - `name` – `size`
   - `value` – `50`

2. Make the following changes so the `Suburb` field can participate in autofill.
   a. Press `Ctrl+N` and enter `AddressFillableExtension`, and then double-click `AddressFillableExtension` in the search results to open this class.
   b. Add the following lines of code:

   ```
   property get Suburb() : String
   property set Suburb(value : String)
   ```

   c. Press `Ctrl+N` and enter `AddressFillableExtensionImpl`, and then double-click `AddressFillableExtensionImpl` in the search results to open this class.
   d. Add the following line of code:

   ```
   var _Suburb : String as Suburb
   ```

   e. Press `Ctrl+N` and enter `UnsupportedAddressFillable`, and then double-click `UnsupportedAddressFillable` in the search results to open this class.

f. Add the following lines of code:

```
override property get Suburb(): String {
  return _unsupportedBehavior.getValue<String>()
}
override property set Suburb(value: String) {
  _unsupportedBehavior.setValue("Surburb")
}
```

g. Press `Ctrl+N` and enter `AddressAutofillableDelegate`, and then double-click `AddressAutofillableDelegate` in the search results to open this class.

h. Add the following lines of code:

```
override property set Suburb(value : String) { _af.Suburb = value }
override property get Suburb() : String { return _af.Suburb }
```

i. Press `Ctrl+N` and enter `AddressEntityDelegate`, and then double-click `AddressEntityDelegate` in the search results to open this class.

j. Add the following lines of code:

```
override property get Suburb() : String {
  return _ao.Address.Suburb }
override property set Suburb(value : String) {
  _ao.Address.Suburb = value }
```

3. Add a New Zealand locale type and a `suburb` zone type.

a. Navigate in the **Project** window to **configuration** → **config** → **Metadata** → **Typelist** and right-click `LocaleType.tti`.

b. Choose **New** → **Typelist extension** to create the file `LocalType.ttx`.

Note: If you have a language pack installed, you see a **Typelist Extension** dialog enabling you to choose one of several `LocaleType.ttx` locations. Choose the one in `configuration/config/extensions/typelist/` and then click OK.

Note: If you see a message saying that typelist extension is not allowed, then the file `LocalType.ttx` already exists. Open that file instead.

c. In the editor for `LocaleType.ttx`, right-click the top element on the left, **typelistextension** `LocaleType`, and choose **Add new** → **typecode**.

d. Enter the following values for the new typecode:

- `code` – `en_NZ`
- `name` – `New Zealand (English)`
- `desc` – `New Zealand (English)`

e. Navigate in the **Project** window to **configuration** → **config** → **Extensions** → **Typelist** and double-click `ZoneType.ttx`.

f. In the editor, right-click the top element on the left, **typelistextension** `ZoneType`, and choose **Add new** → **typecode**.

g. Enter the following values for the new typecode:

- `code` – `suburb`
- `name` – `Suburb`
- `desc` – `Suburb`

h. In the editor right-click the `suburb` typecode and choose **Add new** → **Add categories**.

i. Click **suburb** in the list of typecodes and then click **Next**.

j. In the **Typelist** list, click `Country`, and in the **Category** list, click `NZ`, and then click **Finish**.

4. Add a localization folder for New Zealand add required configuration files:

a. Navigate in the **Project** window to **configuration** → **config** → **Localizations**.

b. Right-click **Localizations** and choose **New** → **Package**. Name the new package `en_NZ`.

c. Copy the `localization.xml` file under the `en_US` folder to this new folder.

d. Double-click the new `localization.xml` file to open it in the editor.

e. Configure the `<GWLocale>` element to use `en_NZ` as the country and to use date configurations appropriate for New Zealand:

```
<GWLocale code="en_NZ" name="New Zealand (English)" typecode="en_NZ">
  <DateFormat long="E d MMM yyyy" medium="d MMM yyyy" short="dd/MM/yyyy"/>
  <TimeFormat long="h:mm:ss a z" medium="hh:mm:ss a" short="h:mm a"/>
  <NumberFormat decimalSymbol="." thousandsSymbol=","/>
  <CurrencyFormat negativePattern="($#)" positivePattern="$#" zeroValue="-"/>
</GWLocale>
```

5. Add a folder for New Zealand in `geodata`, and add configuration files for country, address, and zone:

a. Navigate in the **Project** window to **configuration** → **config** → **geodata**.

b. Right-click **geodata** and choose **New** → **Package**. Name the new package `NZ`.

c. Copy the three files under the `geodata/AU` folder to this new folder. The three files are `address-config.xml`, `country.xml`, and `zone-config.xml`.

d. Double-click the copied `country.xml` file to open it in the editor.

e. Add `Suburb` to the `visibleFields` definition and remove `State`:

```
visibleFields=
  "Country,AddressLine1,AddressLine2,AddressLine3,Suburb,City,PostalCode"
```

f. Double-click the copied `address-config.xml` file to open it in the editor.

g. Set the file up to use `NZ` as the country and the name, and to use `Suburb` instead of `State`:

```
<AddressConfig xmlns="http://guidewire.com/address-config">
  <AddressDef name="NZ">
    <Match>
      <Field name="Country" value="NZ"/>
    </Match>
    <Fields>
      <Field name="City" zonecode="city">
          <AutoFillFromZone code="suburb"/>
          <AutoFillFromZone code="postalcode"/>
      </Field>
      <Field name="Suburb" zonecode="suburb">
      <!-- Do not autofill from the city because a definitive match
           to suburb from the city isn't possible-->
          <AutoFillFromZone code="postalcode"/>
      </Field>
      <Field name="PostalCode" zonecode="postalcode">
          <!-- Autofill first from suburb, because most people use suburb and
               postcode is a new concept. -->
          <!-- Postcodes are not unique by city. For example, 0420 maps to the
               following cities: Cable Bay, Coopers Beach, Mangonui, Taipa -->
          <AutoFillFromZone code="suburb"/>
          <AutoFillFromZone code="city"/>
          <ValidatorDef value="[0-9]{4}"
            description="Validator.PostalCode.NZ" input-mask="####"/>
      </Field>
      </Fields>
  </AddressDef>
</AddressConfig>
```

h. Double-click the copied `zone-config.xml` file to open it in the editor.

i. Change the country code to `NZ`:

```
countryCode="NZ"
```

j. Add `Suburb` to the zone definitions, change its relationship to postal code and city, and remove `State`:

Note: This file defines the CSV file order for each zone data entry to be postal code, suburb, city.

```
<Zones countryCode="NZ">
<!-- The uniqueness of the postcode is based on the containing zone.
     The postcode's container is the country and the deciding factor is whether
     or not the postcode is unique within the country. and not the fact that the same
```

```
    postal code is shared by multiple cities. For example, 0420 maps to the following
    cities: Cable Bay, Coopers Beach, Mangonui, Taipa -->
  <Zone code="postalcode" fileColumn="1" granularity="1" regionMatchOrder="1">
    <AddressZoneValue>Address.PostalCode.substring(0,4)</AddressZoneValue>
    <Links>
      <Link toZone="suburb" lookupOrder="1"/>
      <Link toZone="city" lookupOrder="2"/>
    </Links>
  </Zone>
  <Zone code="suburb" fileColumn="2" granularity="2" regionMatchOrder="2">
  <!-- As described for the postcode, the containing zone for the suburb
       is the city and therefore the suburb is not unique per city.
       Some examples: HillCrest is a suburb of both Hamilton and Rotorua.
       Avondale is a suburb of Auckland and Christchurch -->
    <ZoneCode> city + ":" + suburb </ZoneCode>
    <AddressZoneValue>
      Address.City + ":" + Address.Suburb
    </AddressZoneValue>
    <Links>
      <Link toZone="postalcode" lookupOrder="1"/>
      <Link toZone="city" lookupOrder="2"/>
    </Links>
  </Zone>
  <Zone code="city" fileColumn="3" granularity="3" regionMatchOrder="3">
  <!-- City (and town) names are unique in the country. They might have similar
       names such as Palmerston North, which is in the North Island,
       and Palmerston, which is in the South Island, but there is
       always a distinction. -->
      <AddressZoneValue> Address.City </AddressZoneValue>
    <Links>
      <Link toZone="postalcode" lookupOrder="1"/>
      <Link toZone="suburb" lookupOrder="2"/>
    </Links>
  </Zone>
</Zones>
```

6. Add a field validator file for New Zealand and add the field validator for a New Zealand postal code to it:

   a. Navigate in the **Project** window to **configuration** → **config** → **fieldvalidators**.

   b. Right-click **fieldvalidators** and choose **New** → **Package**, and then enter `NZ` and click **OK**.

   c. Copy `fieldvalidators.xml` from **AU** to the new **NZ** folder.

   d. In the new file, change `Validator.PostalCode.FourDigit` to `Validator.PostalCode.NZ`. The new validator definition is:

```
<ValidatorDef
  description="Validator.PostalCode.NZ"
  input-mask="####" name="PostalCode"
  value="[0-9]{4}"/>
```

7. Configure the following typelists to set up typecodes for currency and jurisdiction:

   a. Navigate in the **Project** window to **configuration** → **config** → **Extensions** → **Typelist**.

   b. Double-click `Currency.ttx` to open this typelist in the editor.

   c. In the editor, right-click the top element on the left, **typelistextension** `Currency`, and choose **Add new** → **typecode**.

   d. Enter the following values for the new typecode:

   - `code` – `nzd`
   - `name` – `NZD`
   - `desc` – `New Zealand Dollar`

   e. Double-click `Country.ttx` to open this typelist in the editor.

   f. In the editor, right-click the `NZ` typecode and choose **Add new** → **Add categories**.

   g. Click **NZ** in the list of category typecodes and then click **Next**.

   h. In the **Typelist** list, click `Currency`, and in the **Category** list, click `nzd`, and then click **Finish**.

i. Double-click `Jursdiction.ttx` to open this typelist in the editor.

j. In the editor, right-click the top element on the left, **typelistextension** `Jursdiction`, and choose **Add new** → **typecode**.

k. Enter the following values for the new typecode:

- `code` – `NZ`
- `name` – `New Zealand`
- `desc` – `New Zealand`

l. In the editor, right-click the `NZ` typecode and choose **Add new** → **Add categories**.

m. Click **NZ** in the list of category typecodes and then click **Next**.

n. In the **Typelist** list, click `Country`, and in the **Category** list, click `NZ`, and then click **Finish**.

o. In the editor, right-click the `NZ` typecode again and choose **Add new** → **Add categories**.

p. Click **NZ** in the list of typecodes and then click **Next**.

q. In the **Typelist** list, click `JurisdictionType`, and in the **Category** list, click `insurance`, and then click **Finish**.

8. Configure the currency XML file for New Zealand currency:

a. Navigate in the **Project** window to **configuration** → **config** → **currencies**.

b. Right-click **currencies** and choose **New** → **Package**, and then enter `nzd` and click **OK**.

c. Copy the `currency.xml` file from the **aud** folder to the new **nzd** folder.

d. In the editor for `nzd/currrency.xml`, change the currency code from `"aud"` to `"nzd"` and the currency description from `"Australian Dollar"` to `"New Zealand Dollar"`. The resulting code is:

```
<Currency xmlns="http://guidewire.com/currency">
  <CurrencyType code="nzd" desc="New Zealand Dollar" storageScale="2">
    <CurrencyFormat zeroValue="-"/>
  </CurrencyType>
</Currency>
```

9. Edit supporting files for the user interface and add the New Zealand suburb field:

a. Press `Ctrl+Shift+N` and enter `Address.en`, and then double-click `Address.en` in the search results to open it in the editor.

b. In the top part of the editor, click + and then enter the following values:

- **Name** – `suburb`
- **Entity Path** – `Address.Suburb`

c. Below this top pane, there is a code pane with a `Default` tab. Click this tab and add a line of code defining a `Suburb` to the list of `formatter` address field definitions. If necessary, add a comma to the end of the preceding line:

```
:CEDEXBureau = CEDEXBureau,
:Suburb = suburb
```

d. In ContactManager only, this code pane also has a `Full` tab. Click this tab and make the same addition as you did in the `Default` tab.

e. Press `Ctrl+N` and enter `AddressOwnerFieldId`, and then double-click `AddressOwnerFieldId` in the search results to open this class in the editor.

f. Add the following line of code to the constant declarations for available fields:

```
public static final var SUBURB : AddressOwnerFieldId =
  new AddressOwnerFieldId("Suburb")
```

g. Add the `Suburb` field to the constant representing all PCF address fields:

```
public final static var ALL_PCF_FIELDS : Set<AddressOwnerFieldId> =
  { ADDRESSLINE1, ADDRESSLINE2, ADDRESSLINE3, SUBURB, CITY, COUNTY,
    STATE, POSTALCODE, COUNTRY, ADDRESSLINE1KANJI, ADDRESSLINE2KANJI,
    CITYKANJI, CEDEX, CEDEXBUREAU }.freeze()
```

h. Hide the `Suburb` field from the default search screens by changing the settings in `AddressOwnerFieldId` for `HIDDEN_FOR_SEARCH` and `HIDDEN_FOR_PROXIMITY_SEARCH`:

```
public final static var HIDDEN_FOR_SEARCH : Set<AddressOwnerFieldId> =
  { ADDRESSLINE1, ADDRESSLINE2, ADDRESSLINE3, SUBURB, COUNTY, ADDRESSLINE1KANJI,
    ADDRESSLINE2KANJI, CEDEX, CEDEXBUREAU }.freeze()

public final static var HIDDEN_FOR_PROXIMITY_SEARCH : Set<AddressOwnerFieldId> =
  { ADDRESSLINE1KANJI, ADDRESSLINE2KANJI, SUBURB, CITYKANJI,
    CEDEX, CEDEXBUREAU }.freeze()
```

Note: You can change this setting later if New Zealand is to be the default region.

i. Press `Ctrl+N` and enter `AddressFormatter`, and then double-click `AddressFormatter` in the search results to open this class in the editor.

j. Add the following case to `switch(_addrCountry)` to indicate the suburb, city, and postal code format for a New Zealand address:

```
case TC_NZ:
  append(cszBuf, fieldId.SUBURB, delimiter, addr.Suburb)
  append(cszBuf, fieldId.CITY,  " ", addr.City)
  append(cszBuf, fieldId.POSTALCODE, " ", addr.PostalCode)
  break
```

10. Add the New Zealand suburb field to the user interface:

a. Navigate in the **Project** window to **configuration** → **config** → **Page Configuration** → **pcf** → **address** and double-click `GlobalAddressInputSet.default` to open it in the editor.

b. Click `InputSet: GlobalAddressInputSet` at the top of the screen to open its `Properties` pane at the bottom.

c. Click the **Variables** tab, and then click + to add a new variable. Enter the following values:

- **initialValue** – `contact.AddressAutocompleteHandler.createHandler( "Suburb","Suburb,City,County,State,PostalCode,Country",true)`
- **name** – `suburbhandler`

d. In the screen, select the Address 3 `Input` widget.

e. Drag an `AddressAutofillInput` widget from the **Toolbox** to the Address 3 `Input` widget and drop it so it appears after this widget. Select the new widget.

f. In the **Properties** panel, set the following values:

- **editable** – `addressOwner.isEditable(fieldId.SUBURB)`
- **id** – `Suburb`
- **label** – `displaykey.Web.AddressBook.AddressInputSet.Suburb`

  Note: Click the red icon and choose **Create Display Key** to create this display key. Under `en_US`, enter `Suburb` and then click **OK**. If no red icon appears, navigate to **configuration** → **config** → **Localizations** → **en_US** and double-click `display.properties`. Then add the key `Web.AddressBook.AddressInputSet.Suburb = Suburb` to that file.
- **required** – `addressOwner.isRequired(fieldId.SUBURB)`
- **value** – `addressOwner.AddressDelegate.Suburb`
- **action** – `if (addressOwner.AutofillIconEnabled) gw.api.contact.AddressAutocompleteUtil.autofillAddress( addressOwner.AddressDelegate, "Suburb")`
- **actionavailable** – `addressOwner.AutofillIconEnabled`

- **autoComplete** – `suburbhandler`
- **autoCompleteArgIds** – `Suburb,City,County,State,PostalCode,Country`
- **available** – `addressOwner.isAvailable(fieldId.SUBURB)`
- **visible** – `addressOwner.isVisible(fieldId.SUBURB)`

11. Close and restart Studio to clear any errors you see reported in the classes that you changed.
12. **Note:** This step and the remaining steps apply only if you have ContactManager installed and integrated with PolicyCenter. For more information, see "Integrating ContactManager with Guidewire Core Applications" on page 49 in the *Contact Management Guide*.

    Modify the `ContactMapper` class in both PolicyCenter and ContactManager to pass the field to and from ContactManager:

    a. Navigate in the **Project** window to **configuration** → **gsrc** and then to `gw/contactmapper/ab800/ContactMapper`.

    b. Add a call to `fieldMapping` for `Address#Suburb` after the field mapping call for `Address#CEDEXBureau`:

    ```
    fieldMapping(Address#CEDEXBureau),
    fieldMapping(Address#Suburb),
    ```

13. Make the following web service change in ContactManager only:

    a. If necessary, open ContactManager Studio.

    b. Navigate in the **Project** window to **configuration** → **gsrc** and then to `gw/webservice/ab/ab801/abcontactapi/AddressInfo`.

    c. Add the following variable definition to the list of variables at the top of the class:

    ```
    public var Suburb : String
    ```

    d. Add the following line of code to the method `construct(address : Address)`:

    ```
    this.Suburb = address.Suburb
    ```

14. Restart ContactManager.
15. In PolicyCenter studio, reload the ContactManager web service, as follows:

    a. In the **Project** window, navigate to **configuration** → **gsrc** → **wsi** → **remote** → **ab** → **ab801** and double-click `ab801.wsc` to open it in the editor.

    b. In the Resources pane, click `${ab}/ws/gw/webservice/ab/ab801/abcontactapi/ABContactAPI?wsdl` to select it.

    c. At the bottom of the `Resources` pane, click `Fetch Updates`.

See also

- "Understanding Global Addresses" on page 132
- "Configuring Address Data and Field Order for a Country" on page 135
- "Address Modes in Page Configuration" on page 138
- "Address Owners" on page 139
- "Address Autocompletion and Autofill" on page 140
- "Configuring Zone Information" on page 122

Additional Steps for Configuring New Zealand Localization

To test autofill and autocomplete in the New Zealand configuration, you need to import New Zealand sample data supporting the zones defined in `zone-config.xml`. If you use the previous `zone-config.xml` configuration, the format of the data is postal code, suburb, city. For example:

```
0110,Abbey Caves,Whangarei
9018,Abbotsford,Dunedin
3330,Acacia Bay,Taupo
8011,Addington,Christchurch
```

- For information on creating zone information in a file you can import, see "Configuring Zone Information" on page 122.
- For information on importing the file you create, see "Zone Import Command" on page 169 in the *System Administration Guide*.

Additional configurations you can perform are:

- Adding New Zealand jurisdictions to `Jursidiction.ttx` to provide drop-down lists of jurisdictions in the user interface. The previous example adds only New Zealand itself as a country typecode to this typelist.
- Adding New Zealand regions in the PolicyCenter Administration tab. You can then associate these regions with user groups and use them to assign work.
  - See "Working with Regions" on page 660 in the *Application Guide*.
- Configuring New Zealand language support and regional formats. For example, after completing the previous configuration for the `Suburb` field, the only English language support is `en_US`. See:
  - "Working with Languages" on page 21
  - "Working with Regional Formats" on page 79

chapter 16

Configuring Phone Information

Phone numbers are specific to each country. Guidewire PolicyCenter uses country information to determine the appropriate fields to show for user entry of the phone number. PolicyCenter also uses country information to correctly format a phone number in read-only mode.

This topic includes:

Working with Phone Configuration Files

You can access phone-related configuration files in Studio by navigating in the **Project** window to **configuration** → **config** → **phone**.

Important files in this folder include the following:

File	Description
`nanpa.properties`	Area codes as defined by the North American Numbering Plan Administration (NANPA). These area codes apply to North American countries other than the United States.
`PhoneNumberMetaData.xml`	Area codes and validation rules for international phone numbers. See the comments at the beginning of the file for more information.
`PhoneNumberAlternateFormats.xml`	Additional area codes and validation rules for international phone numbers. See the comments at the beginning of the file for more information.

Any change to an XML file in the `phone` folder requires that you regenerate the phone data in the `data` subdirectory. To regenerate phone data, run the following gwpc utility from the application `bin` directory:

```
gwpc regen-phone-metadata
```

For details on how to use the gwpc commands, see “Build Tools” on page 119 in the *Installation Guide*.

Setting Phone Configuration Parameters

You use configuration parameters to set phone-related information in PolicyCenter:

Configuration parameter	Sets...
`DefaultPhoneCountryCode`	The default ISO country code to use for phone data. The country code must be a valid ISO country code that exists as a typecode in the `PhoneCountryCode` typelist. See the following web site for a list of valid ISO country codes: `http://www.iso.org/iso/english_country_names_and_code_elements` The base application default phone country code is `US`.
`DefaultNANPACountryCode`	The default country code for region 1 phone numbers. If mapping file `nanpa.properties` does not contain the area code for this region, then PolicyCenter defaults to the area code value configured by this parameter. The base application default NANPA phone country code is `US`.
`AlwaysShowPhoneWidgetRegionCode`	Whether the phone number widget in PolicyCenter always shows a selector for phone region codes. The base application value for this parameter is `false`.

See “Globalization Parameters” on page 57 in the *Configuration Guide* for more information on this configuration parameter.

Phone Number PCF Widget

PCF widget `GlobalPhoneInputSet` provides a way to show phone-related fields in PolicyCenter. The phone-related fields that you see in PolicyCenter depend on

- Country information that the user selects.
- Screen mode, editable or read-only.

If the screen is in edit mode, a PolicyCenter user has access to the following phone-related fields, set in the PCF widget `GlobalPhoneInputSet`:

- Country code
- National subscriber number
- Extension

If the screen is in read-only mode, a PolicyCenter user views previously entered phone-related information, the format of which depends on the phone country code.

You initialize the `GlobalPhoneInputSet` by providing the `InputSetRef` with a `PhoneOwner`. You initialize a `PhoneOwner` by providing a `PhoneFields` object.

Phone Numbers in Edit Mode

International phone numbers begin with a country code, according to the following format.

```
+phoneCountryCode phoneNumber extensionNumber
```

It is also possible to have an extension to the phone number as well.

For ease of entering phone information, it is possible to configure the `GlobalPhoneInputSet` widget to show a list of **Region Code** values in PolicyCenter from which the user can chose. For PolicyCenter to show the **Region Code** drop-down, the following must be true:

- Configuration parameter `AlwaysShowPhoneWidgetRegionCode` in `config.xml` must be set to `true`.
- The user must initially enter a plus sign in the phone number field.

If the user initially enters a plus sign (+) in the phone number field, the `GlobalPhoneInputSet` widget issues a post-on-change event to expose a **Region Code** drop-down list. Application logic maps the region code to a country code by using `CountryCodeToRegionCode.xml` and identifies the corresponding country. PolicyCenter formats the phone number based on the user's phone region and the country selected for the phone number. For example:

User phone region	Phone number country	PolicyCenter formats number for...
US	US	Domestic United States
US	CN	International Chinese
CN	CN	Domestic Chinese
CN	US	International United States

If a user enters a numeric phone number without first entering a country code, then PolicyCenter invokes only the format action on a targeted post-on-change event. Also, PolicyCenter invokes only the format action if the country code is the same as the user's selected default, or, if none, the default configured for the server.

See also

- "AlwaysShowPhoneWidgetRegionCode" on page 60 in the *Configuration Guide*

Phone Numbers in Read-only Mode

The read-only phone number field of the `GlobalPhoneInputSet` widget formats phone numbers based on one of the following:

- The default phone region selected by the user
- The default phone country code configured for the server

Users select a default phone region on the standard **Preferences** screen in PolicyCenter, available by clicking **Preferences** on the Options menu. A user's selection for default phone region overrides the default phone region set for the server.

Converting Phone Numbers from Previous Formats

PolicyCenter provides a plugin interface, `IPhoneNormalizerPlugin`, that you can use to customize the conversion of phone numbers from a format that does not match the format used in PolicyCenter. The plugin is called by the Phone Number Normalizer work queue. The plugin is also called whenever an entity containing a phone number is modified in PolicyCenter or restored from the archive.

PolicyCenter provides a default implementation of the plugin in the class `gw.api.phone.DefaultPhoneNormalizerPlugin`. If you disabled the phone number input mask or imported phone numbers, you might need to customize the plugin implementation. If you added new phone fields on existing objects or extension objects, customize or extend the plugin implementation to handle the additional phone fields. The `gw.api.util.PhoneUtil` class contains helper methods to facilitate formatting and parsing phone number records.

For full information on this plugin and how to modify it, see "Upgrading Phone Numbers" on page 192 in the *Upgrade Guide*.

chapter 17

Linguistic Search and Sort

This topic describes how to configure PolicyCenter to perform search and sort operations in languages other than the default `en_US`. PolicyCenter provides support for language-appropriate text search and sort for a single language.

> **Note:** This topic does not cover localization of free-text search. The base configuration of PolicyCenter provides free text search only in United States English. You must configure Guidewire Solr Extension to be able to perform free text searches in languages other than United States English. This configuration requires expertise in configuring Apache Solr. For information on basic configuration of free-text search, see "Free-text Search Configuration" on page 341 in the *Configuration Guide*.

This topic includes:

Linguistic Search and Sort of Character Data

There are two primary ways to search for and sort character data:

- Treat the character data as binary code points and compare and sort the data numerically.
- Treat the character data linguistically. This approach applies specific collation rules to order words in a list that reflect the commonly accepted practices and expectations for a particular language.

Linguistic search applies a specific collation to the character data. A *collation* is an overriding set of rules that applies to the ordering and comparison of the data. Collation *strength* refers to the elements of the collation process that the search and sort code applies to the data.

For example:

- Collation strength controls whether the search and sort code respects or ignores differences in case and accent on a character, such as the leading character on a word.
- In the Japanese language, collation strength also controls whether the search and sort code respects or ignores the differences between:
  - Katakana and Hiragana.
  - Full-width and half-width Katakana character differences.

PolicyCenter uses the value of configuration parameter `DefaultApplicationLocale` and the `language.xml` and `collations.xml` configuration files to implement localized search and sort functionality.

Note: PolicyCenter displays the default locale value when the current locale value is missing. However, PolicyCenter always sorts by the current locale value. As a result, sorting of a localized column of a list view can appear to be broken when the column has one or more untranslated values.

Effect of Character Data Storage Type on Searching and Sorting

Guidewire PolicyCenter stores character data in the following ways:

- Database storage
- In-memory storage

PolicyCenter handles searching and sorting of character data differently for these storage types.

Character Data in the Database

PolicyCenter writes most application data directly to the database. This action stores the data on a physical disk storage system. Each discrete piece of data is an entry in a table column, with each table being organized by rows. During a comparison and sort of data in the database, the database management system (DBMS) performs the operations and applies rules that control these operations.

Character Data in Memory

PolicyCenter writes some application data to volatile memory devices, such as the local machine RAM memory. PolicyCenter typically uses this kind of memory storage for the display of certain kinds of data in the user interface. For example, PolicyCenter uses in-memory storage for drop-down lists and the results of list views that do not use database queries. During a comparison and sort of data in memory, programming code provided in the base configuration controls the operations.

Searching and Sorting in Configured Languages

Your PolicyCenter installation provides support for searching and sorting for a single language. PolicyCenter reads the localization code from the `DefaultApplicationLocale` configuration parameter, which you set in `config.xml`. The default is United States English, `en_US`.

You set the value of `DefaultApplicationLocale` once, before you start the application server for the first time. PolicyCenter stores this value in the database and checks the value at server startup. If the application server value does not match a database value, then PolicyCenter throws an error and refuses to start.

IMPORTANT You must set the value of configuration parameter `DefaultApplicationLocale` before you start the application server for the first time. You cannot change this value after you start the application server without dropping the database.

Guidewire also provides support for language-appropriate searching and sorting for display keys for each supported localization code. You define and manage language characteristics in `language.xml` files. You define these files for each localization folder, such as `en_US`. You can access the localization folders in Studio by navigating in the **Project** window to **configuration** → **config** → **Localizations**.

Each `language.xml` file contains a `<GWLanguage>` element. This element supports the following subelements that you can use to configure the behavior of searching and sorting operations in the Guidewire application:

- `<SortCollation>`

  Element `<SortCollation>` has a `strength` attribute that you use to define the sorting collation strength for this language. The exact meaning of the `SortCollation` `strength` attribute value depends on the specific language. For more information on this attribute, see "Configuring Database Sort in language.xml" on page 163.
- `<LinguisticSearchCollation>`

  Element `<LinguisticSearchCollation>` supports a `strength` attribute that you use to define the searching collation strength for a language. For more information on this attribute, see "Configuring Oracle Search in language.xml" on page 160.

See also

- "Configuring Search in the PolicyCenter Database" on page 159
- "Configuring Sort in the PolicyCenter Database" on page 162
- "Setting Linguistic Search Collation" on page 59 in the *Upgrade Guide*

Configuring Search in the PolicyCenter Database

For a column to be eligible for inclusion in the database search algorithm, the `supportsLinguisticSearch` attribute on that column must be set to `true`. Setting this column attribute to `true` marks that column as searchable, regardless of which DBMS you use.

See "<column>" on page 180 in the *Configuration Guide* for more information on the `<column>` element and the attributes that you can set on it.

IMPORTANT You cannot use the `supportsLinguisticSearch` attribute with an encrypted column. If you attempt to do so, the application server refuses to start.

How Guidewire PolicyCenter handles searching of data depends on the database involved. See the following:

- Searching and the Oracle Database
- Searching and the SQL Server Database

Searching and the Oracle Database

To implement linguistic searching in Oracle, the database compares binary values that PolicyCenter modifies for searching. For functional and performance reasons, PolicyCenter does not use Oracle collations.

- For primary, accent-insensitive searching, Guidewire uses the configurable Java class described in "Configuring Oracle Search in collations.xml" on page 160 to compute the comparison values. Guidewire also uses this Java class to define the search semantics for searching in the Japanese and German languages.
- For secondary, case-insensitive searching, Guidewire transforms the search values to lower case.

Configuring Oracle Search in language.xml

Guidewire provides the ability to configure language-appropriate linguistic search capabilities through the `<LinguisticSearchCollation>` element. This subelement of `<GWLanguage>` is defined in `language.xml` in the appropriate localization folder. You use the `strength` attribute of this subelement to configure and control specialized search behavior.

IMPORTANT Any change to the `<LinguisticSearchCollation>` element in `language.xml` requires a database upgrade. If you make a change to this element, then you must restart the application server to force a database upgrade.

The meaning of the `strength` attribute depends on the specific language. In general, the settings mean the following:

- A `strength` of `primary` considers only character weights. This setting instructs the search algorithms to consider just the base, or primary letter, and to ignore other attributes such as case or accents. Thus, the collation rules consider the characters é and E to have the same weight. For more information on this attribute, see "Configuring Database Sort in language.xml" on page 163.
- A `strength` of `secondary`, the default, considers character weight and accent differences, but, not case differences. Thus, the collation rules consider the characters e and é to be different and thus the rules treat them differently. The collation rules do not, however, treat e and E differently.

To summarize, the `strength` attribute can take the following values, with the default being `secondary`.

Strength	Search description
`primary`	• accent-insensitive • case-insensitive
`secondary`	• accent-sensitive • case-insensitive

Note: Localized search supports only two levels for the `strength` value, in contrast to localized sorting, which supports three levels for the `strength` value.

The following `language.xml` file is an example of this file in the localization folder **ja**, with suggested settings.

```
<?xml version="1.0" encoding="UTF-8"?>
<Language xmlns="http://guidewire.com/language">
  <GWLanguage code="ja" name="Japanese" typecode="ja">
    <LinguisticSearchCollation strength="primary"/>
    <SortCollation strength="primary"/>
  </GWLanguage>
</Localization>
```

Configuring Oracle Search in collations.xml

For the Oracle database, Guidewire provides specialized search rules through the use of a Java class that you can configure. PolicyCenter exposes this Java class as a `CDATA` element in the `<Database>` element for Oracle in file `collations.xml`. You access `collations.xml` in the Studio **Project** window by navigating to **configuration** → **config** → **Localizations**.

In this file, search for the following:

```
<Database type="ORACLE">
  ...
  <DBJavaClass> <![CDATA[...]]></DBJavaClass>
  ...
</Database>
```

Guidewire PolicyCenter uses this Java code as the source code for a Java class. In the base configuration, the provided Java class defines:

- General rules for primary-strength searching in the database

- Specialized rules for searching in the Japanese language
- Specialized rules for searching in the German language

As defined in the comments in `collations.xml`, it is possible to modify the embedded `GWNormalize` Java class directly to meet your business needs. It is useful to modify the `GWNormalize` class under the following circumstances:

- You are using an Oracle database and either Japanese or German language strings
- You are using an Oracle database and primary, accent-insensitive search collation

General Search Rules

In the base configuration, PolicyCenter uses the following general rules as it performs a database search on a column that is configured to support linguistic searching:

- All searches are case insensitive, regardless of the value of the `strength` attribute on `<LinguisticSearchCollation>`. PolicyCenter regards the characters `e` and `E` as the same.
- All searches take punctuation into account. PolicyCenter regards O'Reilly and OReilly as different.
- All searches in which the `strength` attribute on `<LinguisticSearchCollation>` is set to `primary` ignore accent marks. PolicyCenter regards the characters `e` and `è` as the same in this type of search.
- All searches in which the `strength` attribute on `<LinguisticSearchCollation>` is set to `secondary` take into account any accent marks. PolicyCenter regards the characters `e` and `è` as different in this type of search.

PolicyCenter searches only database columns for which you set the `supportsLinguisticSearch` attribute to `true`.

General Search Rules for the Japanese Language

In the base configuration, Guidewire provides specialized search algorithms specifically for the Japanese language. Guidewire sets these rules in `collations.xml`, as described at the beginning of this topic. This Java class provides the following behavior for searching in a Japanese-language database:

Search case	Rule
Half-width/Full-width	All searches in Japanese ignore the difference between half-width and full-width Japanese characters.
Small/Large characters	All searches in Japanese in which the `strength` attribute on `<LinguisticSearchCollation>` is set to `primary`, meaning accent-insensitive, ignore Japanese small/large letter differences in Katakana or Hiragana. Searches in which this attribute is set to `secondary` take small/large letter differences into account.
Katakana and Hiragana	All searches in Japanese ignore the difference between Katakana and Hiragana characters. This type of search is known as *kana-insensitive* searching.
Long dash (—)	All searches in Japanese ignore the long dash character.
Sound marks (`` and °)	All searches in Japanese in which the `strength` attribute on `<LinguisticSearchCollation>` is set to `primary` ignore sound marks. Searches in which this attribute is set to `secondary` take sound marks into account.

If you modify the contents of `collations.xml` or the embedded Java class, PolicyCenter forces a database upgrade the next time the application server starts.

General Search Rules for the German Language

In the base configuration on Oracle, Guidewire provides specialized search algorithms specifically for the German language. Guidewire sets these rules through the use of a configurable Java class that it exposes as a `CDATA` element in the `collations.xml`. This is the same Java class that the discussion on rules for the Japanese language covered.

This Java class provides the following behavior for searching in a German-language database:

Search case	Rule
Vowels with umlauts	All searches in German compare as equal a vowel with an umlaut or the same vowel without the umlaut but followed by the letter e. Thus, all searches in German explicitly treat the following as the same value: • ä and ae • ö and oe • ü and ue
German letter Eszett	All searches in German treat the Eszett character ß (also known as Sharp-S) the same as the characters ss.

Searching and the SQL Server Database

In SQL Server, the collations provided by the Windows operating system are effective in providing language-appropriate searching. To work correctly, Guidewire requires that you create a SQL Server database with case-insensitive collation. Guidewire uses this collation for all character data sorting and searching by default, as well as to provide case-insensitive table and column names.

Through the linguistic search configuration, it is possible to specify a different collation for searching on columns that support linguistic searching:

- If simple, case-insensitive searching meets your requirements, then configure file `collations.xml` to select the same collation as the database collation.
- If you need different search semantics, then configure the SQL Server entry in `collations.xml` for a `primary` strength search collation, which will give you accent-insensitive searching.

The semantics of linguistic searching for SQL Server are those of the Windows collation selected from the `collations.xml` file. The collation is based on the default language and linguistic search collation strength from `language.xml`, in which `secondary` strength is the default. Microsoft controls the Windows collation rules, not Guidewire.

With reference to the discussion about Japanese and German search rules on Oracle, the Windows collations configured in the base configuration in `collations.xml` provide the following:

- Kana-insensitivity and width-insensitivity for Japanese collations
- Umlaut and Eszett handling in the German collations

If you are currently using SQL Server in those languages, your IT staff is mostly likely familiar with these issues.

Configuring Sort in the PolicyCenter Database

PolicyCenter handles the ordering of data as consistently as possible between database sorting and in-memory sorting. PolicyCenter derives the collation to use for sorting from the following:

- The default localization code set in the configuration parameter `DefaultApplicationLocale` in `config.xml`.
- The collation strength setting. This value is set in the localization folder's `language.xml` file.

PolicyCenter uses these values along with the database type to look up the collation in `collations.xml`.

Note: PolicyCenter uses in-memory sorting in the application interface for various elements, such as drop-down lists and list views that do not result from queries. To perform in-memory sorting, PolicyCenter uses a language-specific `Collator` object that is modified with the collation strength setting for that language.

This topic includes:

- "Configuring Database Sort in language.xml" on page 163

- "Configuring Database Sort in collations.xml" on page 163

Configuring Database Sort in language.xml

For optional use, the `<SortCollation>` subelement of the `<GWLanguage>` element in `language.xml` controls specialized sorting behavior. Each localization folder has a separate `language.xml` file. To access the localization folders, navigate in the Guidewire Studio **Projects** window to **configuration** → **config** → **Localizations** → *`localizationFolder`*.

The `<SortCollation>` element has a single `strength` attribute that determines *collation strength*—how PolicyCenter sorting algorithms handle accents and case during the sorting of character data for the following:

- Sorting of in-memory data
- Sorting of data in the database

The `strength` attribute, which defaults to `secondary`, can take the following values:

- `primary`
- `secondary`
- `tertiary`

The specific meaning of the `strength` attribute depends on the language. In general:

- A `strength` of `primary` instructs the search and sort algorithms to consider just the base, or primary letter, and to ignore other attributes, such as case or accents. With this setting, the collation rules consider the characters `e` and `E` to have the same weight.
- A `strength` of `secondary` instructs the search and sort algorithms to consider character weight and accent differences. This value is the default setting. With this setting, the collation rules consider the characters `e` and `è` to be different and order them differently.
- A `strength` of `tertiary` instructs the search and sort algorithms to consider character weight, accent differences, and case. With this setting, the collation rules consider the characters `e` and `è` and `E` to be different and order them differently.

The following list describes these differences.

Strength	Case-sensitive	Accent-sensitive
`primary`	No	No
`secondary`	No	Yes
`tertiary`	Yes	Yes

Configuring Database Sort in collations.xml

Guidewire uses `collations.xml` as a lookup file. To access `collations.xml`, navigate in the Guidewire Studio **Projects** window to **configuration** → **config** → **Localizations**. PolicyCenter uses the following definitions in this file to look up the sort collation name and apply it:

- The application localization code
- The `strength` attribute value from the `<SortCollation>` element in `language.xml`
- The database management system (DBMS) type

PolicyCenter primarily uses these values to look up the sort collation. For example, suppose that the following are all true:

- The database is Oracle.
- The user language is German.
- The `strength` value of `SortCollation` in `language.xml` is set to `secondary`.

PolicyCenter then looks at the following for instructions on how to set `NLS_SORT` for Oracle sessions and sets it to `GERMAN_CI`.

```
<Database type="ORACLE">
  ...
  <Collation locale="de" primary="GERMAN_AI" secondary="GERMAN_CI" tertiary="GERMAN"/>
...
```

Determining the Order of Typekeys

PolicyCenter uses the language collation rules defined in `language.xml` as part of determining the ordering of typekeys from the database. To sort typekeys, PolicyCenter applies the following criteria:

1. If there is no `.sort` file defined for the typelist in the localization folder, then PolicyCenter:
   a. Uses the priority associated with each typekey in its typelist to order the typekeys by priority order.
   b. For typekeys with the same priority, applies the language collation rules to the typekey display names.
2. If there is a `.sort` file defined for the typelist in the localization folder, PolicyCenter:
   a. Uses the order of the typekeys specified in that file.
   b. For typekeys from the typelist that are not defined in the `.sort` file, PolicyCenter orders them after the defined typekeys, applying the language collation rules to these typekey display names.

Note: You typically need the `.sort` file only if you are supporting Japanese with other languages on the same server. Otherwise, the preferred technique is to specify the sort order by defining priority in the typelist and language collation in `language.xml`.

PolicyCenter applies the collation rules to the typekey columns in database query `ORDER BY` clauses that sort database query results. File `collations.xml` contains multiple language collations because PolicyCenter supports storing typekey values in multiple languages in one database, enabling PolicyCenter to sort the typekey names correctly for each language. This storage scheme enables users with different language settings to see different translations of a typekey.

See also

- For information on `.sort` files, see “Setting Localized Sort Orders for Localized Typecodes” on page 49.
- For information on setting typecode priority, see “Entering Typecodes” on page 271 in the *Configuration Guide*.

chapter 18

Configuring National Field Validation

Field validation in PolicyCenter generally relies on regular expressions and input masks to validate data that users enter in specific fields. Field validators define specific regular expressions and input masks. Sometimes, field validation varies by country. For example, many countries issue taxpayer IDs, but the validation rules for taxpayer IDs vary by country.

This topic includes:

See also

- "Field Validation" on page 259 in the *Configuration Guide*

Understanding National Field Validation

Field validators provide basic validation for data that users enter in specific fields.

- Field validators apply only to the value in a single field.
- Field validators do not enforce the uniqueness of values in that field.
- Field validators generally ignore relationships between values in that field and values in other fields.

A field validator typically defines a *regular expression*, which is a pattern of characters and special symbols that a value entered in a text field must match to be valid. Optionally, field validators can define an *input mask*, which provides a visual indication to the user of the expected format for values to enter in the field.

Note: You cannot define an input mask for input of Japanese characters—katakana and hiragana.

You can configure national field validation for fields of data type `LocalizedString` only. In addition, any entity definition that contains localized string fields must have an additional field to store a country code associated with each entity instance. PolicyCenter applies national field validation based on the value of the country code associated with specific entity instances.

You configure field validation by editing `fieldvalidtors.xml` files in various locations in the `fieldvalidators` folder.

- You define global field validators once in the `fieldvalidtors.xml` file located in the root of the `fieldvalidators` folder.
- You define national field validators in `fieldvalidtors.xml` files located in country-specific packages in the `fieldvalidators` folder.

See "Localizing Field Validators for National Field Validation" on page 166.

See also

- "Data Types" on page 233 in the *Configuration Guide*

Localizing Field Validators for National Field Validation

You define national field validators in `fieldvalidtors.xml` files located in country-specific folders in the **field-validators** folder. Country-specific folder names must match typecodes from the `Country` typelist.

To define national field validators for a specific country

1. In Guidewire Studio, navigate in the **Project** window to **configuration** → **config** → **fieldvalidators**.
2. Right-click **fieldvalidators**, and then select **New** → **package** from the context menu.
3. Enter the typecode from the `Country` typelist for the country, and then click **OK**.
4. Copy the `fieldvalidators.xml` file from the root of the **fieldvalidators** package to the new country-specific package.
5. Modify the copy of `fieldvalidators.xml` that you just made to define national field validators for the country.

See also

- "Understanding National Field Validation" on page 165

Gosu Field Validation

A more advanced kind of field validator defines a Gosu class that handles field validation programmatically. You can develop Gosu classes that act as field validators. PolicyCenter provides a class framework for developing Gosu-based field validators, located in the `gw.api.validation` package. Gosu-based field validators must extend the abstract `FieldValidatorBase` class.

One such class is the `PhoneValidator` class, described in the topic that follows.

Enabling National Field Validation for Phone Data

PolicyCenter uses the Gosu class `gw.api.validation.PhoneValidator` as the default mechanism to validate phone number correctness.

To enable the new `PhoneValidator` validation functionality, you configure `fieldvalidators.xml` with the fully qualified name of the Gosu class. For example:

```
<ValidatorDef description="Validator.Phone"
              name="LocalizedPhoneValidator"
              validation-type="gosu"
              value="gw.api.validation.PhoneValidator"/>
```

As shown in the previous example, you must set the validation type to `gosu`.

The validator does not trigger validation unless the associated phone country is set. This trigger functionality provides backwards compatibility with old data.

See also

- “Configuring Phone Information” on page 153