

Internationalizing and Localizing Rhythmyx

6.7

Copyright and Licensing Statement

All intellectual property rights in the SOFTWARE and associated user documentation, implementation documentation, and reference documentation are owned by Percussion Software or its suppliers and are protected by United States and Canadian copyright laws, other applicable copyright laws, and international treaty provisions. Percussion Software retains all rights, title, and interest not expressly granted. You may either (a) make one (1) copy of the SOFTWARE solely for backup or archival purposes or (b) transfer the SOFTWARE to a single hard disk provided you keep the original solely for backup or archival purposes. You must reproduce and include the copyright notice on any copy made. You may not copy the user documentation accompanying the SOFTWARE.

The information in Rhythmyx documentation is subject to change without notice and does not represent a commitment on the part of Percussion Software, Inc. This document describes proprietary trade secrets of Percussion Software, Inc. Licensees of this document must acknowledge the proprietary claims of Percussion Software, Inc., in advance of receiving this document or any software to which it refers, and must agree to hold the trade secrets in confidence for the sole use of Percussion Software, Inc.

Copyright © 1999-2009 Percussion Software.
All rights reserved

Licenses and Source Code

Rhythmyx uses Mozilla's JavaScript C API. See <http://www.mozilla.org/source.html> (<http://www.mozilla.org/source.html>) for the source code. In addition, see the *Mozilla Public License* (<http://www.mozilla.org/source.html>).

Netscape Public License

Apache Software License

IBM Public License

Lesser GNU Public License

Other Copyrights

The Rhythmyx installation application was developed using InstallShield, which is a licensed and copyrighted by InstallShield Software Corporation.

The Sprinta JDBC driver is licensed and copyrighted by I-NET Software Corporation.

The Sentry Spellingchecker Engine Software Development Kit is licensed and copyrighted by Wintertree Software.

The Java™ 2 Runtime Environment is licensed and copyrighted by Sun Microsystems, Inc.

The Oracle JDBC driver is licensed and copyrighted by Oracle Corporation.

The Sybase JDBC driver is licensed and copyrighted by Sybase, Inc.

The AS/400 driver is licensed and copyrighted by International Business Machines Corporation.

The Ephox EditLive! for Java DHTML editor is licensed and copyrighted by Ephox, Inc.

This product includes software developed by CDS Networks, Inc.

The software contains proprietary information of Percussion Software; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

Due to continued product development this information may change without notice. The information and intellectual property contained herein is confidential between Percussion Software and the client and remains the exclusive property of Percussion Software. If you find any problems in the documentation, please report them to us in writing. Percussion Software does not warrant that this document is error-free.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Percussion Software.

AuthorIT™ is a trademark of Optical Systems Corporation Ltd.

Microsoft Word, Microsoft Office, Windows®, Window 95™, Window 98™, Windows NT® and MS-DOS™ are trademarks of the Microsoft Corporation.

This document was created using *AuthorIT™, Total Document Creation* (see AuthorIT Home - <http://www.author-it.com>).

Percussion Software

600 Unicorn Park Drive

Woburn, MA 01801 U.S.A.

781.438.9900

Internet E-Mail: technical_support@percussion.com

Website: <http://www.percussion.com>

Contents

Introduction to Internationalization and Localization in Rhythmyx	7
---	---

Determining System Requirements	9
---------------------------------	---

Modeling Internationalization of a Rhythmyx Content Management System	11
---	----

Modeling Assembly for Internationalization	12
Modeling Internationalized Workflows	13
Defining the Content Relationship Model for Translations.....	13
Modeling Internationalized Workflow Processes	13
Spinning Off Translation Versions of Content Items	14
Modeling Publishing.....	15

Implementing Internationalization	17
-----------------------------------	----

Internationalization Support on the Rhythmyx Server.....	18
Localizing the EditLive Text Editor	21
Localizing the EditLive Text Editor	22
Implementing Internationalized Assembly	23
Implementing Internationalization of Templates.....	23
Implementing Internationalized Numbers and Date Formats	25
Changing Output Character Sets	26
Creating Automatic Links to Other Localized Versions of Content.....	26
Localizing the Business User Interface.....	27
Maintaining Locales	28
Preparing XSL Files for Localization.....	33
Localizing JavaScript Files.....	35
Localizing JSPs	36
Using Locale-specific Images	37
Rhythmyx Language Tool.....	37
Defining Translation Properties	41
Translation Setting Editor.....	42
Creating Translation Settings	43
Modifying Translation Settings	43
Removing a Translation Setting	44
Implementing Automated Translation	45
Example Auto Translation Implementation.....	46
Configuring the sys_createTranslations Workflow Action	50
Implementing Internationalized Publishing	51
Implementing Language-centric Publishing Models.....	51
Implementing Content-centric Publishing Models.....	51

Full-text Search in Globalized Environments52

Index **53**

CHAPTER 1

Introduction to Internationalization and Localization in Rhythmyx

Internationalizing and Localizing Rhythmyx explains how to model a Rhythmyx Content Management System (CMS) for internationalization and then how to implement your model. The document assumes that you are developing a new system for internationalization. It also assumes that you are generally familiar with the process of modeling and implementing a non-internationalized CMS, either through training on Rhythmyx or through reading the *Rhythmyx Implementation Guide*. If you are not familiar with that document, you should read it before proceeding with this document.

Internationalization is the process of implementing a Rhythmyx CMS to support content in multiple languages. The Rhythmyx Content Model provides a firm foundation for internationalization because it provides you with control over each item. Rhythmyx has functionality that allows you to determine which items to translate, and the languages into which to translate them. You can implement your system to create translation Versions of items automatically or allow your users to determine whether an item should be translated.

Rhythmyx uses the Java definition of Locale, which identifies a combination of language and geographical, political, and cultural region. For more details about internationalization in Java, see <http://java.sun.com/j2se/1.5.0/docs/guide/intl/overview.html> (<http://java.sun.com/j2se/1.5.0/docs/guide/intl/overview.html>).

CHAPTER 2

Determining System Requirements

When internationalizing Rhythmyx, the only special requirement you need to consider is the character set of your database. The default character set for Rhythmyx is UTF-8, which is generally adequate for most European languages. If you will store Asian languages that require multiple-bit characters, such as Korean or Japanese, you should install your database with support for UTF-16. It is best to install the database with support for the correct character set from the beginning. Converting a database to a different character set is possible, but very complicated.

CHAPTER 3

Modeling Internationalization of a Rhythmyx Content Management System

Internationalization of a Rhythmyx content management system involves three of the four engines in Rhythmyx, Assembly, Workflow, and Publishing. You do not need to modify your content model to accommodate internationalization.

Implementing internationalization is much easier if you model your internationalized CMS. You will then save time implementing the model.

Modeling Assembly for Internationalization

When modeling internationalization of Content Assembly, Templates fall into three groups:

- Templates that need no modification;
- Templates you can internationalize by customizing the formatting to add localization processing;
- Templates that must be unique for each Locale.

A number of assemblers will not need modification. Generally, Templates that produce small, simple Snippets, such as title links used in navigation or abstracts used on index pages, do not require localization. These Snippets consist of a few lines of text from managed content, with no static text or images, and thus require no special modification for localization.

Templates require internationalization if they include static text or if the formatting differs significantly for one or more Locales. A Template that includes images may also require internationalization, depending on the content of the images. If an image includes text or culture-specific content, you probably need to internationalize the Template that uses that image.

You can offload internationalization of static text in Template to the server by adding translation keys to the Template markup.

Another approach for static text is to use Dispatch Templates, which are also necessary for internationalized images. The Dispatch Template evaluates the Locale of the content and processes the Template accordingly, displaying the correct text or retrieving the correct image.

If the Templates differ significantly for different Locales, such as using different table layouts, you probably need to develop Locale-specific Templates. If you have a significant amount of conditional processing, it may also be easier to manage unique Templates for each Locale rather than using excessive Dispatch Templates.

You can combine both approaches to internationalizing Templates. You may have sets of Locales that can use common images, for example, but that also use unique static text for specific Locales within the Group. For example, you may have several Spanish Locales that use the same set of images, but that use different static text. In that case, you might develop a set of Dispatch Templates for your Spanish Locales to display the appropriate text.

Modeling Internationalized Workflows

Internationalizing Workflows involves three steps:

- Defining the Relationship dependency model between original and translation content items;
- Defining the model for the Workflow processes
- Deciding when to spin off translation Versions of content items.

Defining the Content Relationship Model for Translations

When a user creates a translation Version of a Content Item, the translation Version has a Relationship association with the original item. In many cases, the two items proceed through their respective Workflow processes, become Public, and are published independent of one another. This association is referred to as a *weak dependency*.

In other cases, items from multiple Locales must become Public together. For example, in some jurisdictions, content must be available in multiple languages; it would be illegal to publish a content item in one language if the associated translation was not complete. Both items must be ready to enter a Public State before either can make the Transition. This association is referred to as a *strong dependency*.

Your circumstances determine the model you choose. As indicated above, in some cases, regulations dictate a strong dependency model. Business requirements may also dictate a strong dependency model if your priority is to keep the content on all sites synchronized. A strong dependency model may entail some delay in publishing the content, however, while translation is completed.

Much of the time you have more flexibility about keeping content synchronized. If it is not important that content for all Locales stay synchronized, a weak dependency model may be more appropriate. A weak dependency model may be more appropriate if timely publication of the original content is more important than synchronization of content for all Locales, such as delivering breaking news.

Relationship dependencies can be defined by content type, so you do not have to use a pure model. For example, a system using strong dependencies may include “quick edit” States so you can fix minor mistakes, such as misspellings. A misspelling in an original item is unlikely to affect its translation, so there’s no reason that the translation item should change States while the original is edited.

Modeling Internationalized Workflow Processes

To internationalize Workflow processes, decide whether you need any Locale-specific Workflows, or whether a single translation Workflow will suit your needs. In most cases, a single translation Workflow should be sufficient. You may need Locale-specific Workflows if a Locale has specific translation requirements that differ from other Locales.

Workflow Processes in Strong Dependency Models

In systems that use a strong dependency model, you must pay special attention to your Workflow Processes. A strong dependency model may require that all content enter a Public State together. If you have not designed your Workflow process and your content Relationships correctly, content can become trapped in the Workflow, unable to progress to a Public State.

Best Practice when designing Workflow processes for a strong dependency model is to create a “pending” State prior to the Public State in each Workflow. In this State, all work is effectively done on the Content Item and it is ready to Transition to Public, but must wait for dependent items to reach the same State before they can all make the Transition.

Spinning Off Translation Versions of Content Items

In a strong dependency model, you will need to decide at what phase in the Workflow to spin off the translation Version of the Content Item. Usually, it is best to generate the translation Versions automatically using the `sys_CreateTranslations` Workflow Action. This Action creates a Translation Content Item of the original Content Item in each Locale in which the original Content Item does not already have a corresponding Translation Content Item.

Best Practice is to spin off the translation Versions as late in the Workflow process as possible, ideally when the original item enters the “pending” State. At this point, all revision of the original item is complete, so the translator uses the final version of the original. If you spin off the translation Versions too early, the text of the original may still be changing and the translation may not account for all edits to the original item.

Modeling Publishing

Two delivery models are available for publishing content in Rhythmyx.

- A “language-centric” approach involves creating a file system for each language or modifying your output database with additional columns or tables to store localized content.
- A “content-centric” approach involves creating a single file system or database schema with unique pages or rows for each language.

Both approaches are feasible using Rhythmyx.

If you deliver to remote sites, such as mirrors in the Locale’s country, you should use a language-centric approach rather than a content centric approach. A language-centric approach is also more appropriate if much of the content of your sites is Locale-specific. For example, if you offer different products in different Locales, you probably should use a language-centric approach.

A language-centric approach is also more appropriate if you re-translate content frequently, or if your translators are in-house or otherwise easily accessible. A language-centric approach can be more expensive, however, since it may require multiple servers and databases.

A language-centric delivery model makes it easier to manage all of the content for a single Locale because all of the Locale-specific content resides together. You can also define different Publishing rules for different Locales and administer each Locale separately. Using this model, however, it is difficult to tell whether the content is in sync across Locales (in other words, whether all of the content on the `en-us` site is also on the `fr-ca` site). Sites may be out of sync if you use scheduled Publishing and the schedules for the sites are not synchronized.

If you deliver content from a single server, a content-centric approach may be useful particularly if your content does not differ significantly across Locales. If you translate less frequently (for example, if you outsource translations or if your Localized content is relatively stable), you might prefer a content-centric approach. A content-centric approach to content delivery can also be less expensive because it requires fewer delivery servers and databases.

Using a content-centric approach, it is easier to monitor whether specific content is published across Locales because content for all Locales is together. However, you cannot define unique publishing rules for different Locales and all Locales are administered together.

If you choose a language-centric approach, you will only need to design the additional file trees or database schemas for your localized sites. You will need to define contexts for each site, as well as global variables and scheme generators to deliver content into each site.

If you choose a content-centric delivery model into a file system, you will need to decide on a naming convention for localized content. A common convention incorporates the Locale identifier into the page name:

```
index_en-us.htm
index_fr-ca.htm
index_de-de.htm
home_en_us.htm
home_fr-ca.htm
home_de-de.htm
```


CHAPTER 4

Implementing Internationalization

This section explains how to implement your internationalization model you developed for Content Assembly, Workflow, and Publishing. Since you do not need to develop a content model for internationalization, no additional implementation is required to internationalize content.

This section also discusses support for internationalization on the Rhythmyx server.

Internationalization Support on the Rhythmyx Server

Several default Rhythmyx CMS tables have been internationalized. You can use these tables in both Content Editors and Content Assemblers. Internationalized tables include:

Table	Keys
COMPONENTS	psx.component.<NAME>@<DISPLAYNAME
CONTENTTYPES	psx.contenttype.<CONTENTTYPEID>@<CONTENTTYPENAME>
CONTENTVARIANTS	psx.variant.<VARIANTID>@<VARIANTDESCRIPTION>
ROLES	psx.role@<ROLENAME>
RXCOMMUNITY	psx.community@<NAME>
RXLOCALE	psx.locale@<LANGUAGESTRING>
RXLOOKUP	psx.keyword.<LOOKUPID>@<LOOKUPDISPLAY>
RXMENUACTION	psx.menuitem.<NAME>@<DISPLAYNAME>
RXSITES	psx.site.name.<SITEID>@<SITENAME> psx.site.description.<SITEID>@<SITEDESC>
RXSLOTTYPE	psx.slot@<SLOTNAME>
STATES	psx.workflow.state@<STATENAME>
TRANSITIONS	psx.workflow.transition.<WORKFLOWAPPID>.<TRANSITIONID>@<TRANSITIONLABEL>
WORKFLOWAPPS	psx.workflow.workflow.workflow.<WORKFLOWAPPID>@WORKFLOWAPPNAME

Translation keys consist of two parts, a UI identifier and a label identifier. The UI identifier specifies what part of the user interface uses the data. It consists of the following elements

- Prefix (always the Percussion identifier, psx)
- UI Mode (identifies which general portion of the interface uses the data, for example, workflow, component, or slot)
- UI Context (identifies in detail the portion of the mode that uses the data; this element is optional)

For example, the UI identifier for Workflow TRANSITIONS table is psx.workflow.transition, while the UI identifier of the RXLOOKUP table is psx.keyword.

The label identifier specifies the label to use. Some tables only require the single column storing the label, but other may require an identifier column as well. In some cases, two identifiers are required. For example, the key for a Workflow Transition requires the WORKFLOWAPPID column as well as the TRANSITIONID column, while the key for a Role only requires the ROLENAME column.

You can use the Rhythmyx Language Tool to create Localized translations of display data from these tables.

The RXLOOKUP table is the most commonly used table in this list. It stores lookup data for drop lists and similar controls used in Content Editors.

Several Rhythmyx resources have been internationalized to display localized data for these tables. These resources include:

Application Name	Resources
sys_cmpUserCommunity	usercommunity
sys_caCommunityViews	All resources
sys_caContentSearch	All query resources
sys_caSites	casites_gen and casites_comm
sys_ceDependency	Depend, children and parents
sys_ceInlineSearch	resultpage, contenttypelookup
sys_ceSupport	contentstatus, variantlist, Variantlookup, lookup, variantlist
sys_cmpCaMenuNewContent	All resources
sys_cmpMenuViews	All resources
sys_i18nSupport	All resources
sys_commSupport	usercommunities, communityworkflows(2), communitynamelookup
sys_ca	All resources
sys_psxRelationshipSupport	workflowlookup, communityworkflowlookup
sys_rcSupport	relatedcontent, contentslotvariantlist, itemslotvariantlist, contenttypelookup, slotctypevariantlist, variantlist(2), variantlistwithslots(2), communitycontentlookup

Application Name	Resources
sys_relatedSearch	resultpage, relatedsearch
sys_searchSupport	contentlookup, communitycontentlookup, stateslookup
sys_uiSupport	ActionList, ActionListChildren
sys_Variants	variants

If your application links to these resources, it will already include localized data. For example, droplists that refer to the RXLOOKUP table usually use the Lookup resource in the application rx_ceSupport. This resource is already internationalized. If your Content Editor derives its data from this resource, it will present the user with localized data.

However, you will probably have to build applications that do not include these resources, particularly Content Assemblers, which are specific to each implementation. To retrieve localized data from internationalized tables for these applications, Rhythmyx includes two UDFs:

- `sys_LocalizedTextLookup`; this UDF requires you to specify how to derive the Locale used to lookup localized data. For example, you may use the Locale of the Content Item. Use this UDF when you cannot derive the Locale from the User Context, such as on Content Assemblers.
- `sys_LocalizedTextLookupUser`; this UDF derives the Locale from the User Context when looking up localized data. Use this UDF only when you can derive the Locale from the User Context, such as on Content Editors.

To use these UDF's map them to the DTD element associated with the localized data. The following graphic shows an example:

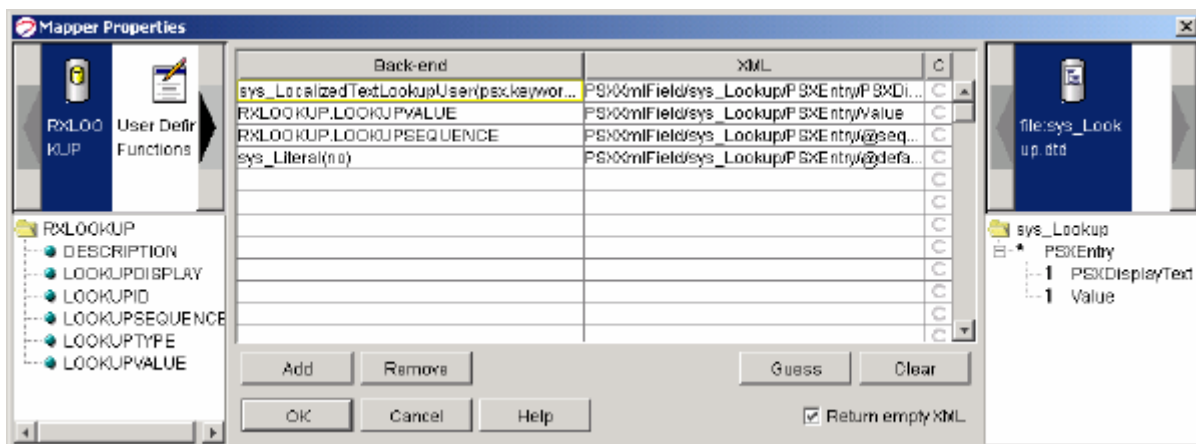


Figure 1: Mapping an Internationalization UDF

Both UDFs require you to specify the translation keys you want to use for the data. Key1 must specify the UI identifier while the remaining keys specify the table columns. In addition, the `sys_LocalizedTextLookup` UDF has a `LanguageString` parameter that specifies the location of the Locale for the translation keyword. When using the UDF on an assembler, for example, this parameter would probably be mapped to the Locale of the item.

Localizing the EditLive Text Editor

You can localize the interface of the `sys_EditLive` inline text editor control. For details see the topic "Adding Custom Menu and Toolbar Actions" in Rhythmyx's *Technical Reference Manual* or the Ephox EditLive! for Java Developer's Guide n <Rhythmyx root>/sys_resources/ephox/ephox_developerguide.pdf.

The following translation keys have also been added to the server TMX file to support the Inline Template Actions Dialog:

- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Inline Template Actions`
- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Select an Action:`
- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Add Line Break Before`
- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Add Line Break After`
- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Remove Line Break Before`
- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Remove Line Break After`
- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Delete Inline Template`
- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Ok`
- `com.percussion.ephox.extensions.ui.PSEditLiveInlineTemplateActionsDialog@Cancel`

Localizing the EditLive Text Editor

You can localize the interface of the `sys_EditLive` inline text editor control. For details see the topic "Adding Custom Menu and Toolbar Actions" in Rhythmyx's *Technical Reference Manual* or the Ephox EditLive! for Java Developer's Guide on the developer section of the Ephox Web Site (www.ephox.com) (<http://www.ephox.com>).

Implementing Internationalized Assembly

In addition to implementing Templates for internationalization, you may need to modify dates based on Locale. Publishing some languages may also require modifying the output character set. Finally, you may want to automate the creation of links to localized versions of content.

Implementing Internationalization of Templates

When developing Templates for internationalization, be sure to include a binding on the Locale. The Locale is stored in the property `sys_lang`, so your binding will resemble the following screenshot:

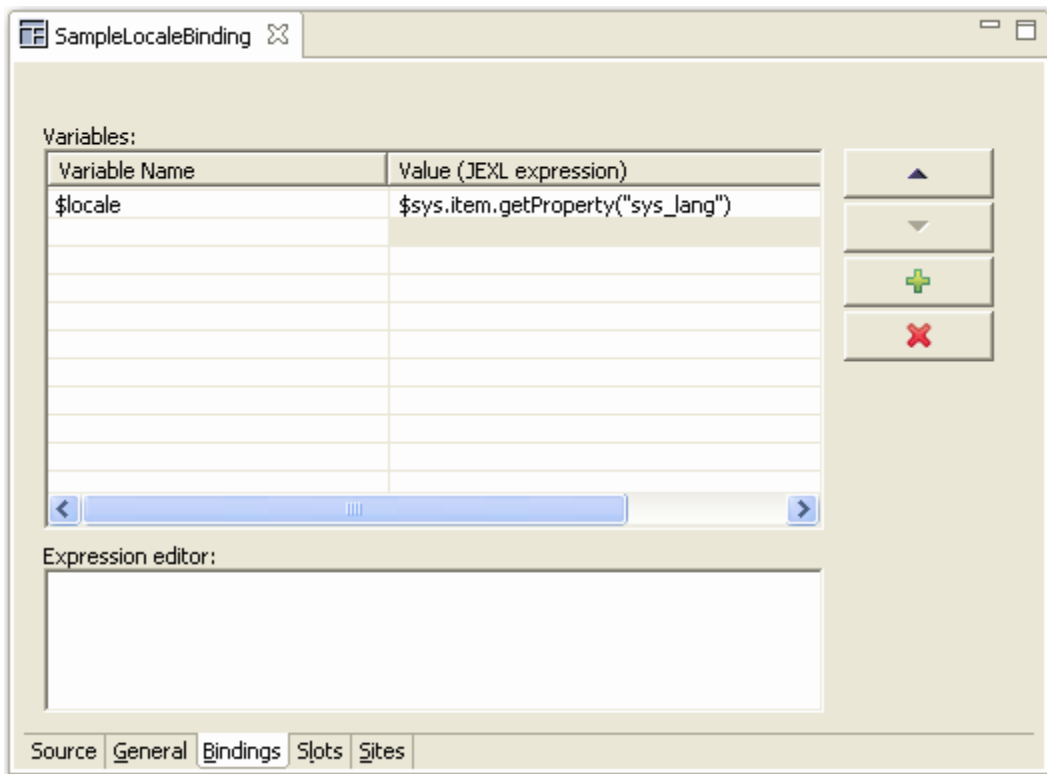


Figure 2: Example binding for Locale

Implementing Localized Templates

A localized Template is a Template that includes only the static text and image references specific to one locale. You should include some indication of the Locale for the Template in the Template name.

If all of the Templates in a localized Site are Localized Templates, you can call all of these Templates directly. If you use some internationalized Templates, however, you may have to use Dispatch Templates to choose the correct Localized Template for a Locale.

Implementing Internationalized Templates

An internationalized Template is a single Template that produces different static text and images for different Locales.

To internationalize images, you must use Dispatch Templates to select the correct image based on the Locale.

Two options are available for internationalizing static text:

- Use Dispatch Templates to choose the correct text to display for the output Locale.
- Use the Translation Bundle on the Rhythmyx Server.

To use the Rhythmyx Translation Bundle:

- a) Define a set of Translation Keys for localized text and add them to the Rhythmyx TMX file.
- b) Create a translation XML.
- c) *Translate the keys for the localized text* (see page 38).
- d) *Upload the translated XML file* (see "Merging TMX Files with the Master TMX Document" on page 39).

Preparing Templates for Localization

You do not have to modify all Templates for Localization. You only have to modify Templates files if they include the following elements:

- Static text
- Alt text
- Locale-specific images or cascading stylesheets.

To modify a Template for translation:

- 1 Define Translation Keys for each static text or alt text string. Use the following conventions:

- Static text: `assemblyPlugin.templateName@textToBeTranslated`

where

- `assemblyPlugin` is the name of the Assembly Plug (Velocity by default);
- `templateName` is the name of the Template where the key is used; and
- `textToBeTranslated` is the string to be translated into other Locales.

For example, if you wanted define a key for the text Search in the Template `rffPgEIGeneric`, you would define the key `velocity.rffPgEIGeneric@search`

- Alt text: `assemblyPlugin.templateName.alt@altText`

where

- o `assemblyPlugin` is the name of the Assembly Plug (Velocity by default);
- o `templateName` is the name of the Template where the key is used; and
- o `altText` is the alt text string to be translated into other Locales.

For example, if you wanted define a key for the altText Search Icon Search in the Template `rffPgEIGeneric`, you would define the key `velocity.rffPgEIGeneric.alt@SearchIcon`.

- 2 Add the Translation Keys to the Rhythmyx server ResourceBundle.tmx file (`<Rhythmyxroot>/rxconfig/I18N.ResourceBundle.tmx`). For details about the correct formulation of entries in the TMX file, see www.lisa.org/tmx (<http://www.lisa.org/tmx>). In the `<prop type="sectionname">` element for each translation key, specify *XSL Stylesheet*. (`<prop type="sectionname">XSL Stylesheet<prop>`). (This option is used by the Rhythmyx Language Tool to retrieve keys used in XSL Stylesheets and Velocity Templates.)

- 3 If you develop Locale-specific images, place them in Locale-specific subdirectories of the `rx_resources/images` directory. For example:

```
../rx_resources/images/fr-ca
../rx_resources/images/de-de
../rx_resources/images/ja-jp
```

Use Dispatch Templates to choose the correct image subdirectory for each Locale.

Implementing Internationalized Numbers and Date Formats

Formats for numbers and dates differ by Locale, and you often need to convert them from the default format in which they are stored in the database to the format appropriate to the Locale.

Numbers within content item text are part of the text string, which Rhythmyx does not translate automatically. The translator of the item must translate these numbers and dates.

To modify the format of dates in system fields, use the Velocity Date Tools.

Changing Output Character Sets

The output character set defines the character set used in the pages Rhythmyx generates. The default output character set of Rhythmyx is UTF-8, but a variety of additional character sets are available.

The output character set is defined for each Template in the **Character set** field on the General tab of the Template editor. If different Locales require different output character sets, you will have to use different Templates for each Locale or set of Locales that use a particular character set. For example, suppose you were outputting to the following Locales using the specified character sets:

- sa-ar: iso-8859-6)
- en-us: UTF-8
- fr-ca: UTF-8
- jp-ja: EUC-JP
- kr-ko: EUC-KR

In this case, you could use the same internationalized Template for the en-us and fr-ca Locales, but would have to use unique localized Templates for the sa-ar, jp-ja, and kr-ko Locales.

Creating Automatic Links to Other Localized Versions of Content

You may want content to cross-link to other localized versions (for example, a Page in English would include cross-links to versions of the same Page in French, German, and Spanish). To create automatic links from one piece of content to other localized versions of the content, create a new Slot using the `sys_TranslationContentFinder`. This Content Finder populates the Slot with a list of Content Items associated in a Translation Relationship with the Content Item being Assembled.

When adding the `sys_TranslationContentFinder` to a Slot, you must specify the Template that will be used to format the links to the associated Translation Content Items in the `template` parameter. You can also specify a maximum number of Content Items to include in the Slot in the `max_results` parameter, and define the order in which the Content Items will be listed in the `order_by` parameter.

Add the Slot to the Page where you want the links to be included.

Localizing the Business User Interface

Implementing internationalized Workflow processes does not require any special work. While you may need to develop special processes for specific Locales, those processes do not need to change to accommodate different Locales.

The Business User interface, however, does require translation, or localization. To localize the interface, add a new language to Rhythmyx. Because Rhythmyx uses the Java definition of language by Locale, you can add multiple variants of each language to Rhythmyx. For example, you can add French as spoken and used in France and French as spoken and used in Canada.

When adding a new language, you translate text inherent in Rhythmyx including:

- Text in Rhythmyx CMS tables;
- Labels in Content Explorer (other than the //Sites and //Folders roots, which cannot be translated);
- Displayable references in Templates;
- Displayable references in extension and system resources (System resources produce labels and links within the Rhythmyx interface [for example, the Insert, Edit, and View Dependents labels in the Action Menu], as well as error codes for extensions provided by Percussion Software that have been localized.)

Rhythmyx uses Translation Memory Exchange (TMX), an XML-standard method of defining translatable data. For details about TMX, see <http://www.lisa.org/tmx/tmx.htm>. The Rhythmyx server installation includes a command-line tool, the Rhythmyx Language Tool, that you can use to create TMX files for translation, and upload translated TMX files to Rhythmyx. (Note: Although the Rhythmyx Workbench includes functionality that allows you to maintain Locales, you can also use the Rhythmyx Language Tool to maintain Locales; For details, see Appendix I, Using the Rhythmyx Language Tool to Maintain Locales.)

While CMS table text and Content Editor labels are available automatically, you must modify Templates that contain any of the following:

- Static text
- Alternative text
- Locale-specific image references or cascading stylesheets

You may also want to localize cascading stylesheets.

Any JavaScript files included in Templates must also be localized if they include alert or prompt messages.

You may also want to use Locale-specific images in your content editor, especially if the images include any text.

Note that when you add a new Locale to Rhythmyx, the Rhythmyx Language Tool only copies system resources (Locale-specific resources shipped as part of Rhythmyx) to the new Locale. Any resources specific to an implementation, such as graphics or JavaScript files, must be copied to Locale-specific folders manually.

You must define the Locales in your system and translate the labels of the interface elements in the Business User interface.

Maintaining Locales

The default Locale for Rhythmyx is the US English (en-us) Locale. You must create any other Locale you want to use in the system.

New Locale Wizard

The New Locale wizard allows you to create a new Locale in the Locale folder of the Localization node in Content Design view.

To access the New Locale wizard:

- From the Menu bar choose *File > New > Other*. In the Select a wizard dialog, choose *Locale*.
- In Content Design view, right-click the Locale category under the Localization folder and choose *New > Locale*.

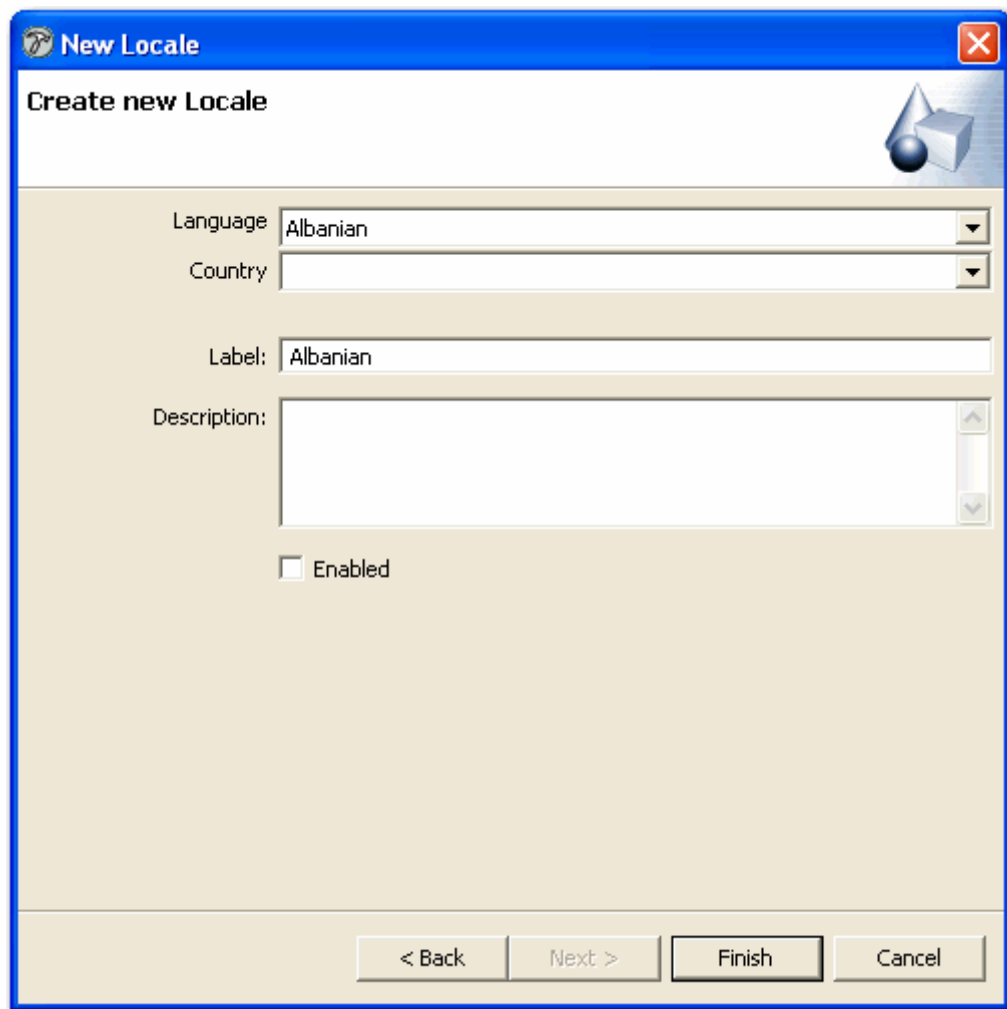
The image shows a Windows-style dialog box titled "New Locale". The main heading inside is "Create new Locale". There is a small 3D cube icon in the top right corner. The dialog contains several input fields: "Language" is a dropdown menu with "Albanian" selected; "Country" is an empty dropdown menu; "Label:" is a text box containing "Albanian"; "Description:" is a large empty text area with scrollbars. Below these fields is a checkbox labeled "Enabled" which is currently unchecked. At the bottom of the dialog, there are four buttons: "< Back", "Next >", "Finish", and "Cancel".

Figure 3: New Locale Wizard

Field Definitions

Language - Drop list of languages.

Country - Drop list of countries and regions that you can associate with the language.

Label - Label for Locale that Rhythmyx displays to users.

Description - Optional. Description of Locale.

Enabled - Whether or not Locale is available to users. Default is unchecked (disabled).

Locale Editor

The Locale editor allows you to change Locale properties.

To access the Locale editor:

- After finishing the *New Locale wizard* (on page 29), click [**Finish**].
- In any view that displays Locales, right-click a Locale and choose *Open*.
- In any view that displays Locales, double-click a Locale.

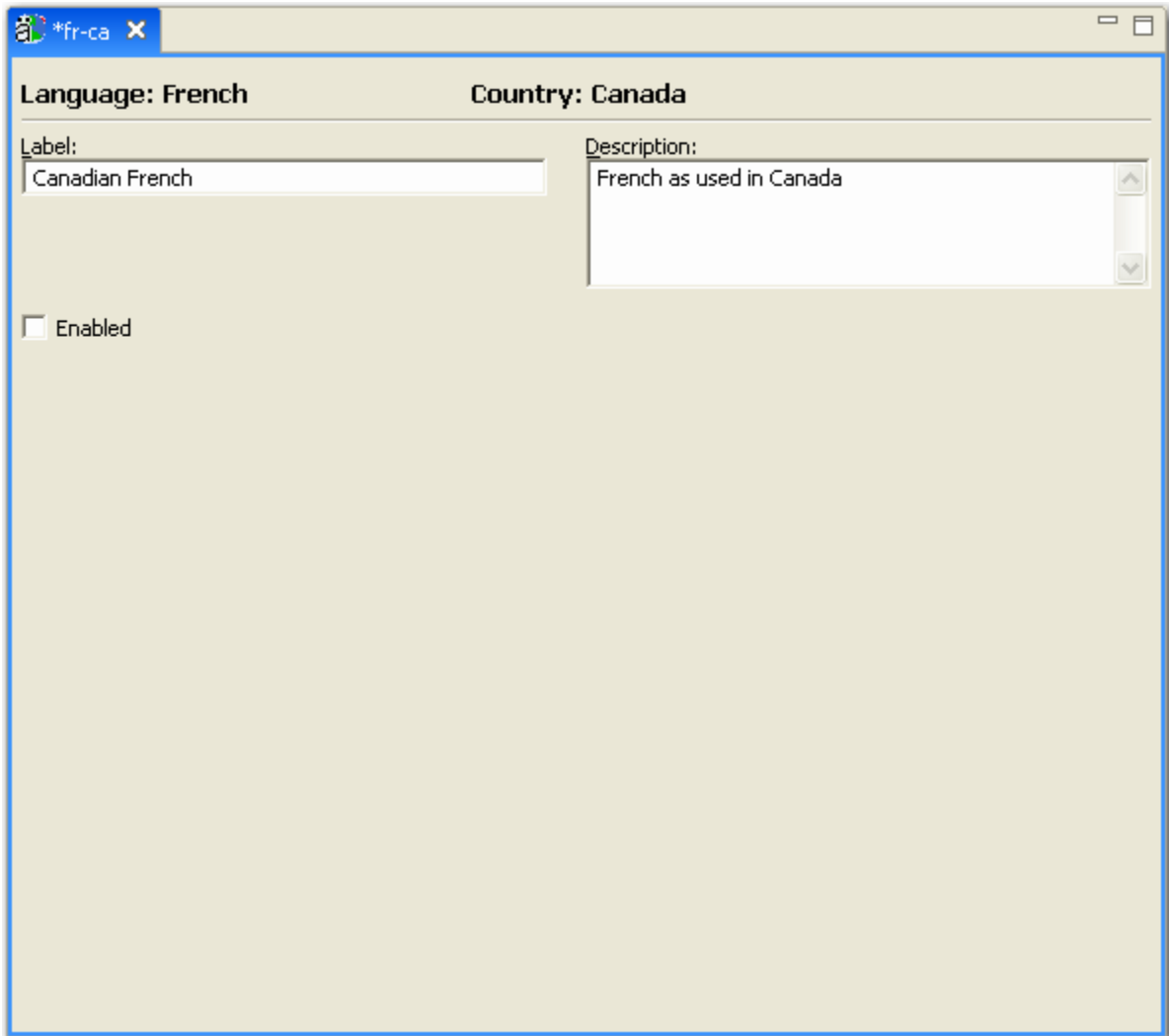


Figure 4: Locale Editor

Field Definitions

Language - Read-only. Locale language.

Country - Read-only. Locale country or region.

Label - Label for Locale displayed in Content Explorer.

Description - Optional. Description of Locale.

Enabled - Whether or not Locale is available to users.

Creating a Locale

To create a Locale, complete the Locale wizard.

For a graphic of the wizard and definitions of the fields discussed below, see *New Locale Wizard* (on page 29).

To create a Locale:

- 1 In Content Design view, under the Localization node, right-click on the Locale folder and choose *New > Locale*.

The *Locale wizard* (see "New Locale Wizard" on page 29) opens.

- 2 In the Language drop list, choose a language.
- 3 In the Country drop list, choose a country or region.
- 4 In Label, enter the name that Rhythmyx will display for the Locale in Content Explorer.
- 5 In Description, optionally enter a description for the Locale.
- 6 By default, the Locale is disabled. If you want to make it available to users, check **Enabled**.
- 7 Click [**Finish**].

The Locale wizard closes and the *Locale editor* (on page 31) opens. The Locale appears under the Locales category of the Localization folder in Content Design View.

Now you can:

- *Make changes to the Locale properties* (see "Editing a Locale" on page 32) in the Locale editor.
- Localize components of your Rhythmyx system. See the document *Internationalizing and Localizing Rhythmyx* for instructions.

Editing a Locale

When you edit a Locale, you can modify all of the properties except the country and region.

For a graphic of the Locale editor and definitions of the fields discussed below, see *Locale Editor* (on page 31).

To edit a Locale:

- 1 In any view where the Locale appears, right-click on the Locale and choose *Open*.

OR

Double-click the Locale.

The *Locale editor* (on page 31) opens with data filled in for the Locale.

- 2 Change the Label or Description or check/uncheck **Enabled**.
- 3 Save and close the Locale editor.

Deleting a Locale

To delete a Locale:

- 1 In any view that displays the Locale, right-click the Locale and choose *Delete*.
The Locale is deleted.

Preparing XSL Files for Localization

You do not have to modify all XSL files for localization. You only have to modify XSL files if they include the following elements:

- Static text
- Alt text
- Locale-specific images or cascading stylesheets.

To modify an XSL file for translation:

- 1 Add the internationalization namespace to the XSL file:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:psxil8n="urn:www.percussion.com/il8n" exclude-result-
prefixes="psxil8n">
```

- 2 Import the `sys_I18nUtils.xsl` file:

```
<xsl:import href="file:sys_resources/stylesheets/sys_I18nUtils.xsl"/>
```

- 3 Add a global variable called 'lang' just after all the imports.

```
<xsl:variable name="lang" select="//@xml:lang"/>
```

If you import the XSL file into any other XSL files, the main XSL file in the set (the XSL into which the all others are imported) must have the variable definition. The `select` attribute in the variable definition assumes that the root element of the XML document has the `xml:lang` attribute. Note that the `xml:lang` attribute is reserved for the user's Locale string. If you want to add a different Locale (for example, the Locale of the item), you must use a different name.

- 4 To localize static text, replace the text you want to localize with a template call. The template call should have two parts: the first is a unique key for the text and the other is a text string. The recommended format is:

```
applicationname.xslfilename@texttobetranslated
```

where

`applicationname` is the name of the Rhythmyx application;

`xslfilename` is the name of the XSL file; and

`texttobetranslated` is the actual text that will be localized.

For example, suppose you wanted to localize the following:

```
<tr>
  <td class="Getlocalestring[1font">Search Results</td>
</tr>
```

The modified code would look like this:

```
<tr>
  <td class="Getlocalestring[1font">
    <xsl:call-template name="getLocaleString">
      <xsl:with-param name="key"
select="'psx.applicationname.xslfilename@Search Results'"/>
      <xsl:with-param name="lang" select="$lang"/>
    </xsl:call-template>
  </td>
</tr>
```

- 5** Use the same technique to localize alt text that you use to localize static text. The recommended format is:

```
applicationname.xslfilename.alt@AltText
```

Where

applicationname is the name of the Rhythmyx application;

xslfilename is the name of the XSL file; and

alt indicates that the template is for alt text; and

AltText is the text that will be localized.

For example:

```

<xsl:attribute name="alt">
<xsl:call-template name="getLocaleString">
<xsl:with-param name="key"
select="'psx.sys_cmpCaSearchBox.searchbox.alt@Calendar Pop-up'"/>
<xsl:with-param name="lang" select="$lang"/>
</xsl:call-template>
</xsl:attribute>
</img>
```

- 6** If you develop Locale-specific images, place them in Locale-specific subdirectories of the rx_resources/images directory. For example:

```
../rx_resources/images/fr-ca
../rx_resources/images/de-de
../rx_resources/images/ja-jp
```

Use the following code to define the src attribute of localized images:

```
src="concat('../../rx_resources/images/', $lang, '/imagename.gif)'"
```

For example:

```

```

- 7** To make all of the keys available to the Rhythmyx Language Tool, list them all as children of the psxi18n:lookupkeys element at the end of the XSL file. For example:

```
<psxi18n:lookupkeys>
<key name="psx.applicationname.xslfilename@Search Results">Search
Results text for Search Box.</key>
<key name="sys_cmpCaSearchBox.searchbox.alt@Calendar Pop-up">Alt text
for calendar pop-up.</key>
</psxi18n:lookupkeys>
```

The value of the name attribute is the translation keyword recovered by the Rhythmyx Language Tool, while the value of each key is the note to the translators for the translation unit.

- 8 When processing, the XSL file needs to know the user's Locale. The Locale is user's session object `sys_lang` variable. In the mapper of the application associated with the XSL, add `xml:lang` as an attribute of the root element of the result document.

PSXUserContext/User/SessionObject/sys_lang	PSXXmlField/RootElement/@xml:lang
--	-----------------------------------

Add the same mapping to any application that imports this XSL.

Localizing JavaScript Files

Localize JavaScript files if they include alert messages or prompts.

To localize JavaScript files, replace the text of alert messages and prompts with a Localized Message function call and a keyword:

```
alert(LocalizedMessage "alert_message_keyword");
```

Localized error messages are stored in the

`rx_resources/js/Locale/globalErrorMessages.js` JavaScript file. Each Locale has a unique version of this JavaScript file located in a Locale-specific subdirectory of the `rx_resources/js` directory. For example, if your system include US English, German, and Canadian French Locales, you would have the following `globalErrorMessages.js` files:

```
rx_resources/js/en-us/globalErrorMessages.js
```

```
rx_resources/js/de-de/globalErrorMessages.js
```

```
rx_resources/js/fr-ca/globalErrorMessages.js
```

Keyword entries are added to the `Psx_MessageMap_Local` variable. The format for keyword entries in these files is:

```
alert_message_keyword: "Locale specific translation of the keyword.",
```

NOTE: All entries **MUST** end with a comma.

For example, if you had the following JavaScript function:

```
function search_submit()
{
    if(document.formx.itemx.value=="")
    {
        alert("Search name is a required field");
    }
    return false;
}
document.formx.submit();
}
```

You modify the alert message:

```
function search_submit()
{
    if(document.formx.itemx.value=="")
    {
        alert(LocalizedMessage("search_name_required"));
    }
    return false;
}
```

```
document.formx.submit();
}
```

Next, add an entry for the keyword to the `PsxMessageMap_Locale` variable of the `rx_resources/js/en-us/globalErrorMessages.js` JavaScript file.

```
var PsxMessageMap_Locale = {
  search_name_required: " Search name is a required field.",
  save_search_prompt: "Please enter a name for this search.\nNote: search
will be overwritten if name already exists.",
  .....
  .....
};
```

Add the keyword entry to the `globalErrorMessages.js` JavaScript file for each Locale. Assuming your implementation included the three Locales specified above, you would also have to add an entry to the `rx_resources/js/de-de/globalErrorMessages.js` and `rx_resources/js/fr-ca/globalErrorMessages.js` JavaScript files.

Any XSL files that use localized JavaScript files must include both the system (`sys_resources`) and local (`rx_resources`) `globalErrorMessages.js` JavaScript files and the Global Variable `lang` mapped to the `xml:lang` attribute of the root element of the XML. See **“Preparing XSL Files for Localization** (see “Preparing Templates for Localization” on page 24, on page 33)”, Step 8 for instructions about mapping the `lang` Global Variable. To add the `globalErrorMessages.js` JavaScript files, include the following script calls in the XSL file:

```
<script language="javascript"
src="../../sys_resources/js/globalErrorMessages.js"></script>
<script language="javascript"
src="{concat('../../rx_resources/js/', $lang, '/globalErrorMessages.js')} "></script>
```

Localizing JSPs

To localize a JSP for Rhythmyx:

- 1 Add a tag library reference to the Rhythmyx JSP tag library:

```
<%@ taglib url="http://rhythmyx.percussion.com/components"
prefix="rxnamespace"%>
```

Where `rxnamespace` is the namespace that will be used to refer to Rhythmyx JSP EL functions in EL function references for localized content derived from Rhythmyx.

- 2 Mark up each instance of localized content derived from Rhythmyx using the following code:

```
${rxnamespace:i18ntext('jspname@keyname', locale)}
```

Where:

- `rxnamespace` is the namespace you specified in the tag library reference to refer to the Rhythmyx SAP EL function library.
- `i18ntext` is Rhythmyx `i18ntext` JSP EL function (this is currently the only supported function).
- `jspname` is the name for the collection of keys. You can name keys by JSP, by project, or according to some other convention.
- `keyname` is the name of the specific key to be localized.

- `locale` specifies the locale into which the text should be localized.

The `locale` parameter can be specified in two different ways:

- In most cases, the request will include the HTTP parameter `sys_lang`; in that case, you can specify `param.sys_lang`: for example

```
${rxnamespace:i18ntext('jspname@keyname',param.sys_lang)}
```

- If the request does not include the HTTP parameter `sys_lang`, you can bind it to a variable elsewhere. For example, in the Rhythmyx login page (`rxlogin.jsp`), the system locale is bound to the variable `locale`:

```
<%
    String locale = PSI18nUtils.getSystemLanguage();
    pageContext.setAttribute("locale", locale);
%>
```

This variable is used later:

```
<title>${rxcomp:i18ntext('jsp_login@Rhythmyx
Login',locale)}</title>
```

Using Locale-specific Images

Images may need to be localized, particularly if they include text. Rhythmyx does not include tools for localizing images themselves. Use appropriate graphic tools to develop Locale-specific images. The images for each Locale should be stored in a Locale-specific subdirectory of the `rx_resources/images` directory. For example, if your implementation includes US English, Canadian French, and Japanese, you would have the following subdirectories:

```
rx_resources/images/en-us
rx_resources/images/fr-ca
rx_resources/images/ja-jp
```

When coding XSL files, use the following code to define the `src` attribute of localized images:

```
src="concat('../rx_resources/images/', $lang, '/imagename.gif)'"
```

For example:

```

```

Rhythmyx Language Tool

The Rhythmyx Language Tool is a command-line interface that allows you to:

- create TMX files for translation; and
- merge translated TMX files with the master internal TMX file.

The Rhythmyx Language Tool is installed with the Rhythmyx server, and is stored in the `rxlft` directory of the Rhythmyx server directory.

Run the language tool from a command line. To run the Rhythmyx Language Tool, change to the `rxlft` directory, and run the `rxlft` executable or shell script.

When the Rhythmyx Language Tool starts, it will display the available actions:

- 1 Generate TMX File

- 2 Merge Translated TMX File with server TMX File
- 3 Exit

Generating TMX Files for Translation

Before generating the TMX file, you should prepare all *XSL file* (see "Preparing Templates for Localization" on page 24, "Preparing XSL Files for Localization" on page 33)s and *JavaScript* (see "Localizing JavaScript Files" on page 35) files for localization, and *localize all images* (see "Using Locale-specific Images" on page 37).

To generate a TMX file for translation:

- 1 Start the Rhythmyx Language Tool.
- 2 Enter 1 (Generate TMX Resources).
- 3 Rhythmyx displays a list of interface sections to translate:
 1. CMS Tables
 2. XSL Stylesheets
 3. Content Editors
 4. Extension Resources
 5. JSP Files
 - A. All
 - B. Back to action menu
- 4 Enter the sections you want to include. You can enter more than one section. Separate sections with commas. NOTE: If you enter "A", the Rhythmyx Language Tool ignores any additional characters you enter.
- 5 Press the <Enter> key.
- 6 Specify the path and name of the output file. The default file and location is rxlt/tobetranlated.tmx.
- 7 The Rhythmyx Language Tool asks you whether you want to include only missing keywords from the master resource in the generated TMX file. Enter "yes" (default) to include only missing keywords. Enter "no" to include all keywords in the TMX file. Usually, "yes" is the correct option.

Rhythmyx generates the TMX file in the location you specified.

Translating TMX Files

Rhythmyx TMX files use only a subset of the TMX DTD.

Each TMX file starts with a header that is generated by the Rhythmyx Language Tool. Do not modify the data in the header. For example, if you attempt to add a language by adding it to the header of a TMX file instead of registering it through RLT, Rhythmyx will not recognize the new language.

The header includes one <prop> element for every Locale currently supported in the CMS implementation.

```
<prop type="supportedlanguage">de-de</prop>
```

```

<prop type="supportedlanguage">en-us</prop>
<prop type="supportedlanguage">fr-ca</prop>
<prop type="supportedlanguage">it-it</prop>
<prop type="supportedlanguage">ja-jp</prop>

```

The Rhythmyx Language Tool generates translation keywords automatically. Each translation keyword is defined by a Translation Unit (<tu>) element. The Translation Unit Identifier (tuid) attribute specifies the translation keyword.

```
<tu tuid="psx.ce.action@Check-out">
```

Each Translation Unit element must include one <prop> child. (NOTE: The TMX DTD allows multiple <prop> children, but Rhythmyx permits only one.) The <prop> child of the <tu> element specifies the section name, which defines the CMS area in which the keyword is used.

Each <tu> element may include one <note> child. This element provides information to all translators about translating the keyword. (NOTE: The TMX DTD allows multiple <note> children, but Rhythmyx permits only one.)

Each <tu> element has a Translation Unit Variant (<tuv>) child for each Locale defined in the header. Each <tuv> element has a <seg> child. Enter the translation for the keyword for the Locale in the <seg> element. The <tuv> element can also include a <note> child to provide information about translating the keyword to a specific Locale. No XML comments are allowed in TMX documents, however. XML comments in a TMX document will not be uploaded with the document.

Merging TMX Files with the Master TMX Document

To merge a translated TMX file with the Master TMX document:

- 1 Start the Rhythmyx Language Tool.
- 2 Enter 2 (Merge translated TMX file with master).
- 3 Specify the path and name of the TMX file you want to merge with the Master TMX document. For example, `rxlt/tobetranslated.tmx`.
- 4 Press the <Enter> key.

Updating TMX Files

You can update the translations in the server TMX file. For example, if you change terminology, you would update the TMX file to use the new terminology. You can even change the terms in the default Locale of the server TMX file. For example, you may want to use “Checkout” instead of the default “Check-out”. You can modify the translation of the keyword to the new value.

To update a TMX File, generate a TMX file for the interface elements you want to retranslate, retranslate the file, and merge the retranslated file with the server TMX file.

Removing Stale Keywords

Keywords may become stale or out of date if you no longer use them. For example, if you delete a Workflow, the Transitions no longer exist and the Transition labels are no longer useful. You can leave these keywords in your server TMX file without harm to your system, but the server TMX file will be unnecessarily large. Therefore, you may want to remove these stale keywords.

The Rhythmyx Language Tool does not provide an option for removing stale keywords, but you can use the following procedure:

- 1** Run the Rhythmyx Language Tool and generate a TMX file. Choose “A” to include all interface sections in the file.
- 2** When the RLT asks whether you want to include only missing keywords, enter “no”.
- 3** Generate the TMX file and save it to the Rhythmyx Language Tool directory (`<Rhythmyxroot>/rxlt`) as `ResourceBundle.tmx`.
- 4** Create a backup copy of the server TMX file (`./rxconfig/I18n/ResourceBundle.tmx`) and store it in a safe location.
- 5** Replace the server copy of `ResourceBundle.tmx` with the new copy saved in the Rhythmyx Language Tool directory.

Defining Translation Properties

By default, when you create a new Translation Content Item, Rhythmyx adds it to the same Workflow and Community as its owner. To specify a different Community and Workflow, you must define a configuration that specifies a different Workflow and Community for each new Content Item of a specific Content Type in a specific Locale. Use the Translation Settings dialog in the Rhythmyx Workbench.

Use the *Translation Settings editor* (see "Translation Setting Editor" on page 42) to enter and change translation settings. With the Translation Settings editor you can

- **Create a new translation setting** (see "Creating Translation Settings" on page 43).
- **Modify a translation setting** (see "Modifying Translation Settings" on page 43).
- **Remove a translation setting** (see "Removing a Translation Setting" on page 44).

NOTE: If you want to allow users to specify the Workflow and Community when creating a Translation Content Item, you must create a new Workflow Action that provides an alternate interface where the user can specify these values rather than using the default interface.

Translation Setting Editor

The Translation Setting editor allows you to add and maintain translation settings.

To access the Translation Setting editor:

- In Content Design view, under the Localization folder, right-click Translation Settings and choose *Open*.

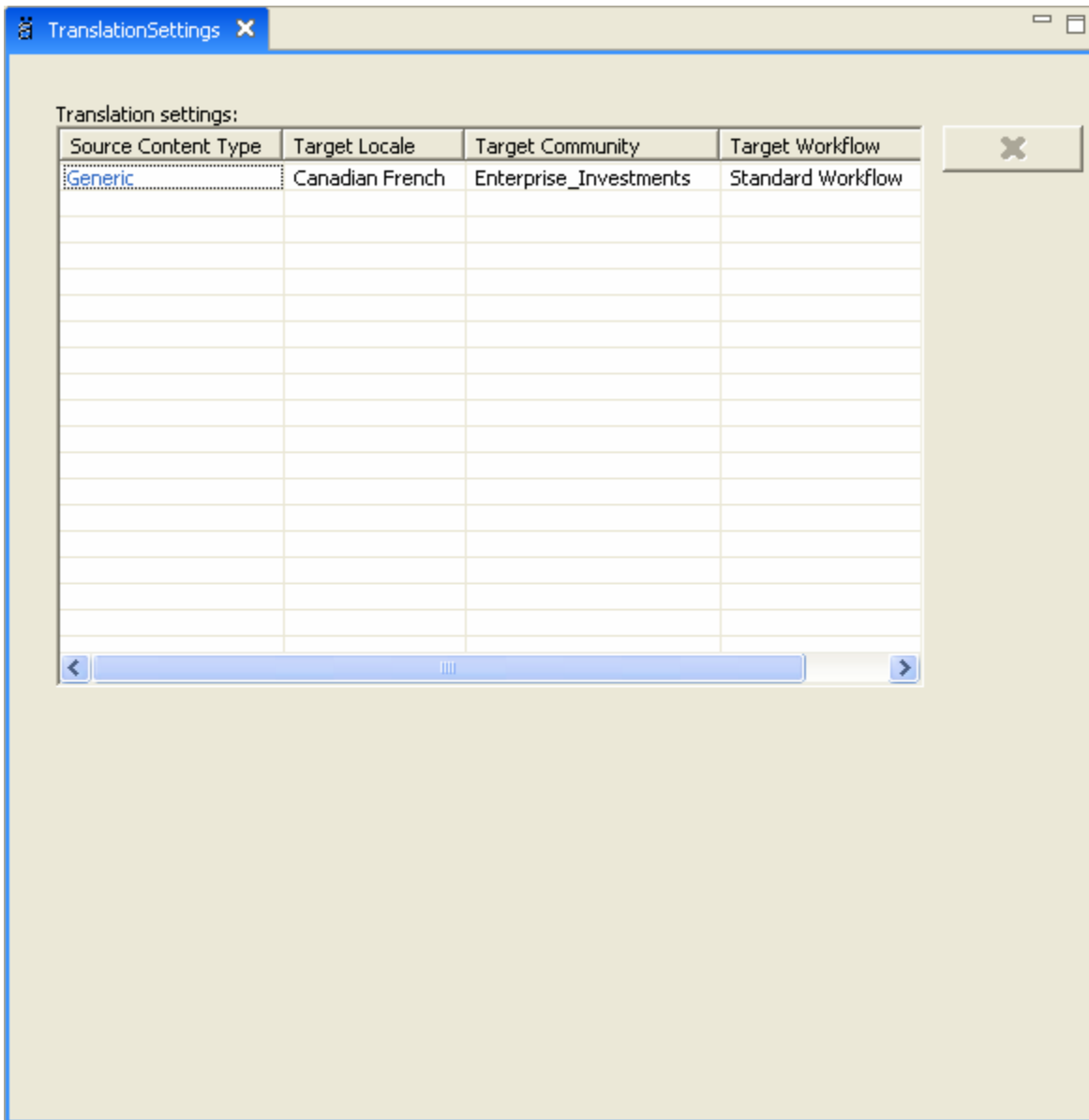


Figure 5: Translation Settings Editor

Field Definitions

Translation settings - Table of translation settings.

Source Content Type - Content Type for this translation setting.

Target Locale - Locale for this translation setting.

Target Community - Community to assign to the Content Type in the specified Locale.

Target Workflow - Workflow to assign to the Content Type in the specified Locale.

Creating Translation Settings

To create translation settings, enter information into the Translation Setting editor's Translation settings table.

For a graphic of the editor and definitions of the fields discussed below, see the *Translation Settings editor* (see "Translation Setting Editor" on page 42).

To create translation settings:

- 1 In Content Design view, under the Localization node, right-click Translation Settings and choose *Open*.
The *Translation Settings editor* (see "Translation Setting Editor" on page 42) opens.
- 2 In the first available row:
 - e) Click the drop list in the **Source Content Type** cell and choose the Content Type for the setting.
 - f) Click the drop list in the **Target Locale** cell and choose the Locale for the setting.
 - g) Click the drop list in the **Target Community** cell and choose the Community for the setting.
 - g) Click the drop list in the **Target Workflow** cell and choose the Workflow for the setting.
- 3 Repeat step 2 to create additional auto translation settings.
- 4 Save and close the editor.

Modifying Translation Settings

To modify a translation setting, in the Translation settings table, change any of values in the row for the setting.

For a graphic of the Translation Setting editor and definitions of the fields discussed below, see *Translation Setting editor* (on page 42).

To edit a Translation Setting:

- 1 In Content Design view, under the Localization node, right-click on Translation Settings and choose *Open*.
The Translation Setting editor opens. The Translation settings table displays any existing translation settings.
- 2 To change a setting, choose a different value for **Source Content Type**, **Target Locale**, **Target Community**, or **Target Workflow**.
- 3 Save and close the Translation Setting editor.

Removing a Translation Setting

To remove a translation setting:

- 1 In Content Design view, under the Localization folder, right-click Translation Setting and choose *Open*.

The Translation Settings editor opens.

- 2 In the Translation settings table, click on the row of the translation setting that you want to remove.

- 3 Click .

- 4 The row is removed from the table.

- 5 Save and close the Translation Setting editor.

Implementing Automated Translation

When working with a Strong Dependency model, you usually want to generate Translation Content Items automatically at a specific point in the Workflow. Before implementing Automated Translation, the following Rhythmyx elements must be in place:

- Translation Content Types
- Locales to be translated (must also be enabled)
- Workflows
- Communities (at least Roles and Content Types must be associated with each Community to make it eligible to use in Auto Translation.
- Translation Configurations

To implement Auto Translation, Add the `sys_createTranslations` Workflow Action to the Workflow Transitions where you want to generate the Translation Content Items.

Example Auto Translation Implementation

To demonstrate implementation of Auto Translation, let us begin with a system with in which Canadian French (fr-ca) and Mexican Spanish (es-mx) locales have been enabled. The Translations will use the Generic Content Type.

The system includes a Central American Marketing Community and a North American Marketing Community. The Central American Marketing Community includes a Spanish Translation Workflow and the North American Marketing Community includes a French Translation Workflow:

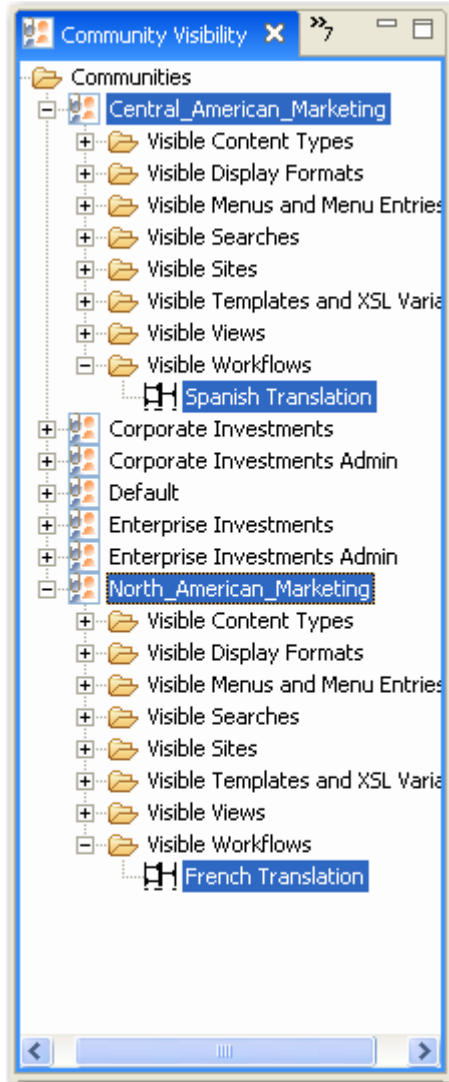


Figure 6: Community Visibility view

Example Auto Translation Configurations

When generating Generic Content Items in the Mexican Spanish (es-mx) Locale, we want the Content Items to go into the Central American Marketing Community and into the Spanish Translation Workflow. When generating Generic Content Items in the Canadian French (fr-ca) Locale, we want the Content Items to go into the North American Marketing Community and the into the French Translation Workflow:

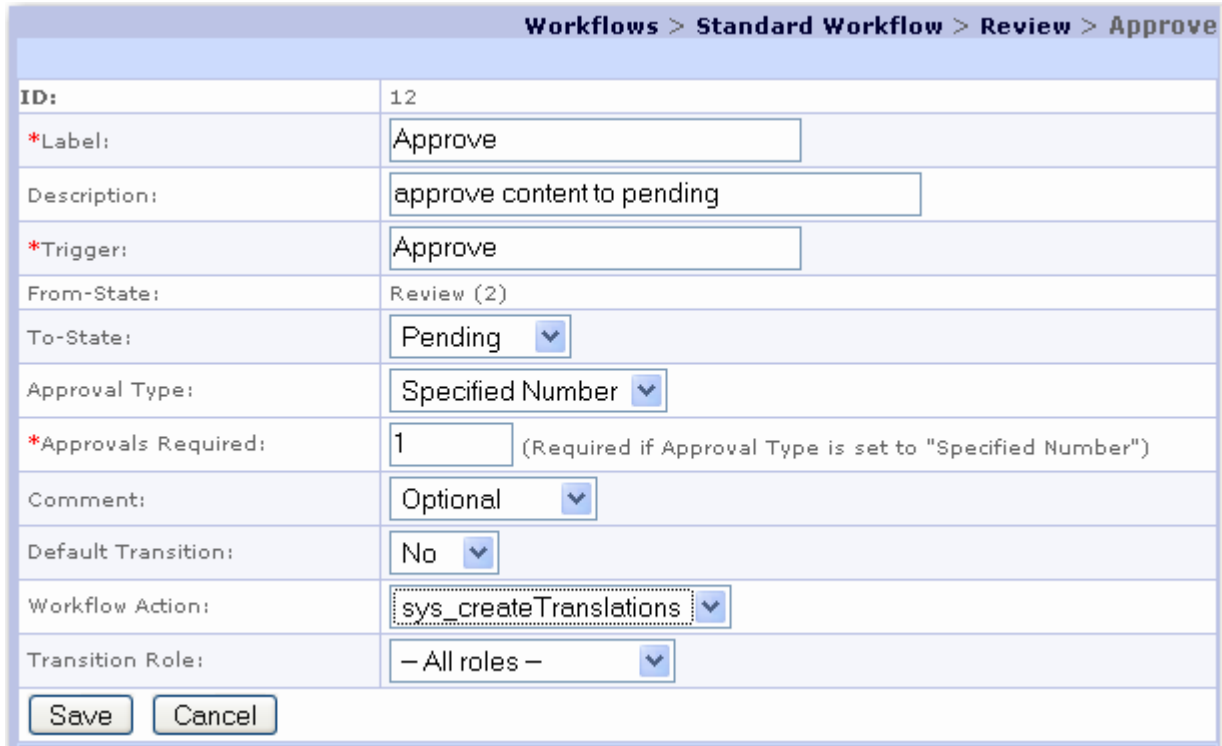
Translation settings:

Source Content Type	Target Locale	Target Community	Target Workflow
Generic	es-mx	Central_American_...	Spanish Translation
Generic	fr-ca	North_American_Ma...	French Translation

Figure 7: Automated Translation Configurations

Example Transition for Auto Translation

Best practice when generating Translation Content Items automatically is to spin off the Translation Content Item as late in the Workflow as possible, so most of the work on the original is complete. Therefore, we will add the `sys_createTranslations` Workflow Action to the Approve Transition from the QA State to the Pending State in the Standard Workflow:



The screenshot shows the 'Workflow Transition Registration' screen for the 'Approve' transition. The breadcrumb navigation at the top reads 'Workflows > Standard Workflow > Review > Approve'. The form contains the following fields:

ID:	12
*Label:	Approve
Description:	approve content to pending
*Trigger:	Approve
From-State:	Review (2)
To-State:	Pending
Approval Type:	Specified Number
*Approvals Required:	1 (Required if Approval Type is set to "Specified Number")
Comment:	Optional
Default Transition:	No
Workflow Action:	sys_createTranslations
Transition Role:	- All roles -

At the bottom of the form are two buttons: 'Save' and 'Cancel'.

Figure 8: Workflow Transition Registration screen

Example Auto Translation Content Item

With the implementation complete, let us take a Content Item through the Workflow. We begin with a sample Content Item:

*** System Title:** Lorem ipsum dolor sit amet

*** Title:** Lorem ipsum dolor sit amet

*** Start Date:** 2004-02-15

Expiration Date: 2004-05-31

Reminder Date: 2004-01-31

Keywords: Lorem ipsum dolor sit amet

Description: Lorem ipsum dolor sit amet

Callout:

veniam, quis nostrud exercitation ullam corper suscipit lobortis nisi ut auquip ex ea commodo consequat. Duis autem velem iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel willum lunombro dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue dui dolore te feugait nulla facilisi.

Figure 9: Example Content Item

When this Content Item Transitions from the QA State to the Public State, two Translation Content Items are generated automatically:

Content Path://Views/All Content/All			
△ Name	State	Community Name	Workflow Name
[es-mx] Copy (2) of Lorem i...	Translate	Central American Marketing	Spanish Translation
[fr-ca] Copy (1) of Lorem ip...	Translate	North American Marketing	French Translation
Lorem ipsum dolor sit amet	Quick Edit	Default	Article

Figure 10: Automatically generated Translation Content Item

Configuring the sys_createTranslations Workflow Action

By default, the sys_createTranslations Workflow Action creates a new Content Item of the same Content Type as the Owner in each enabled Locale in the system, with a Translation - Mandatory Relationship between the Owner and each Dependent.

You can change the configuration of the Action to create a different Relationship or to skip certain Locales. These properties are controlled by the sys_createTranslation.properties file (<Rhythmyxroot>\rxconfig\I18N\sys_createTranslations.properties).

To change the Relationship between the Owner and the Translation Dependent, change the value of the sys_relationshiptype property to the name of the Relationship you want to use instead of Translation - Mandatory.

To skip automatic creation of Translation Content Items in a specific Locale, to the sys_excludelocales property. Use commas or semicolons to separate values.

Implementing Internationalized Publishing

Each internationalization model of Publishing requires a minor modification to implementation.

Implementing Language-centric Publishing Models

To implement a language-centric publishing model, you will need to create a Site for each Locale, with associated Locale-specific Editions and Content Lists. The Content List queries in this model selects Content Items from the Locale-specific Sites. The registrations of the Site, Editions, and Content Lists do not require any special features to implement this model.

Implementing Content-centric Publishing Models

To implement a content-centric publishing model, you only need to define the Location Schemes for the Contexts to include the Locale in the path or the name of each output. For example, the graphic below shows the registration of a Location Scheme that would produce the following path:

```
../EI_Home.war/AboutEnterpriseInvestments/item335_[Locale].html
```

Location Scheme(id): Generic(314)

*Name: Generic

Description: Generic location generation for publishing

*Generator: sys_JexlAssemblyLocation

*Content Type: Generic

*Template Type: P - EI Generic

Save Cancel

Location Scheme Parameters				
New Location Scheme Parameter				
Name(id)	Type	Sequence	Value	
✘ expression	String	1	\$sys.pub_path + \$sys.template.prefix + 'item' + \$sys.item.getProperty ('rx:sys_contentid').String + '-' + \$sys.item.getProperty('rx:sys_lang').String + \$rx.location.getFirstDefined (\$sys.item,'rx:activeimg_ext,rx:sys_suffix', '.html')	

Figure 11: Content-centric Scheme Generator Registration

For example,

```
../EI_Home.war/AboutEnterpriseInvestments/item335_en-us.html
```

In the Value for the Location Scheme Parameter, the binding `$sys.item.getProperty('rx:sys_lang').String` gets the value of the Locale. For more information about this binding and the others in the Location Scheme, see the topic "Creating the Publish Context and its Generic Location Scheme" in the *Rhythmyx Implementation Guide*.

Full-text Search in Globalized Environments

The full-text search Engine shipped with Rhythmyx supports Locales that use the following languages:

- English
- French
- German
- Italian
- Portuguese
- Spanish
- Danish
- Dutch
- Finnish
- Norwegian
- Swedish
- Chinese
- Japanese
- Korean

Each Locale is associated with a specific text analyzer. Text analyzers know how to transform extracted text strings into search terms in different languages through a variety of operations such as removing common words, reducing words to root forms, and discarding punctuation.

If you are using a Locale that is not supported out of the box, you can create a custom text analyzer that supports the Locale, and register the custom text analyzer with the Locale in the Rhythmyx Server Administrator. See the topic *How to Override the Default Text Analyzer* in the Server Administrator online help.

By default, a search query is performed on the user's log-on Locale, and the text analyzer associated with the Locale also analyzes the term entered in **Search for**. In an advanced search, the user can select multiple Locales to search on. If multiple Locales are chosen, separate searches are performed for each Locale using the appropriate analyzer, and the results are combined.

Index

C

- Changing Output Character Sets • 26
- Character sets • 9
- Configuring the sys_createTranslations Workflow Action • 50
- Content Engine • 11, 17
- Content Relationships • 13
- Content-centric Publishing • 15, 51
- Creating a Locale • 32
- Creating Automatic Links to Other Localized Versions of Content • 26
- Creating Translation Settings • 41, 43

D

- Database character set • 9
- Defining the Content Relationship Model for Translations • 13
- Defining Translation Properties • 41
- Deleting a Locale • 33
- Determining System Requirements • 9

E

- Editing a Locale • 32
- Example Auto Translation Configurations • 47
- Example Auto Translation Content Item • 49
- Example Auto Translation Implementation • 46
- Example Transition for Auto Translation • 48

F

- Full-text Search in Globalized Environments • 52

G

- Generating TMX Files for Translation • 38

I

- Images • 37
- Implementing Automated Translation • 45
- Implementing Content-centric Publishing Models • 51
- Implementing Internationalization • 17

- Implementing Internationalization of Templates • 23

- Implementing Internationalized Assembly • 23
- Implementing Internationalized Numbers and Date Formats • 25

- Implementing Internationalized Publishing • 51
- Implementing Internationalized Templates • 24
- Implementing Language-centric Publishing Models • 51

- Implementing Localized Templates • 23
- Internationalization Support on the Rhythmyx Server • 18

- Introduction to Internationalization and Localization in Rhythmyx • 7

J

- JavaScript • 35
- JSP • 36, 38

L

- Language-centric Publishing • 15
- Locale Editor • 31, 32
- Localizing JavaScript Files • 35, 38
- Localizing JSPs • 36
- Localizing the Business User Interface • 27
- Localizing the EditLive Text Editor • 21, 22

M

- Maintaining Locales • 28
- Merging TMX Files with the Master TMX Document • 24, 39
- Modeling Assembly for Internationalization • 12
- Modeling Internationalization of a Rhythmyx Content Management System • 11
- Modeling Internationalized Workflow Processes • 13
- Modeling Internationalized Workflows • 13
- Modeling Publishing • 15
- Modifying Translation Settings • 41, 43

N

- New Locale Wizard • 29, 31, 32

P

- Preparing Templates for Localization • 24, 36, 38
- Preparing XSL Files for Localization • 33, 36, 38
- Publishing Engine • 15, 51

R

- Removing a Translation Setting • 41, 44
- Removing Stale Keywords • 40
- Rhythmyx Language Tool • 37, 38, 39, 40

S

- Spinning Off Translation Versions of Content
Items • 14
- String dependency • 13, 14
- System requirements • 9

T

- Translating TMX Files • 24, 38
- Translation keys • 18
- Translation Setting Editor • 41, 42, 43

U

- Updating TMX Files • 39
- User-defined Functions • 18
- Using Locale-specific Images • 37, 38

W

- Weak dependency • 13
- Workflow Engine • 13
- Workflow Processes in Strong Dependency
Models • 14