Developing Multilingual J2EE Web Applications using Oracle JDeveloper 10*g*

An Oracle Whitepaper
April 2004

ORACLE

Developing Multilingual J2EE Web Applications using Oracle JDeveloper 10*g*

Introduction 4
Multilingual Application Basic Concepts 4
Stages of a Multilingual Application Development Process 4
Oracle Application Development Framework (ADF) Overview 5
  Oracle Application Development Framework (ADF) 5
  Model-View-Controller (MVC) 5
  MVC Model 1 vs. Model 2 6
Java Platform i18n and l10n Basic Concepts 6
  Determination of User's Locale 8
  Locales and ResourceBundles: 8
  ResourceBundle Inheritance 9
Multilingual Architectural Models 9
  Clone & Translate: 9
  Resource Bundles: One JSP / Multiple Languages 10
    Choosing the User Locale: 11
  Resource Bundles: One JSP per language 11
    Choosing the User Locale: 12
Character Encoding 12
  Request: 12
  Page 12
  Response 13
What to Localize in a Multilingual Application 13
Applications Framework Configuration: 14
  Struts configuration: 14
  ADF UIX Configuration: 14
  Business Components Configuration 15
Business Component Generated ListResourceBundle 15
Working with Properties Files 16
  Create the properties files: 16
  Add the Key-Value pairs: 16
  Convert Properties Content to Escaped Unicode 16
Modify Pages to Reference ResourceBundles 17
  Modifying UIX Pages 17
    Specify ResourceBundle Data Source 17
    Modify the UIX page 17
  Modifying JSP's 17
    Jakarta Bean Tag: 17
    JSTL fmt Tag: 18

Large Project Considerations: 18
Implement 3rd Party Tools into Localization Process 18
CASE STUDY: Using Alchemy Software's Catalyst 5.0 19
Write ezParse Rules (Text Files) 19
Exporting Rules to *.ezp files 20
Performing an update on Properties Files 20
Browse to the translated version. 21
Conclusion 22
Bibiliography 22

Developing Multilingual J2EE Web Applications using Oracle JDeveloper 10*g*

INTRODUCTION

Today's global economy means that enterprise applications must work effectively for non-English speaking users who are distributed globally across many different countries and regions.

The Java 2 Platform Enterprise Edition, or J2EE as it's commonly known, consists of a set of coordinated specifications and practices providing solutions for developing, testing, localizing, deploying, and managing multi-tier server-centric applications. This paper summarizes the process of creating multilingual J2EE Web Applications with Oracle JDeveloper 10g, focusing on Oracle's ADF application development framework.

This paper reviews Java Server Pages (JSP) and Java Standard Tag Library (JSTL) support for translatable resources, architectural scenarios for multilingual applications, and demonstrates how using Oracle's ADF UIX component library simplifies the development and localization of multilingual applications.

MULTILINGUAL APPLICATION BASIC CONCEPTS

To be considered "Multilingual", an application has to support the following 4 elemental categories of features:[1]

- *Data representation*: support all the character set encodings that any end users may use
- *Data manipulation:* basic data functions, such as collation, direction of text and internal representation, must be supported
- *Data display:* display strings, currency, number, date formats weights and measures, etc., in a format an end user will understand.
- *Data input*: must understand the format of the data submitted to the application. For example, a British user submits the date "02/09/04" for the 2nd September 2004, while an American would submit the same date as "09/02/04". Knowing the format of submitted data is essential to data integrity.

STAGES OF A MULTILINGUAL APPLICATION DEVELOPMENT PROCESS

Every organization has their favorite development and process improvement methodologies. However, in order to develop multilingual applications, at least the first three of the following processes must be incorporated into their development process:

1. ***Internationalization*** is the process of designing a generalized product so that it supports multiple languages, locale and cultural requirements. This process must be mostly complete before the

[1] G. Nicol, "The Multilingual World Wide Web," http://www.oasis-open.org/cover/nicol-multwww.html.

other processes can begin. It is known by its nickname *i18n*, which stands for “18 characters between *i* and *n*” in the word “internationalization”.

2. ***Localization*** is the process of making a product appropriate for a particular locale’s linguistic (language), cultural or financial requirements. It is sometimes referred to as *l10n*, which stands for “10 characters between *l* and *n*” in the word “localization”.
3. ***Translation*** is the process of converting text written in one language to another language.
4. ***Re-use Translation (optional)*** once the first three processes have been successfully completed, reusing previous translations can save a lot of money especially if you’re making only a few small changes. You can develop your own tools to leverage translation, or you can choose among a number of 3rd party tools that are purpose built for the task. This topic will be covered in detail later in this paper.

ORACLE APPLICATION DEVELOPMENT FRAMEWORK (ADF) OVERVIEW

Before diving into i18n and l10n specific issues, it’s first necessary to understand the underlying architecture of J2EE Web Applications developed using Oracle JDeveloper 10g.

Oracle Application Development Framework (ADF)

Oracle JDeveloper 10*g* includes Oracle Application Development Framework (ADF), an integrated J2EE design time and runtime application framework. Oracle ADF is based on the Model-View-Controller design pattern (MVC).

Oracle ADF takes translation into account at all levels, and simplifies the localization of web applications. For example, ADF UIX provides us with support for localizing the text contained in stylized buttons without digitizing them. We simply translate the strings, and ADF UIX creates the digitized button text for us.

Model-View-Controller (MVC)

MVC was developed at Xerox Palo Alto Research Center (PARC) in the 70’s. Its primary function is to distill into three main subsystems all the functionality that exist in every application:

- **Model** Component: access to a back-end database or remote system
- **View** Component: the User Interface layer.
- **Controller** Component: Workflow automation and design logic of the application, event and error handling.

It’s certainly possible to represent all three functions within a single module – and early JSP code in fact usually was done that way – but HTML is usually written by different people than the developers who write Java code for controlling processing. MVC provides the ability to separate the HTML from the Java code, which simplifies and streamlines the development process.

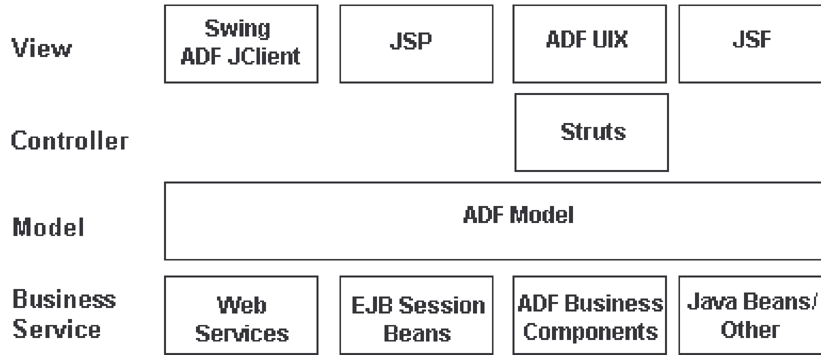


Figure 1: Oracle ADF Model View Controller architecture

MVC Model 1 vs. Model 2

Java Server Pages (JSP) version 0.92 spec defines "Model 1" and "Model 2" MVC implementations. The key differences between Model 1 and Model 2 are:

- In Model 1, an HTTP request is sent to the JSP file, and all the processing is done by the JSP itself (or Bean it interacts with), and the HTTP response comes directly from the JSP.
- For Model 2, a servlet rather than a JSP file receives the initial HTTP request, processes the tasks required for the request, stores data to the bean, passes the data onto the JSP, which then pulls the data from the bean and renders the HTTP response.

With MVC Model 2, the presentation layer (User Interface) is separated from the underlying business logic code. In a typical software localization process, identifying and extracting program resources that need localization is one of the more tiresome and error producing tasks. MVC Model 2 takes care most of this work automatically, and thus simplifies and improves the localization engineering process. The separation of processing code from presentation code also reduces the likelihood of bugs being introduced during localization.

"Jakarta Struts" is a de facto open-source, XML based MVC Model 2 framework for building Web Applications. It's based on servlet and JSP technology, and has been in existence since about 2000. The current revision level is 1.1.

Struts is the controller used by Oracle ADF. Oracle JDeveloper 10g provides a new visual page flow diagrammer for Jakarta Struts that lets you build new pages and actions visually and connect them to design the flow of your web user interface.

JAVA PLATFORM I18N AND L10N BASIC CONCEPTS

J2EE, Struts, ADF UIX and JSP support for i18n and l10n is based on standard Java classes that support internationalization and localization. Using these classes we can establish our preferred language / locale. Once our locale is determined, our application will automatically load UI resources to suit our language/locale, ensure applications data is encoded in an appropriate character set so it can be displayed and stored without corruption, and format our numbers and dates correctly.

These classes are abstracted through user-friendly tag libraries, so we probably won't need to use them directly, but it's very important to understand the underlying Java design and runtime functionality.

Java's base i18n / l10n classes are:

- ***Locale:*** The basic Java i18n class. This class supports a given user's language (and possibly language variant) and country specific requirements such as number and date formats. The Locale object is identified by RFC3066, which combines an ISO 639 language code (i.e., **en** for English), and an ISO 3166 country code (i.e., **en_IR** for Ireland), and possibly a variant string that indicates the preferred operating system (i.e., **en_IR_UNIX**). The Java runtime environment presently supports in excess of 100 locales and 40 languages. Since Java uses the same ISO standards that HTTP uses for its locale identifiers, this simplifies development of web based J2EE applications.

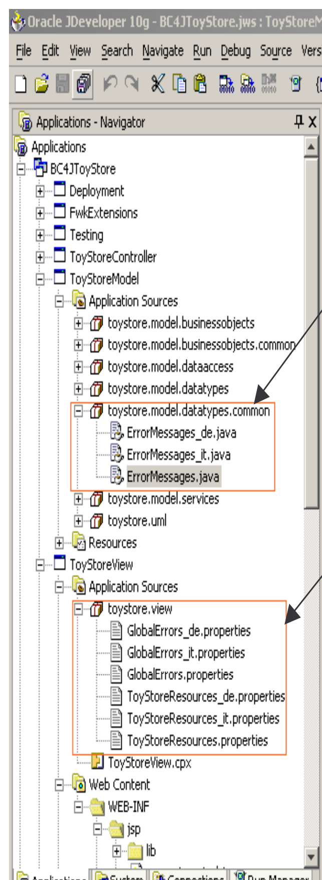


Figure 2 – JDeveloper Applications Navigator

- ***ResourceBundle***: the `java.util.ResourceBundle` class supports messages in multiple languages. The abstract class ResourceBundle has two subclasses: `PropertyResourceBundle` and `ListResourceBundle`. The `ResourceBundle` class is flexible – you can move the key / value pairs from one subclass to the other as long as they're strings.
    - ***ListResourceBundle*** Each ListResourceBundle is stored in a class file. To add support for any additional Locale, you create another source file by copying the base class, and compile it into a class file. You can store any locale-specific object in a ListResourceBundle. If performance is a concern, store all your string resources in a ListResourceBundle, as its performance is marginally better than PropertyResourceBundles.
    - ***PropertyResourceBundle:*** is a `ResourceBundle` implementation that allows for storage of text based resources using the same "name=value" syntax. A `PropertyResourceBundle` is stored as a properties file, a plain-text file that contains translatable text. Properties files are not part of the Java source code, are not compiled, and can contain values for String objects only. If you need to store other types of objects, use a `ListResourceBundle` instead.

      Important note: properties files may <u>only</u> contain characters that can be represented using the ISO 8859-1 character set. "€ ", which is not supported in 8859-1, must be represented as `€` or `\u20ac in` Escaped Unicode. One convenient option is to translate the properties files using whatever non-ASCII encoding is preferred, and then convert the entire translated file to Unicode Escaped characters using the "native2ascii" utility provided by the Java JDK.

      For example, the command line:

      ```
      native2ascii –encoding UTF8 Err_ja.properties Err_ja_UTF8.properties
      ```

      converts the file "Err_ja.properties" into UTF-8 escaped code, and writes the result into the output file "Err_ja_UTF8.properties".

- ***MessageFormat***: The `java.text.MessageFormat` class supports the building of dynamic message strings using arguments specified at run time. The string token `{0}` in the message is replaced by the first argument, while `{1}` is replaced by the second runtime argument, etc. Although this is a powerful method of creating a sentence, the words appear in different orders across different languages, and customization will be required on a language-by-language basis.
- ***MessageResources:*** The `org.apache.struts.util.MessageResources` Struts class uses resource bundles like a database, returning message strings for the user specified locale instead of for the default locale of the server.

Determination of User's Locale

There are mechanisms for an J2EE application to determine a users language / locale preferences:

- Default Browser Language/Locale Preference settings: the HTTP request header field "`Accept-Language`" tells the server what a user's browser language / locale preferences are, and the server automatically loads the appropriate language / locale resource bundles if they are available.
- The user chooses their preferred language / locale settings from a provided list. Their setting can then be stored in a user profile, or remain active for the session.
- The locale may be set using the <fmt:setLocale> JSTL tag. i.e., `<fmt:setLocale value="en" scope="session"/>`.

The best practice is to start with language / locale set according to user's browser settings, and permit them to change their setting manually.

Locales and ResourceBundles:

As described earlier, each ResourceBundle is a set of related language / locale specific subclasses sharing the same base name. The following example shows a set of related subclasses that comprise a `ResourceBundle`:

`TextLabel` is the base name. The characters following the base name indicate the language code, country code, and variant of a Locale. Examples of the naming convention are:

- `TextLabel_en_CA` refers to English language, Canada
- `TextLabel_fr_CA` refers to French language, Canada
- `TextLabel_en_GB` refers to English language, Great Britain.

Java's search algorithm for determining which language / locale subclass to load is as follows:

1. `(baseclass)+(specific language)+(specific country)+(specific variant)`
2. `(baseclass)+(specific language)+(specific country)`
3. `(baseclass)+(specific language)`
4. `(baseclass)+(default language)+(default country)+(default variant)`
5. `(baseclass)+(default language)+(default country)`
6. `(baseclass)+(default language)`
7. `(baseclass)`

So a user with their browser set to Spanish / Spain, TextLabel_es_ES is the desired subclass and assuming the default Locale is en_US, the search for classes occurs in the following order:

1. `TextLabel_es_ES`
2. `TextLabel_es`
3. `TextLabel_en_US`
4. `TextLabel_en`
5. `TextLabel`

If our application fails to find a match, the exception "MissingResourceException" is thrown., but this will only happen if a base class without suffixes doesn't exist.

ResourceBundle Inheritance

Another useful locale related feature of ResourceBundles is inheritance. We can create default key-value pairs in our base classes that are overridden by key-value pairs in our subclasses. For example, if I have an application where my base ResourceBundle is called CarParts_en, and it contains the following key-value pairs:

CarParts_en.properties

```
car.part.door=door
car.part.mirror=mirror
car.part.steeringwheel=steering wheel
car.parts.clovecompartment=glove compartment
car.parts.enginecompartment=hood
car.parts.luggagecompartment=trunk
car.parts.frontglass=windscreen
```

We can create a subclass for terms specific to English spoken in Great Britain by creating a subclass CarParts_en_GB as below which will contain only values that override the base class.

CarParts_en_GB.properties

```
car.parts.enginecompartment=bonnet
car.parts.luggagecompartment=boot
car.parts.frontglass=windscreen
```

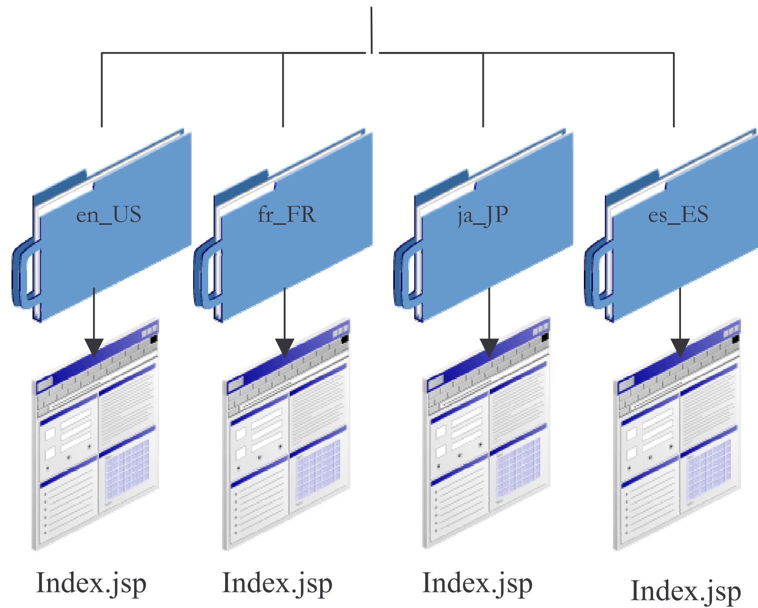
Now when our application is invoked by a user with browser language /locale set to English /Great Britain, their application will display the term "bonnet", while an English language user in the US will see "hood". All the other key-value pairs from the base language are inherited from the CarParts_en subclass, so we need not repeat them in our CarParts_en_GB subclass, unless we intend to modify the value specific to British users. The algorithm for search order (precedence) for inherited key/value pairs is the same as the locale determination algorithm.

MULTILINGUAL ARCHITECTURAL MODELS

Now that we understand how language / locale preferences are determined by J2EE, we need to chose an implementation architecture for our application. There are 2 main architectural designs to choose from:

Clone & Translate:

In this instance, we create our application for a given language/locale, and then make a copy, storing it in its own language/locale. For our example below, we created the application in English initially, and then cloned the entire folder to another language/locale folder, repeating as necessary. Thus our application consists of 4 different JSP's, illustrated below.



Although "Clone and Translate" is a viable option, it's not practical if future modifications are required. For example, if you create full clones of your application for just 3 languages, any change made to one will require an identical change to all the others language versions. Keeping the code in sync will require considerable development resources. If more than a couple of languages must be supported, this becomes a totally impractical option.

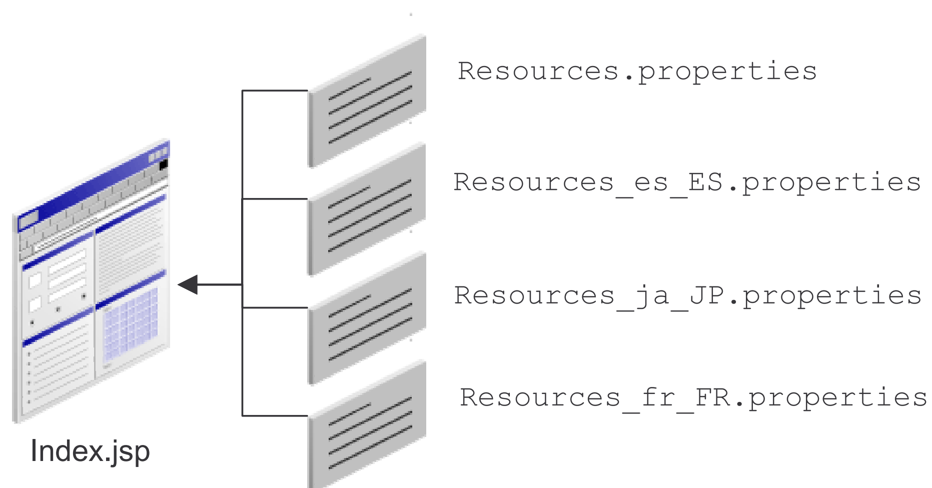
The advantages this architecture provides are:

- the code is somewhat easier to debug
- easier to main content that differs substantially from locale to locale

Other than to satisfy either of these two scenarios, Clone and Translate is not an efficient option, and is not recommended.

Resource Bundles: One JSP / Multiple Languages

In this architecture we have just one JSP, and many locale / language specific resources stored in Java ResourceBundles. This is the recommended architecture for localizing web applications.



The advantages of this design are:

- One page supports all languages.
- The content and logic are the same for all language/locales, so it's easier to maintain code. Just one change to the JSP and the modification is applied to all language / locales.
- It's much easier to add support for additional language/locales – just add another set of resources for the language / locale of choice.

The main disadvantage is:

- It's more difficult to customize content and logic for a specific locale.

Choosing the User Locale:

When implementing the ResourceBundle design architecture, you'll still need to decide how to set the user's language / locale. There are three primary models to choose from:

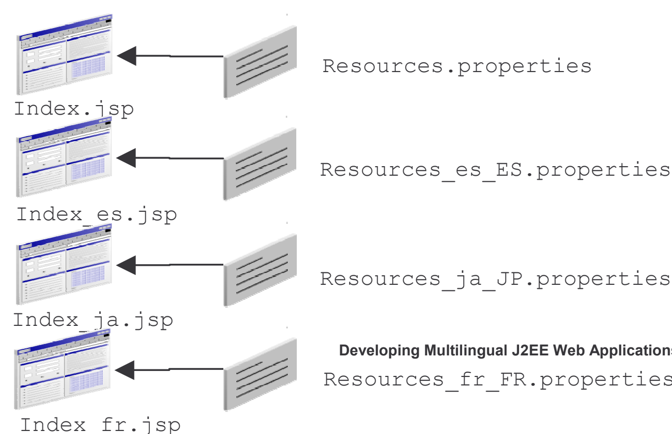
1. **Manual Language Selection**: the initial homepage displays in the base language (say, English) and the user chooses their preferred language / locale. Once the selection is made, the application sets the user's locale to the one selected, and the specified language's resources will be loaded and displayed subsequently until the session terminates, or another language / locale is selected by the user. As previously mentioned, the users's locale may be set using the <fmt:setLocale> JSTL tag.e.g:

   ```
   <fmt:setLocale value="en" scope="session"/>
   ```

2. **Automatic Language /Locale Determination**: In this navigation scenario, there is no initial index JSP for selecting the preferred language / locale from a list. Instead, when a user requests the action page, the HTTP request contains the user's most preferred language/locale as set in their browser per "Accept-Language" header. Further pages are sent to the user in this language. It is possible to combing this with "Manual Language Selection" so that the user can override their browser setting by selecting the language from a list. From that point onward, further pages will be sent to the user in their selected override language/local
3. **Repository Based Settings:** In this navigation scenario, once again there is no initial index JSP for selecting the preferred language / locale from a list. The action page is initially presented in the base language, and once a user successfully logs in, or if a cookie exists, the application looks up their settings from a repository and sets their language/locale preferences accordingly. Subsequent requests for pages will be returned in their preferred language/locale. It is possible to combing this with "Manual Language Selection", especially if using cookies to set the language.

Resource Bundles: One JSP per language

In this architecture, we have one JSP per language with matching locale / language specific resources stored in Java ResourceBundles.



This does not mean it's OK to put lots of hard coded text in the JSP's – the objective of this design is to modify the layout and styles, etc., to suit each individual language as needed.

The advantages of this design are:

- It's much easier to add support for additional language locales – just add another set of resources for the language / locale of choice.
- Each language-locale has it's own JSP, so content and layout can be customized.

Disadvantages:

- The main disadvantage is that it is quite challenging to synchronize code modifications across customized language JSP's.

Choosing the User Locale:

If implementing this design architecture, you'll need a method for loading the specific language / locale page. A useful trick to implementing this design is to store the page name in each language properties file, and use the `<fmt:message>` tag to automatically generate a link to a separate page for each locale.

For example, our `ApplicationsResources_es.properties` file contains:

```
index_page=index_es.jsp

Index_link_text=en Español
```

So we can dynamically generate a link to the page with the following code:

```
<fmt:setBundle basename="view.ApplicationResources" scope="session"/>

<a href="<fmt:message key="index_page"/>">

       <fmt:message key="index_text"/>

</a>
```

CHARACTER ENCODING

J2EE web applications are based on Java and its architecture, so Unicode is the encoding used internally. UIX's preferred, default character encoding is "UTF-8", so no additional work is needed to support multi-lingual UIX applications. However, if your application uses JSP's, there are three types of encoding that must be considered:

Request:

This is the encoding of the parameters in an incoming request. Parameters are submitted as byte values, in the specified charset. The Request encoding may be set automatically to client browser's "Accept-Language" header with the following JSTL tag:

```
<fmt:requestEncoding/>,  or the first i18n action
```

Or we can set it explicitly with:

```
<fmt:requestEncoding value="UTF-8"/>
```

Page

This is the encoding of the bytes in the page itself. If not set, encoding defaults to contentType charset attribute setting. Usually determined by editing tools conventions.

Page encoding is set by the JSP page directive:

```
<%@ page pageEncoding="Shift_JIS" contentType="text/html;charset=UTF-8" %>
```

The Page encoding charset must be an extension of ASCII (i.e., no EBCDIC or UTF-16). UTF-8 is recommended.

If not explicitly set, page encoding defaults to contentType charset attribute value, but not all browsers set this value correctly, so it's recommended to set it explicitly.

Response

The Response encoding defines the encoding of generated web pages, and is determined by the capabilities of the targeted browser.

We set this using the page directive, in the contentType attribute:

```
<%@ page pageEncoding="Shift_JIS" contentType="text/html;charset=UTF-8" %>
```

WHAT TO LOCALIZE IN A MULTILINGUAL APPLICATION

Once you've implemented architecture, you're ready to localize the text and objects in your application. The following is a list of resources that should be modified to correspond to a user's language / locale preferences:

- Titles
- Button Text (UIX)
- Button Graphics / Text (JSP)
- Links
- Labels
- HTML and XML Markup
- Error and validation messages
- Database Row or Column names
- Hint (balloon) text
- Help text and links
- Message Formatting
- Date Formats:
    - Input masks
    - Output masks
    - Date picker controls
- Numeric Formats:
    - Currency
    - General number format
    - Percents

APPLICATIONS FRAMEWORK CONFIGURATION:

When we use an Applications Framework, we must configure it for our multilingual application's requirements.

Struts configuration:

If our application is built using the Jakarta Struts framework, we must configure it by specifying the location of localizable resources. Configuration of `ResourceBundle` location is set in the `Struts-Config.xml` file, using the `message-resources` tag.

Example:

```
<message-resources parameter="view.ApplicationResources"/>
```

This setting says that our sample application properties file is located in the `\View\src\view` directory; since "`\View\src`" is the root directory.

Once defined, the resources can be referenced using the Jakarta `<bean>` tag.

ADF UIX Configuration:

ADF UIX components are provided with pre-translated content and formats for numerous languages and locales. By default UIX applications do not have any language / locale limitations, and will therefore load translated component resources matching the client's browser language preferences, if they are available.

For reasons of maintaining consistency of the User Interface, it may be desirable to limit UIX language / locale support to match only the languages we translate in our UI.

To configure UIX language / locale preferences, we edit the file `\View\public_html\WEB-INF\uix-config.xml`, and add these tags to the `<default-configuration>` tag:

```
<supported-locales>
    <default-locale>en-US</default-locale>
    <supported-locale>es</supported-locale>
</supported-locales>
```

where `default-locale` specifies what locale to display when browser language preference does not match any of the supported locale.. Locale format is the RFC3066 2 char ISO 639 language code+"_"+(optional) 2 char ISO 3166 country code. Add as many supported-locales as your application supports.

You may add as many `<supported-locales>` as your application supports, but `<default-locale>` MUST precede any `<support-locale>` definitions.

Figure 3 - Generating ListResourceBundle

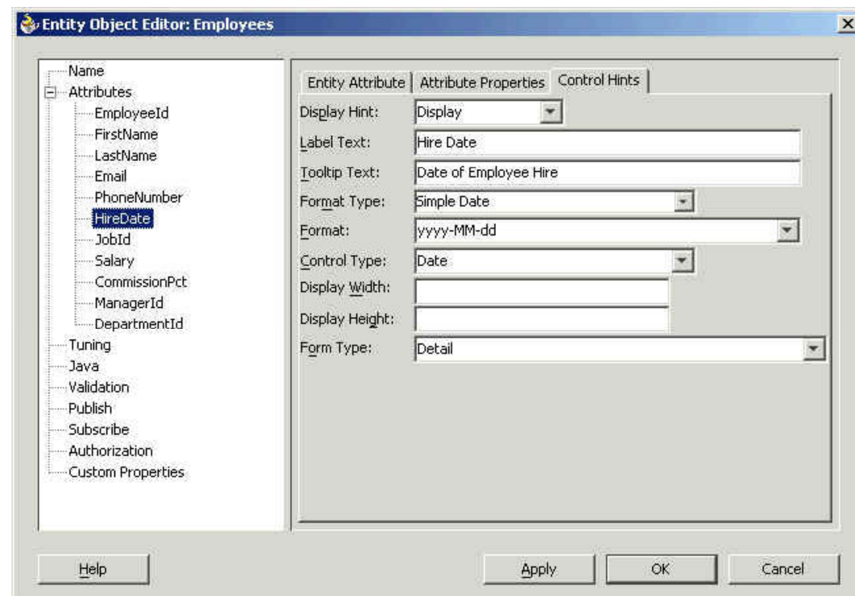
Business Components Configuration

Business Components do not require any configuration.

BUSINESS COMPONENT GENERATED LISTRESOURCEBUNDLE

JDeveloper 10g automatically generates a ListResourceBundle Java file when strings are set in the Business Component's Entity Object Editor. Most strings and format masks are set in the "Control Hints" screen.

The resulting file `EmployeesImplMsgBundle.java` is generated in the `\Model\src\model\common` directory. Next, we copy this file to `EmployeesImplMsgBundle_es.java` in the same directory, and we subclass the constructor (our changes in bold):




```
public class EmployeesImplMsgBundle_es extends EmployeesImplMsgBundle
{
 public EmployeesImplMsgBundle_es()
  {
  }
```

Now the content of the ListResourceBundle is translated into Spanish. Once translated, the Java file is ready to be compiled. Note that when working with non-ASCII characters, the ListResourceBundle must either contain UNICODE escaped, or the encoding must be explicitly specified when compiling into byte-code. There are two ways to approach this requirement:

- Either convert the contents of the ListResourcesBundle Java file to escaped Unicode before building the application:

  ```
  native2ascii -encoding UTF8 MsgBundle_es.java MsgBundle_es_UNICODE.java
  ```

- Or, compile the Java file with runtime argument explicitly specifying the character set encoding of the Java file

  ```
  javac -encoding ISO8859_1 EmployeesImplMsgBundle_es.java
  ```

No further coding is required in order for translations to be displayed in either UIX or JSP pages.

WORKING WITH PROPERTIES FILES

PropertyResourceBundle, or "properties files", contains the strings within our application's user interface. This file contains key-value pairs, and we can add new key-value pairs as new strings are added to the UI. Properties file values do not require compilation and the values are acquired by the application at runtime.

Create the properties files:

The default properties file is `ApplicationsResources.properties`, and it resides in the `\View\src\view` directory.

So for our sample, we created

```
ApplicationsResources_es.properties
```

in same directory (\View\src\view) as the base file

```
ApplicationsResources.properties
```

Add the Key-Value pairs:

Strings within the JSP or UIX page are not automatically exported to the properties file, so we must identify all the strings in our application visually, and then create "keys" for them. The keys are unique names, and we choose names that will help us identify where the text comes from. To help identify the origin of the strings, they can be grouped geographically using prefixes

e.g., the key / value :

```
errorpage.title=Application Error Page
```

tells us the string is a title string from the error page, or we can group them functionally,

```
buttonsubmit=Submit
```

which identifies the string as the name of a button.

Once all the strings in the application are represented in the properties file, we can copy the contents of the English file to the Spanish file, and translate the strings.

Convert Properties Content to Escaped Unicode

Once the properties files are translated, we must ensure that the strings are in a character set encoding that will be correctly interpreted by our application and correctly displayed by the client's browser. Although Java uses Unicode internally, properties files must contain characters that can be represented by the ISO 8859-1 character set. Any non-8859 character sets must be converted to escaped UTF-8 characters, or they will appear corrupt in the browser.

1. Convert to Unicode Escaped characters (i.e., € = € or \u20ac) by using the "native2ascii" utility provided by the JDK

```
native2ascii -encoding UTF8 ApplicationsResources_es.properties
ApplicationsResources_es.propertiesUTF8
```

2. Save the original properties files to `*.propertiesNATIVE`, then rename the `*.propertiesUTF8` to `*.properties`

MODIFY PAGES TO REFERENCE RESOURCEBUNDLES

Once our strings are represented in ResourceBundles, we still must modify our UIX and JSP pages to reference them. We use different mechanisms for referencing ResourceBundles in JSP and UIX pages.

Modifying UIX Pages

Once all the strings in our application are defined in ResourceBundles, we must substitute all the hard coded strings in our UIX page with references to the ResourceBundle keys.

Specify ResourceBundle Data Source

At the top of each UIX file, we must specify the Data Provider that defines a provider name, and the location of the resource file using the <bundle> tag:

Example:

```
<provider>
     <data name="nls">
        <bundle class="view.ApplicationResources"/>
     </data>
</provider>
```

Modify the UIX page

Once defined, we reference the key using the syntax: `${<providername>.<key>}`. For example, a title string becomes:

```
<head title="${nls.empmainheader}"/>
```

Continue until all strings are resourced in this fashion. Add new key - value pairs as needed.

Modifying JSP's

JSP's use a completely different mechanism than UIX file for defining and using ResourceBundles. We can use one of two tags reference the ResourceBundles:

1. Jakarta <bean> tag library
2. and the JSTL <fmt> tag libraries.

You could use both, but ordinarily you should choose only one. No compelling technical reason exists for choosing one over the other, so the choice comes down to personal preference.

Jakarta Bean Tag:

The `<bean>` tag is provided by the Jakarta initiative, which means it is related to and works well with Struts. `<bean>` works in conjunction with the `<message-resources>` tag in the `Struts-Config.xml` file - see the "configuration" section above for more information.

At the top of the JSP, we must specify the tag Library directive, in order to load the `<bean>` library:

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
```

Now we replace all the hard coded strings tags that reference the string via its "key": For example, a page Title sting would be replaced with:

```
<bean:message key="empmaintitle"/>
```

And we continue until strings are resourced. And, of course, we can add new key - value pairs as needed.

JSTL fmt Tag:

JSTL (Java Standard Tag Library) provides us with robust support for internationalization, providing extensive support for locale specific date, numeric, and message formats.

To use the <fmt> tag in our JSP, we must specify the tag Library directive as follows:

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
```

and we specify the location of the resources and the scope of the setting. In this example, we set our resources basename to "view.ApplicationResources" for the life of the current session:

```
<fmt:setBundle basename="view.ApplicationResources" scope="session"/>
```

and we replace the hard coded strings with the following <fmt:message> tag. For example:

```
<fmt:message key="buttonsubmit"/>
```

Continue replacing hard coded strings until strings are resourced. And, of course, feel free to add new key – value when necessary.

LARGE PROJECT CONSIDERATIONS:

For smaller projects, it's sufficient to use manual processes for sending your resource bundles to translators, and keeping the content to date. However, large projects, which can contain thousands or millions of strings, with several project milestone dates, and frequently changing code, require additional localization process considerations. In this section we'll discuss specific strategies for localizing larger projects.

Implement 3rd Party Tools into Localization Process

Many 3rd party tools exist to simplify and improve the localization process. When shopping for such a tool, consider the following highly desirable features:

- **Pseudo Translation**: most 3rd party tools support the validation of the internationalization of your application by Pseudo translating it. Pseudo Translation enables your development team to run a test cycle early in the development project cycle, before a single word is translated.
- **Translation Memory**: 3rd party localization tools will often provide tools for reusing previous translations when your project changes. Needlessly retranslating your strings when versions change is a needless expense. Translation Memory provides reuse of existing translations.
- **Automation:** Automation and scripting are important when you need to maintain consistency of your process across many different languages, or when you have many individual files being sent out to translators as well as translations coming back in that must be integrated into your application.

CASE STUDY: Using Alchemy Software's Catalyst 5.0

One of the many 3rd party localization tools is Alchemy Software's Catalyst 5.0.   In this section, we'll examine how to localize ListResourceBundles using a 3rd party tool. Catalyst is one of many tools available, and is not endorsed by Oracle Corporation. Other highly effective tools include: Multilizer, PASSOLO, SDLX, TRADOSl and WizTom.

The following case study illustrates how one 3rd party tool - Catalyst – supports the reuse of previous translations of our ListResourceBundles for a new version.

Write ezParse Rules (Text Files)

This is a simple task made very easy with Catalyst's previewing technology.

Select Tools | Options | ezParse.

Step 1: Add New Extension

From the File Groups list, select Text Based Files. If your file extension does not exist in the list, then add it.

Step 2: Add New Rule

Select the extension for which you which you wish to write a rule. By default, a rule called Standard is automatically added for each extension. Add a more meaningful rule name for your purposes, e.g. UI Strings

Step 3: Write Methods

Select the rule and press Edit Methods… or simply double-click the rule name. The Edit Methods Dialog appears with your file extension and rule name selected.

Step 4: Preview

In the Preview section, use the Browse button to locate a sample of the file type. Press the Preview button. You will see the file on the left and the result of your parsing rule on the right. As the methods are written, pressing preview will display the result of using this rule.

Step 5 Start Tag

Localisable text is identifiable as text in double quotes that is preceded by an id in double-quotes and a comma. Here is the regular expression start tag saying the same thing -
`[^"]*"([^"]*)",[^"]*"`

`[^"]*`	Any character excluding a double-quote, any number of times (can have a variable number of tabs, spaces, etc. before the double-quote)
`"([^"]*)",`	This is the ID portion preceded by a double-quote, and followed by a double-quote and a comma. To identify the ID portion, it is surrounded with braces. Within braces, the actual id is defined as any character excluding a double-quote, any number of times

`[^"]*"`	Between the comma and the double-quote commencing the string, there can be a variable number of tabs, spaces, etc. – So to make the rule flexible, this allows for any character excluding a double-quote, any number of times, followed by a double quote.

If it is possible to identify the ID portion of the regular expression, there is great advantage to doing so:

- performs accurate Import Translations.
- leverages memos, locks, signed-off status

Step 6 End Tag

The End of String tag is simply identified with a terminating double-quote.

Press OK to accept and keep the rule Press OK again to close the main Tools | Options dialog also. This rule is now saved in the registry of the machine and is available for all future use.

Exporting Rules to *.ezp files

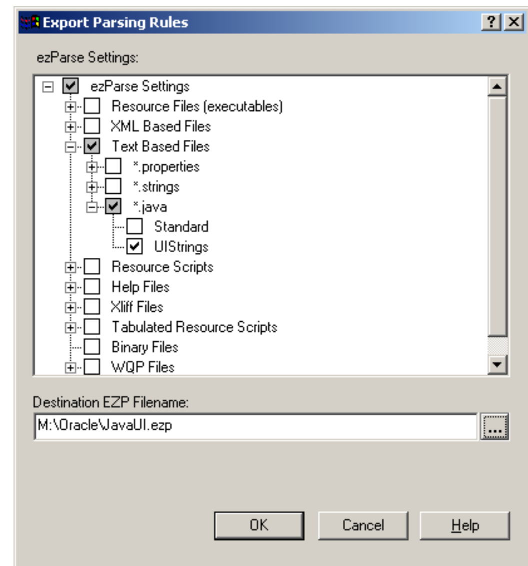
Having written and verified a rule, it may be necessary to pass it on to other team members; this is facilitated within the ezParse feature using

Import / Export.

Select Tools | Options | ezParse

Choose Export…

This yields the Export Parsing Rules dialog containing a tree of selectable rules to export. Select the desired rules and enter a filename for transferring the rules, Select OK.



Performing an update on Properties Files

Having previously created rules for the text files in question, it is now possible to put these files through all the typical processes that localisation involves.

In this example, a single file is inserted into a TTK. It is possible to extend this to as many files as desired per TTK.

Pre-requisites

Version 1 English file (ViewRowImplMsgBundle.java)

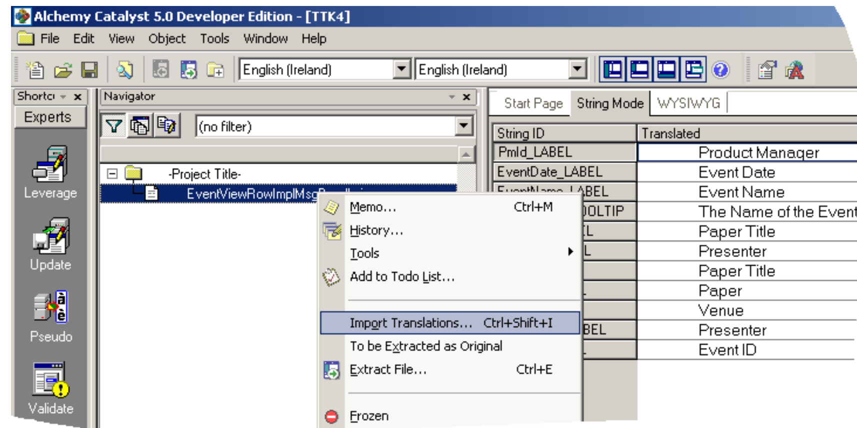
Version 1 translated file (ViewRowImplMsgBundle_es.java)

Version 2 English file (ViewRowImplMsgBundle_V2.java)

Step 1: Generate the Translation Memory

- Using Catalyst, create a new TTK

- Insert ViewRowImplMsgBundle.java. (Catalyst looks for previously written. java rules. If more than one exist, you will be prompted to choose a rule to use for this insertion)
- Now select that file in the Navigator, right-click and choose Import Translations…



Browse to the translated version.

Once that process completes, Catalyst displays a list of the signed-off translations. These matches were determined using the ID specified when writing the UIStrings rule. This TTK should now be saved as ViewRowImplMsgBundle_v1.ttk. This file is a valid TM that can be kept indefinitely. It could, for example, be stored in source control system for use in the future.

Step 3 – Leveraging from the TM

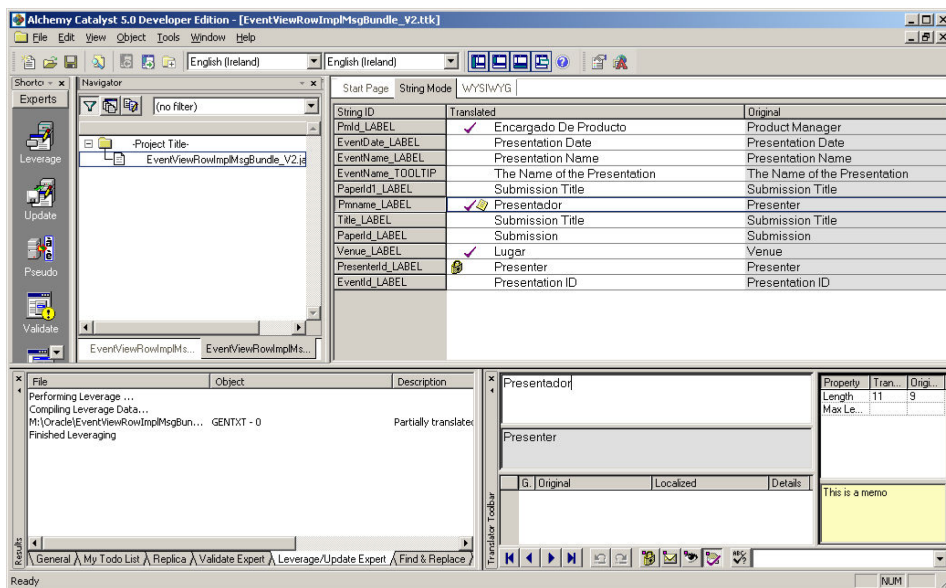
Now that we have a valid TM (Translation Memory), it can be used to leverage into future versions of the same, or any other file. When a new version of the file arrives, the following steps are performed to leverage the translations from the TM

Using Catalyst, create a new TTK

- Insert ViewRowImplMsgBundle_V2.java. (Select the same rule as before)
- Save TTK as Insert ViewRowImplMsgBundle_V2.ttk
- Select the file in the Navigator and choose Leverage from the Experts bar on the left.

Accept the default Options and choose OK. Catalyst will identify any strings that have not changed and bring across the translations. In addition any extra information such as memos, locks or signed off-status information can optionally be leveraged.

The eye symbol with an arrow ( ) indicates that a leveraged translation has been applied. If in the Options page, 'Retain Signed-Off status' was selected, the result would be as follows. In this example, some memos and locks had also applied to ViewRowImplMsgBundle.ttk (v1)



This process can serve as a model for similarly localizing PropertyResourceBundle (*.properties) file content. For more information, visit http://www.alchemysoftware.ie/index.html

CONCLUSION

JDeveloper and ADF support and simplify the creation and localization of multilingual applications well beyond the basic features provided by J2EE and Struts.

Our localization process is greatly simplified by developing our application using ADF UIX, which provides us with pre-translated components, and a simplified method of adjusting the basic look and feel of the application's user interface.

If we choose to implement our web application using JSP's, we can take advantage of JSTL and Struts tag library's robust i18n support.

Further process improvement can be achieved by incorporating 3rd party tools into the localization process. These tools are specifically designed to validate the i18n capabilities of our application, as well as reducing the cost of translation by reusing exiting translatoins.

BIBILIOGRAPHY

G. Nicol, "The Multilingual World Wide Web," http://www.oasis-open.org/cover/nicol-multwww.html.

World-Wide-Web Character Sets, Languages, and Writing Systems http://www.w3.org/International/

Hans Bergsten, "JSTL 1.0: What JSP Applications Need, Part 2", http://www.onjava.com/pub/a/onjava/2002/09/11/jstl2.html?page=1

Greg Murray, "Best Practices in Internationalizing Web-Based Enterprise Applications", http://java.sun.com/javaone/javaone2001/pdfs/2335.pdf

David Czarnecki, "Designing Internationalized User Interface Components for Web Applications", http://java.oreilly.com/pub/a/oreilly/java/news/javaintl_0701.html - ex1

April 2004
Author: Tony Jewtushenko
Contributing Authors:

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Oracle Corporation provides the software
that powers the internet.

Oracle is a registered trademark of Oracle Corporation. Various
product and service names referenced herein may be trademarks
of Oracle Corporation. All other product and service names
mentioned may be trademarks of their respective owners.

Copyright © 2004 Oracle
All rights reserved