

CHAPTER 18

Internationalization

Think of your users first! For whom are you developing? What languages do they speak? Will application training be provided, or is this a self-service application for which proper documentation must be shipped along with the product?



Internationalization is an important step in developing applications and is recommended even if an application may not be localized to support different languages. Localization is one of the last steps in application development, when an application is shaped for specific languages by translating text and number formats for languages, countries, and/or regions. Localization allows application users to interact with the application in the way that is most efficient for them—in their native language.

Creating Internationalizable JSF Applications

Regardless of the need for localizing an application for different languages or countries, every application should be configured so that it may be localized. The process of configuring an application for localization is called *internationalization*, sometimes abbreviated *i18n* (the *18* is for the number of characters between the *i* and the *n*). It is performed by the developer providing data and text labels in separate files to which the application refers by a key value, instead of hard-coding the text. For example, a Hello World message could be statically defined as follows:

```
<af:outputText value="Hello World"/>
```

However, it is preferable instead to define the value field using Expression Language (EL) that looks up a key value from a resource file. The expression used to refer to a key value is `#{bundlevar.keyname}`; here's an example:

```
<af:outputText value="#{vcb.home_greeting}"/>
```

The value associated with the key is defined in a file that serves as the resource bundle for the application. For example, the `home_greeting` key might be defined as follows:

```
home_greeting=Hello World
```

This value could then be translated and saved in an additional resource bundle for the application. For example, a resource bundle for the German language might contain the following value for the same `home_greeting` key, and the bundle would be used when the user's browser default language is set to the German language:

```
home_greeting=Hallo Welt
```

The EL defined in our example includes the use of a variable (`vcb`) and the key value, `home_greeting`. The variable can be defined in several ways. First, an `f:loadBundle` tag from the JavaServer Faces (JSF) core tag library can be used to define the resource bundle for the page, as shown:

```
<f:loadBundle basename="view.ViewControllerBundle" var="vcb"/>
```

This is the pre-JSF 1.2 way of defining resource bundles, because every page that refers to the resource file must include the `f:loadBundle` tag. In this example, `view` is the package name in which the bundle `ViewControllerBundle` resides. Another way to define resource bundles is to declare them globally in the `faces-config.xml` of the application, as described in the next

Defining Strings Using Expression Language

Our code example shows the `value` attribute of an `af:outputText` component defined using EL to retrieve the value from a resource bundle. Of course, resource strings can be used not only for output text components, and not only for the `value` attribute. Common ADF Faces components that should use EL instead of literal strings include layout component strings such as the `disclosedText` and `undisclosedText` attributes of the `af:showDetail` component, the `title` attribute for the `af:dialog` and `af:document` components, and the `text` attribute of the `af:showDetailItem` and `af:panelHeader` components. It's also very typical for all command components to use resource strings, defined in the `text` attribute of the `af:commandButton`, `af:commandLink`, `af:commandToolBarButton`, and `af:commandNavigationItem` components, for example. Additionally, components that do not have their label or value determined by EL that refers to the data model can specify one using a resource string. For example, the `label` attribute of the `af:panelLabelAndMessage` component can be defined using EL that refers to a resource string.

section. A third way is to use some functionality provided by the ADF framework called the *ADF bundle*, described later in the section “Using the ADF Bundle to Create Resource Strings.”



NOTE

By default, JDeveloper sets the encoding for JSF pages to windows-1252. However, for properly internationalized applications, pages should specify an encoding that will support all languages that are intended to be supported, such as UTF-8. The default encoding can be modified by choosing Tools | Preferences in JDeveloper, selecting the Environment node, and choosing the appropriate value from the Encoding drop-down list.

Defining Resource Bundles Globally

To define resource bundles globally, use the `resource-bundle` element of the `faces-config.xml` file to refer to a bundle. Here's an example:

```
<resource-bundle>
  <base-name>view.ViewControllerBundle</base-name>
  <var>vcb</var>
</resource-bundle>
```

In this case, the EL used to refer to a string is similar to the EL for resource bundles that are declared for each page, but the variable is used directly and refers to a variable defined in the `faces-config.xml`, instead of a variable defined in the page's `f:loadBundle` tag. Here's an example:

```
<af:outputText value="#{vcb.home_greeting}"/>
```

This is the most common way for JSF 1.2 applications to be configured, and this can be easier to maintain because every page in the application refers to the resource bundle using the same variable, which makes searching for strings easier when you're trying to locate a string in a particular page at design time.

However, this global configuration is not the default way that ADF applications are configured, because ADF applications created in JDeveloper will be configured for use with the ADF bundle functionality, discussed in the next section. Due to this fact, when choosing the Select Text Resource option in the Property Inspector for a value, text, or label property, the ADF bundle functionality does not use a globally defined variable. Therefore, when using a resource bundle that is globally defined in the faces-config.xml file, do not use the Select Text Resource option when defining strings or labels in the Property Inspector. Instead, choose Expression Builder, as shown in Figure 18-1, and then choose the appropriate key to use for the string resource from the available values in the Faces Resource Bundles node of the Expression Builder dialog, as shown in Figure 18-2.

Using the ADF Bundle to Create Resource Strings

The ADF bundle functionality in JDeveloper allows developers to create customizable resource strings for applications. For example, a set of resource strings can be specified for a particular user or user role, and another set of strings might be used for another user or role. This provides a great deal of flexibility in developing customizable applications.

This functionality is typically used with XML Localization Interchange File Format (XLIFF) resource bundle types, which provide a more structured way of defining resource strings for ease in customizing resources in an application. As stated earlier, however, JDeveloper will assume the use of an ADF bundle regardless of the type of resource bundle used. Even if the additional functionality of an ADF bundle is not required, the design-time benefits to using this method far outweigh using a globally defined resource bundle in faces-config.xml, even though the use of an ADF bundle means that the resource bundle will be defined in each JSF page. (At the time of writing, there is an outstanding enhancement request for JDeveloper that will enable the Select Text Resource dialog to use the ADF bundle and thus allow the resource bundle to be defined globally.)

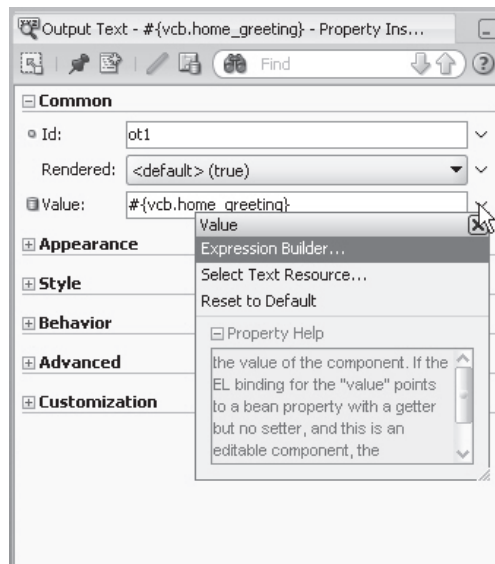


FIGURE 18-1 Property Inspector for text, value, and label properties

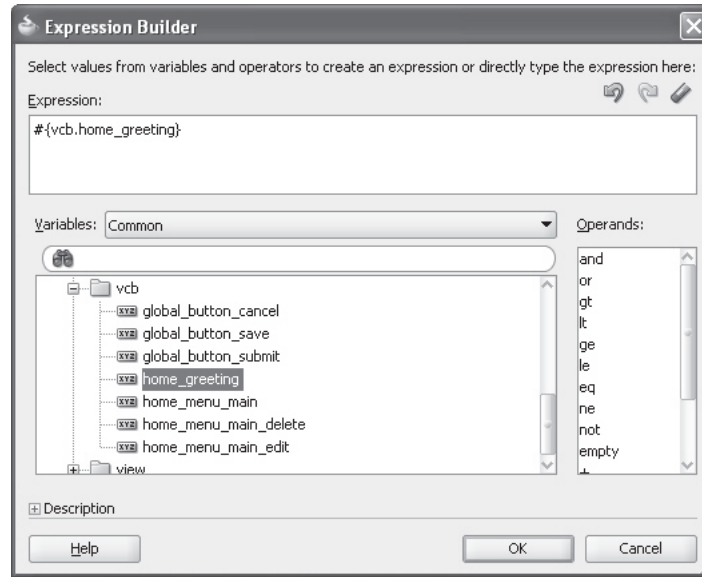


FIGURE 18-2 *Expression Builder dialog for Faces Resource Bundles*

Consider the preceding example where the Expression Builder is used to select a resource string key, as shown in Figure 18-2. In that example, only those keys that already exist are shown in the Faces Resource Bundles node. However, when using the ADF bundle, two important design-time benefits are provided, both related to creating new strings resources. The first benefit is the ability to define new resource keys and values using the Select Text Resource dialog. In the Property Inspector, click Select Text Resource in the drop-down list for a text, value, or label property to open the Select Text Resource dialog shown in Figure 18-3.

The Select Text Resource dialog allows developers to filter the list of existing strings by typing values into the Display Value or Key field, which greatly assists in selecting a string from a long list. The dialog also allows developers to create new strings directly without having to edit the resource file manually. This saves time during development; however, teams should ensure that new resource strings are created appropriately so that inconsistent labels are not generated. (For example, if the agreed-upon string for buttons that save user changes is *Save*, by using the Select Text Resource dialog, it becomes very easy for developers to create new string resources such as *Submit* or *Commit* for a button with the same functionality.)

The second benefit of using an ADF bundle is the ability for strings to be created automatically as developers enter values, labels, and text in the JSF visual editor. The Resource Bundle node of the Project Properties dialog contains the setting Automatically Synchronize Bundle, as shown in Figure 18-4. If selected, this setting will automatically create resource keys and values as they are typed into the visual editor. Note that this feature should be used only for small development projects, because the ability to create duplicate or inconsistent resource strings is significant.

Figure 18-4 also shows the Warn About Hard-coded Translatable Strings option. This option is extremely useful (whether used in conjunction with synchronization or not) for providing visual cues to developers when literal strings are entered into label, text, and value properties.

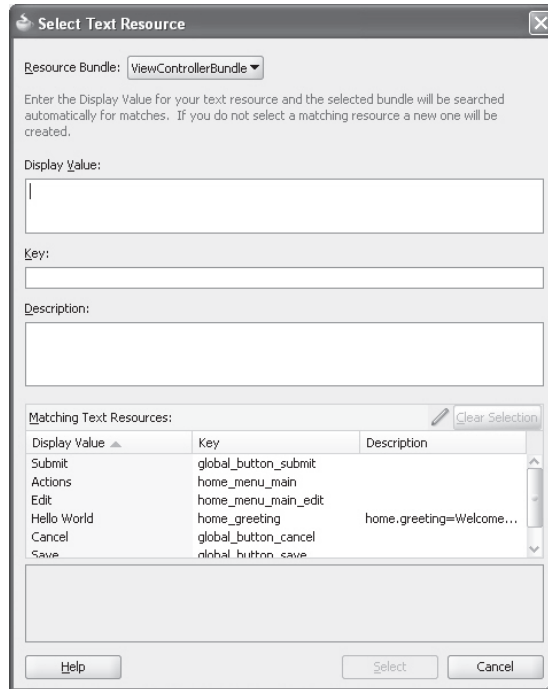


FIGURE 18-3 The Select Text Resource dialog

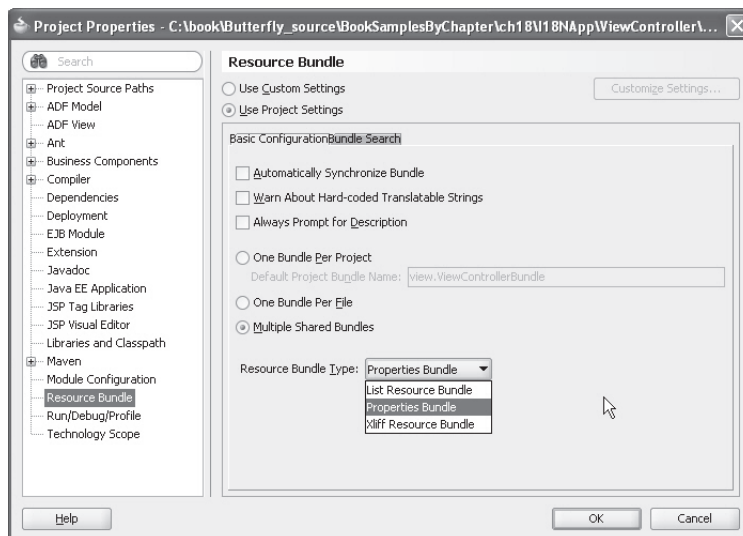


FIGURE 18-4 Project Properties dialog for synchronizing resource strings

When this option is selected, properties that contain literal strings will be highlighted in the Property Inspector with a warning.

As mentioned, ADF applications will use the ADF bundle functionality by default. When a value is selected from the Select Text Resource dialog, a `c:set` tag from the Java Standard Tag Library (JSTL) core library will be added to the page. Here's an example:

```
<c:set var="vcb" value="#{adfBundle['view.ViewControllerBundle']}"/>
```

The `c:set` tag refers to the implicit object `adfBundle` and defines a variable (`vcb` in this example) in each JSF page that uses a text resource. The EL syntax for referring to the bundle is the same as in previous examples:

```
<af:outputText value="#{vcb.home_greeting}"/>
```

Defining Resource Strings

In the preceding sections, the key used to refer to the resource string was `home_greeting`. The key/value mapping for this resource string can be specified in a list resource bundle, a properties file, or an XLIFF resource bundle. The options to configure the resource type are defined in the Project Properties dialog, as shown in Figure 18-4. The most common case is the use of a properties file, where key/value pairs are separated by line and contained in a text file with the extension `.properties`. For example, the `home_greeting` key would be contained in a file named `ViewControllerBundle.properties`, where this file is used as the default resource bundle for the entire `ViewController` project. As shown in Figure 18-4, multiple files may be specified for a project, or each JSF page may use its own resource bundle.

NOTE

Only strings can be defined in a resource properties file. If objects other than strings need to be internationalized for the application, a list resource bundle should be used instead.

In this example, the `ViewControllerBundle.properties` file would contain the key `home_greeting`, mapped to a string value, along with other key/value pairs:

```
home_greeting=Hello World
global_button_submit=Submit
global_button_cancel=Cancel
global_button_save=Save
home_menu_main=Actions
home_menu_main_edit=Edit
home_menu_main_delete>Delete
```

Resource Bundles as Marketing Tools

Defining resource bundles does not have to be done solely for the purposes of supporting multiple languages in an application. Applications that require different labels for marketing purposes, for example, can also use resource bundles so that labels can be easily replaced across the application depending on variations in promotions or seasonal sales.

Translating Access Keys

Access keys are defined by page developers to provide keyboard access to command components by using the ALT-access key (or ALT equivalent). The access key must be a letter within the string. Thus, these keys must be translated and included in the resource bundle for each supported language. For example, if the `ViewControllerBundle.properties` file defines the `home_menu_main_view` key as `Vie&w`, where the `w` is defined as the access key, then the `ViewControllerBundle_de.properties` file should contain the translated version of that key value, as well as specifying a valid access key—such as this: `home_menu_main_view=&Ansicht`

Note the use of underscores in the resource properties file. This is done for both functionality as well as readability. It is helpful to categorize types of strings by their purpose and/or specific page, especially when one properties file is used for an entire application. This helps developers maintaining the properties file to determine which labels are used for various parts of the application, and makes it much easier for page developers to find the appropriate string for a UI component. The use of dot notation in the properties file is allowed, but the EL syntax must be changed to support this. For example, suppose the `home_greeting` key was changed to the following:

```
home.greeting=Hello World
```

You might be inclined to access this key using the following expression:

```
<af:outputText value="#{vcb.home.greeting}"/>
```

However, this code would fail at runtime with a `javax.el.PropertyNotFoundException`, because the expression is attempting to evaluate the property by the name of `greeting` in the `home` object, which of course does not exist. Instead, the EL to access a key value containing a dot would need to include single quotes around the key:

```
<af:outputText value="#{vcb['home.greeting']}"/>
```

When using the Select Text Resource dialog, the syntax is generated correctly and so no effort is required on the part of the developer to modify the syntax. However, when using a globally defined resource bundle and thus selecting the expression from the Expression Builder dialog, the expression will be created without the quotes and therefore would require a manual modification. Thus, when using globally defined resource property files (instead of the default ADF bundle functionality), keys should not include the use of dot notation to delineate categories.

NOTE

Setting resource bundle preferences for a project should be performed before resource strings are used in an application. Any settings specified in the Resource Bundle node of the Project Properties dialog will be relevant from that point forward. This means that if resource bundles have already been created, configured in the project, and/or used in UI components prior to specifying the resource bundle properties, the key/value pairs in the existing bundle file(s) will not be merged to the new file(s), and EL that refers to the original file(s) will not be modified automatically.

Using Tokens in Resource Strings

When the content of the key's value isn't known at design time, a substitution parameter, or *token*, can be provided in the string. To use tokens in a resource string, simply include a token placeholder within curly braces in the message. For example, in a properties file, the following string is specified:

```
home.param.greeting=Hello {0}. You have been logged in since {1}.
```

The values for these tokens can be applied at runtime in several ways. The easiest way is to use the `h:outputFormat` tag from the JSF core library and include nested `f:param` tags to supply the values of the tokenized strings. For example, the following tag produces the output "Hello Lynn. You have been logged in since Mon Jul 13 23:35:14 MDT 2009."

```
<h:outputFormat value="#{vcb['home.param.greeting']}">
  <f:param value="Lynn"/>
  <f:param value="#{userInfoBean.loggedInTimeFormatted}"/>
</h:outputFormat>
```

In this example, a static string is supplied for the first token value, and the value of the `loggedInTimeFormatted` property from a managed bean is supplied as the second value. The parameters will be processed in the order they are supplied, regardless of any value supplied for the optional name attribute of the `f:param` tag.

The ADF Faces equivalent `af:outputFormatted` component does not support the use of nested `f:param` tags. However, internal `af:format` and `af:formatNamed` EL functions can be used within the `value` attribute of ADF Faces components to supply token values. The next example shows the use of the `af:format2` tag which allows two token values to be passed to the resource string:

```
<c:set var="myvar" value="#{userInfoBean.loggedInTimeFormatted }"/>
<af:outputText value="#{af:format2(vcb['home.param.greeting'], 'Lynn',
myvar)}"/>
```

In addition to the `af:format2` function, several other functions are available that allow up to four substitution values to be specified. The signature for each of these methods is listed here:

- `af:format(String base, String param0)`
- `af:format2(String base, String param0, String param1)`
- `af:format3(String base, String param0, String param1, String param2)`
- `af:format4(String base, String param0, String param1, String param2, param3)`
- `af:formatNamed(String base, String key0, String value0)`
- `af:formatNamed2(String base, String key0, String value0, String key1, String value1)`
- `af:formatNamed3(String base, String key0, String value0, String key1, String value1, String key2, String value2)`

- `af:formatNamed4(String base, String key0, String value0, String key1, String value1, String key2, String value2, String key3, String value3)`

Defining Resource Strings Using XLIFF

XLIFF files (.xlf extension) allow developers to maintain a properly formatted XML structure for specifying string resources. The process for creating a resource bundle using the XLIFF format is most easily accomplished by specifying XLIFF as the resource type in the Project Properties dialog, and then choosing Select Text Resource from the Property Inspector when specifying a text value. This action creates the XLIFF resource bundle and adds the newly created element automatically. For example, the `ViewControllerBundle.xlf` might contain the following code:

```
<?xml version="1.0" encoding="windows-1252" ?>
<xliff version="1.1" xmlns="urn:oasis:names:tc:xliff:document:1.1">
  <file source-language="en" original="view.ViewControllerBundle"
    datatype="x-oracle-adf">
    <body>
      <trans-unit id="home_greeting">
        <source>Hello World</source>
        <target/>
      </trans-unit>
      <trans-unit id="global_button_submit">
        <source>Submit</source>
        <target/>
      </trans-unit>
    </body>
  </file>
</xliff>
```

More information about XLIFF, a project of the Oasis group, can be found at <http://docs.oasis-open.org/xliff/xliff-core/xliff-core.html>.

Defining Resource Strings Using Classes

List resource bundles are most commonly used when substitution parameters for resource string values do not provide enough flexibility and coding is required to determine resource values.

Like XLIFF files, the most straightforward way to configure an application to use the list resource bundle is to specify the list resource bundle resource type in the Project Properties dialog, and then use the Select Text Resource dialog to enter string resources that will then be automatically created in a list resource bundle. The file is a Java class that extends `java.util.ListResourceBundle` and implements the `getContents()` method, which returns an array of key/value pairs. For example, the `ViewControllerBundle.java` list resource file is implemented as follows:

```
import java.util.ListResourceBundle;

public class ViewControllerBundle extends ListResourceBundle {
  private static final Object[][] contents = {
    { "home_greeting", "Hello World" },
    { "global_button_submit", "Submit" }
  };
};
```

```

public Object[][] getContents() {
    return contents;
}
}

```

Note that the class is not as human-readable as the properties file and therefore might not be as easily translated by nontechnical translation specialists.

Internationalizing ADF Business Components

Internationalizing labels, date formats, error messages, and other strings such as tool tips in ADF BC is very straightforward. As with UI projects, the Resource Bundle node of the Project Properties dialog is used to define how resources will be configured for an application, including the type of resource bundle and whether one bundle will be used for the project or one bundle per file. In the case of UI projects, it is common to define one resource bundle for the entire application. However, for projects containing ADF Business Components, it may be beneficial to define one resource bundle per file (for example, a resource bundle for every entity object). The benefits of this are seen at design time; JDeveloper nests resources for a particular file with the Application Navigator (when the default Group Related Files option is selected), as shown Figure 18-5. This allows data model developers, who typically work with one entity object or view object at a time, to locate the resource bundle easily for the component with which they are working.

Another reason for using one resource bundle per file is that (especially in the case of entity objects) the labels, data formats, and other internationalizable strings are typically unique to a particular entity. For example, the appropriate `DepartmentId` attribute label for a `Department` entity object might be `Id`, whereas the label for the `DepartmentId` attribute for an `Employees` entity object might be `Department`.

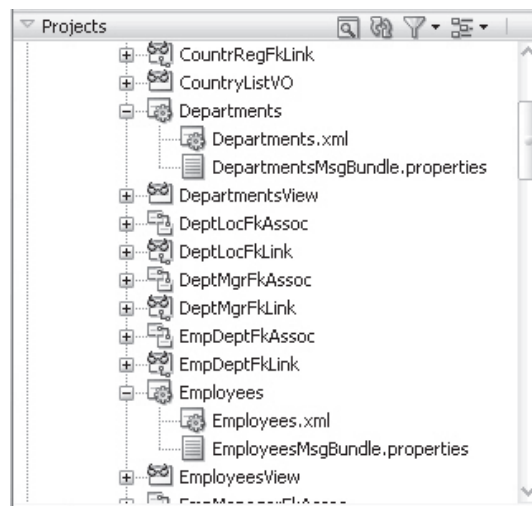


FIGURE 18-5 JDeveloper nests resources with Application Navigator.

Internationalizing ADF BC Attributes

When creating ADF Business Components, column names from the database are converted into appropriate attribute names for Java naming conventions. For example, the `DEPARTMENT_ID` column name is converted to the attribute name `DepartmentId`. This same value is used as the label for the attribute by default. To modify the label so that it is more descriptive, use the Control Hints node of the attribute editor to define a label. Figure 18-6 shows the Control Hints node for the `DepartmentId` attribute where the label and tool tip text are specified.

These control hints can also be entered in the Property Inspector, but regardless of how they are specified, corresponding entries will be automatically added (or modified if the keys already exist) to the default resource property file as shown:

```
DepartmentId_LABEL=Department Number
DepartmentId_TOOLTIP=Identifier for the Department
```

Internationalizing Date, Number, and Currency Attributes To internationalize date, number, and currency attributes (attributes with type `Number`, `Date`, `Timestamp`, `Long`, and so on), specify a format type (`Date`, `Currency`, or `Number`) for the attribute as appropriate, and define a format if necessary. This will add an entry for the built-in ADF BC formatter to the resource bundle, as well as an entry for any format specified. For example, the following resource keys are added to a resource bundle when specifying the `Salary` attribute as type `Currency`, the

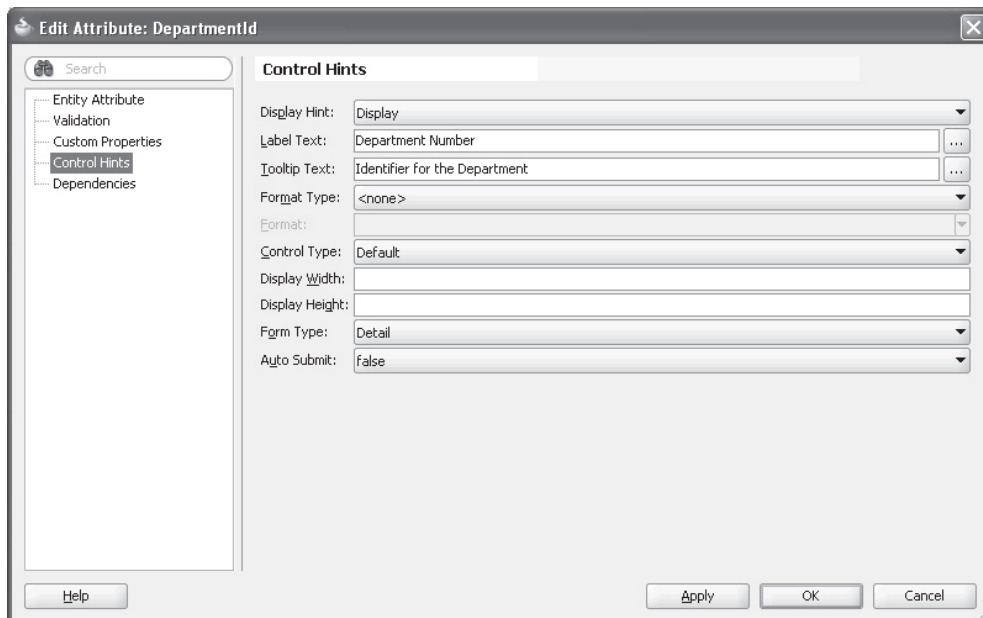


FIGURE 18-6 Specifying control hints for attributes

CommissionPct attribute as type Number with format of ##.##, and the HireDate attribute as type SimpleDate with a format of MM-dd-yyyy:

```
Salary_FMT_FORMATTER=oracle.jbo.format.DefaultCurrencyFormatter
HireDate_FMT_FORMATTER=oracle.jbo.format.DefaultDateFormatter
HireDate_FMT_FORMAT=MM-dd-yyyy
CommissionPct_FMT_FORMATTER=oracle.jbo.format.DefaultNumberFormatter
CommissionPct_FMT_FORMAT=\#\#\.\#\#\
```

The list of valid formats for an attribute with type SimpleDate is available at <http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html>, and the list of valid formats for an attribute with type Number is available at <http://java.sun.com/j2se/1.5.0/docs/api/java/text/DecimalFormat.html>. The currency format type has a non-editable format.

TIP

To add format masks that appear in the format drop-down list for attributes, modify the formatInfo.xml file located in the o.BC4J directory of the system directory (typically C:\Documents and Settings\<user>\Application Data\JDeveloper\system11.1.1.2.x.x\o.BC4J). A restart of JDeveloper is necessary to pick up the change.

Internationalizing ADF BC Error Messages

Like attribute control hints, business component error messages are automatically internationalized when an error message is added to a validator. For example, Figure 18-7 shows the Select Text Resource dialog that is launched when clicking the magnifying glass icon in the Failure Handling tab of the Validation Rule editor.

Defining an error message in this way adds the key and to the appropriate resource bundle. Here's an example:

```
INVALID_LOCATION_ID=Invalid Location Id.
```

Error messages can also be entered directly in the Failure Handling tab of the Validator dialog and will use a generated key. However, to make it easier to locate keys for reuse, use the Select Text Resource dialog to specify the value, key, and an optional description.

Built-in validators are automatically created for attributes that are based on database columns that contain constraints such as not null, precision, and so on. Custom error messages can be added for these validators (again, using

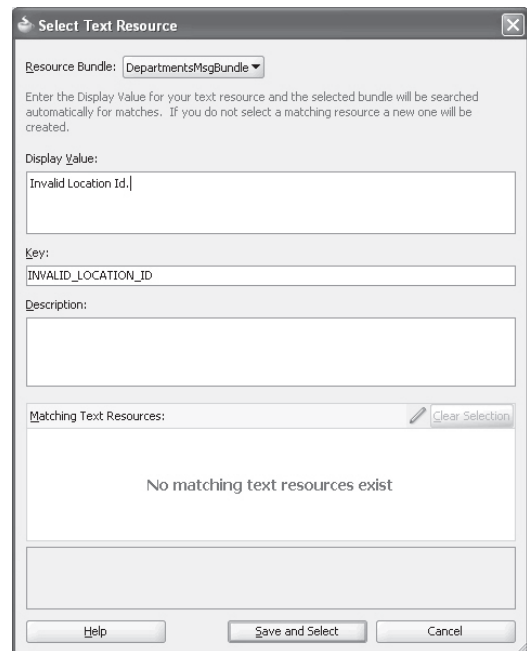


FIGURE 18-7 Select Text Resource dialog

the Failure Handling tab of the Validator dialog), but note that these validators are already translated into many languages, as discussed later in the section “Localizing ADF BC Applications.”

Using Tokens in Validation Error Messages Tokens can also be used in error messages and warnings for ADF BC validators. Expressions are used to evaluate the value for the token. For example, if a validator is added to the `LocationId` attribute so that it must correspond to a `LocationId` that is present in a view accessor, the error message could be defined with a token that indicates the appropriate value. First, the error message is defined as shown in Figure 18-8 (defining the string resource with a token adds the key and value to the resource bundle file as usual).

Notice in Figure 18-8 that the Message Token field accepts an expression to evaluate the value for the token. In this example, the token would be set to a Groovy expression as follows:

```
def locationvalue = []
while ( LocationsView1.hasNext() ) {
LocationsView1.next()
locationvalue.add(LocationsView1.currentRow.LocationId) }
return locationvalue
```

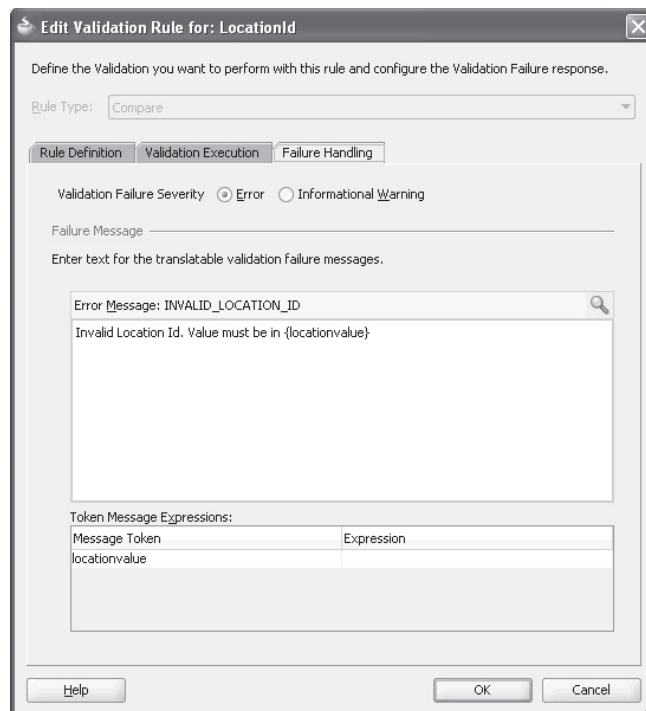
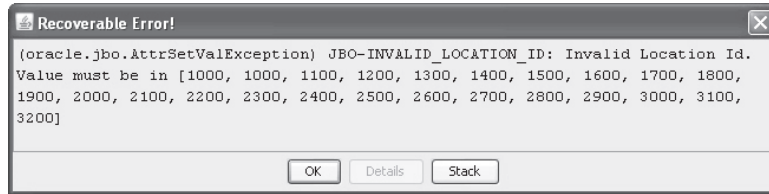


FIGURE 18-8 Defining a validation error message that contains a token

At runtime, the key value replaces the `locationvalue` token with the value defined in the expression, as shown here:



Localizing Applications

While internationalization of applications should be performed throughout development, localizing applications should be performed after application testing is completed using the default language. This way, resource bundle files that are defined for the application's default language can be translated and subsequently added to the project as a last step before testing, without having to synchronize multiple resource files.

Localizing ADF Faces Applications

Once internationalization for an application is completed and the resource file keys are translated, localizing an application is very straightforward. For example, to support German and French languages using properties files, save the `ViewControllerBundle.properties` file as `ViewControllerBundle_de.properties` and `ViewControllerBundle_fr.properties`, respectively. Follow this same convention for XLIFF files. For list resource bundles, save the `ViewControllerBundle.java` file as `ViewControllerBundle_de.java` and `ViewControllerBundle_fr.java`, and change the class signature to match the filename. The `de` and `fr` codes represent the ISO 639 language code. Additionally, an ISO 3166 country code can be appended if a language is used by more than one country.

Creating Properties Files for Special Character Sets

To create properties files for character sets that are outside of the ISO-8859-1 encoding (such as multibyte character sets), you need to use Unicode escaping to allow the special characters to be encoded properly—for example, `\u<char-code>`. Properties files must be saved in the 8-bit Latin-1 (ISO-8859-1) character set, and not doing so results in question marks (?) appearing at runtime rather than the proper characters. Typically, however, human translators will save a working copy of the properties file in an encoding that supports the character set for the language they are translating to, and directly type in the characters to provide the translation. It is impractical to ask translators to create the translated file using Unicode character codes instead of native characters. To solve this dilemma, use the `native2ascii` conversion tool provided in `<JDK_HOME>\jdk\bin` to convert properties files saved with encoding other than ISO-8859-1 to the Unicode format.

**NOTE**

List resource bundles and resource properties files can be used in parallel if required. The list bundle will be searched first for a key. If the key does not exist, then the properties file will be searched.

For a full list of the ISO 639 and 3166 country codes, refer to http://www.loc.gov/standards/iso639-2/php/code_list.php and http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm.

Specifying Supported Locales in a JSF Application

Once the values are translated, add default and supported locales for the application to the faces-config.xml file. In the Overview tab of the faces-config.xml editor, select the Application node and in the Locale Config section, and add language and/or country and region codes for the supported locales, as shown in Figure 18-9.

This adds the following code to the faces-config.xml file:

```
<locale-config>
  <default-locale>en</default-locale>
  <supported-locale>en</supported-locale>
  <supported-locale>en_US</supported-locale>
  <supported-locale>de</supported-locale>
  <supported-locale>de_DE</supported-locale>
  <supported-locale>de_AT</supported-locale>
</locale-config>
```

Typically, the user's browser settings determine the locale requested, as shown in Figure 18-10.

At runtime, the application will search locales according to a restrictive match with the user's browser settings. For example, if the user's browser locale is set to de_AT, an exact match is found, so the `ViewControllerBundle_de_AT` resource will be used to load resource strings. If the user's browser locale is set to de_CH, an exact match isn't found, but the generic German locale (de) is supported, so the `ViewControllerBundle_de` resource will be used. Further, if a locale that is not among the list of supported locales is set in the browser, the default locale (`ViewControllerBundle_en`) will be used.

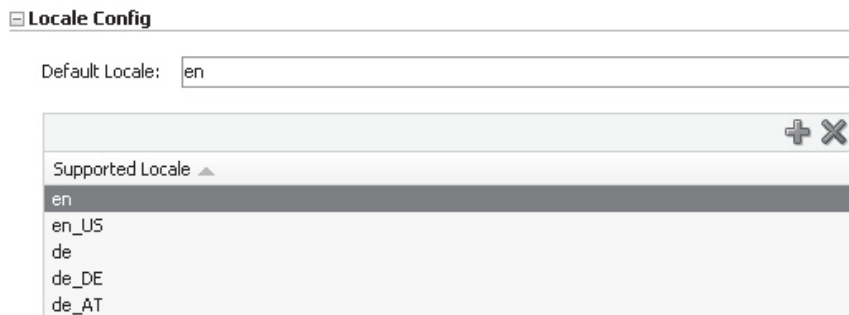


FIGURE 18-9 *Locale configuration in faces-config.xml*



FIGURE 18-10 *Language dialog*

Localizing ADF BC Applications

The same technique for localizing UI resource bundles is used to localize ADF BC resource bundles, and localization can be tested in the business components browser. For example, consider the case where key/value pairs for entity attributes are specified in a resource properties file as shown in `DepartmentsMsgBundle.properties`:

```
DepartmentId_LABEL=Department Number
DepartmentName_LABEL=Department Name
ManagerId_LABEL=Manager's Employee Id
LocationId_LABEL=Location Code
```

To localize these strings, the file would be saved as `DepartmentsMsgBundle_de.properties` and translated appropriately, as follows:

```
DepartmentId_LABEL=Abteilungs Nummer
DepartmentName_LABEL=Abteilungsname
ManagerId_LABEL=Vorgesetzter
LocationId_LABEL=Standort
```

As mentioned previously in this chapter, error messages for built-in validators are translated into several languages. For example, Figure 18-11 shows the default error message dialog for an invalid scale when an application module is run in the `de_DE` locale.

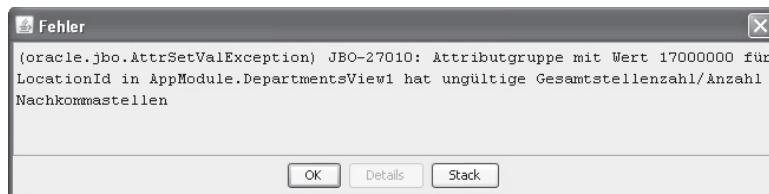


FIGURE 18-11 *Error message dialog*

To run the Business Component browser in a language other than the default `en_US` locale, specify the locale, country, and region properties of the application module configuration. Figure 18-12 shows the configuration dialog where the default locale and country properties have been modified.

Localizing Dates, Numbers, and Currencies

UI components that are created in a JSF page by dragging attributes from the Data Control palette will automatically add nested `af:convertDateTime` and `af:convertNumber` tags as appropriate. For example, if the `HireDate` and `CommissionPct` attributes in the `Employees` view object are created as output text components in a page, the following code results:

```
<af:outputText value="#{bindings.HireDate.inputValue}">
  <af:convertDateTime pattern="#{bindings.HireDate.format}"/>
</af:outputText>
<af:outputText value="#{bindings.CommissionPct.inputValue}">
  <af:convertNumber groupingUsed="false"
    pattern="#{bindings.CommissionPct.format}"/>
</af:outputText>
```

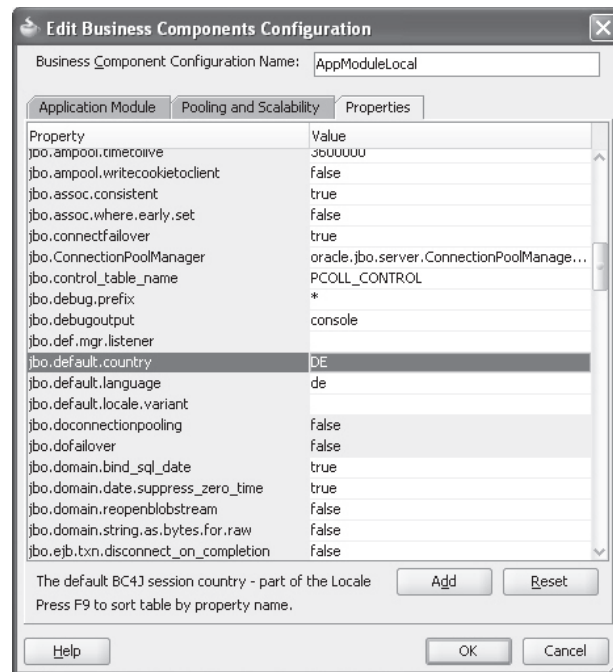


FIGURE 18-12 Specifying a default language for an application module

Notice that the `pattern` attribute of the converter components is set to the format property from the attribute's binding. This refers to the format specified for the attribute in the Control Hints dialog, typically defined in the entity object, and sometimes overridden by the attribute definition in the view object. When a format type and format are defined for a number or date attribute in ADF business components, formatters for these objects are added to the resource bundle, as explained in an earlier section. Decimal separators (commas versus periods) for number and currency types are automatically localized appropriately by the framework. However, date format masks are not automatically localized and should be modified for supported locales. For example, if the following entries are included in the `en_US` properties file, they may need to be localized appropriately for different locales:

```
HireDate_FMT_FORMAT=MM-dd-yyyy
CommissionPct_FMT_FORMAT=00.00
```

In the `de_DE` properties file, these formats could be modified to the following values that reflect the format style of European countries:

```
HireDate_FMT_FORMAT=dd.MM.yyyy
CommissionPct_FMT_FORMAT=00.00
```

Note that the `CommissionPct_FMT_FORMAT` value remains `00.00`. At runtime, the value will automatically display the decimal separator as a comma instead of a period for the `de_DE` locale.

Special Considerations When Localizing Currencies

If no country is defined in the supported locales, a default symbol (¤) will be used in place of an actual currency symbol. Be careful when defining locales with currencies. Consider the case where both `en_US` and `fr_FR` are defined as supported locales. If a value of 20.00 is stored in the database, the value will display as \$20.00 when the browser's language is set to `en_US`. When the browser language is set to `fr_FR`, the value will display as 20,00 €, which does provide the proper formatting for the number itself, but obviously doesn't represent the same value in monetary terms.

TIP

When considering localization of currencies, the country part of the locale should be specified in `faces-config.xml`.

ADF Faces Components Translation

Many ADF Faces components contain text that is embedded in the component itself. For example, the `af:panelCollection` component includes View and Detach menu items as part of the component. Conveniently, this text is automatically translated into the standard languages for end-user facing products: Arabic, Brazilian Portuguese, Czech, Danish, Dutch, English, Finnish, French, German, Greek, Hebrew, Hungarian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Romanian, Russian, Simplified Chinese, Slovak, Spanish, Swedish, Thai, Traditional Chinese, and Turkish. Figure 18-13 shows the runtime view of a page run in the German locale that contains automatically translated ADF Faces component elements.

Ansicht ▼		Zuordnung aufheben	
Abteilungs Nummer	Abteilungsname	Vorgesetzter	Standort
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping55	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance123	108	1700
110	Accounting	205	1700

FIGURE 18-13 Runtime view of page run in German locale

If the application needs to support a language other than those listed earlier, or if your application rules mandate that different text be used to warn the user of invalid entries due to the use of the converter and validator components of the ADF Faces library, you can globally override the default messages. To do so, locate the key value for the converter or validator component that requires an override in Appendix B of *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework 11g Release 1*. For example, to override the error messages displayed for an `af:validateLength` validator when the user exceeds the maximum length, use the following keys, defining custom messages as necessary:

```
org.apache.myfaces.trinidad.validator.LengthValidator.MAXIMUM=Error: There are too many characters.
org.apache.myfaces.trinidad.validator.LengthValidator.MAXIMUM_detail=Attention: Enter {2} or fewer characters, not more.
org.apache.myfaces.trinidad.validator.LengthValidator.MAXIMUM_HINT=Hint: Enter {0} or fewer characters.
```

These keys can be defined in the default properties file for the application, or they can be defined in a separate file. In either case, the file should be configured in the `message-bundle` element of the `faces-config.xml` file, as follows:

```
<message-bundle>view.ViewControllerBundle</message-bundle>
```

Figure 18-14 shows the runtime view of an input text field with an `af:validateLength` component where the key values above have been overridden.

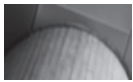


FIGURE 18-14 Runtime view of input text field

In addition to the built-in translation features of ADF Faces, the framework will also automatically set other locale-based features, such as bidirectional text rendering and currency symbols, based on the user's preferred locale. To alter the default functionality, modify the `trinidad-config.xml` file. There are several localization settings, including the following:

- `currency-code`
- `decimal-separator`
- `formatting-locale`
- `number-grouping-separator`
- `right-to-left`
- `time-zone`

Refer to *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework 11g Release 1* for further information regarding these settings.



TIP

Using custom skins, you can override default ADF Faces component labels. In this case, you need to provide the translations for these labels as well. Chapter 16 covers the use of resource bundles in custom skin definitions.

Changing Locales

Typically, locales are changed by modifying a browser's default language. Developers can easily test the functionality of a localized application by adding a language to the browser and running the application. For added flexibility, applications may support changing the locale programmatically, regardless of the browser settings. The remainder of this section describes how to implement this technique.

Changing Locales Programmatically

Instead of using browser settings to control the locale that is loaded for an application, you may sometimes find it necessary to change the locale based on the user's preferred language. For example, a drop-down list can be used to allow the user to choose from languages that the application supports and programmatically change the locale in this way. This is implemented by programmatically retrieving the list of supported languages for an application and then changing the locale of the application dynamically. These two aspects can be implemented separately if desired, but the most dynamic and reusable way of changing locales programmatically is described here.

Programmatically Retrieving Supported Locales To retrieve a list of supported locales, use the `getSupportedLocales()` method from the `FacesContext.getApplication()` method, as shown in the following managed bean code:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Locale;
```

```

import javax.faces.context.FacesContext;
import javax.faces.model.SelectItem;

public class ChangeLocale {

    List suppLocales;

    public void setSuppLocales(List suppLocales) {
        this.suppLocales = suppLocales;
    }
    public List getSuppLocales() {
        FacesContext fctx = FacesContext.getCurrentInstance();
        Iterator<Locale> localeIt = fctx.getApplication().getSupportedLocales();
        suppLocales = new ArrayList();
        while (localeIt.hasNext()) {
            Locale locale = localeIt.next();
            SelectItem item = new
                SelectItem(locale.getLanguage(), locale.getDisplayLanguage(locale));
            suppLocales.add(item);
        }
        return suppLocales;
    }
}

```

The list element in the JSF page is then defined as follows:

```

<af:selectOneChoice>
    <f:selectItems value="#{ChangeLocale.suppLocales}"/>
</af:selectOneChoice>

```

Changing a Locale Dynamically Once the supported locales from the application are determined, your next step is to change the locale of the application. This can be done via various command components, but in this example, a list is used and so the `valueChangeListener` attribute of the list can be used to retrieve the locale that the user has selected. The `selectOneChoice` component would be modified as follows:

```

<af:selectOneChoice valueChangeListener="#{ChangeLocale.listChanged}">
    <f:selectItems value="#{ChangeLocale.suppLocales}"/>
</af:selectOneChoice>

```

The `listChanged` method is implemented in the managed bean and calls a method named `changeLocale` that accepts the newly selected value and sets the locale for the application appropriately:

```

import javax.faces.event.ValueChangeEvent;
import java.util.Locale;
import javax.faces.context.FacesContext;
...
public void listChanged(ValueChangeEvent valueChangeEvent) {
    changeLocale(valueChangeEvent.getNewValue().toString());
}

```

```
private void changeLocale (String language){
    Locale newLocale = new Locale (language);
    FacesContext fctx = FacesContext.getCurrentInstance();
    fctx.getViewRoot().setLocale(newLocale);
}
```

The `changeLocale()` method sets the locale for the application, but this isn't applied to the running page at this point. What's required is an instance variable for the user's current locale. This is provided by creating accessors as well as a command component action that sets the value to the updated locale, as shown in this managed bean code:

```
Locale preferredLocale;
public void setPreferredLocale(Locale preferredLocale) {
    this.preferredLocale = preferredLocale;
}
public Locale getPreferredLocale() {
    return preferredLocale;
}
public String cb1_action() {
    setPreferredLocale
        (FacesContext.getCurrentInstance().getViewRoot().getLocale());
    return null;
}
```

Finally, after creating a command component on the page that has the action attribute set to the `cb1_action` method, a phase listener is registered with the application in the `adf-settings.xml` file. The phase listener intercepts the ADF lifecycle before the prepare model phase and applies the selected locale to the JSF `uiViewRoot` of the page:

```
import java.util.Locale;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;
import oracle.adf.controller.v2.lifecycle.ADFLifecycle;
import oracle.adf.controller.v2.lifecycle.PagePhaseEvent;
import oracle.adf.controller.v2.lifecycle.PagePhaseListener;

public class CustomPhaseListener implements PagePhaseListener {
    public CustomPhaseListener() {
        super();
    }
    public void afterPhase(PagePhaseEvent event) {
    }
    public void beforePhase(PagePhaseEvent event) {
        Integer phase = event.getPhaseId();
        if (phase.equals(ADFLifecycle.PREPARE_MODEL_ID)) {
            FacesContext fctx = FacesContext.getCurrentInstance();
            ChangeLocale changelocale = (ChangeLocale)fctx.getApplication()
                .evaluateExpressionGet(fctx, "#{ChangeLocale}", Object.class);
            Locale preferredLocale = changelocale.getPreferredLocale();
            UIViewRoot uiViewRoot =
                fctx.getCurrentInstance().getViewRoot();
```

```

    if (preferredLocale == null) {
        changelocale.setPreferredLocale(uiViewRoot.getLocale());
    } else {
        uiViewRoot.setLocale(preferredLocale);
    }
}
}
}

```

In this example, the `CustomPhaseListener` class retrieves the `preferredLocale` set via the command button action in the managed bean and applies that to the current page. The resulting runtime view is shown in Figure 18-15, where an `af:outputText` field that contains an internationalizable value is included in the page, as well as a form based on a localized business components data model. The ADF application lifecycle phases and the `adf-settings.xml` file are explained in detail in Chapter 3.

Localizing an Application Using the Database as a Resource Store

Applications with a database-centric architecture might store resource strings and translations in the database. For example, consider a database table `CATEGORIES` with the following values:

CATEGORY_ID	CATEGORY_NAME	LANGUAGE
1	Books	EN
2	Movies	EN
3	Magazines	EN
1	Buecher	DE
2	Filme	DE
3	Magazine	DE

NOTE

To support non-Latin-based character sets using the database, additional database configuration may be required.

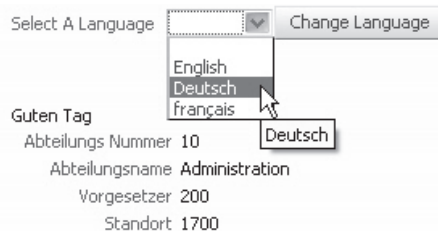


FIGURE 18-15 Runtime view of setting the Locale programmatically

To localize an ADF application to determine which set of these rows to use at runtime, a dynamic database parameter can be set that stores the preferred language for the current session. For example, the following procedure stores the value of a dynamic parameter (named `user_lang`) to the built-in database procedure `dbms_application_info.set_client_info`:

```
CREATE OR REPLACE
PACKAGE BODY USER_CONTEXT_PKG IS
    PROCEDURE set_app_user_lang ( user_lang IN VARCHAR2) IS
    BEGIN
        dbms_application_info.set_client_info(user_lang);
        EXCEPTION WHEN OTHERS THEN raise_application_error (-20001, 'Error in
            user_context_pkg.set_app_user_lang: ' || SQLERRM);
    END;
END USER_CONTEXT_PKG;
```

The procedure is called by a custom application module class that extends the `ApplicationModuleImpl` class and overrides the `prepareSession` method. The following code demonstrates how to determine the default language for the application using the `jbo.default.language` property of the application configuration. Additionally, the code calls the `set_app_user_lang` procedure using the default language as the argument.

```
import java.sql.CallableStatement;
import java.sql.SQLException;
import oracle.jbo.JboException;
import oracle.jbo.common.PropertyMetadata;
import oracle.jbo.Session;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.DBTransactionImpl;

public class CustomApplicationModuleImpl extends ApplicationModuleImpl {
    @Override
    protected void prepareSession(Session session) {
        super.prepareSession(session);
        setCurrentUserLanguage();
    }

    public void setCurrentUserLanguage(){
        DBTransactionImpl dbti = (DBTransactionImpl)getDBTransaction();
        CallableStatement statement = dbti.createCallableStatement(("BEGIN " +
            "user_context_pkg.set_app_user_lang(?); " + "END;"), 0);
        try {
            statement.setString(1, getApplicationLanguage());
            statement.execute();
        } catch (SQLException sqlerr) {
            throw new JboException(sqlerr);
        } finally {
        }
        try {
            if (statement != null) {
                statement.close();
            }
        } catch (SQLException closeerr) {
```

```

        throw new JboException(closeerr);
    }
}
}
public String getApplicationLanguage(){
    String appLanguage = "EN";
    appLanguage = getAMLanguage();
    return appLanguage;
}
public String getAMLanguage(){
    PropertyMetadata langProperty =
        PropertyMetadata.findProperty("jbo.default.language");
    String amLanguage = langProperty.getProperty();
    return amLanguage.toUpperCase();
}
}
}

```

View Objects that require filtering based on the user's preferred language should include a where clause to filter the query by the database parameter that has been set. This is stored in the database as 'CLIENT_INFO' in the built-in database USERENV namespace, which describes the current session.

```

SELECT CategoriesEntity.CATEGORY_ID, CategoriesEntity.CATEGORY_NAME,
       CategoriesEntity.LANGUAGE, CategoriesEntity.ROWID
FROM CATEGORIES CategoriesEntity
WHERE CategoriesEntity.LANGUAGE = USERENV('CLIENT_INFO')

```

When running the application module with this code in place, the default language is determined by the application module configuration, and the View Object will return only those rows that have a LANGUAGE value set to that value. However, this does not handle the case when the language should be determined by the web browser, and not by the application module's `jbo.default.language` property (to which users might not have access). To provide this functionality, the application module class needs to include the ability to set the language for the current user based on the browser's default language. For example, the following modified `getApplicationLanguage()` method determines whether the user is accessing the application via the Web, and if so, it attempts to match the user's preferred language to the supported languages of the application:

```

public class CustomApplicationModuleImpl extends ApplicationModuleImpl {
    public static String preferredLanguage;
    public static boolean isWebUser=false;
    private String[] supportedLanguages = {"EN","DE"};

    public String getApplicationLanguage(){
        String appLanguage = "EN";
        if (isWebUser){
            for (int index=0; index<supportedLanguages.length; index++){
                if (preferredLanguage.equals(supportedLanguages[index])){
                    appLanguage = preferredLanguage;
                    break;
                }
            }
        }
    }
}

```

```

    } else{
        appLanguage = getAMLanguage();
    }
    return appLanguage;
}
}

```

The `isWebUser` and `preferredLanguage` variables in the code can be set using a phase listener that implements the `beforePhase()` method to augment the default functionality when the model is prepared for delivery. The following code in the phase listener retrieves the locale from the `FacesContext` object (which is introspected from the browser) and sets that value to the `preferredLanguage` variable of the custom application module class:

```

import oracle.adf.controller.v2.lifecycle.ADFLifecycle;
import javax.faces.context.FacesContext;
import oracle.adf.controller.v2.lifecycle.PagePhaseEvent;
import oracle.adf.controller.v2.lifecycle.PagePhaseListener;
import model.CustomApplicationModuleImpl;

public class CustomPhaseListener implements PagePhaseListener {
    public void afterPhase(PagePhaseEvent pagePhaseEvent) {
    }
    public void beforePhase(PagePhaseEvent pagePhaseEvent) {
        Integer phase = pagePhaseEvent.getPhaseId();
        if (phase.equals(ADFLifecycle.PREPARE_MODEL_ID)) {
            FacesContext fctx = FacesContext.getCurrentInstance();
            String language =
                fctx.getExternalContext().getRequestLocale()
                    .getLanguage().toUpperCase();
            CustomApplicationModuleImpl.preferredLanguage = language;
            CustomApplicationModuleImpl.isWebUser = true;
        }
    }
}

```

Providing Help Topics

ADF Faces lets you define instructional information for components in a central file, rather than hard-coding the information within the component. This allows developers to specify a pointer to help information for components, and the source of the information can be dynamically defined, localized, and reused across components as necessary. For example, an application may have many input components that are duplicated across pages but have the same function in the application. An e-mail field is a good example of this. If the login page as well as several other pages in an application contain an input text field for entering an e-mail address, helpful information for the correct format and example values could be stored in a centralized help bundle containing help text for the entire application. The various input text fields can refer to the help text dynamically, so that if a change in the help text is required, they need to be applied only in the help bundle and not in every page containing the e-mail fields.

Help text can be defined for two types of information: `DEFINITION` and `INSTRUCTIONS`. The runtime view of the help text “Enter the employee’s user id without a domain.” defined for the instructional help type is shown in Figure 18-16.

A screenshot of a web form for creating an employee. The form contains the following fields: EmployeeId (with an asterisk), FirstName, LastName (with an asterisk), Email (with an asterisk), PhoneNumber, HireDate (with an asterisk and a calendar icon), JobId (with an asterisk), Base Salary, ManagerId, and DepartmentId. At the bottom are 'Create' and 'Submit' buttons. A callout box points to the Email field with the text: "Enter the employee's user id without a domain."

FIGURE 18-16 Runtime view of help text

Notice that the help text appears at runtime as a note window when the mouse hovers over the component to which the help topic is attached. This is because the instructional help type was used, and it was used with an input component (`af:inputText`). For header components, the instructional text appears beneath the header text. For definition help types, a help icon is displayed instead of the help text, and hovering over the icon displays the definition help text. The help icon appears before the label of input components, at the end of header components, next to the close icon in window and dialog components, and below table headers in table components. The runtime view of the definition help text type for an input component is shown in Figure 18-17.

The help text can be stored in resource property files, XLIFF files, managed beans, or class files. No matter which type of file is used for the help text storage, the application must configure a help provider in the `adf-settings.xml` file and define a prefix to be used to access the provider. For example, the following code specifies a help provider to be used across the application. The prefix for the help provider is `APP_HELP_`, the provider class is the built-in

A screenshot of a web form for creating an employee, similar to Figure 18-16. The fields are filled with the following values: EmployeeId (145), FirstName (John), LastName (Russell), Email (JRUSSEL), PhoneNumber (011.44.1344.429268), JobId (SA_MAN), Base Salary (\$14,000.00), ManagerId (100), and DepartmentId (80). A help icon is visible next to the Email field label. A callout box points to the Email field with the text: "The employee's email."

FIGURE 18-17 Runtime view of the definition help text type

ResourceBundleHelpProvider for resource property file storage, and the property for the help provider is defined with a property name and value that refer to the storage file.

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
  <adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/settings">
    <help-provider prefix="APP_HELP_">
      <help-provider-class>
        oracle.adf.view.rich.help.ResourceBundleHelpProvider
      </help-provider-class>
      <property>
        <property-name>baseName</property-name>
        <value>view.ViewControllerBundle</value>
      </property>
    </help-provider>
  </adf-faces-config>
</adf-settings>
```

This example contains values unique to the resource properties file method of storing help text. The following tables describe what the values should be for the other types of storage files.

Storage File	help-provider-class Value
Resource property file	oracle.adf.view.rich.help.ResourceBundleHelpProvider
XLIFF	oracle.adf.view.rich.help.ELHelpProvider
Managed bean	Fully qualified class path to the managed bean—such as view.MyHelpBean
Class file	Fully qualified class path to the class—such as view.MyHelpProvider

Storage File	Help Provider Property <value>
Resource property file	Fully qualified name of the property file—such as view.ViewControllerBundle.
XLIFF	EL value for the XLIFF file—such as #{adfBundle['view .ViewControllerBundle']}. Note that the property-name for XLIFF-based help text storage must be helpSource.
Managed bean	EL value for the helpMap property of the managed bean. For example, if the managed bean is defined as <pre><managed-bean> <managed-bean-name> helpMapBean </managed-bean-name> <managed-bean-class> view.MyHelpBean </managed-bean-class> <managed-bean-scope> session </managed-bean-scope> </managed-bean></pre> the property value would be #{helpMapBean.helpMap}.
Class file	Custom value for the custom property.

NOTE

You may need to create the `adf-settings.xml` file if it does not already exist in your application. To create the file, choose *File | New in JDeveloper* to create a new XML file, and save the file in the `src\META-INF` directory of the UI project. You may also need to create the `META-INF` directory if it does not already exist; however, any project that contains ADF data bindings will already include a `META-INF` directory that contains the `adfm.xml` file.

Specifying Help Text

Configuring help text is similar to specifying resource strings for an application. A key/value pair is entered into the help storage file, where the key contains the help provider prefix, a custom topic (such as a descriptive name of the field to which the text should be applied), and the appropriate help type (`INSTRUCTIONS` or `DEFINITION`). The generally accepted naming conventions include the use of underscores to separate the key elements. For example, the following key/value pairs are defined in a resource properties file (`ViewControllerBundle.properties`) and contain the help provider prefix `APP_HELP_`, the topic `EMPLOYEE_EMAIL`, and the help type:

```
APP_HELP_EMPLOYEE_EMAIL_INSTRUCTIONS=Enter the employee's user id without a domain.
APP_HELP_EMPLOYEE_EMAIL_DEFINITION= The employee's email.
```

These same values can be configured in an XLIFF (.xlf) file, as shown next:

```
<?xml version="1.0" encoding="windows-1252" ?>
<xliff version="1.1" xmlns="urn:oasis:names:tc:xliff:document:1.1">
  <file source-language="en" original="view.ViewControllerBundle"
    datatype="xml">
    <body>
      <trans-unit id="APP_HELP_EMPLOYEE_EMAIL_INSTRUCTIONS">
        <source>Enter the employee's user id without a domain.</source>
        <target/>
      </trans-unit>
      <trans-unit id="APP_HELP_EMPLOYEE_EMAIL_DEFINITION">
        <source>The employee's email.</source>
        <target/>
      </trans-unit>
    </body>
  </file>
</xliff>
```

NOTE

Definition text may not include any HTML formatting and is typically less than 100 characters or so. Instructional text may be longer, as it is displayed in a note component, and it may include simple HTML formatting.

Configuring Components for Use with Help Topics

Regardless of the storage mechanism used for help text, components are configured to display help topics by setting the `helpTopicId` attribute. The attribute should be set to the value of the help provider prefix and the help topic only. All available help types defined for the help topic are determined by the help provider and displayed at runtime as necessary. For example, the `af:inputText` component for the e-mail field would be defined as follows:

```
<af:inputText value="#{bindings.Email.inputValue}"
  label="#{vcb.EMAIL}"
  helpTopicId="APP_HELP_EMPLOYEE_EMAIL"/>
```

Note that this is an `af:inputText` field, which directly supports help topics by providing the `helpTopicId` attribute.

Adding Help Text for Other Components

You can display help text for a component that does not directly support help topics via the `helpTopicId` attribute. You might bind the `shortDesc` attribute of a component to a help topic, create an output text component and bind the value to the help topic, or create a command component to launch the help topic external URL (external URLs are covered in the next section). The following code shows each of these examples in practice, using EL to retrieve the help provider from the ADF Faces context:

```
<af:menuBar>
  <af:menu text="Actions">
    <af:commandMenuItem text="Edit"
shortDesc="#{adfFacesContext.helpProvider['APP_HELP_ACTIONS_MENU_EDIT']
  .definition}"/>
    <af:goImageLink text="Help for this Menu"
      destination="#{adfFacesContext.helpProvider['APP_HELP_ACTIONS_MENU']
        .externalUrl}"/>
  </af:menu>
  <af:outputFormatted
    value="#{adfFacesContext.helpProvider['APP_HELP_ACTIONS_MENU']
      .instructions}"/>
</af:menuBar>
```

Providing Help Using Web Documents

In addition to the definition and instruction types of help, a help provider may include an external URL help type. This type of help topic renders a help icon as is done for the definition help type, but the help icon will launch an external URL when clicked. Since resource bundles should not include URLs, the `ResourceBundleHelpProvider` does not provide an implementation for returning external URL values. Therefore, to create a help provider that uses external URL help types, create a class that extends `oracle.adf.view.rich.help.ResourceBundleHelpProvider` and overrides the `getExternalUrl` method, as shown here:

```
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import oracle.adf.view.rich.help.ResourceBundleHelpProvider;
```

```

public class MyHelpProvider extends ResourceBundleHelpProvider{
    public MyHelpProvider() {
    }
    private static String APP_HELP_EMPLOYEE_EMAIL_URL =
        "http://<site>";
    private static String APP_HELP_DEPT_URL = "http://<anothersite>";
    private static String DEFAULT_URL = "http://www.oracle.com";

    @Override
    protected String getExternalUrl(FacesContext context,
        UIComponent component, String topicId) {
        if (topicId == null)
            return null;
        if (topicId.contains("APP_HELP_EMPLOYEE_EMAIL") )
            return APP_HELP_EMPLOYEE_EMAIL_URL;
        if (topicId.contains("APP_HELP_DEPT")){
            return APP_HELP_DEPT_URL;
        } else {
            return DEFAULT_URL;
        }
    }
}

```

Using Oracle Help for the Web (OHW)

Oracle Help for the Web (OHW) is an application that provides comprehensive online help systems via HTML files. Documentation writers create the help topics using an authoring tool, and the application is deployed as an Enterprise Java application (EAR). For more information on how to author and deploy an OHW application, refer to *Oracle Fusion Middleware Developer's Guide for Oracle Help 11g*.

OHW can be integrated with an ADF Faces application by deploying the OHW application and then referring to the front controller servlet of the OHW application in the `adf-settings.xml` file of the ADF Faces application. The `adf-settings.xml` file should be configured as follows to use OHW as the help provider:

```

<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
    <help-provider>
        <help-provider-class>
            oracle.help.web.rich.helpProvider.OHWHelpProvider
        </help-provider-class>
        <property>
            <property-name>ohwConfigFileURL</property-name>
            <value>/helpsets/ohwconfig.xml</value>
        </property>
        <property>
            <property-name>baseURI</property-name>
            <value>
                http://localhost:7100/help-ohw-rcf-context-root/ohguide/
            </value>
        </property>
    </help-provider>
</adf-settings>

```

This example contains two properties within the `OHWHelpProvider` definition. The first is `ohwConfigFileURL`. The value for this property should be set to the location of the `ohwconfig.xml` file that is located in the `public_html` directory of the accessing ADF Faces application. Thus, the entire contents of the helpsets (including the `ohwconfig.xml` file) should be copied into the `public_html` folder of the application that will use OHW. In this case, the files were copied into a new folder named `helpsets`. The second property, `baseURI`, should be set to the location of the OHW servlet that is deployed with the OHW application to launch the appropriate page.

Finally, to use OHW topics as the help for a component, specify the topic ID defined in the map of the OHW application as the value of the `helpTopicId` attribute for components. The help will appear as for the external URL type of help, where a user can click a help icon for a component to launch the help file.

Summary

This chapter covered two significant aspects of Web application development that enable end users to interact with an application effectively: providing instructional information and providing the ability to run an application in the end user's native language. Think of your users first! For whom are you developing? What languages do they speak? Will application training be provided, or is this a self-service application for which proper documentation must be shipped along with the product? Answering these questions early on in development will greatly affect application development policies, standards, and procedures. Failing to recognize the importance of these aspects can cause headaches later on. For example, retrofitting an application to be internationalizable is a difficult and time-consuming task. For this reason, it is recommended that every application use resource bundles, regardless of the need for multiple language support.

