Best Practices for Globalization using the Oracle 9*i* Internet Application Server

Author:	Simon Wong
Last Updated:	July 12, 2001
Version:	1.0

Contents

Introduction

Globalizing Internet applications to support many languages is getting more attention as users around the world access applications through the Internet. The Oracle9*i* Internet Application Server (Oracle9*i*AS) is fully internationalized to provide a global platform for users to develop and deploy Internet applications supporting languages of their choice.

By supporting a national language, an Internet application not only presents HTML pages that are constructed with a proper character set and content specific to the language, but also formats them using cultural conventions, such as the date format and currency symbol, of the region in which the language is spoken. The language and the region in which the language is spoken are collectively known as the locale.

Building an Internet application or a web site that supports different locales for Oracle9*i*AS requires good practices in programming and deployment. This paper describes some of the best globalization practices based on your requirements:

- Deploying a web site or Internet application that supports multiple locales
- Configuring a database to store data from around the world
- Configuring an application server to minimize character set conversion and eliminate data loss from conversion
- Handling non-English HTML output and user input reliably based on industry Internet standards
- Making your application extensible to support additional locales
- Accessing a database with data in multiple languages
- Formatting your HTML content according to the language and culture conventions of a user
- Making your Internet application translatable
- Manipulating translated content for your application

Concepts

To better understand the materials presented in this paper, you should understand the following terms:

- A *locale* refers to a national language and the region in which the language is spoken. From a programming point of view, a locale mainly consist of two pieces of information:
  1. *Language* --- The language of a locale refers to the national language of a user.
  2. *Territory* --- The territory defines the region in which the language is spoken. The territory of a locale usually determines the cultural conventions such as date format and number format. Sometimes, it also implies the dialect of the language.

  Some locale models also associate a character set to a locale. In this document, the ISO locale name is used to refer a locale of a user unless explicitly specified. For example, fr-CA is the ISO locale name for French (Canada).
- A *character set* is sometimes called a character encoding, a code page or simply an encoding. It defines how characters are mapped to their binary values. The character set of a locale defines how the character data of the language should be encoded. For example, the ISO-8859-1 character set can be used to encode most Western European languages. In this document, the Internet Assigned Numbers Authority (IANA) character set names are used to refer to a character set unless explicitly specified.
- *Unicode* is a universal character set that defines characters in almost all languages around the world. Unicode characters can be encoded in 1 to 4 bytes as UTF-8, 2 or 4 bytes as UTF-16 or 4 bytes as UTF-32. For more information on the Unicode standard, please refer to the Unicode 3.0 book published by the Unicode Consortium.

Design and Configuration

The following areas need to be considered when developing and deploying a global Internet application:

- Internet application models
- Global database configuration
- Application server configuration

Multilingual versus Monolingual Models

There are two alternatives to deploy a global web site or a global Internet application on Oracle9*i*AS depending on the globalization requirements. Which alternative to use affects how the Internet application is built and how Oracle9iAS is configured in the middle-tier server. The two alternatives are:

· Multiple instances of monolingual Internet applications

Internet applications that support one single locale at a time in a single binary are classified as monolingual applications. This level of globalization support is suitable for customers who want to support one locale per instance of the application. Users need to have different entry points to access the applications for different locales.

· Single instance of a multilingual application

Internet applications that support multiple locales simultaneously in a single binary are classified as multilingual applications. This level of globalization support is suitable for customers who want to support a number of locales in an Internet application simultaneously. Users of different locale preferences use the same entry point to access the application.

The ways in which an Internet application is developed and Oracle9*i*AS is configured are very different between a multilingual model and a monolingual model. This document use the following symbols to identify content specific to monolingual applications and multilingual applications respectively.

- Mono
- Multi

Also, to simplify the description on each model and deployment alternative, the document assumes that each middle tier application server runs a single instance of Oracle9*i*AS.

Mono **Monolingual Internet Application Deployment**

Deploying a global web site with multiple instances of monolingual Internet applications is depicted in Figure 1: Multiple Monolingual Internet Applications Deployment. Oracle9*i*AS is configured in a middle tier server for the locale that it serves. This deployment alternative assumes that one instance of an Internet application runs in the same locale as Oracle9*i*AS in the middle-tier server. Due to the complexity and limitations, Oracle does not recommend deploying multiple instances, each instance serves a different locale, of the monolingual application on a single instance of Oracle9iAS.

The Internet applications access a back-end database in the native encoding used for the locale.

The major advantages of deploying monolingual Internet applications are that:

- You separate the support of different locales into different servers so that different locales can be supported independently in different time zones and work load can be distributed accordingly. For example, customers may want to support Western European locales first and then support Asian locales with multiple character sets such as Japanese (Japan) later.
- You avoid the code complexity of supporting multiple locales simultaneously. The amount of code you have to change is significantly less than a multilingual Internet application.

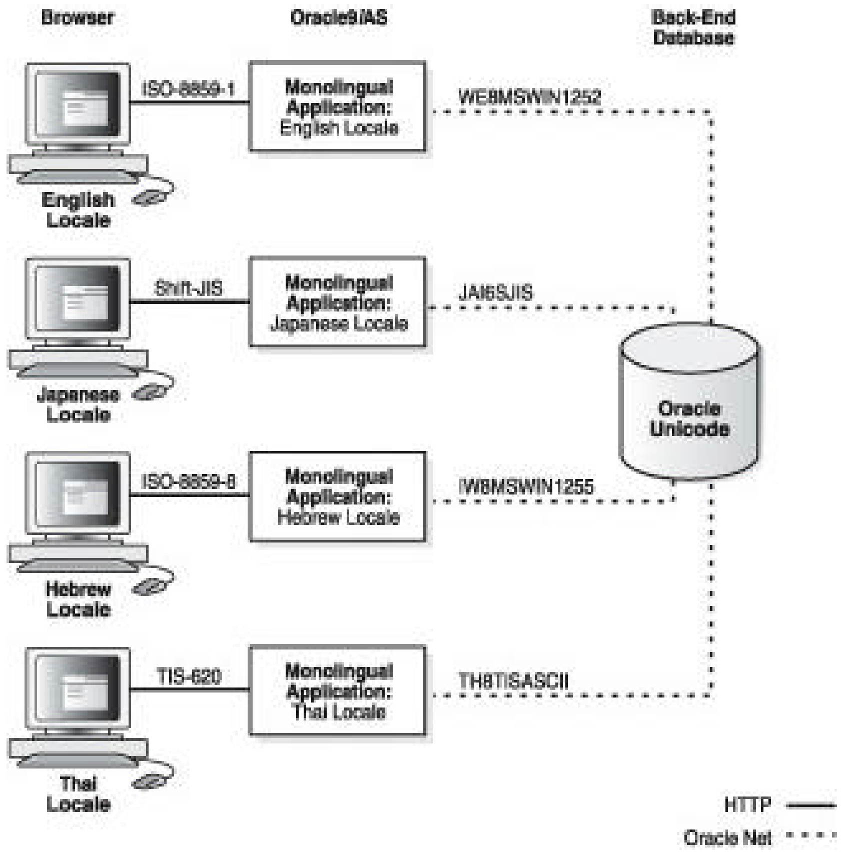


Figure 1: Multiple Monolingual Internet Applications Deployment

The major disadvantages of deploying monolingual Internet applications are that:

- More work is required to maintain and manage multiple servers for different locales. Different Oracle9*i*AS configurations are required for different middle-tier servers.
- The minimum number of middle-tier servers required is tied to the number of locales your web site supports regardless of whether the site traffic will reach the capacity provided by the middle-tier servers.
- Load balancing for middle-tier servers is limited to the group of middle-tier servers for the same locale.
- More testing resources, both human and machine, are required for multiple configurations of middle-tier servers. Internet applications running on different locales must be certified on the corresponding middle-tier server configuration.
- It is not designed to support multilingual content. For example, a web page containing Japanese and Arabic data cannot be easily supported in this alternative.

As more and more locales are supported, the disadvantages of this alternative outweigh the advantages. This alternative is more appropriate for customers who only need to support one or two locales in their web sites.

Multi

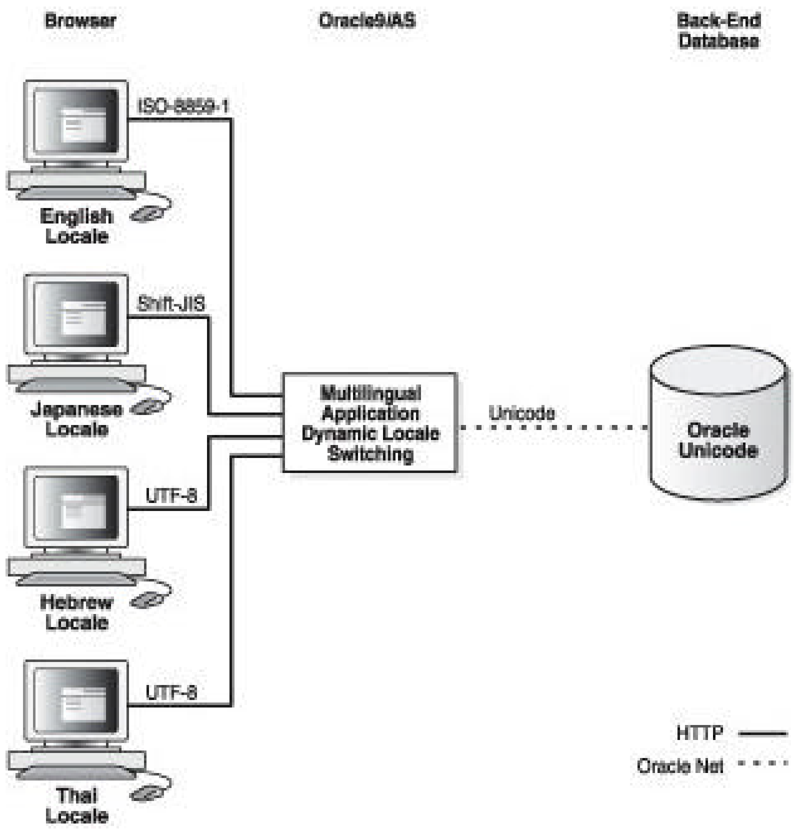
Multilingual Internet Application Deployment

Multilingual Internet applications are deployed to middle-tier servers with a single consistent Oracle9*i*AS configuration that works for all locales. Oracle9*i*AS is configured to support multiple locales simultaneously. Figure 2: Multilingual Internet Application Deployment depicts the configuration of a multilingual Internet application deployment.

In order to support multiple locales in a single application instance, an application may need to:

- Dynamically detect the locale of the users and adapt to the locale by constructing HTML pages in the language and cultural conventions of the locale.
- Process character data in Unicode so that data in any language can be supported. Character data can be entered by users or retrieved from back-end databases.
- Dynamically determine the HTML page encoding (or character set) to be used for HTML pages and convert content from Unicode to the page encoding and vice versa.

Figure 2: Multilingual Internet Application Deployment



The major advantages of deploying multilingual Internet application are that:

- Using a single Oracle9*i*AS configuration for all middle-tier servers simplifies the deployment configuration and hence reduces the cost of maintenance.
- Performance tuning and capacity planning do not depend on the number of locales supported by the web site.
- Supporting additional languages is relatively easy. You do not need to add more machines for the new locales.
- Testing the application for multiple locales can be done in a single testing environment.
- It is designed to support multilingual content with data in multiple languages.

The major disadvantage of deploying multilingual Internet applications is that it requires extra coding work up front to handle dynamic locale detection and Unicode handling, which is costly when only a couple languages need to be supported.

Deploying multilingual Internet applications is more appropriate than deploying monolingual applications when web sites support many different locales.

Configuring Centralized Back End Database for Multilingual Support

Configuring an Oracle9*i* database is an important part of deploying a global Internet application for Oracle9*i*AS. Instead of storing regional data into many distributed databases, Oracle recommends that you store all your data, regardless of language, in a single centralized database. There are many advantages of using a centralized database over using multiple distributed databases. For example:

- A centralized database provides a complete view of your data so that you can easily query information such as the number of customers worldwide or the inventory level of a product worldwide.
- It is much easier to professionally manage a centralized database than many distributed databases.

An Oracle9*i* database enables you to store and manipulate data in multiple locales in the form of Unicode. Unicode is a universal character set that defines characters in almost all languages around the world. Unicode data can be stored in Oracle9*i* databases in the following two encoding forms:

- UTF-8 - each character spans 1 to 4 bytes
- UTF-16 - each character is either 2 bytes or 4 bytes long

You can set up your Oracle9*i* database to store Unicode data in the following ways:

1. as UTF-8 in the SQL CHAR data types (CHAR, VARCHAR2 and CLOB)
2. as UTF-16 in the SQL NCHAR data types (NCHAR, NVARCHAR2 and NCLOB)
3. as UTF-8 in the SQL NCHAR data types

For more details about how an Oracle9*i* database should be configured to support Unicode and which of the above alternatives should be used for Unicode data storage, please refer to chapter 5 of the Oracle9*i* Globalization Support Guide.

A good practice is to configure your database to support both the first and second alternatives because they fulfill most of the requirements for Unicode support. In order to store and manipulate data in the UTF-8 encoding with the SQL CHAR datatypes and the UTF-16 encoding with the SQL NCHAR datatypes within the same database, the database character set and national character set have to be properly configured. You should specify UTF8 for the database character set and AL16UTF16 for the national character set at database creation time. For example, the following command creates an Unicode database in which the SQL NCHAR datatypes use the UTF-16 encoding form and SQL CHAR datatypes use the UTF-8 encoding form:

```
CREATE DATABASE myunicodedatabase
    CONTROL FILE REUSE
    LOGFILE '/u01/oracle/utfdb/redo01.log' SIZE 1M REUSE
            '/u01/oracle/utfdb/redo02.log' SIZE 1M REUSE
    DATAFILE '/u01/oracle/utfdb/system01.dbf' SIZE 10M REUSE
        AUTOEXTENT ON
        NEXT 10M MAXSIZE 200M
    CHARACTER SET UTF8
    NATIONAL CHARACTER SET AL16UTF16
```

If you want to store UTF-8 Unicode data in SQL NCHAR datatypes, use UTF8 as the national character set in the CREATE DATABASE command.

Configuring Oracle HTTP Server Powered by Apache

At the minimum, there are two configurations that need to be considered for Oracle HTTP Server Powered by Apache to support globalization:

- The NLS_LANG environment variable

  The NLS_LANG environment variable controls the character set, language and territory (or region) information used for database connections in an Internet application. The value of NLS_LANG is written in the following format:

  ```
  <language>_<territory>.<character set>
  ```

  The <`language`>, <`territory`> and <`character set`> values refer to valid Oracle language names, Oracle territory names, and Oracle character set names, respectively. Using the Oracle9*i* data conversion facility, data retrieved from and inserted into the database is automatically converted to and from the character set specified in NLS_LANG. The language and territory specified in NLS_LANG are used to initialize the locale that governs the default date time format, number format and sorting sequence used in a database session.

- Default locale of the application runtime environment

  The default locale of a runtime environment for an Internet application usually determines the default date-time format, number format, and national languages used in the application. Different runtime environments have different ways of setting this runtime default locale. Among others, Oracle HTTP Server Powered by Apache supports PL/SQL, Java, Perl and C/C++ runtime environments.

Configuring NLS_LANG

In order to properly configure NLS_LANG, you need to know:

- where the NLS_LANG-related settings should be specified
- what value of NLS_LANG should be specified

First, the NLS_LANG parameter should be specified in the following Apache configuration files or locations:

- In `<ORACLE_HOME>/Apache/Jserv/etc/jserv.properties`, add the following line.

  ```
  wrapper.env=NLS_LANG=<NLS_LANG value>
  ```

- In `<ORACLE_HOME>/Apache/Apache/conf/oracle_apache.conf`, add the following line.

  ```
  PassEnv NLS_LANG
  ```

- For UNIX platforms, add the following line to `<ORACLE_HOME>/Apache/Apache/bin/apachectl`

  ```
  NLS_LANG=${NLS_LANG=<NLS_LANG value>}; export NLS_LANG
  ```

- For Windows platforms, the Apache server is usually started as a Windows service, and the NLS_LANG value is specified in the NLS_LANG registry key in the following location.

  ```
  HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\HOME<ORACLE_HOME number>
  ```

Second, the value of NLS_LANG should be set differently depending on whether this is a monolingual application deployment or a multilingual application deployment.

Mono

- For a monolingual Internet application deployment, the NLS_LANG environment variable should specify the language, territory and character set corresponding to the locale which a middle-tier server is configured to serve. Assuming that most clients are running on Windows platforms, it is a good practice to use the NLS_LANG character set corresponding to the Windows code page of the locale. For example, when the middle-tier server is configured to serve Japanese clients, the NLS_LANG value should be JAPANESE_JAPAN.JA16SJIS where JA16SJIS corresponds to code page 932 of the Japanese Windows operating system. Table 1 lists the NLS_LANG values for the most commonly used locales.

Table 1: Locale and NLS_LANG Value Mappings

Locale	NLS_LANG
Arabic (U.A.E.)	ARABIC_ UNITED ARAB EMIRATES.AR8MSWIN1256
Germany (German)	GERMANY_GERMAN.WE8MSWIN1252
English (U.S.A)	AMERICAN_AMERICA. WE8MSWIN1252
English (United Kingdom)	ENGLISH_UNITED KINGDOM. WE8MSWIN1252
Greek	GREEK_GREECE.EL8MSWIN1253
Spanish (Spain)	SPANISH_SPAIN.WE8MSWIN1252
French (France)	FRENCH_FRANCE.WE8MSWIN1252
French (Canada)	CANADIAN FRENCH_CANADA.WE8MSWIN1252
Hebrew	HEBREW_ISRAEL.IW8MSWIN1255
Italian (Italy)	ITALIAN_ITALY.WE8MSWIN1252
Japanese	JAPANESE_JAPAN.JA16SJIS
Korean	KOREAN_KOREA.KO16MSWIN949
Portuguese (Portugal)	PORTUGUESE_PORTUGAL.WE8MSWIN1252
Portuguese (Brazil)	BRAZILIAN PORTUGUESE_BRAZIL.WE8MSWIN1252
Turkish	TURKISH_TURKEY.TR8MSWIN1254
Thai	THAI_THAILAND.TH8TISASCII
Chinese (Taiwan)	TRADITIONAL CHINESE_TAIWAN.ZHT16MSWIN950
Chinese (P.R.C)	SIMPLIFIED CHINESE_CHINA.ZHS16GBK

Multi

- For a multilingual Internet application deployment, the language and territory parts of the NLS_LANG parameter are not as important as they are in a monolingual application deployment. This is because a multilingual application needs to handle different locales dynamically at run time and cannot rely on fixed settings defined by NLS_LANG. The NLS_LANG character set should always be set to UTF8 so that Unicode data are retrieved from and inserted into the database. An example of the NLS_LANG value for a multilingual application deployment is:

```
NLS_LANG=AMERICAN_AMERICA.UTF8
```

Configuring Application Runtime for Different Locales

Mono

The configuration for application runtime default locale is required only for monolingual application deployments, and the steps used to configure it varies depend on the runtime environments and operating systems. The following describes how the runtime default locale can be initialized for the main runtime environments supported by Oracle9*i*AS:

- Apache mod_jserv runtime for Java

  For UNIX, the LANG or LC_ALL environment variable not only defines the POSIX (also known as XPG4) locale used for a process but also governs how Java VM initialize its default locale. To configure the Java VM for JServ, you need to define the LANG or LC_ALL environment variable with a corresponding POSIX locale name in `jserv.properties`. For example, the following line in `jserv.properties` defines Japanese (Japan) to be the default locale of Java VM for Jserv on Solaris.

```
wrapper.env=LANG=ja_JP
```

  Note that the values for the LANG and LC_ALL environment variables should refer to the same POSIX locale available in your operating system, and the LC_ALL environment variable always overrides the LANG environment variable if they are different. For Windows platforms, the default locale of the Java VM for JServ is controlled by the regional settings of the Control Panel. You should change the regional settings to the desired locale from the Control Panel before starting Apache.

- Apache mod_plsql runtime for PL/SQL and PSP

PL/SQL and PSP run on an Oracle9*i* database in the context of a database session. Therefore the runtime default locale is controlled by the NLS_LANG parameter which should have been configured as described in Table 1: Locale and NLS_LANG Value Mappings.

- Apache mod_perl runtime for Perl script

  Perl scripts run on the Perl interpreter provided by the Apache mod_perl module. The locale support in Perl is based on the POSIX locale system available in the operating system, and use the underlying POSIX C libraries as the foundation. To configure the Perl runtime default locale, you can follow the same procedure described for the C/C++ runtime. For more information on how Perl scripts use POSIX locales, see the Perl doc pages:

  ```
  <ORACLE_HOME>/Apache/Perl/<Perl Version>/bin/perldoc perllocale
  ```

- C/C++ runtime

  The C/C++ runtime makes use of the POSIX locale system provided by an operating system. The locale system can be configured via the LC_ALL (or LANG) environment variable. You need to define LC_ALL with a valid locale value provided by the operating system. Unfortunately, these locale values are different on different operating systems. Table 3 provides a list of commonly used POSIX locale for Solaris.

  For UNIX platforms, you need to define LC_ALL as follows:

  - In `<ORACLE_HOME>/Apache/Apache/conf/oracle_apache.conf`, add the following line.

    ```
    PassEnv LC_ALL
    ```

  - For UNIX platforms, add the following line to `<ORACLE_HOME>/Apache/Apache/bin/apachectl`

    ```
    NLS_LANG=${LC_ALL=<OS locale>}; export LC_ALL
    ```

  For Windows platforms, the POSIX locale value should be inherited from the regional settings of the Control Panel instead of specifying it in the LC_ALL environment variable. Changing the regional settings will effectively change the default runtime POSIX locale.

Configuring Front-end Database for Running PL/SQL

If your front-end applications are written in PL/SQL, you may want to:

- Create a database in a middle-tier server for running the PL/SQL front-end programs to reduce the load of a back-end database
- Use database links to seamlessly access data from the back end database.

This front-end database should be installed on the machine where Oracle9*i*AS is installed.

Mono For monolingual application deployments, front-end databases should be configured so that the database character set is the same as the NLS_LANG character set as described in Table 1. Also, the national character set should be AL16UTF16.

Multi For multilingual application deployments, databases in the middle-tier servers should be configured with the same database character set and national character set as those of the back-end database. Using the same character sets for a front-end database and a back-end database avoids implicit character set conversion between them.

A database access descriptor (DAD) describes the connect string and Oracle parameters of a target database, to which Apache mod_plsql connects. Among other things, the DAD specifies the NLS_LANG parameter that is used to initialize the locale of a database session to the target database. NLS_LANG should be configured with the same value as listed in Table 1: Locale and NLS_LANG Value Mappings.

Programming

On Oracle9*i*AS, you can develop and deploy Internet applications written in the following programming environments.

- Java Server Page (JSP) and Java Servlet, callable from Apache mod_jserv
- PL/SQL and PL/SQL Server Page (PSP), callable from Apache mod_plsql
- Perl, callable from Apache mod_perl and mod_cgi
- C/C++

When developing a global Internet application supporting multiple languages, there are some good coding practices developers should be aware of. These coding practices are described in subsequent sections for each of the above programming languages where applicable.

Page Encoding for HTML

The encoding (or character set) of an HTML page is a very important piece of information to a browser and an Internet application. The browser needs to know so that it can use correct fonts and character set mapping tables for displaying pages, and the Internet applications need to know so they can safely process input data from a HTML form based on the encoding. You can view the page encoding as the character set used for the locale to which an Internet application is serving. In order to correctly specify the page encoding for HTML pages, Internet applications must:

- Choose a desired page encoding.
- Encode HTML content in the desired encoding.
- Correctly specify the HTML pages with the corresponding encoding name.

Choosing a Page Encoding

Mono The page encoding of HTML pages is determined based on the locale of the user to which the application is serving. If your Internet application needs to support only a single locale per instance, HTML pages should be encoded in the native encoding for that locale, and the encoding should be equivalent to the NLS_LANG character set specified for Apache. Table 2 lists the NLS_LANG character set for the native encodings of the most commonly used locales and their corresponding Internet Assigned Numbers Authority (IANA) encoding names and Java encoding names. These character sets should be used for monolingual application deployments.

Table 2: Native Encoding Mappings for Commonly Used Locales

Language	NLS_LANG character set names	IANA encoding names	Java encoding names
Western European	WE8MSWIN1252	ISO-8859-1	ISO8859_1
Central European	EE8MSWIN1250	ISO-8859-2	ISO8859_2
Japanese	JA16SJIS	Shift_JIS	MS932
Traditional Chinese	ZHT16MSWIN950	Big5	MS950
Simplified Chinese	ZHS16GBK	GB2312	GBK
Korean	KO16MSWIN949	EUC-KR	MS949
Arabic	AR8MSWIN1256	ISO-8859-6	ISO8859_6
Hebrew	IW8MSWIN1255	ISO-8859-8	ISO8859_8
Cyrillic	CL8MSWIN1251	ISO-8859-5	ISO8859_5
Baltic	BLT8MSWIN1257	ISO-8859-4	ISO8859_4
Greek	EL8MSWIN1253	ISO-8859-7	ISO8859_7
Thai	TH8TISASCII	TIS-620	TIS620
Turkish	TR8MSWIN1254	ISO-8859-9	ISO8859_9
Universal	UTF8	UTF-8	UTF8

Multi

If an Internet application supports multiple languages simultaneously, the application needs to determine the encoding used for the locale of the current user at run time and map the locale to the encoding described in Table 2: Native Encoding Mappings.

As an alternative to using different native encodings for different locales, you may choose to use UTF-8 as the page encoding always. Using the UTF-8 encoding not only simplifies the coding for multilingual applications but also supports multilingual content. In fact, if a multilingual Internet application is written Perl or PL/SQL, the best choice for the page encoding is UTF-8 because these programming environments do not provide an intuitive and efficient way to convert HTML content from UTF-8 to the native encodings of various locales.

However, there are some limitations when using UTF-8 as the page encoding in Netscape 4.x browser. They are listed below. Netscape6 resolves the second and third issues.

- Non-ASCII file names cannot be specified in a HTTP multipart request
- Asian characters are corrupted in tool tips on localized version of Windows NT 4.0.
- Font used for UTF-8 page requires to be specified manually. Users have to specify the font to be used in one of the preference pages.

Specifying the Page Encoding for HTML Pages

The encoding of HTML pages not only can tell the browser to switch to the encoding of the pages automatically, but also to return user input in the specified encoding. In fact, it is always the best practice to specify the encoding of HTML pages returned to the client browser.

There are basically two ways to specify the encoding of an HTML page. If both are used together, the first one has precedence over the second one.

1. Specify the encoding in the HTTP header
   - The HTTP specification includes the Content-Type HTTP header to specify the content type and character set information of a document. This header is correctly interpreted by the most commonly used browsers, such as Netscape 4.0 and Internet Explorer 4.0 or later. The Content-Type HTTP header is of the form:

     ```
     Content-type: text/plain; charset=iso-8859-4
     ```

   - The `charset` parameter specifies the encoding for the HTML page. The possible values for the `charset` parameter are the IANA names for the character encoding supported by the browser.

2. Specify the encoding in the HTML page header
   - This is mainly used for static HTML pages. The character encoding of these pages should be specified in the HTML header as follows:

     ```
     <meta http-equiv="Content-Type" content="text/html;charset=utf-8">
     ```

   - An HTML page encoding should be specified in the `charset` parameter in the same way as in the Content-Type HTTP header.

Table 2 shows the IANA names of the commonly used encodings for various language groups.

Handling Page Encoding in Java Servlet and JSP

You can specify the encoding of an HTML page in the Content-Type HTTP header in a JSP using the `contentType` page directive. An example is shown below.

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

This is the MIME type and character encoding the JSP file uses for the response it sends to the client. You can use any MIME type or IANA character set name that is valid for the JSP container. The default MIME type is `text/html`, and the default character set is `ISO-8859-1`. In the above example, the character set is set to UTF-8. The character set of the `contentType` page directive directs the JSP engine to encode the dynamic HTML page and set the HTTP Content-Type header with this character set.

For Java Servlets, you can specify a page encoding in the HTTP header by calling the `setContentType()` method of the Servlet API. The following `doGet()` function shows how this method should be called.

```
public void doGet(HttpServletRequest req, HttpServletResponse res)throws
ServletException, IOException
{

    // generate the MIME type and character set header
    res.setContentType("text/html; charset=utf-8");
    ...
    // generate the HTML page
    Printwriter out = res.getWriter();
    out.println("<HTML>");
    ...
    out.println("</HTML>");
}
```

Note that the `setContentType`() method should be called before the `getWriter`() method because the `getWriter`() method initializes an output stream writer using the character set specified in the `setContentType()` method call. Any HTML content written to the writer and eventually to a browser would be encoded in the encoding specified in the `setContentType()` call.

Handling Page Encoding in PL/SQL and PSP

You may specify a page encoding in a PSP in the same way you specify it in a JSP page. The following directive directs the PSP compiler to generate code to set the page encoding in the HTTP Content-Type header for this page.

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

To specify the encoding in the Content-Type HTTP header for PL/SQL procedures, you should use the following Web Toolkit API (from the OWA_UTL package) from within PL/SQL procedures:

```
owa_util.mime_header('text/html', false, 'utf-8')
```

The above API should be called in the context of the HTTP header. It generates the following header in the HTTP response:

```
Content-type: text/html; charset=utf-8
```

Alternatively, you can also specify a page encoding in generated HTML content by adding the following PL/SQL code:

```
htp.headopen;
htp.meta('content-type', null, 'text/html; charset=utf-8');
htp.headclose;
```

For PL/SQL front-end applications and PSPs, the HTML pages generated are always encoded in the NLS_LANG character set specified in the corresponding DAD (which must also be the same as the database character set). Therefore, the character set specified in the NLS_LANG of a DAD must match the encoding being specified for a HTML page. Otherwise, the HTML page is delivered in a different character set from the one it is tagged with. In other words, the configuration of the database and the NLS_LANG character set of the DAD control the encoding of the HTML output instead of the application itself.

Handling Page Encoding in Perl

For Perl scripts running in the Apache mod_perl environment, you can specify an encoding to a HTML page in the HTTP Content-Type header as follows:

```
$page_encoding = 'utf-8';
$r->content_type("text/html; charset=$page_encoding");
$r->send_http_header;
return OK if $r->header_only;
```

Mono For monolingual Internet applications, the encoding of a HTML page should be equivalent to the character set used for the POSIX locale on which a Perl script runs and to the NLS_LANG character set if the Perl script also accesses a back-end database.

Multi For multilingual Internet applications, Perl scripts should run on an environment where:

- Both the NLS_LANG character set and the character set used for the POSIX locale are equivalent to UTF-8.
- The UTF8 Perl pragma should be used (This pragma tells the Perl interpreter to encode identifiers and strings in the UTF-8 encoding). For more information on the UTF8 pragma, please refer to the perldoc:

```
<ORACLE_HOME>/Apache/Perl/<Perl Version>/bin/perldoc perlunicode
```

This environment allows the scripts to process data in any languages in the form of UTF-8. The page encoding of the dynamic HTML pages generated from the scripts, however, could be different from UTF-8. If this is the case, the `Unicode::MapUTF8` Perl module, which can be downloaded from www.cpan.org, should be used to convert data from UTF-8 to whatever the page encoding is. The following example illustrates how this module should be used to generate HTML pages in the Shift_JIS encoding:

```
use Unicode::MapUTF8 qw(from_utf8)
# This is to show how the UTF8 Perl pragma is specified,
# and is NOT required by the from_utf8 function.
use utf8;
...
$page_encoding = 'Shift_JIS';
$r->content_type("text/html; charset=$page_encoding");
$r->send_http_header;
return OK if $r->header_only;
...
#html_lines contains HTML content in UTF-8
print (from_utf8({ -string=>$html_lines, -charset=>$page_encoding}));
...
```

The `from_utf8()` function converts dynamic HTML content from UTF-8 to the character set specified in the `charset` argument.

Form Input Handling

Applications generate HTML forms to get input from the user. For both Netscape and Internet Explorer browsers, the encoding of the input always corresponds to the encoding of the forms for both POST and GET requests. In other words, if the encoding of a form is in UTF-8, input text returned from a browser is encoded in UTF-8 as well. Based on this fact, Internet applications are able to control the encoding of the form input by specifying the corresponding encoding in a requesting HTML form.

How user input is passed from a browser in a POST request is different from that in a GET request:

- For POST requests, input is passed as part of the request body, and 8-bit data is allowed.
- For GET requests, input is passed as part of a URL as an embedded query string where every non-ASCII byte is encoded as `%XX` where `XX` is the hexadecimal representation for the binary value of the byte.

HTML standards allow for named and numbered entities. These special codes allow users to specify characters. For example, "`&aelig;`" and "`&#230;`" both reference the same character - æ. Tables of these entities are available at http://www.w3.org/TR/WD-htm140-970708/sgml/entities.html.

Some browsers generate *numbered* or *named* entities for any input character that cannot be encoded in the encoding of an HTML form. For example, the Euro character "€" and the character "à" (Unicode values 8364 and 224 respectively) cannot be encoded in Big5 encoding and will be sent as "&#8364;" and "&agrave;" when the HTML encoding is Big5. However, these cases will not happen if the page encoding of the HTML form is UTF-8 because all characters can be encoded in UTF-8. Internet applications which support page encoding other than UTF-8 should handle these cases.

Form Input Handling in Java

In most JSP and Servlet containers (including Apache JServ), the Servlet API implementation assumes that incoming form input is in the ISO-8859-1 encoding. As a result, when the `HttpServletRequest.getParameter()` API is called, all embedded `%XX` data of the input text is decoded, and the decoded input is converted from ISO-8859-1 to Unicode and returned as a Java string. The Java string returned is incorrect if the encoding of the HTML form is not ISO-8859-1. However, you can work around this problem. When a JSP or Java Servlet receives form input in a Java string, it needs to convert them back to the original form in bytes, and then convert the original form to a Java string based on the correct encoding.

```
String original = request.getParameter("name");
try
{
    String real = new String(original.getBytes("8859_1"),"UTF8");
}
catch (UnsupportedEncodingException e)
{
    String real = original;
}
```

In the above example, the Java string `real` will be initialized to store the correct characters from a UTF-8 form. In addition to Java encoding names, IANA encoding names can be used as aliases in Java functions. See Table 2 for some mappings between IANA and Java encoding names.

With Servlet API 2.3, you can get the correct input by setting the `CharEncoding` attribute of the HTTP request object before calling the `getParameter()` function. The code is provide below:

```
request.setCharacterEncoding("UTF8");
String real = request.getParameter("name");
```

Form Input Handling in PL/SQL

Form input is passed to PL/SQL procedures as PL/SQL procedure arguments. When a POST or GET request is issued from a browser, the form input is first sent from the browser to the Apache mod_plsql module in the encoding of the requesting HTML form. All `%XX` escape sequences in the input are then un-escaped to their actual binary representations before being passed to the PL/SQL procedure serving the request.

Mono For a monolingual application deployment, the page encoding of a HTML page generated should be the same as the database character set of the front-end database where PL/SQL front-end programs run. Hence the HTML input that is passed as the PL/SQL procedure arguments should already be encoded in the database character set and ready to be used within the PL/SQL procedure.

Multi For a multilingual application deployment, Oracle recommends using the UTF-8 encoding for both HTML forms and front-end databases. Form input passed as PL/SQL procedure arguments are encoded in UTF-8 and ready to be used within the PL/SQL procedure running in the front-end Unicode databases.

Form Input Handling in Perl

In the Apache mod_perl environment, from input is passed to a Perl script differently in a GET request and a POST request. It is a good practice to handle both types of requests in the script. The following piece of code gets the input value of the `name` parameter from a HTML form.

```
my $r = shift;
my %params = $r->method eq 'POST' ? $r->content : $r->args ;
my $name = $params{'name'} ;
```

Multi For multilingual Perl scripts, the page encoding of a HTML form may be different from the UTF-8 encoding used in the Perl scripts. In this case, input data should be converted from the page encoding to UTF-8 before being processed. The following example illustrates how strings are converted from Shift_JIS to UTF-8 by using the `Unicode::MapUTF8` Perl module.

```
use Unicode::MapUTF8 qw(to_utf8);
# This is to show how the UTF8 Perl pragma is specified,
# and is NOT required by the from_utf8 function.
use utf8;
...
my $page_encoding = 'Shift_JIS';
my $r = shift;
my %params = $r->method eq 'POST' ? $r->content : $r->args ;
my $name = to_utf8({-string=>$params{'name'}, -charset=>$page_encoding});
...
```

The `to_utf8()` function converts any input string from the encoding specified to UTF-8.

Encoding URL

If HTML pages contain URLs with embedded query strings, any non-ASCII byte in the query strings must be escaped in the %XX format where XX is the hexadecimal representation of the binary value of the byte. Assuming that an Internet application embeds a URL pointing to a UTF-8 JSP page containing the German name “Schloß”, the URL should be encoded in the following way:

```
http://<host.domain>/actionpage.jsp?name=Schlo%c3%9f
```

where `c3` and `9f` represent the binary value in hexadecimal of the ß character in the UTF-8 encoding.

To encode a URL, you need to make sure the following two things are correctly done:

1. Convert the URL into the encoding expected from the target object. This encoding is usually the same as the page encoding you choose to use in your application.
2. Escape non-ASCII bytes of the URL into the `%XX` format.

Most programming environments provide APIs to encode and decode URLs, and they are described in the following sections.

Encoding URL in Java

If a URL is constructed in a JSP or Java Servlet, all 8-bit bytes must be escaped using their hexadecimal values prefixed by a percent sign as described above. The `URLEncoder.encode()` function provided in JDK 1.1 and JDK 1.2 only works if a URL is encoded in the Java default encoding. To make it work for URLs in any encodings, you need to add your own code to escape any non-ASCII characters in a URL into their hexadecimal representation based on the encoding of your choice. The following code shows you an example on how to encode a URL based on the UTF-8 encoding.

```
String unreserved = new String("-
_.!~*'()ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz 0123456789");
byte [] urlBytes;
try
{
    urlBytes = url.getBytes("UTF8");
}
catch (UnsupportedEncodingException e)
{
    urlBytes = url.getBytes();
}
StringBuffer out = new StringBuffer(urlBytes.length);
for (int i = 0; i < urlBytes.length; i++)
{
    int c = (int) urlBytes[i];
    if (unreserved.indexOf(c) != -1)
    {
        if (c == ' ') c = '+';
        out.append((char)c);
    }
    else
       out.append(“%” + Long.toHexString((long)(c&0xff)).toUpperCase());
}
encodedUrl = out.toString();
```

Encoding URL in PL/SQL

In Oracle9*i* database, you can call the `ESCAPE()` function in the UTL_URL package to encode a URL in PL/SQL. The `ESCAPE()` function can be called as follows:

```
encodedURL varchar2(100);
url varchar2(100);
charset varchar2(40);
...
encodedURL := UTL_URL.ESCAPE(url, FALSE, charset);
```

The `url` argument is the URL you want to encode, and the `charset` argument specifies the character encoding used for the encoded URL. You should use a valid Oracle character set name for the `charset` argument. Some of them are listed in Table 2. To encode a URL in the database character set, the `charset` argument should always be specified as NULL.

Encoding URL in Perl

Encoding a URL in Perl can be done by using the `escape_uri()` function of the `Apache::Util` module as follows:

```
use Apache::Util qw(escape_uri);
...
$escaped_url   = escape_uri( $url );
...
```

The `escape_uri()` simply takes the bytes from the `$url` input argument and encode them into the `%XX` format. If you want to encode a URL in a different character encoding, you need to first convert the URL to the target encoding before calling the `escape_uri()`function. Perl provides a few modules for character conversion. Please refer to the CPAN web site at www.cpan.org.

Formating HTML

The format of HTML pages should be designed according to the following guidelines:

- Allow table cells to resize themselves as the enclosed text expands, instead of hard-coding the widths of the cells as follows:

  ```
  <TD WIDTH="50">
  ```

  If the widths of cells must be specified, it is better to externalize the width values as well so that translators are able to adjust them with the translated text.

- Fonts should not be specified directly in the pages because they may not contain glyphs for all languages that the application supports. Instead, each element should inherit from a class in a Cascading Style Sheet (CSS) where fonts and font sizes are specified.

- For bi-directional languages such as Arabic and Hebrew, the pages should have a DIR attribute in its <HTML> tag to indicate that the direction of the language displayed is from right to left. For example, with the <HTML DIR="RTL"> tag, all components of an HTML page will follow the direction of the HTML tag. To make direction settings seamless to developers, the direction should be set in the CSS file as follows:

  ```
  HTML{ direction:rtl }
  ```

  The direction property was introduced in CSS level 2, which is support in IE 5.0.

- Text alignment should be sensitive to the direction of the text. Instead of using the absolute alignments such as LEFT and RIGHT, use the following alignments:
  - START, which is the leading alignment relative to the direction of the text.
  - END, which is the trailing alignment relative to the direction of the text.
  - CENTER, which centers the text regardless of the direction of the text.

It is a good practice to provide Cascade Style Sheet (CSS) for different locales or groups of locales and use them to control how HTML pages are rendered. Using a CSS isolates the locale-specific formatting information from HTML pages. Applications should dynamically generate CSS references in HTML pages corresponding on the locale of a user so that the pages can be rendered with the corresponding locale-specific formats. Locale-specific information in the CSS file should include:

- Font names and sizes
- Alignments (for bi-directional language support only)
- Direction of text (for bi-directional language support only)

Database Server Access

Internet applications access databases via various data access products. Any Java-based Internet applications using technologies such as Java Servlets, JSPs, and EJBs uses the Oracle JDBC drivers for database connectivity.

Since Java strings are always Unicode-encoded, JDBC transparently converts text data from the database character set to Unicode and vice versa. Java Servlets and JSPs that interact with an Oracle database should make sure that the Java strings returned from the database are converted to the encoding of the HTML page being constructed, and that form inputs are converted from the encoding of the HTML form to Unicode before being used in calling the JDBC driver.

For non-Java Internet applications using programming technologies such as Perl, PL/SQL and C/C++, text data retrieved from or inserted into a database are encoded in the character set specified in NLS_LANG. The character set used for the POSIX locale should match the NLS_LANG character set so that data from the database can be directly processed with the POSIX locale-sensitive functions in the applications. The values for the NLS_LANG parameter for different locales and different deployment alternatives are described in the Configuring NLS_LANG section.

Multi For multilingual applications, the NLS_LANG character set and the page encoding may both be UTF-8 to avoid character set conversion and possible data lost.

Database Access using JDBC

You should use the Oracle JDBC drivers provided in Oracle9*i*AS for Oracle9*i* database access when using JSPs and Java Servlets. Oracle9*i*AS provides two client-side JDBC drivers that can be deployed with middle-tier applications:

- JDBC OCI driver, which requires the Oracle client library
- JDBC Thin driver, which is a pure Java driver

Oracle JDBC drivers transparently convert character data from the database character set to Unicode for the SQL CHAR datatypes and the SQL NCHAR datatypes. As a result of this transparent conversion, JSPs and Java Servlets calling Oracle JDBC drivers may bind and define database columns with Java strings, and fetch data into Java strings from the result set of a SQL execution. An example of using a Java string to bind the NAME and ADDRESS columns of a customer table is shown below. These database columns are defined as VARCHAR2 and NVARCHAR2 respectively.

```
String caddr = request.getParameter("caddress");
String cname = request.getParameter("cname")
OraclePreparedStatement pstmt = conn.prepareStatement("insert into" +
        "CUSTOMERS (NAME, ADRESS) values (?, ?) ");
pstmt.setString(1, cname);
pstmt.setFormOfUse(2, OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(2, caddr);
pstmt.execute();
```

To bind a Java string variable to the `ADDRESS` column defined as NVARCHAR2, the `setFormOfUse()` method should be called before the `setString()` method. For more information on accessing databases from Java, please refer to the Oracle9*i* JDBC Developer's Guide and Reference.

Mono In addition, the Oracle JDBC drivers set the values for the NLS_LANGUAGE and NLS_TERRITORY session parameters to the ones corresponding to the default Java locale when the database session was initialized. For monolingual applications, the Java default locale is configured to match the locale to which the applications are serving, and hence the database connection is always synchronized with the locale of the target users.

Database Access in PL/SQL

PL/SQL procedures running in a front-end database access a back-end Oracle9*i* database via database links and SQL. For example, the following PL/SQL procedure callable from the Apache mod_plsql module inserts a record to a customer table with the customer name column defined as VARCHAR2 and the customer address column defined as NVARCHAR2.

```
procedure addcustomer( cname varchar2, caddress varchar2) is
    caddr nvarchar2(200);
begin
    caddr := TO_NCHAR(cddress);
    insert into customers@remote (name, address) values (cname, caddr);
    commit;
end;
```

Note that Apache mod_plsql module does not support NVARCHAR argument passing and as a result, PL/SQL procedures have to use VARCHAR2 for arguments and convert them to NVARCHAR2 explicitly before executing the INSERT statement. The database link, called `remote` in the example, is configured for accessing data from a back-end database.

Mono For monolingual applications, data retrieved from and inserted into the columns of the SQL CHAR datatypes in the back-end database would be transparently converted from the database character set of a back-end database to that of a front-end database. Care must be taken for any data expansion as a result of this data conversion. To avoid character set conversion and data expansion, data can be stored in the columns of the SQL NCHAR datatypes in the back-end database and PL/SQL procedures use NVARCHAR2 variables to bind those columns and manipulate the data in the NVARCHAR2 datatype. In PL/SQL, you can use the SQL NCHAR datatypes for string data in the same way as the SQL CHAR datatypes.

Multi For multilingual applications, both the database and national character sets of a front-end database and a back-end databases are UTF-8, and there is no character set conversion when PL/SQL procedures retrieve data from or insert data into the back-end database.

Database Access in Perl

Perl scripts access Oracle9*i* databases using the DBI/DBD driver for Oracle. The DBI /DBD driver calls Oracle Callable Interface (OCI) to access the databases. The data retrieved from or inserted into the databases is encoded in the NLS_LANG character set. It is important for Perl scripts to initialize a POSIX locale with the locale specified in the LC_ALL environment variable and use a character set equivalent to the NLS_LANG character set so that data retrieved from the databases can be processed with POSIX string manipulation functions. The following shows how to insert a row to a customer table to an Oracle9*i* database through the DBI/DBD driver.

```
Use Apache::DBI;
...
# Connect to the database
```

```
$constr = 'host=dlsun1304.us.oracle.com;sid=icachedb;port=1521' ;
$usr = 'system' ;
$pwd = 'manager' ;
$dbh = DBI->connect("dbi:Oracle:$constr", $usr, $pwd, {AutoCommit=>1} ) ||
       $r->print("Failed to connect to Oracle: " . DBI->errstr );

# prepare the statement
$sql = 'insert into customers (name, address) values (?, ?)';
$sth = $dbh->prepare( $sql );
$sth->execute( $cname, $caddress );
$sth->finish();
$dbh->disconnect();
```

Multi

In order to properly process UTF-8 data in a multilingual application, Perl scripts should:

- Use a POSIX locale associated with the UTF-8 character set
- Use the UTF8 Perl module to indicate that all strings in the Perl scripts are in UTF-8

Database Access in C/C++

C/C++ applications access Oracle9*i* databases via OCI or Pro*C/C++. You can call OCI directly or use the Pro*C/C++ interface to retrieve and store Unicode data in a UTF-8 database and the SQL NCHAR datatypes. Generally speaking, data retrieved from and inserted into a back-end database are encoded in the NLS_LANG character set. C/C++ programs should use the same character set for their POSIX locale as the NLS_LANG character set. Otherwise, the POSIX string functions cannot be used on the character data retrieved from the back-end database and, by the same token, the character data encoded in the POSIX locale can possibly be corrupted when they are inserted into the back-end database.

Multi

For multilingual applications, you may want to use the Unicode API provided in the OCI library instead of relying the NLS_LANG character set. This alternative is good for applications written for platforms such as Windows NT/2000, where the `wchar_t` datatype is implemented using UTF-16 Unicode. Using the Unicode API for those platforms bypasses some unnecessary data conversions that are required when using the regular OCI API.

Using the Regular OCI API

The following example shows you how to bind and define the VARCHAR2 and NVARCHAR2 columns of a customer table in C/C++ using OCI based on the NLS_LANG character set. Note that the `text` datatype is a macro for `unsigned char`.

```
text *sqlstmt= (text *)"SELECT name, address FROM customers
                        WHERE id = :cusid";
text cname[100];                      /* Customer Name */
text caddr[200];                      /* Customer Address */
text custid[10] = "9876";             /* Customer ID */
ub2 cform = SQLCS_NCHAR;              /* Form of Use for NCHAR types */
...
OCIStmtPrepare (stmthp, errhp, sqlstmt,
                (ub4)strlen ((char *)sqlstmt),
                (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));
/* Bind the custid buffer */
OCIBindByName(stmthp, &bnd1p, errhp, (text*)":custid",
              (sb4)strlen((char *)":custid"),
              (dvoid *) custid, sizeof(cust_id), SQLT_STR,
              (dvoid *)&insname_ind, (ub2 *) 0, (ub2 *) 0,
              (ub4) 0,(ub4 *)0, OCI_DEFAULT);

/* Define the cname buffer for VARCHAR */
OCIDefineByPos (stmthp, &dfn1p, errhp, (ub4)1, (dvoid *)cname,
                (sb4)sizeof(cname), SQLT_STR,
                (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);

/* Define the caddr buffer for the address column in NVARCHAR2 */
OCIDefineByPos (stmthp, &dfn2p, errhp, (ub4)2, (dvoid *)caddr,
                (sb4)sizeof(caddr), SQLT_STR,
                (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfn2p, (ub4) OCI_HTYPE_DEFINE, (void *) &cform, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_FORM, errhp);
...
```

Multi

Using the Unicode API in OCI

You turn on the Unicode API by specifying Unicode mode when creating an OCI environment handle. Any inherited handle from the OCI environment handle will be set to Unicode mode automatically. By changing to Unicode mode, all text data arguments to the OCI functions are assumed to be in the Unicode text (`utext*`) datatype and in the UTF-16 encoding. For binding and defining, the data buffers are also assumed to be `utext` buffers in the UTF-16 encoding. The following Windows program shows how you can create an OCI environment handle with Unicode mode on and bind / define the same columns, name in VARCHAR2 and address in NVARCHAR2, of the customer table:

```
utext *sqlstmt= (text *)L"SELECT name, address FROM customers
                        WHERE id = :cusid";
utext cname[100];                     /* Customer Name */
utext caddr[200];                     /* Customer Address */
text custid[10] = "9876";             /* Customer ID */
ub1 cform = SQLCS_NCHAR;              /* Form of Use for NCHAR types */
...
/* Use Unicode OCI API by specifying UTF-16 mode */
status = OCIEnvCreate((OCIEnv **)&envhp, OCI_UTF16, (void *)0,
                      (void *(*) ()) 0, (void *(*) ()) 0, (void(*) ()) 0,
                      (size_t) 0, (void **)0);
...
OCIStmtPrepare (stmthp, errhp, sqlstmt,
                (ub4)wcslen ((char *)sqlstmt),
                (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));
/* Bind the custid buffer */
OCIBindByName(stmthp, &bnd1p, errhp, (constant text*) L":custid",
              (sb4)wcslen(L":custid"),
              (dvoid *) custid, sizeof(cust_id), SQLT_STR,
              (dvoid *)&insname_ind, (ub2 *) 0, (ub2 *) 0,
              (ub4) 0,(ub4 *)0, OCI_DEFAULT);

/* Define the cname buffer for the name column in VARCHAR2 */
OCIDefineByPos (stmthp, &dfn1p, errhp, (ub4)1, (dvoid *)cname,
                    (sb4)sizeof(cname), SQLT_STR,
                    (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);

/* Define the caddr buffer for the address column in NVARCHAR2 */
OCIDefineByPos (stmthp, &dfn2p, errhp, (ub4)2, (dvoid *)caddr,
                    (sb4)sizeof(caddr), SQLT_STR,
                    (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfn2p, (ub4) OCI_HTYPE_DEFINE, (void *) &cform, (ub4) 0,
             (ub4)OCI_ATTR_CHARSET_FORM, errhp);
...
```

Basically, the program code for the Unicode API is similar to that of non-Unicode except that

- All `text` datatypes are changed to `utext` datatypes which is a macro of the `unsigned short` datatype
- All literal strings are changed to Unicode literal strings
- All `strlen()` functions are changed to the `wcslen()` functions to calculate the string length in number of Unicode characters instead of bytes

Multi

*Using Unicode Bind and Define in Pro*C/C++*

Pro*C/C++ lets you specify UTF-16 Unicode buffers for bind and define operations. There are two ways to specify UTF-16 buffers in Pro*C/C++:

1. Use the `utext` datatype, which is actually an alias for the `unsigned short` datatype in C/C++
2. Use the `uvarchar` datatype provided by Pro*C/C++. It will be preprocessed to a `struct` with a length field and `utext` buffer field.

```
struct uvarchar
{
   ub2 len;         /* length of arr */
   utext arr[1] ;   /* UTF-16 buffer */
};
typedef struct uvarchar uvarchar ;
```

In the following example code, there are two host variables, `cname` and `caddr`. The `cname` host variable is declared as a `utext` buffer containing 100 UTF-16 code units (`unsigned short`) for the customer name column in VARCHAR2. The `caddr` host variable is declared as a `uvarchar` buffer containing 50 UCS2 characters for the customer address column in NVARCHAR2, the `len` and `arr` fields are accessible as fields of a `struct`.

```
#include <sqlca.h>
#include <sqlucs2.h>

main()
{
   ...
   /* Change to STRING datatype:    */
   EXEC ORACLE OPTION (CHAR_MAP=STRING) ;
   utext cname[100] ;             /* unsigned short type */
   uvarchar caddr[200] ;          /* Pro*C/C++ uvarchar type */
   ...
   EXEC SQL SELECT name, address INTO :cname, :caddr FROM customers;
```

```
        /* cname is NULL-terminated */
        wprintf(L"ENAME = %s, ADDRESS = %.*s\n", cname, caddr.len, caddr.arr);
        ...
    }
```

User Locale Awareness

To be locale-aware or locale-sensitive, Internet applications need to determine the locale of a user.

Mono For monolingual applications, the locale of a user is always fixed and usually follows the default locale of the runtime environment so that they can be easily configured to run on a different locale.

Multi For multilingual applications, the locale of a user differs in HTTP requests and should be dynamically detected. The code for a multilingual Internet application must be sensitive to the locale of a user, not the default runtime locale of a programming environment.

Once the locale of a user is determined, applications should:

- Construct HTML content in the language of the locale
- Use the cultural conventions implied by the locale

Locale-sensitive functions, such as date formatting, are built into various programming environments such as C/C++, Java and PL/SQL. Applications may use them to format the HTML pages according to the cultural conventions of the locale of a user. A locale is represented differently in these programming environments. For example, the French (Canada) locale is represented differently in these environments as follows:

- In the ISO standard, it is represented as “fr-CA” where “fr” is the language code defined in the ISO 639 standard and “CA” is the country code defined in the ISO 3166 standard.
- In Java, it is represented as an Java Locale object constructed with “fr”, the ISO language code for French, as the language and “CA”, the ISO country code for Canada, as the country. The Java Locale name is “fr_CA”.
- In C/C++, it is represented as a POSIX locale name which is implementation-specific. French (Canada) is represented as “fr_CA” on Sun Solaris. POSIX locale names may include a character set for the locale to overwrite the default character set when it is not sufficient. For example, the de.ISO8859-15 locale is used to support the Euro symbol.
- In PL/SQL and SQL, it is represented mainly by the NLS_LANGUAGE and NLS_TERRITORY session parameters where the values of the NLS_LANGUAGE parameter equal to “CANADIAN FRENCH” and the value of the NLS_TERRITORY parameter equal to “CANADA”.

Table 3 shows the locale representations of some commonly used locales in various programming environments.

Table 3: Locale Representations in Various Programming Environments

Locale	ISO	Java	POSIX Solaris	NLS_LANGUAGE/NLS_TERRITORY
Arabic (U.A.E.)	ar	ar	ar	ARABIC/ UNITED ARAB EMIRATES
Germany (German)	de-DE	de_DE	de	GERMANY/GERMAN
English (U.S.A)	en	en_US	en_US	AMERICAN/AMERICA
English (United Kingdom)	en-GB	en_GB	en_UK	ENGLISH/UNITED KINGDOM
Greek	el	el	el	GREEK/GREECE
Spanish (Spain)	es-ES	es_ES	es	SPANISH/SPAIN
French (France)	fr	fr_FR	fr	FRENCH/FRANCE
French (Canada)	fr-CA	fr_CA	fr_CA	CANADIAN FRENCH/CANADA
Hebrew	he	he	he	HEBREW/ISRAEL
Italian (Italy)	it	it	it	ITALIAN/ITALY

Japanese	ja-JP	ja_JP	ja_JP	JAPANESE / JAPAN
Korean	ko-KR	ko_KR	ko_KR	KOREAN / KOREA
Portuguese (Portugal)	pt	pt	pt	PORTUGUESE / PORTUGAL
Portuguese (Brazil)	pt-BR	pt_BR	pt_BR	BRAZILIAN PORTUGUESE / BRAZIL
Turkish	tr	tr	tr	TURKISH / TURKEY
Thai	th	th	th	THAI / THAILAND
Chinese (Taiwan)	zh-TW	zh_TW	zh_TW	TRADITIONAL CHINESE / TAIWAN
Chinese (P.R.C)	zh-CN	zh_CN	zh_CN	SIMPLIFIED CHINESE / CHINA

Applications written for more than one programming environment require synchronizing a locale from one environment with another. For example, applications written in Java that call PL / SQL procedures should map the Java locale of a user to the corresponding NLS_LANGUAGE and NLS_TERRITORY values and alter the corresponding NLS parameters with these values before calling the procedure.

Getting the User Locale

Mono

To be locale-aware, an application has to know the locale of a user it is currently serving. Monolingual applications always serve users with the same locale, and that locale should also be equivalent to the default runtime locale of the corresponding programming environment.

Multi

Multilingual applications can determine a user locale dynamically in three ways. All of them have some pros and cons, but they can be used together in the applications to complement each other:

- Based on the user profile information from a LDAP directory server such as the Oracle Internet Directory (OiD)

  The information of user profile can be stored in a OiD server provided by Oracle9*i*AS, and the LDAP schema for the user profile should include a preferred locale attribute to indicate the locale of a user. This way of determining a locale user does not work if a user has not been logged on.

- Based on the default locale of the browser

  Get the default ISO locale setting from a browser. The default ISO locale of the browser is sent via the Accept-Language HTTP header in every HTTP request. If the Accept-Language header is NULL, the desired locale should default to English. The drawback of this approach is that the Accept-Language header may not be a reliable source of information for the locale of a user.

- Based on user Input

  Allow users to select a locale from a list box, and switch the locale to the one selected on the fly.

Once the locale is determined, it should be mapped to the one corresponding to the programming environments on which the applications run and used on any locale-sensitive functions.

Locale Awareness in Java

A user locale can be represented in Java by a corresponding Java locale object. The Java encoding used for the locale is required for Java to properly convert Java string to byte data and vice versa. It should also be considered when making your Java code aware of a user locale. Every locale and encoding-sensitive method in Java provides two entry points:

- One that use the default Java locale and default Java encoding for the method implicitly
- One that allows you to explicitly specify a Java locale or a Java encoding for the method

Mono

Monolingual applications should be written to run on the default Java locale and default Java encoding so that they can be easily configured for a different locale. To make the applications sensitive to the default Java locale and default Java encoding, use the first entry point of the locale-sensitive and encoding-sensitive functions provided in Java. For example, to create a date formatter using the default Java locale, use the following method call:

```
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL);
dateString = df.format(date); /* Format a date */
```

Multi

Conversely, multilingual applications should be written independent of any fixed default locale or encoding. To make Java applications independent of the default Java locale and Java encoding but sensitive to the locale of the user, use the second entry points of the Java locale and encoding-sensitive methods and explicitly specify the Java locale and Java encoding corresponding to the locale of the current user. For example, specify the Java locale object corresponding to the locale of the user, identified by `user_locale` in the example below, in the `getDateTimeInstance()` method as follows:

```
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL,
user_locale);
dateString = df.format(date); /* Format a date */
```

Multi

By the same token, encoding-sensitive methods that assume the default Java encoding should not be used in a multilingual application. For example, the `String.getBytes()` method is encoding-sensitive and should not be used in a multilingual application. Instead, use the entry point that accepts an encoding as an argument for this method, that is `String.getBytes(String encoding)`, and make sure that the encoding used for the locale of the user is specified.

Do not use the `Locale.setDefault()` method to change your default locale because

- It changes the Java default locale for all threads and make your applications unsafe to threads
- It does not affect the Java default encoding

Locale Awareness in Perl and C/C++

Mono

Both Perl and C/C++ use the same POSIX locale model for internationalized applications. Monolingual applications should be sensitive to the default POSIX locale which is configured by changing the value of the LC_ALL environment variable or changing the operating system locale from the Windows' Control Panel. To run on the default POSIX locale, the applications should call `setlocale()` to set the default locale to the one defined in LC_ALL and use the POSIX locale-sensitive functions such as `strftime()` thereafter. Note that the `setlocale()` function affects the current process and all the threads associated with it so that any multithread application should assume the same POSIX locale in each thread. The following example gets the current time in the format specific to the default locale in Perl.

```
use locale;
use POSIX qw (locale_h);
...
$old_locale = setlocale( LC_ALL, "" );
$dateString = POSIX::strftime( "%c", localtime());
...
```

Multi

Multilingual applications should be sensitive to the locales that are dynamically determined. The same `setlocale()` function should be called to initialize a locale used for subsequent locale-sensitive functions before calling these functions. For example, the following C code gets the local time in the format of a user locale identified by `user_locale`:

```
#include <locale.h>
#include <time.h>
    ...
    const char *user_locale = "fr";
    time_t ltime;
```

```
          struct tm *thetime;
          unsigned char dateString[100];
          ...
          setlocale(LC_ALL, user_locale);
          time (&ltime);
          thetime = gmtime(&ltime);
          strftime((char *)dateString, 100, "%c", (const struct tm *)thetime))
          ...
```

The mappings of user locales to the corresponding POSIX locale names are required for applications to initialize the correct locale dynamically in C/C++ and Perl, and these POSIX locales depend on the operating system implementation dependent.

Locale Awareness in PL/SQL and SQL

PL/SQL procedures run in the context of a database session whose locale is initialized by the NLS_LANG parameter in the DAD. The NLS_LANG parameter specifies top level NLS parameters, NLS_LANGUAGE and NLS_TERRITORY, for the database session. There are several other NLS parameters such as NLS_SORT and NLS_DATE_LANGUAGE that are inherited from the values of these top-level parameters. The locale of a database session is defined by these NLS parameters.

Similar to Java, every locale-sensitive SQL and PL/SQL function have at least two entry points:

- One that is based on the NLS parameters of the current database session
- One that allows you to explicit specify the NLS parameters

Mono

Generally speaking, the initial values of the NLS parameters inherited from NLS_LANG are sufficient for monolingual PL/SQL procedures. Applications should use the first entry point of the locale-sensitive functions and rely on the NLS settings inherited from the NLS_LANG parameter for the functions to work properly. For example, the following PL/SQL code gets the formatted date by calling the `TO_CHAR()` function which uses the current values of the NLS_DATE_FORMAT and NLS_DATE_LANGUAGE parameters

```
mydate date;
dateString varchar2(100);
...
select sysdate into mydate from dual;
dateString = TO_CHAR(mydate);
```

Should you find the initial values of the NLS parameters not appropriate, you may issue an ALTER SESSION command to overwrite them for the current database session. An example to alter the NLS_DATE_FORMAT is shown below:

```
ALTER SESSION SET NLS_DATE_FORMAT='Day Month, YYYY';
```

Multi

Multilingual PL/SQL procedures should either issue the following ALTER SESSION commands to change the locale of the database session to that of a user before calling any locale-sensitive SQL or PL/SQL function:

```
ALTER SESSION SET NLS_LANGUAGE=<NLS_LANGUAGE of the user locale>
ALTER SESSION SET NLS_TERRITORY=<NLS_TERRITORY of the user locale>
```

or explicitly specify the corresponding NLS parameters in every SQL function that accepts a NLS parameter as an argument. For example, the following PL/SQL code gets a date string based on the language of the locale of a user.

```
mydate date;
```

```
      dateString varchar2(100);
      ...
      select sysdate into mydate from dual;
      dateString TO_CHAR(mydate, 'DD-MON-YYYY HH24:MI:SSxFF',
                          'NLS_DATE_LANGUAGE=<langauge>' );
      ...
```

The `<language>` placeholder specifies the Oracle language name for the locale of the user.

Organizing Content in Different Languages

The user interface (UI) and content presented in HTML pages should be translated. Translatable sources for the content of an HTML page can be categorized as follows.

1. Static files such as HTML, images and CSS
2. Static UI strings stored as Java resource bundles used by Java Servlets and JSPs
3. Static UI strings stored as POSIX message files used by C/C++ programs and Perl scripts
4. Static UI strings stored as relational data in a database used by PL/SQL procedures and PSPs
5. Dynamic content such as product information stored in a back-end database

Translatability Guidelines

When creating translatable content, developers should follow the practices described below.

1. All static and translatable UI strings used in programs such as Java Servlets, JSPs, Perl scripts, PL/SQL procedures and PSPs should be externalized to resource files. These resource files can then be translated independent of program code.
2. All dynamic text in a HTML page must be able to expand by at least 30% to cater for the text expansion as the result of translation without overlapping adjacent objects. The HTML page should look acceptable after expanding strings by 30% for the placeholders.
3. Avoid concatenating strings to form a sentence at runtime. The concatenated translated strings will not have the same meaning as the original English one. Use the string formatting functions provided by different programming languages to substitute runtime values for placeholders.
4. Avoid embedding text into images and graphics because they are often not easy to translate.
5. JavaScript code must not include any translatable string. JavaScript is hard to translate. Instead, applications should externalize all translatable strings, if any, into resource files or message tables and construct JavaScript code at run time and replace the dynamic text with the one corresponding to the locale of a user.
6. Since translations are often not available in the initial release of an application, it is important to make the application work even when the corresponding translation is not available by putting a fall-back mechanism in the application. The fallback mechanism can be as simple as using English information instead or as complex as using the closest language available possible. For example, the “fr-CA” locale stands for Canadian French. The fallback for this language can be “fr” which is French or “en” which is English. A simple way to find the closest language possible is to remove the territory part of the ISO locale name. It is up to the application how the fallback mechanism behaves.

Organizing Static Files

Translatable HTML, images, CSS files should be organized into different directories from non-translatable static files so that files under the locale-specific directory can be zipped for translation. There are many possible way to define the directory structure to hold these files. Here is just one example:

```
/docroot/images        - Non-translatable images
/docroot/html          - HTML common to all languages
/docroot/css           - Style sheets common to all languages
/docroot/<lang>        - Locale directory such as en, fr, ja etc.
/docroot/<lang>/images - Images specific for <lang>
/docroot/<lang>/html   - HTMLs specific for <lang>
/docroot/<lang>/css   - Style sheets specific for <lang>
```

The <`lang`> placeholder can be replaced with the ISO locale names. Based on the above structure, a utility function called `getLocalizedURL()` must be written to take a URL as parameter and look for the available language file from this structure. Whenever a HTML, image or CSS file is referenced in a HTML page, the Internet application should call this function to construct the path of the translated file corresponding to the current locale and fall back appropriately if the translation does not exists. For example, if the path `/docroot/html/welcome.html` is passed to `getLocalizedURL()` and the current locale is fr_CA , this function looks for the following files in the order shown below.

- `/docroot/fr_CA/html/welcome.html`
- `/docroot/fr/html/welcome.html`
- `/docroot/html/en/html/welcome.html`
- `/docroot/html/welcome.html`

The function returns the first file that exists in the above list. This function always reverts to English when the translated version corresponding to the current locale does not exist.

For Internet applications using UTF-8 as the page encoding, the encoding of the static HTML files should also be encoded in UTF-8. However, translators usually encode translated HTML files in the native encoding of the target language. To convert the translated HTML into UTF-8, you may use the JDK `native2ascii` utility shipped with Oracle9*i*AS. For example, to convert a Japanese HTML file encoded in Shift_JIS into UTF-8, follow the steps below:

1. Replace the `charset` parameter in the Content-Type HTML header in the `<meta>` tag with UTF-8.
2. `native2ascii -encoding MS932 japanese.html japanese.unicode`
3. `native2ascii -reverse -encoding UTF8 japanese.unicode japanese.html`

Organizing Translatable Strings for Java Servlets and JSPs

Translatable strings within Java Servlets and JSPs should be externalized into Java resource bundles so that these resource bundles can be translated independent of the Java code. After being translated, the resource bundles carry the same base class names as the English bundles but with the Java locale name as the suffix, and they should be placed in the same directory as the English resource bundles for the Java resource bundle lookup mechanism to function properly. For more information on Java resource bundles, please refer to JDK documentation.

Some people may disagree about externalizing JSP strings to resource bundles because it seems to defeat the whole purpose of using JSPs. There are two reasons for externalizing JSPs strings:

1. Translating JSPs is error prone simply because they consist of Java code that are not familiar to translators.
2. Separate the translation process from the development process so that translation can take place in parallel to development on JSPs. This is to eliminate the huge effort of merging the translated JSPs with the most up to date JSPs that contain bug fixes to the embedded Java code.

Java supports two types of resource bundles, the list resource bundle and the property resource bundle. Another good practice is to make use of list resource bundles instead of property resource bundles. The main reasons are

- List resource bundles are essentially Java programs that required to be compiled. Translation errors will be caught at compile time. Property resource bundles are text files read directory from Java. Translation errors can only be caught at run time.
- Property resource bundles expose all string data of your Internet application to users. There will be potential security and support issues for your application.

An example of a list resource bundle is shown below:

```
import java.util.ListResourceBundle;
public class Resource extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents =
    {
       {"hello", "Hello World"},
       ...

    };
}
```

Translators usually translate list resource bundles in the native encoding of the target language. Japanese list resource bundles encoded in Shift_JIS cannot be compiled on a English system because the Java compiler expects source files that are encoded in ISO-8859-1. In order to build your translated list resource bundles in a platform-independent manner, you need to run the JDK `native2ascii` utility to escape all non-ASCII characters to Unicode escape sequences in the `\uXXXX` format where `XXXX` is Unicode code value in hexadecimal. For example:

```
native2ascii -encoding MS932 resource_ja.java resource_ja.tmp
```

Mono At run time, monolingual applications can get strings from a resource bundle of the default Java locale as follows:

```
ResourceBundle rb = ResourceBundle.getBundle("resource");
String helloStr = rb.getString("hello");
```

Multi Since the locale of a user is not fixed, multilingual applications should call the `getBundle()` method by explicitly specifying a Java locale object, `user_locale` in the example below, corresponding to the locale of a user.

```
ResourceBundle rb = ResourceBundle.getBundle("resource", user_locale);
```

Java provides a default fall-back mechanism for resource bundles when translated resource bundles are not available. An application only needs to make sure that a base resource bundle without any locale suffix always exists in the same directory. The base resource bundle should contains strings in the fallback language. As an example, Java looks for a resource bundle in the following order when the Java locale “fr_CA” is specified to `getBundle()`.

```
resource_fr_CA
resource_fr
resource_en_US where en_US is the default Java locale
resource_en
resource (base resource bundle)
```

Organizing Translatable Static Strings in C/C++ and Perl

For C/C++ programs and Perl scripts running on UNIX platforms, static strings in C/C++ or Perl scripts should be externalized to POSIX message files. For programs running on Windows platforms, static strings should be externalized to message tables in a database as described in the next section because Windows does not support POSIX message files.

Message files (with .po file extension) associated with a POSIX locale are identified by their domain names, and they need to be compiled into binary objects (with .mo file extension) and placed into the directory corresponding to the POSIX locale. The path name for the POSIX locale is implementation specific. For example, a Canadian French message file, resource.po, is compiled into resource.mo by msgfmt and placed into the /usr/lib/locale/fr_CA/LC_MESSAGES directory on Solaris. For more information, please refer to the man pages of `gettext`, `msgfmt` and `xgettext`.

An example of a resource.po message file is shown below.

```
domian "resource"
msgid "hello"
msgstr "Hello World"
...
```

Note that the encoding used for the message files must match the encoding used for the corresponding POSIX locale.

Instead of putting binary message files into a implementation specific directory, you should put them into an application-specific directory and use the `binddomain()` function to associate a domain with a directory. The following piece of Perl script uses the `Locale::gettext` Perl module downloadable from the CPAN site to get a string from a POSIX message file:

```
use Locale::gettext;
use POSIX;
...
setlocale( LC_ALL, "fr_CA" );
textdomain( "resource" );
binddomain( "resource", "/usr/local/share");
print gettext( "hello" );
```

The domain name for the resource file is “resource”, the ID of the string to be retrieved is “hello”, the translation to be used is Canadian French, and the directory for the binary .mo files is /usr/locale/share/fr_CA/LC_MESSAGES.

Organizing Translatable Static Strings in Message Tables

Message tables mainly store static translatable strings used by PL/SQL procedures and PSPs. Sometimes, they can also be used for C/C++ programs and Perl scripts. The tables should have a language column to identify the language of static strings so that accessing applications can retrieve messages based on the locale of the user. The table structure should be similar to the one below:

```
CREATE TABLE messages
( msgid   NUMBER(5)
, langid  VARCHAR2(10)
, message VARCHAR2(4000)
);
```

The primary key for this table consists of the `msgid` and `langid` column. One good choice of the values for this column is the Oracle language abbreviations of corresponding locales. For the list of all Oracle language abbreviates, please refer to the Oracle9*i* Globalization Support Guide. Using the Oracle language abbreviation allows applications to retrieve translated information transparently by issuing a query on the message table. See below.

To provide a fallback mechanism when the translation of a message is not available, you should create the following views on top of the message table defined above.

```
-- fallback language is English which is abbreviated as 'US'.
CREATE VIEW default_message_view AS
 SELECT msgid, message
 FROM messages
 WHERE langid = 'US';
/
-- create view for services, with fall-back mechanism
CREATE VIEW messages_view AS
SELECT d.msgid,
       CASE WHEN t.message IS NOT NULL
            THEN t.message
            ELSE d.message
       END AS message
FROM default_view d,
     translation  t
WHERE t.msgid (+) = d.msgid AND
      t.langid (+) = sys_context('USERENV', 'LANG');
```

Messages should be retrieved from the `messages_view` view that provides the logic to fall back a message to English by joining the `default_message_view` view with the `messages` table. The SQL function `sys_context()` returns the Oracle language abbreviation of the locale for the current database session. This locale should be initialized to the locale of the user at the time when the session is created.

To retrieve a message, an application should use the following query:

```
SELECT message FROM message_view where msgid = 'hello';
```

The query retrieves the message in the language defined in the NLS_LANGUAGE parameter of a database session. Note that there is no language information needed for the query with this message table schema.

In order to minimize the load to a back-end database, all message tables and their associated views should be set up on a front-end database where PL/SQL procedures and PSPs runs.

Organizing Translatable Dynamic Content in Application Schema

An application schema stores translatable dynamic information, such as product names and product description, used by the application. The following shows an example of a table that stores all the products of an Internet store, and the translatable information for the table is the product description and the product name.

```
CREATE TABLE product_information
    ( product_id          NUMBER(6)
    , product_name        VARCHAR2(50)
    , product_description VARCHAR2(2000)
    , category_id         NUMBER(2)
    , warranty_period     INTERVAL YEAR TO MONTH
    , supplier_id         NUMBER(6)
    , product_status      VARCHAR2(20)
    , list_price          NUMBER(8,2)
    );
```

In order to store product names and product descriptions in different languages, the following table should be created where the primary key consist of the `product_id` and `language_id` columns:

```
CREATE TABLE product_descriptions
    ( product_id             NUMBER(6)
    , language_id            VARCHAR2(3)
    , translated_name        NVARCHAR2(50)
    , translated_description NVARCHAR2(2000)
    );
```

Also, a view is required on top of these two tables to provide fallback when information are not available in the language the user request.

```
CREATE VIEW product AS
SELECT i.product_id
,      d.language_id
,      CASE WHEN d.language_id IS NOT NULL
            THEN d.translated_name
            ELSE i.product_name
       END    AS product_name
,      i.category_id
,      CASE WHEN d.language_id IS NOT NULL
            THEN d.translated_description
            ELSE i.product_description
       END    AS product_description
,      i.warranty_period
,      i.supplier_id
,      i.product_status
,      i.list_price
FROM   product_information  i
,      product_descriptions d
WHERE  d.product_id  (+) = i.product_id
AND    d.language_id (+) = sys_context('USERENV','LANG');
```

This view performs a outer join on the `product_information` and `production_description` tables and selects the rows with the `language_id` equal to the Oracle language abbreviation of the current database session.

To retrieve a product name and product description from the `product` view, an application should use the following query:

```
SELECT product_name, product_description FROM product
      where product_id = '1234';
```

This query retrieves the translated product name and production description corresponding to the value of the NLS_LANGUAGE session parameter. Note that there is no language information needs to be specified in the query.

Summary

Oracle9*i*AS provides a reliable and open platform to develop and deploy Internet applications for a global market. The best practices described in this paper provides practical guidelines for customers to develop multilingual Internet applications by

- Following the open standards, such as Java Servlet API, Java Server Page (JSP), HTML, HTTP and Perl, and making the best use of the internationalization support of those standards.
- Considering the behaviors of industrial Internet software such as the Netscape and Internet Explorer so that applications are built for the real life environment on the Internet.
- Following the common practices used by the industry to make the applications translatable so that translation and development works can be done independently.
- Avoiding common mistakes made in the applications to support non-English languages, especially multibyte languages, on the Internet.
- Leveraging the globalization support of Oracle9*i* such as transparent character set conversion, Unicode data storage, comprehensive locales support.

With the best practices as the central guidelines and Oracle9iAS as the global platform, Internet applications supporting multiple languages becomes more reliable, extensible, maintainable, and portable.