An Overview on Globalizing Oracle PHP Applications

An Oracle White Paper
January 2004

An Overview on Globalizing Oracle PHP Applications

An Overview on Globalizing Oracle PHP Applications

INTRODUCTION

When an application is considered fully globalized it can be used in a multitude of languages and is essentially ready for deployment World Wide. From the users perspective, this means that all aspects of the interface are translated into their local language and conventions, from the developers point of view, creating a Globalized application introduces a new set of issues that need to be carefully planned from the early stages. Luckily, PHP offers all the necessary building blocks so that the developer can implement fully Globalized applications. This document provides an introduction on how to effectively use these features when deploying in the Oracle Application Server.

ASPECTS OF GLOBALIZATION

The act of globalizing your application can be broken into four distinct stages. The concepts will be introduced then the methods to achieve them will be discussed later in the document.

Data Integrity

Ensuring data maintains its meaning is perhaps the most basic but important concept for database driven applications. As data flows between user, middle tier and database without explicit manipulation, it must maintain integrity between all tiers of the application. This is especially pertinent in globalized applications where character set conversion and string expansion could take place and potentially cause data corruption under incorrect configuration. If, for example, your database is configured to store Unicode data while your middle tier application is configured only for Western European data then any non Western European data within the database will most likely end up corrupt when presented to the end user through character set conversion. The easiest way to achieve data integrity here is to ensure that between the Oracle database, PHP, and the end user browser, all layers are in agreement as to the character sets that are used to convey data and each layer performs any necessary character set conversions to support this.

String Manipulation

Any tier that needs to manipulate text data needs to have awareness of the character set that data is encoded in so that the results may be correctly interpreted. As a simple example, suppose an application is configured to accept a field whose character length was not to exceed 3 characters. Most likely there would be a string length function employed somewhere to perform this checking but without knowledge of the character set that data is in, the function cannot correctly determine how many characters are presented and hence could not correctly perform this checking. String manipulation functions typically include string length, sub-string, search, replace, and regular expression operations and all are at the heart of many middle tier applications.

Text Translation

The application should be translated into all the intended target languages. Translation at this level means transforming the base text to be something equivocally meaningful in the target language and includes translating the user interface, translating associated documentation and on-line help, and possibly translating any template text that could be used for emails or logging. The first rule in building an easily translatable product is to externalize from the source code all text that will require translation. By doing this you are separating logic from data presentation that makes the handling of translations far less complex than it would otherwise be. Externalizing messages means that translators may process and deliver files without special knowledge of your application, and your application can be configured to select the correct set of languages at run time. The development environment must provide a method to externalize text into message catalogues and a means to obtain the intended translation through identification.

Data Presentation

Runtime data may need special preparation in order to be correctly presented to the end user depending on the language they require. For example, '12th October 2003' may be presented to a user in the United States as '10/12/03' but a user in the United Kingdom as '12/10/03' and application logic will have to be aware and support this requirement.

Applications dealing with lists, such as a to do list, a list of email headers, a product list, and so on, may want to present this data to the user sorted according to the rules of their language.

Sometimes text also needs to be presented in a special format for consumption by another application or protocol. Some applications might require an encoded format such as Base-64, others might require a particular encoding such as UTF-8.

WORKING WITH PHP

PHP was originally designed to work with Western European data only and, much like C or PERL, does not directly support any of the concepts we have talked about so far. These languages do, however, usually provide an impressive set of building blocks to allow the developer to create a fully Globalized application. Because of its open source nature, PHP is free to make use of available libraries for performing globalization tasks and, much like PERL, borrows from many C APIs to perform such tasks. You should also be aware that within PHP, globalization support is only available in certain components, and even only to certain extents within those components, so it pays to know their level of support before making a commitment to using them. For example, PHP will only allow single byte Western European names for variables.

The remainder of this document aims to describe how you can build fully globalized applications in PHP (4.2.x or higher) with Oracle (9.2.x or higher) as a backend datastore using the Oracle connectivity functions as documented in "LXXIII. Oracle 8 functions" of the PHP Manual.

Establishing the Environment

Getting the right running environment for your PHP application is vital and, when done correctly, will guarantee data integrity across all tiers. First we should concentrate on the connectivity between the PHP engine and Oracle to ensure that we can insert and select data.

Most Internet based standards support Unicode as a character encoding. Unicode is a single encoding that can represent most languages of the world so it makes sense to ensure your application also supports it so that it may deploy in many languages. Given the state of the art, we are going to presume that your application will in fact be entirely based on Unicode, as there is little incentive to support legacy encodings when the entire technology stack you are using can support it.

PHP is basically an OCI application as far as Oracle connectivity is concerned, this means that all the rules that apply to OCI also apply to PHP. OCI applications make use of environment variables to control certain runtime aspects of the client, in our case, the character set. This setting is achieved through the NLS_LANG environment variable that takes the following form:

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

If our application was designed to work in Unicode for a Japanese user we would set this variable as:

```
NLS_LANG=JAPANESE_JAPAN.AL32UTF8
```

This environment variable only affects the client and the basic premise for the character set portion is that data selected from the server will be presented in the specified character set, and more importantly, data presented to the server by the client must be in this character set. A common mistake is to assume that the character set in NLS_LANG must be set to that of the database being connected to. This can be a costly mistake if the character set that the client is running in does not match that of the database character set. With web-based applications such as the one we are trying to build, we will assume that all data being presented to the database is already in Unicode. While the language and territory portions of this setting can be changed at runtime, the character set cannot so it is the only real mandatory setting here. By omitting the optional settings we can configure our application to be a Unicode application with the following:

```
NLS_LANG=.AL32UTF8
```

Where AL32UTF8 is Oracle's naming convention for UTF-8. For test purposes you can set this as an environment variable in your shell and run PHP from the command line. In your runtime application you would need to set this alongside your ORACLE_HOME and ORACLE_SID settings for the Apache user within your runtime application. Setting this variable indicates to Oracle what character set the client provided data will be in and means that Oracle can perform character set conversion for both incoming and outgoing data if required. In order to avoid the overhead of character set conversion and to minimize data loss, ensure your database character set is also AL32UTF8. For cases where your database character set is a subset of Unicode, for example Western European, then data outside that repertoire will not be stored correctly.

Data integrity will now be maintained between PHP and Oracle so all that is left to do is to ensure that it will be maintained between PHP and the end users browser. The browser looks to an HTTP header (namely Content-Type) to determine not only how best to display that content but more importantly, what character set encoding to use to send user provided form input back to PHP. We want to be Unicode everywhere here so we need to find a means of tagging all pages as being such. While Oracle uses 'AL32UTF8' to refer to Unicode, Internet standards use 'UTF-8'. There are many ways to provide this tagging in PHP but the most convenient is to set the default_charset configuration variable in your php.ini file as follows:

```
default_charset = UTF-8
```

This ensures that the following HTTP header will be set in all PHP pages and consequently sent to the browser.

```
Content-Type: text/html; charset=UTF-8
```

While it is guaranteed that the browser will only supply well-formed UTF-8 to the server, it is up to the application developer to ensure that server generated pages are indeed encoded in UTF-8. This setting does not imply any conversion of outgoing pages.

At this point you can take your application for a data integrity test drive. Try to build a form for input in PHP, take the variables as input and insert them into a VARCHAR2 column in Oracle. Prove that it works by selecting the data back out into a report. By using data not normally found in Western European character sets you can convince yourself that it really is working, try cutting and pasting content from an Arabic or Chinese website and comparing with the output. If you need further confirmation that the set up is working, test the data stored in the database using the DUMP() function to examine the byte sequence and ensure it is valid UTF-8. Note also that PHP is able to correctly decode form input that is encoded in the URL Transformation Format, before presenting them for use in your PHP application as variables. Following is a very simple example that inserts a user provided string into the database and selects out all rows, a table with a single character column is assumed:

```
<FORM METHOD=GET>
<INPUT TYPE=TEXT NAME=invar VALUE="">
<INPUT TYPE=SUBMIT VALUE="GO">
</FORM>
<?
  if (isset($invar))
  {
    $conn = ocilogon("scott", "tiger");

    $pars = ociparse ($conn, "INSERT INTO t1
                              VALUES ('$invar')");
    ociexecute($pars);

    $pars = ociparse ($conn, "SELECT c1 FROM t1");
    ociexecute($pars);

    ocifetchstatement($pars, $res);

    for ($i = 0; $i < $nrow; $i++)
    {
      $var = $res["C1"][$i];
      echo "value: ($var)<BR>";
    }
  }
?>
```

It is important that any text you encode in your script be encoded in Unicode. For example, if you have a table name in Oracle that uses characters beyond the ASCII set, you must use UTF-8 encoding in SQL statements otherwise Oracle will throw an error.

Working the Data

Once the correct data flow is established it is important to identify functions available that understand Unicode and can help us manipulate and format our data. As mentioned earlier, PHP was originally designed to work natively with Western European data, specifically the ISO-8859-1 character set, and will not normally return expected results when dealing with other character sets, particularly those that do not encode each character in one byte. To solve this limitation, a set of functions made available in PHP 4.0.6 and documented under the 'Multi-Byte String Functions' section in the PHP manual provide string manipulation functions that support many character sets. Of particular interest to us, PHP also supports Unicode.

To use the multi-byte string feature ensure it is enabled in PHP by compiling with the `--enable-mbstring` configure option. You can check if support is already compiled in by examining the output of phpinfo() in a script or by running 'php -m' on the command line. If you do not have control over your PHP installation then you are out of luck; ask nicely that your administrator include it. The following setting in the php.ini file is all it takes to start deploying applications in Unicode:

```
mbstring.internal_encoding=UTF-8
```

When this is set all mbstring functions will assume the text you are providing them will be in UTF-8 and as we have already made sure this will be the case when we established our environment, we are ready to begin coding. To prove to ourselves that it is working, amend the FOR loop in the previous example to read as follows:

```
for ($i = 0; $i < $nrow; $i++)
{
  $var = $res[“C1”][$i];
  echo “value: ($var)<BR>”;
  echo "strlen: " . strlen($var) . "<BR>";
  echo "mb_strlen: " . mb_strlen($var) . "<BR>";
}
```

Given the word “résumé” as input, observe the following output:

```
value: (résumé)
strlen: 8
mb_strlen: 6
```

Notice that the normal string function returns 8 which is actually a byte count as each é character is encoded in 2-bytes in UTF-8. The real string length is correctly calculated by mb_strlen as it knows that the 2-bytes of é make up just one single character. It is also possible to overload the behaviour of the standard string manipulation functions, such as strlen(), so that they work in terms of UTF-8 with the following configuration:

```
mbstring.func_overload=7
```

Both functions in our example would know the data is in Unicode and the output would then become:

```
value: (résumé)
strlen: 6
mb_strlen: 6
```

If you do decide to overload the standard functions, be warned that it will change the documented behaviour and may return unexpected results for the basic string functions. If you are using `strlen()` to obtain the byte length of a user provided string to determine, for example, whether it will fit within a `CHAR(20)` in your database, you probably don't want to overload `strlen()` to work in terms of characters. It may be a good idea to migrate your function calls that are supposed to work in terms of characters to their mbstring equivalents and leaving `mbstring.func_overload` well alone.

It should be noted that many of the string manipulation features available in PHP are also available in Oracle. Oracle has the added benefit of being fully globalized and all function calls work in terms of the current database character set without extra configuration. An array of sort orders is also provided that is configurable by setting NLS_SORT when data is selected from a table. It can often be to your advantage to perform these routines within the database so your middle tier logic can focus on presentation.

Regular Expressions are also worth a mention as they are perhaps the single most useful means to manipulate and search text data. Currently the mbstring implementations of regular expressions are in Beta meaning you are discouraged from using them in production code. If you are running on Oracle Database 10g, however, regular expressions are available in the SQL REGEXP_LIKE, REGEXP_SUBSTR, REGEXP_INSTR, and REGEXP_REPLACE functions.

It should be noted that while PHP character datatypes are fully interoperable with Oracle's CHAR, VARCHAR2, LONG, and CLOB character datatypes, full support for accessing NCHAR, NVARCHAR2, and NCLOB is currently not provided. When selecting or inserting data into NCHAR datatypes, implicit conversion to their CHAR counterparts takes place. This means that unless the character set of your NCHAR datatypes (as defined by the national character set) is a superset of the character set of your CHAR datatypes (as defined by the database character set), data loss will occur.

Externalizing Text

As mentioned earlier, one of the key areas to globalizing your application is that it is translated into all the required languages. As a means to translate user interface text, we must start by externalizing such text from the application logic and place it somewhere where it can be easily retrieved on a per language basis.

In PHP, there is no one right way to do this, rather, there are many options available to you. The GNU gettext functions are available and allow you to externalize text for translation using a flat file format for messages. The details of these functions are well documented and you can read more at the GNU website. Another good way to externalize text is to look into a template based solution, this is ideal if you have a large amount of static HTML text that you wish to translate. Other potential solutions include using Oracle tables and SQL queries to store and retrieve text.

You may find that a combination of the above is ideal depending on the type of data being translated, whether it is online help pages, UI labels, error messages, or other text.

Bringing it all together

Now that we have data flowing back and forth in Unicode and we have all our text externalized and translated into the different languages that we are going to support, we need to bring it all together to the point where we can deploy our multilingual application.

The first consideration to make is how are you going to determine the language of the user so that you may present them with text in their language. There is a multitude of different ways to do this depending on the model of your application. You may require the user to login and maintain user preferences where the user could identify their native language or you may provide an interface of the application itself where any user can identify their language and that preference is stored in a cookie without necessarily requiring the user to login. Perhaps the simplest approach to this problem is to honor the accept-language preference that is sent by the browser with every request. In its most simplest form, the header looks like this:

```
Accept-Language: language[-territory]
```

Territory is optional and is useful for defining variations on languages such as 'fr-CA' for French as spoken in Canada or 'zh-TW' for Chinese as spoken in Taiwan. This value is stored in the ‘HTTP_ACCEPT_LANGUAGE’ Apache environment variable and you will need to determine how best to parse this value. This approach does have several drawbacks though, often users do not set their language preference in the browser, and in many cases, the user is not allowed to make such settings such as in an internet café of airport lobby. It is always best to have an application level user language and territory preference available.

Once you have determined the preferred language it should be an easy task to obtain the translated resource, be it an externalized flat file, PHP array element or column value from the database, the main challenge here is ensuring that you have a means to translate the tag used to indicate the language preference to that used to tag the translated file so some consideration is needed in this area.

Oracle also offers several features that help to refine the presentation of data when the language preference is known. First set the language with the following:

```
ALTER SESSION SET NLS_LANGUAGE=<language>
```

It should be noted that the Oracle convention is required for the language tag. Setting the language will return translated Oracle error messages, return translated date formats such as day and month name, and will set a default sort order applicable to that language. Note that PHP itself is not a globalized application so error text or logging data will always be in English. Next set the territory:

```
ALTER SESSION SET NLS_TERRITORY=<territory>
```

This will assign a default date format and will format numeric values for the local conventions, such as using a comma or a period for the thousands separator. In depth discussion of these parameters is beyond the scope of this document so refer to the Oracle Globalization Guide for more details.

CONCLUSION

This paper aims to give a quick introduction to some of the issues you will face as an application developer when creating a fully globalized application in PHP and Oracle. It is not intended to be a complete guideline and several major issues, such as how to design an application that can render HTML pages suitable for bi-directional languages, are beyond the scope of this document, as they do not directly pertain to PHP or Oracle.

There are also other subtle character set related issues, such as which character set to use should your application need to send mail, which are not discussed in detail but it is worth mentioning that PHP does offer APIs for this particular issue.

Like most contemporary languages, PHP is a quite dynamic and changes often as new features and packages are added. It is important to keep up to date with changes especially in the area of Globalization support that is typically improved release by release.

Globalized Oracle PHP Applications
January 2004
Author: Peter Linsley
Contributing Authors:

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

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