

PHP / MySQL SPECIALISTS!

Simple, Affordable, Reliable PHP / MySQL Web Hosting Solutions

POPULAR SHARED HOSTING PACKAGES

MINI-ME \$6⁹⁵ /mo

500 MB Storage
15 GB Transfer
50 E-Mail Accounts
25 Subdomains
25 MySQL Databases
PHP5 / MySQL 4.1.X
SITEWORX control panel

SMALL BIZ \$21⁹⁵ /mo

2000 MB Storage
50 GB Transfer
200 E-Mail Accounts
75 Subdomains
75 MySQL Databases
PHP5 / MySQL 4.1.X
SITEWORX control panel

POPULAR RESELLER HOSTING PACKAGES

NEXRESELL 1 \$16⁹⁵ /mo

900 MB Storage
30 GB Transfer
Unlimited MySQL Databases
Host 30 Domains
PHP5 / MYSQL 4.1.X
NODEWORX Reseller Access

NEXRESELL 2 \$59⁹⁵ /mo

7500 MB Storage
100 GB Transfer
Unlimited MySQL Databases
Host Unlimited Domains
PHP5 / MySQL 4.1.X
NODEWORX Reseller Access

interWORX: CONTROL PANEL

All of our servers run our in-house developed PHP/MySQL server control panel: **INTERWORX-CP**

INTERWORX-CP features include:

- Rigorous spam / virus filtering
- Detailed website usage stats (including realtime metrics)
- Superb file management; WYSIWYG HTML editor

INTERWORX-CP is also available for your dedicated server. Just visit <http://interworx.info> for more information and to place your order.

WHY NEXCESS.NET? WE ARE PHP/MYSQL DEVELOPERS LIKE YOU AND UNDERSTAND YOUR SUPPORT NEEDS!

NEW! PHP 5 & MYSQL 4.1.X



We'll install any PHP extension you need! Just ask :)

PHP4 & MySQL 3.x/4.0.x options also available



128 BIT SSL CERTIFICATES

AS LOW AS \$39.95 / YEAR 

DOMAIN NAME REGISTRATION

FROM \$10.00 / YEAR

GENEROUS AFFILIATE PROGRAM
**UP TO 100% PAYBACK
PER REFERRAL**

30 DAY
MONEY BACK GUARANTEE

FREE DOMAIN NAME
WITH ANY ANNUAL SIGNUP

**ORDER TODAY AND GET 10% OFF ANY WEB HOSTING PACKAGE
VISIT [HTTP://NEXCESS.NET/PHPARCH](http://nexcess.net/phparch) FOR DETAILS**

Dedicated & Managed Dedicated server solutions also available

Serving the web since Y2K

Solving the Unicode Puzzle

by Michael Toppa

Many web sites cannot correctly interpret or display anything other than English language characters. Converting your site to UTF-8 (Unicode) enables you to handle characters from almost any language in the world. However, currently available conversion guidelines typically focus on just a single software product, offering little guidance on how to move UTF-8 encoded data between different products. Configuring your web server, PHP, and your database to support UTF-8 is one thing—configuring them so UTF-8 encoded data moves smoothly between them is another. This article guides you through a UTF-8 conversion using PHP, Oracle, and Apache. It also covers data exports to PDF, RTF, email, and plain text.

Unicode is a single character set designed to include characters from just about every writing system on the planet (and off the planet—even Klingon has been written for Unicode, although it is not part of the official standard). In recent years, Unicode has become more prevalent on the web, and all major web browsers, web servers, programming languages, and databases worth their salt now support it. Switching your web applications to Unicode will give you the ability to correctly handle and display any character from any language you're likely to encounter.

Understanding the significance of Unicode requires first understanding some basics of character sets, and their history. The first thing you need to know was said best by Joel Spolsky of *Joel On Software*: "There ain't no such thing as plain text." If you don't know the character set and the encoding that were used in the creation of a string of text, then you won't know how to display it properly. For modern purposes, the story of character sets starts with ASCII. In the 1960s, unaccented English characters, as well as various control characters for carriage returns, page feeds, etc., were each assigned a number from 0 to 127; there was general agreement on these number assignments, and so ASCII was born. The ASCII characters could fit in 7 bits, and computers

used 8-bit bytes, which left an extra bit of space. This led to the proliferation of hundreds of different character sets, with each one using this extra space in a different way. The characters from 0-127 are often referred to as Lower ASCII, and the characters from 128-255 as

REQUIREMENTS



PHP	4.3.10 or higher
OS	Any
Other Software	Oracle 9, Apache, PDFLib
Code Directory	n/a

REFERENCES



UNICODE	http://www.unicode.org/
UNICODE	http://www.alanwood.net/unicode/
ORACLE	http://www.oracle.com/technology/tch/opensource/php/globalizing_oracle_php_applications.html
PHP	http://us3.php.net/manual/en/ref.mbstring.php

Upper ASCII or Extended ASCII. Extended ASCII character sets added characters from non-English languages, special characters like copyright symbols, and line-drawing characters to simplify drawing boxes, etc. With all these different versions of extended ASCII floating around, text generated on, say, a computer in Russia would turn into gibberish if you tried to read it on a computer in the US. This happened because the number codes representing the Cyrillic characters were assigned to totally different characters on the US computer. This became a bit of a problem when everyone started using the internet.

Unicode represents an effort to clean up this mess. The Unicode slogan is: “Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.” Unicode can do this because it allows characters to occupy more than one byte, so it has enough room to store characters from languages around the world—even Asian languages that have thousands of characters. With Unicode, it’s particularly important to understand the distinction between a character set, and character encoding. Unicode is a single character set, but there are three different ways to encode it: they are called UTF-8, UTF-16, and UTF-32 (there’s also UTF-7, but it was never officially adopted by the Unicode Consortium, and for the most part it’s been deprecated in favor of UTF-8). The numbers 8, 16, and 32 indicate the bits used for the Unicode code units (a complete character may occupy more than one code unit—it can be multi-byte). All three encodings can display any Unicode character, and each has its own advantages and disadvantages depending on what’s important in a particular implementation. In the case of web applications, UTF-8 is the encoding of choice because it stores the lower ASCII characters in a single byte format. This makes UTF-8 fully compatible with “plain text,” even if you’re clueless about character encoding.

For the sake of brevity, I’ve glossed over a great number of points related to Unicode and character sets. If you want to learn more, I highly recommend the article *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)* by Joel Spolsky, at www.joelonsoftware.com/articles/Unicode.html. It contains links to a number of other good resources as well.

Why Care About Unicode?

As far as Unicode and UTF-8 are concerned, all web sites can be placed in one of three categories: those that don’t need to care about them, those that should convert to UTF-8, and those that should convert to UTF-8 and internationalize.

The most common character set currently in use on the English-speaking side of the web, other than UTF-8, is Western ISO-8859-1 (aka Latin-1). If your site isn’t

already using UTF-8, then you’re probably using Latin-1. If you’ve had no problems related to character sets so far, and you have absolutely no foreseeable needs to handle text outside the ASCII range, then you fall into the first category: you probably don’t need to do anything. As you’ll see in the rest of this article, converting to UTF-8 is not a painless process, so you should only undertake the work if you have some clearly identifiable, relevant goals to meet.

Here at the University of Pennsylvania School of Medicine, we fall into the second category: our web sites are in English, but we occasionally handle data from a variety of foreign languages that don’t use the English alphabet. We must receive, store, display, and transmit these characters faithfully. Since we can’t reliably predict what sort of characters might come our way, converting our applications to UTF-8 was the logical choice, since it can handle any language we might need to support.

The third category is for sites that don’t just occasionally handle foreign characters—they actually serve an international audience. In addition to using UTF-8, these sites typically employ various mechanisms that allow visitors to choose the language for displaying content. One important term applied here is *internationalization*, defined by the W3C as “[t]he process of designing, creating, and maintaining software that can serve the needs of users with differing language, cultural, or geographic requirements and expectations” (see <http://www.w3.org/TR/ws-i18n-scenarios/>). Another key term is *localization*: “[t]he tailoring of a system to the individual cultural expectations for a specific target market or group of individuals.” Sites that are able to dynamically perform localization for a variety of target audiences can do so because they’ve been configured with a good internationalization framework.

Internationalization and localization are substantial topics, and are not the focus of this article. However, getting all the various components of your web application environment to place nicely together using UTF-8 is a necessary step before you can even try internationalizing your site. So this article will be of interest to those who only want to handle the occasional non-English characters, and to those who are contemplating fully internationalizing their site.

Getting Ready for UTF-8

The first step is determining the scope of your work. At a minimum, you probably have PHP, a web server, and a database to consider. I’ll cover doing a UTF-8 conversion with PHP, Apache, and Oracle. If you are also using Oracle, then you must read *An Overview on Globalizing Oracle PHP Applications* at http://www.oracle.com/technology/tech/opensource/php/globalizing_oracle_php_applications.html. It’s an excellent starting point, but, unfortunately, it doesn’t always explain the reasons

behind its recommendations, which means you'll get stuck if things don't happen to work after you follow its instructions. I'll try to fill those gaps.

You also have to take a look at any other applications that interact with PHP, your web server, or your database, as they will also be affected by a character set conversion. For us, that included Smarty, PDFlib, and exporting data to RTF, text files, and email, so I'll discuss those as well. Even if you have a different mix of applications, the concepts I'll describe are probably applicable to your situation, although the implementation specifics, obviously, will be different.

Configuring Apache, PHP, and Oracle

Most of the time, PHP web applications are run under the Apache web server, which itself is running in a user account (assuming you're in a Unix-ish environment). So, the first step is to set the environment of this

and your database all to UTF-8. This will save you the headache of translating character encodings as you move data around.

NLS_LANG is not the end of the story. It applies to the communication between PHP and Oracle, but it doesn't determine how characters are encoded within PHP, and it doesn't influence how documents are served by Apache. There are a few different approaches to consider for having Apache and PHP serve your web pages in UTF-8.

If you want all of the documents on your server to default to UTF-8, one option is to set the **AddDefaultCharset** directive in the Apache configuration to UTF-8. Note, however, that the Apache documentation at <http://httpd.apache.org/docs-2.0/mod/core.html> does not express enthusiasm about this approach: "**AddDefaultCharset** should only be used when all of

**“Switching your web applications to Unicode
will give you the ability to correctly handle and display
any character from any language you're likely to encounter.”**

account correctly. Since PHP and Oracle are speaking to each other through this account, it's crucial to specify the right character set for it, so they both know what to expect. You do this by setting the **NLS_LANG** environment variable in the Apache configuration. The Oracle *Overview* document mentioned above says to set it to **.AL32UTF8**, but doesn't fully explain why. So when this didn't do the trick for me, I had to do some more research. I looked up the Oracle Character Set descriptions and learned that **.AL32UTF8** corresponds to Unicode 3.1. After talking with our DBA I learned that our Oracle database was set to Unicode 3.0, which meant I needed to set **NLS_LANG=.UTF8**. Note that we ultimately switched to **.AL32UTF8**, since it corresponds to the latest version of Unicode, and in Oracle it allows for conversion between UTF-16 and UTF-8 (just in case you ever need to do that). The moral of the story is that **NLS_LANG** should exactly match the character set you're using in Oracle.

What I just said contradicts the advice of the Oracle *Overview* document, where it says **NLS_LANG** should be set to match the client (in this case, PHP) but that it doesn't need to match the database character set. That's technically true, but a mismatch will quickly lead to trouble if, for example, you try to insert records from PHP that are in an encoding that's not compatible with the Oracle character set. If you're going to switch to UTF-8, do it wholeheartedly: set PHP, your web server,

the text resources to which it applies are known to be in that character encoding and it is too inconvenient to label their charset individually. One such example is to add the charset parameter to resources containing generated content, such as legacy CGI scripts, that might be vulnerable to cross-site scripting attacks due to user-provided data being included in the output. Note, however, that a better solution is to just fix (or delete) those scripts...

If you want all of your PHP-generated content to be served in UTF-8, set **default_charset=UTF-8** in your **php.ini** file. It's OK if the PHP **default_charset** is different from what's specified in Apache **AddDefaultCharset**: the former will apply only to PHP files, and the latter will apply to everything else.

If you want *some* (but not all) of your PHP documents served in UTF-8, you don't have to modify **php.ini**. Instead, specify UTF-8 as the character set in the **Content-type** header of those files. It's important to point out here that you should set this header with the PHP **header()** function. If you try to set it with an HTML Meta tag, and you've used Apache's **AddDefaultCharset** directive to specify a different character set, the Apache directive will override your Meta tag.

Now that you've configured how you want documents served, you need to configure PHP so it can internally handle UTF-8. This means enabling multi-byte character support. You'll need to re-compile PHP

with the `-enable-mbstring` option (unless, of course, you had the foresight to do it previously), and set `mbstring.internal_encoding=UTF-8` in your `php.ini` file.

Look over the PHP documentation for multi-byte string functions at <http://www.php.net/ref.mbstring>. Many of the PHP string functions have multi-byte equivalents. An example is the best way to illustrate what this means. The multi-byte version of `strlen()` is `mb_strlen()`. The `strlen()` function assumes that a character always occupies a single byte, so it actually returns the length of a string in bytes, and does not necessarily indicate the number of characters. In UTF-8, though, a string that is 4 characters long could occupy anywhere from 4 to 24 bytes depending on the presence of multi-byte characters. The `mb_strlen()` function will correctly tell you the number of characters in such a string, but the regular `strlen()` function won't.

Because of all this, you should consider enabling PHP's *function overloading* feature, described at <http://php.net/ref.mbstring#mbstring.overload>. Activating function overloading will cause PHP to automatically assume it's handling multi-byte strings, so—continuing with the example—it will actually execute `mb_strlen()` when you call `strlen()`. If you're making a wholesale conversion to UTF-8, and you don't want to revise all of the string function calls in your existing code, implementing function overloading makes sense. But there are a couple of caveats:

Watch out for calls to `strlen()` (or any other string function) where it really is intended to work with the byte length, not the character length. In that situation, function overloading will end up giving you an unintended result. Fortunately, there is a workaround for `mb_strlen()`: it accepts a character set specification as a second argument and if you pass in 'latin1' (even though it's actually handling a UTF-8 string). This will cause the string to be evaluated as if it were single-byte encoded. `mb_strlen($your_utf8_string, 'latin1')` will give you the number of bytes in a multi-byte string.

You may not want to do function overloading on `mail()`. I'll explain why in the discussion of email below. Note that if you haven't upgraded to PHP 5, the `html_entity_decode()` function will return an error if you pass it a UTF-8 string. This was the only UTF-8 incompatibility we found in PHP 4.3.

Going back to Oracle, starting with Oracle 9i, it provides improved handling for multi-byte characters by giving you a way to distinguish between byte length and character length. When creating a table, you can specify whether its length is defined in terms of characters or bytes. For example, `VARCHAR2(20 BYTE)` will give you a 20-byte length field, and `VARCHAR2(20 CHAR)` will give you a 20-character length field. The default is `BYTE`, which you can alter with the `NLS_LENGTH_SEMANTICS` parameter—see your Oracle documentation for more details.

Beware Windows-1252 in Web Forms

As I mentioned, other than UTF-8, the character encoding you're most likely to find on English-speaking web sites, these days, is Latin-1 (aka Western ISO-8859-1). One of the nice things about UTF-8 is that the first 256 characters are the same as in Latin-1. That is, the Latin-1 ASCII characters and its Extended ASCII characters live in the same numerical locations in UTF-8. If you're currently on Latin-1, this greatly eases the pain of switching to UTF-8.

So, the big "however" comes from—you guessed it—Windows. Fortunately, Windows NT, 2000, and XP use Unicode internally and shouldn't cause headaches for a UTF-8 web site. But Windows 95 and 98 use the Windows-1252 character set. Its standard ASCII characters from 0-127 are the same as Latin-1 and UTF-8, but its Extended ASCII set is different. If you have a form on a web page that's UTF-8 encoded, and someone running Windows 9x fills out the form by copying-and-pasting text from Microsoft Word, Extended ASCII characters may be interpreted properly. You may have experienced this before: for example, the "©" symbol in your Word document turned into something like "ä" when you pasted it into a form. Nothing about the character's underlying data changed—the decimal representation of the character is the same as it was before—it just means something different in UTF-8 than it does in Windows-1252.

This was more of a problem in the past than it is now, as modern browsers try to transparently perform a character set conversion for you as needed in these situations. But the problems are by no means entirely resolved: see *FORM submission and i18n* at <http://ppewww.ph.gla.ac.uk/~flavell/charset/form-i18n.html> for a thorough overview of all the issues related to this, as well as a rundown of how the major browsers behave (if you're wondering about the meaning of *i18n*, it's short-hand for internationalization).

What makes this a truly maddening problem is converting a Latin-1 encoded database to UTF-8 when some of the data in it came from Latin-1 encoded web forms where users pasted in Windows-1252 text, and their browsers didn't convert the characters properly. There is no easy fix for this, as you simply have to look at the records yourself to see if the Extended ASCII characters are displaying as the user intended, or if there was a character set conversion problem along the way.

UTF-8 Support in Smarty

Smarty handles UTF-8 transparently—almost. The one trouble spot is the `escape` modifier. It calls the PHP `htmlentities()` and `htmlspecialchars()` functions, but it doesn't provide them with the necessary `charset` argument so they'll work with UTF-8. The solution is to

override `escape` with your own custom version. Start by making a copy of the Smarty `escape` modifier, and tweak it to pass along a `charset` argument to PHP. Then override the original with your custom version. If you won't always be using UTF-8, set your custom version to accept a `charset` argument, so you can adjust the functionality as needed. Look up the "Extending Smarty with Plugins" section of the manual on the Smarty site—[<http://smarty.php.net/>—for instructions on how to customize Smarty.

Exporting UTF-8 Data to PDF, RTF, Plain Text, and Email

It may not always be wise, or even possible, to keep data encoded in UTF-8 when exporting to other formats. As you'll see below, sometimes you need to change the character set before performing the export. Take a look at PHP's `utf8_decode()` and `iconv()` functions to learn about converting UTF-8 to single-byte encoding. Note that `utf8_decode()`, while easy to use, is limited to the Latin-1 character set (see the user contributed notes on the PHP `utf8_decode()` page for tips on dealing with other character sets).

Our applications require exporting data to PDF, RDF, text files, and email:

To generate PDF, we run the PDFlib application on our web server to create PDF documents on the fly. PDFlib is an application specifically designed for processing PDF data and dynamically generating PDF documents—you can learn more about it at <http://www.pdflib.com/>. For it to work with UTF-8 data, you need to use it with a UTF-8 compatible font. The commonly used Windows TrueType fonts—Arial, Times New Roman, and Courier New—are Unicode compliant. However, that doesn't mean they can display any Unicode character. They are fine for English and most Central and Eastern European languages. For more on this, see the Font section of *Alan Wood's Unicode Resources* at <http://www.alanwood.net/unicode/>. It's important to mention Microsoft's Arial Unicode MS font, which is not the same as the standard Arial font. Arial Unicode MS can display characters from Arabic, Tamil, Thai, Hangul, Chinese, and many other languages. This means the font itself is huge: approximately 23Mb. If you try to use it with PDFlib running on your web server, you may run into performance problems.

If you are using, for example, Microsoft Word, it's easy to take a Unicode document and save it as an RTF file. It's also not difficult to use a tool like RTF File Generator (available at <http://www.paggard.com/projects/rtf.generator/>) to generate RTF files using PHP, as long as the source data does not include characters from multiple languages. It turns out to be quite difficult to use PHP to generate an RTF file when the source data is UTF-8 encoded and

contains characters from several different languages. This is because RTF requires you to specify a character set for displaying the characters, and you can't just say "Unicode." You have to specify one or more ANSI, PC-8, Mac, or IBM PC character sets. This means you must analyze the multi-byte characters in a UTF-8 string and figure out what characters they represent. Then you need to specify in the header of the RTF file what character sets are needed to display them: a Hebrew character set for Hebrew characters, Arabic for Arabic, etc. Then in the body of the file you must flag the various chunks of non-English text and indicate which of these character sets are needed to display them. Rather than attempting this Herculean task, our solution is to do a `utf8_decode()` on our data before generating RTF files, so that the text is all in Latin-1. At the moment we can get away with this since none of the data going into the RTF files we currently generate contain non-English characters. We are planning to eventually discontinue our RTF support, so this will not be a long-term problem. Acquiring an understanding of how RTF works with Unicode data was difficult—of all the applications

“Unicode allows characters to occupy more than one byte, so it has enough room to store characters from languages around the world.”

we encountered in this project, RTF was the least well documented when it came to Unicode.

We export data to text files, primarily in `.csv` format for use in spreadsheets. Surprisingly, current versions of Microsoft Excel do not support importing UTF-8 encoded text files. As with RTF, our solution is to perform a `utf8_decode()` before generating these text files. This doesn't pose any problems for us since the kind of data we put in spreadsheets does not contain any non-English characters.

As I mentioned, I do not recommend doing function overloading on the PHP `mail()` function. The reason has to do with line breaks. In Unix, a line break is represented by a line feed (`LF`, or `\n`) character, on Macs, it's represented by a carriage return (`CR`, or `\r`) character, and on Windows, by a `CR+LF` (`\r\n`). For email to work between platforms, an email standard was agreed upon in the early days of the internet, which is `CR+LF`. So, for example, on Unix, `sendmail` will add a `CR` as

needed to each LF it finds in the body of an email message. But when an email is UTF-8, PHP will first base64 encode it before passing it off to sendmail. This encoding is done so that multi-byte UTF-8 characters can be transported within the 7-bit world of email (for more about this, see *Advanced E-mail Manipulation* by Wez Furlong, *php|architect* Vol. 3, Iss. 5). Sendmail and other mailers do not attempt to wade through the base64 encoding to “fix” the line breaks. Unless you’re careful to put **CR+LF** line breaks in all your PHP generated emails before sending them, you’ll end up sending emails with improper line breaks. This can have unpredictable results, as you’re at the mercy of the recipient’s email client software, and what it chooses to do with malformed line breaks. In our testing, we found that the **LF-only** line breaks in our UTF-8 encoded emails were interpreted as desired in Mac and Unix mail readers, and by Microsoft Outlook on Windows, but not by Eudora 6.2 (and previous versions) on Windows. In Eudora, the messages displayed with no line breaks at all. You can’t say it’s a Eudora bug, since the line breaks weren’t meeting the standard. At this time, the emails we generate only contain basic English characters, so sticking with the standard `mail()` function meets our needs for now.

The Bumpy Road to Unicode Compliance

As you can see, converting your web site to UTF-8 is by no means a painless process. But the payoff is worth it if you plan to support characters from several languages. It’s also a fascinating educational experience: you’ll gain a stronger understanding of how Apache, Oracle, and PHP interact, how Unicode supports so many different languages, some of the gory details of how email works, how browsers deal with mismatching character sets, what a Unicode compliant font is, and much more. Even if you’re not using the same software discussed in this article, hopefully I’ve at least imparted a sense of what kinds of problems you should look out for. If nothing else, hopefully you’ll remember, “there ain’t no such thing as plain text.”

About the Author

?>




*Michael Toppa is a web applications developer at the University of Pennsylvania School of Medicine. He has previously worked for Ask Jeeves, E*TRADE, and Stanford University Libraries' HighWire Press. He can be found on the web at www.toppa.com. Credit for a lot of the research in this article goes to all of the U Penn School of Medicine Web Development team.*

To Discuss this article:




<http://forums.phparch.com/219>


php|architect

Certification Central

Certification Training



Live Training Everyone Can Afford!

Available Right At Your Desk
All our classes take place entirely through the Internet and feature a real, live instructor that interacts with each student through voice or real-time messaging.

What You Get
Your Own Web Sandbox
Our No-hassle Refund Policy
Smaller Classes = Better Learning

Curriculum
The training program closely follows the certification guide—as it was built by some of its very same authors.

Sign-up and Save!
For a limited time, you can get over **\$300 US in savings** just by signing up for our training program!

New classes start every three weeks!
<http://www.phparch.com/cert>