WEB PROGRAMMING

PHP Add a spell-checker

Paul Hudson's threat to replace Rebecca with a very small shell script finally comes true.

Long-time readers of this magazine will know that spelling is not our forte. In fact, you would be forgiven for thinking that our production monkeys have spent the last few months on strike over a raise in their peanut rations, as typos that even OpenOffice.org could have spotted have flown into print. However, we remain a cut above the rest: we rarely say "w00t!", we do try to avoid "Micro$oft", and to the best of my knowledge we have never said anything was "the suXX0rs" no matter *how* negative a review was.

This month we shall examine the ways in which PHP can help proof – and correct – written English on your site. If you found last month's tutorial on validating visitors mathematically challenging, this month is going to be a curve ball to the other hemisphere of your brain as we investigate how phonetics actually have a valuable part to play outside of the classroom.

Spill chucking

Early adopters of mobile phone SMS messages developed their own method of chatting, primarily designed to minimise the amount of thumb use it took to write "see you later". So – much to the disgust of English prescriptivists everywhere – "c u l8r" was born. However, to the mystification of all, this trend reversed on the internet: the same message encoded in messageboard-speak is "OMG LOL! CU L8R! WTF?!" That, admittedly, is beyond correction by even the most advanced computer system; however, correcting errors like spelling Red Hat with two Ts, using percieve rather than perceive and missing out letters entirely is within our grasp.

You will need to recompile PHP to add in spell-checking support. To do this, add **--with-pspell** to your configure line. You will also need *GNU Aspell* installed for your language (including development libraries), as this is the back-end system that PHP uses.

When you have recompiled, run **php -m** and make sure **pspell** is in the list of compiled-in modules. With that over with, we can try our first pspell script using two functions: **pspell_new()**, and **pspell_suggest()**. The first creates a new **pspell**-processing resource for a given language, and the second uses that resource to check for a word, like this:

```
<?php
  $spell = pspell_new("en");
  if (pspell_check($spell, "orange")) {
    echo "Word spelt correctly!\n";
  } else {
    echo "Bad spelling, sorry!\n";
  }
?>
```

If English is not your primary language, change the "en" to your language code, eg "es" for Spanish, "de" for German, or "no" for Norwegian. The return value from **pspell_new()** is stored in **$spell**, because it's needed for the first parameter of **pspell_check()**. The second parameter to **pspell_check()** is the word you want to check, in this case "orange". You need to pass in the language to check each time you call **pspell_check()**, which is a chore if you are only using one language, but makes scripts like this easier:

```
<?php
  $english = pspell_new("en");
  $deutsch = pspell_new("de");
  $word = "alles";
```

```
  if (pspell_check($english, $word)) {
    echo "Word is in English!";
  } else {
    if (pspell_check($deutsch, $word)) {
      echo "Word is in German\n";
    } else {
      echo "Is it Welsh?";
    }
  }
?>
```

Just telling a user they made a spelling mistake is useless, unless they happen to be one of those weird people who keep a dictionary by their PC. What we should be doing is finding suggestions for them, a task accomplished with the **pspell_suggest()** function. This takes the same parameters as **pspell_check()** and returns an array of words that matches the word passed in. So, we can take our script and modify it to provide suggestions for our word, like this:

```
<?php
  $spell = pspell_new("en");
  $word = "naranja"
  if (pspell_check($spell, $word)) {
    echo "Word spelt correctly!\n";
  } else {
    $suggestions = pspell_suggest($spell, $word);
    if (count($suggestions)) {
      echo "Did you mean...\n";
      foreach($suggestions as $suggestion) {
        echo "   $suggestion\n";
      }
    } else {
      echo "Bad spelling, sorry!\n";
    }
  }
?>
```

That script snags the return from **pspell_suggest()**, then iterates over the array in a foreach loop, printing out the suggestions as it goes. This works well, but it's still fairly useless because it only checks one word. Ideally we want it to check a whole sentence at a time, which means we need to break it up into words by exploding the string into an array wherever spaces occur.

```
<?php
  $spell = pspell_new("en");
  $sentence = "The rain in Spain fallls manily on the
Spaniards.";
  $words = explode(" ", $sentence);
  foreach($words as $word) {
    if (pspell_check($spell, $word)) {
      echo "Word spelt correctly!\n";
    } else {
      $suggestions = pspell_suggest($spell, $word);
      if (count($suggestions)) {
        echo "Did you mean...\n";
        foreach($suggestions as $suggestion) {
          echo "   $suggestion\n";
        }
      } else {
        echo "Bad spelling, sorry!\n";
      }
    }
  }
?>
```

There are still some problems with that code – namely that our sentence is split by spaces, which will leave 'Spaniards' with a full stop at the end. This full stop is considered to be part of

the word for **pspell_check()**, which means it will fail the test. Try writing your own script to strip punctuation out, keeping in mind that if someone types with dashes in a line – like this writer frequently does – those dashes will appear as words by themselves and should not be checked.

Hooked on phonics

Leaving spell checks alone for now (we'll come back to them later), PHP has several ways to help you analyse the contents of a string by the letters it contains, rather than the usual functions **strlen()** and friends. The most important of these are **similar_text()**, which is a simple letter-matching algorithm for comparing strings, and **metaphone()**, which is a more complex system for calculating the sound that a string makes when it's pronounced.

The **similar_text()** function is easier to grasp, so we shall start there. This takes a minimum of two parameters (the string to compare), and returns the number of letters that are the same in both strings. The algorithm behind **similar_text()** is smart enough to handle differences between individual letters, for example:

```
<?php
  echo similar_text("hyperbolic", "parabolic");
?>
```

Running this will output 7: it matches the 'p', 'r', and 'bolic' in the words. However,

```
<?php
  echo similar_text("rhypebolic", "parabolic");
?>
```

will only return 6, because the letters of 'hyperbolic' are not in the correct order.

There is an optional third parameter to **similar_text()** that stores a percentage of the difference between the two words. This percentage is usually very long, so take care to round it before printing it out, like this:

```
<?php
  $diffchars = similar_text("hyperbolic", "parabolic",
$diffpercent);
  $diffpercent = round($diffpercent, 2);
  echo "There were $diffchars different characters, making
$diffpercent% difference.\n";
?>
```

The **metaphone()** function uses a basic understanding of English pronunciation to approximate how a word sounds when spoken, then encapsulates that in a string. This allows you to distinguish between words that have different spellings but the »

same sounds, eg:

```
<?php
  $pair = metaphone("pair");
  $pear = metaphone("pear");
  $pare = metaphone("pare");
  $bare = metaphone("bare");
  $bear = metaphone("bear");
  $editor = metaphone("procrastination");
  echo "Pair: $pair\nPear: $pear\nPare: $pare\nBare: $bare\
nBear: $bear\nEditor: $editor\n";
?>
```

That script will output the following:

```
Pair: PR
Pear: PR
Pare: PR
Bare: BR
Bear: BR
Editor: PRKRSTNXN
```

Without simplifying the system too much, you should at least be able to see that it essentially ignores vowels – if you look up the metaphone value of peer you will see it is also PR, like pair, pear, and pare. This actually works in our favour. For example, the most common Google search term in 2003 and 2004 was 'Britney Spears'. But there were many more searches by people who were peculiarly inept at spelling the name of their object of obsession. Fortunately, Google was able to figure out what they meant even if their typing was way off, and the metaphone system lets us emulate the same in our scripts.

Here is an example script that demonstrates the **metaphone()** function handling various ways of spelling Ms Spears' name:

```
<?php
  $b1 = metaphone("britnney speers");
  $b2 = metaphone("britteny spiers");
  $b3 = metaphone("britney spears");
  $b4 = metaphone("briteney spires");
  echo "B1: $b1\nB2: $b2\nB3: $b3\nB4: $b4\n";
?>
```

When run, you will see that each of those strings generates the same metaphone: BRTNSPRS.

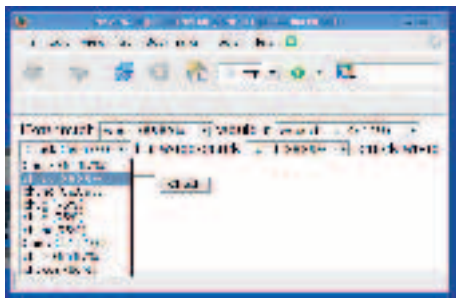
Getting the right use for the metaphone function takes time and consideration, but when used properly it can be a great way of providing some surprising insight into user input. Compared with **pspell_suggest()**, which works its way character by character through a list of possible matches like a machine would, **metaphone()** looks at the text from a different angle, and one that is much more natural to people.

For the last trick...

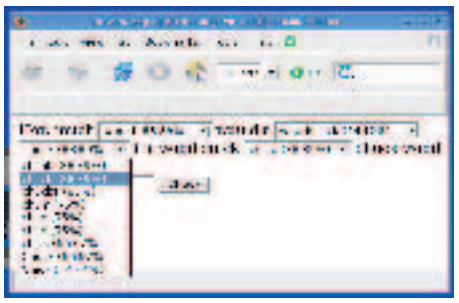
Having covered spell-checking and text similarity matching, we can now produce a script that accepts a line of text from users, spell-checks it, offers suggestions and ranks the suggestions according to their similarity to the source word. The most attractive way to do this is with HTML, using the drop-down boxes to show alternatives wherever a word has not been recognised. The first implementation of this script is based on the last **pspell** example, and mixes in calls to **similar_text**():

```
<HTML>
<HEAD>
<TITLE>The Amazing Spellchecking PHP Script</TITLE>
</HEAD>
<BODY>
<FORM METHOD="POST" ACTION="67final.php">
<?php
  /// COMMENT ONE
  $_POST['input'] = "How much woud would a wodchuck
chuk if a woodchuck culd chuck wood";
            if (isset($_POST['input'])) {
    $sentence = trim($_POST['input']);
    if (!$sentence) {
      echo "You must supply a sentence!\n";
      exit;
    }
          $spell = pspell_new("en");
    $words = explode(" ", $sentence);
foreach($words as $word) {
      if (pspell_check($spell, $word)) {
        echo "$word ";
      } else {
        $suggestions = pspell_suggest($spell, $word);
        if (count($suggestions)) {
          echo "<select>";
 foreach($suggestions as $suggestion) {
            /// COMMENT TWO
          similar_text($word, $suggestion, $similarity);
          $similarity = round($similarity, 2);
          echo "<option>$suggestion ($similarity%)</
option>";
          }
echo "</select> ";
        } else {
          /// COMMENT THREE
          echo "$word ";
        }
      }
    }
  }
?>
<br /><br />
<input type="text" name="input" />
<input type="submit" value="Check" />
</form>
</body>
</html>
```

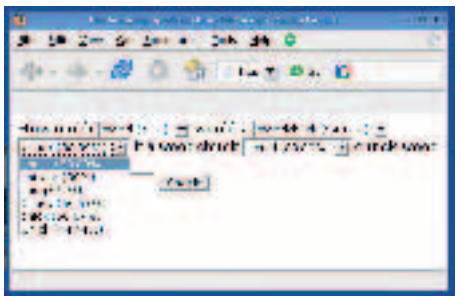
There are three lines starting with /// **COMMENT** in there. The first inserts a specific line of text into the **$_POST** array to simulate user input. This isn't necessary, but means that for the purposes of this article we have a set piece of text to work



1/ All spelling errors have suggestions, but those suggestions have no helpful order.



2/ Each suggestion is now sorted according to the similiarity with the misspelled word.



3/ Each suggestion now gets checked for metaphone equality with the original word.

with. The second comment marks where the suggestion similarity is calculated and printed out in HTML. The last comment shows where unmatched words are printed out, which means that any strange word (perhaps someone's name, or a place?) gets treated as if it were spelled correctly.

Fig 1 shows how the output from this page looks. Our test sentence is 'How much woud would a wodchuck chuk if a woodchuck culd chuck wood' – full of juicy typos. This first script comes up with 'How much would would a woodchuck Chuck if a woodchuck could chuck wood'. If you look at the screenshot, the reason for the lack of quality is apparent: **pspell_suggest**() has not sorted the suggestions by their similarity to the original word, which is why 'Chuck' (66.67% match) appears higher than 'chuck' (88.89% match).

The solution is to create an array of all the matches, then sort it by the similarity with the word, like this:

```
$suggestions = pspell_suggest($spell, $word);
if (count($suggestions)) {
   echo "<select>";
   $similarities = array();
   foreach($suggestions as $suggestion) {
      /// COMMENT TWO
      similar_text($word, $suggestion, $similarity);
      $similarity = round($similarity, 2);
      $similarities[$suggestion] = $similarity;
   }
   arsort($similarities);
   foreach($similarities as $suggestion => $similarity) {
      echo "<option>$suggestion ($similarity%)</option>";
   }
echo "</select> ";
```

Britney – highly sought-after, apparently.

Fig 2 shows the output from this second attempt at the script. This time the result is actually worse: 'How much would would a woodchuck chunk if a woodchuck could chuck wood'. I think I would rather have 'Chuck' than 'chunk'! However, at least we now have the matches sorted by their similarity.

The script is consistently getting 'wood' and 'chuck' wrong but it isn't far off – 'chuck' is as textually similar to 'chuk' as 'chunk' is, which means we need to find something else to differentiate between the two. Of course, if you have been paying attention so far, you should now be screaming "metaphone!" at your magazine: 'chuck' and 'chuk' are phonetically identical, but 'chunk' has a very different sound. The difference between 'would', 'woud', and 'wood' is very minor, but because 'would' has an l in it, the metaphone algorithm should flag it up with a slightly different representation.

So, the final change we need to make is to reject outright any suggestions that are not an exact phonetic match for our input word, like this:

```
///COMMENT TWO
if (metaphone($word) != metaphone($suggestion)) continue;
```

The **COMMENT TWO** part is there to show you where to insert the new line. **Fig 3** shows the final script in action, with all the correct words matched.

Punctuation loops

Our spelling script is now fairly complex. Not only can it match simple typing errors, it can also match spelling errors when people really have no clue how to spell the word they are typing. In cases like searches for Britney this will make a big difference to your users. In Google this kind of spell-prompting is common: 'Did you mean Britney Spears?' Sometimes, if your query has no matches at all, it will actually rewrite the query and give you those results, which can be more helpful – although they are careful to provide a link to the unaltered results, just in case you really were looking for Britenety Spares.

From here you can extend the script to better handle punctuation. The easiest way to do this is to split the entire sentence by spaces, then loop through each word and split them by punctuation and symbols: ',;:"–!?$£%^&*()' and others. These then need to be placed into the right order when reassembling the sentence. This sounds harder than it is, and if you give it a try (recommended) you'll learn a lot.

Whatever you do, keep in mind that some people don't want their spelling changed – either because they have a particular style, or because their country spells things differently from yours. Including spell-checking on your site should not be intrusive or annoying, at least not if you plan to keep your users. Instead, make it optional – perhaps a small button next to Submit Post on your forum, or a checkbox in your account preferences.

So, I will leave you with this one thought in mind: unobtrusive spell-checking is a great addition to your site, whereas badly done it's the suXX0rs – w00t! LXF

NEXT MONTH

Still not upgraded to PHP 5? You'd better get your skates on, because PHP 5.1 is almost here! We'll show you around.