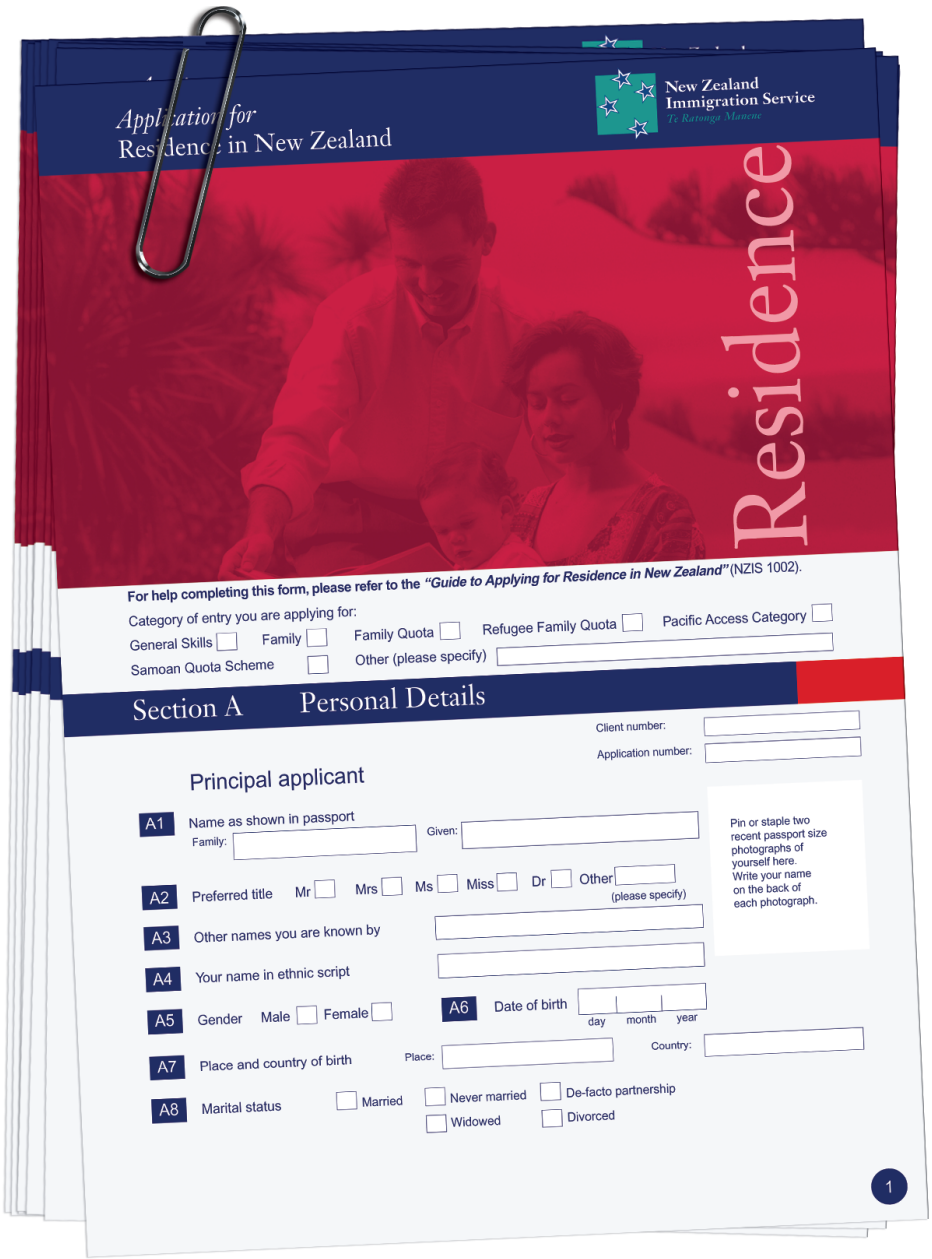
Using PHP Components

Forms to the Max

Like many other languages, the PHP Web scripting language abounds with pre-fabricated components. In the following article we will be looking at practical applications using form management as an example.

BY RICHARD SAMAR



Programmers, and above all their clients, are always interested in quick results. The need to work on parallel projects, and tight deadlines hit Web developers in particular. Thus professional developers look to spending more time on tackling the task in hand, rather than churning out auxiliary functions. PHP [2] supports this aim in many areas application development, but unfortunately not many people know that. And this is something I intend to set straight with the current article.

The idea behind Pear, the PHP Extension and Application Repository [3] arose in parallel with the release of PHP 4 in 1999. Many programmers view this as analogous to CPAN for Perl. But where CPAN provides quantity rather than quality, the PHP class library maintainers have clearly focused on the quality of their collection. As of PHP 4.3m, Pear left the beta development stage and is now an essential component of any Linux distribution that includes PHP.

Package Browser Shows Application Categories

In PHP-speak components are referred to as packages. They are organized in a tree by application category, an underscore and name. The so-called package browser [4] on the Web shows this tree structure (see Figure 1).

This article shows the practical capabilities of Pear, using the *QuickForm* package as an example. *QuickForm* facilitates the use of forms, the major interface to the user and a mandatory aspect of Web applications. *QuickForm* has been assigned to the *HTML* category and can thus be located under *HTML_QuickForm*. As discussed previously, you can use the browser to ascertain the field of application, version, changelog, download statistics and dependencies on other packages.

Installing Pear & *QuickForm*

If PHP 4.3 or later is installed on your system, you can safely assume that Pear is, too. In this case, no installation steps are required. In rare cases where PHP

has been explicitly configured and compiled *--without-pear*, admins will need to re-compile PHP without the option set.

If your distribution still uses an older version, you can use a workaround – as an alternative to a PHP update – to acquire the Pear package installer, which is essential for package setup tasks. To do so, open up a connection to the Internet and enter the following in a shell:

```
lynx -source http://go-pear.org➥
 | php
```

This line downloads a PHP script, runs the script and takes care of everything else automatically.

The so-called *include_path* in the PHP configuration file (typically in */etc/php.ini*) has to be set correctly to avoid problems when including Pear components. If PHP is installed in */usr/local*, Pear will typically be set up in */usr/local/lib/php/*.

If you do need to troubleshoot an installation, a quick look at the configuration file will reveal whether the entries are pointing at the correct directories.

Displaying Installed Packages in the Shell

The *pear list* command shows a list of installed packages in the shell:

```
$pear list
Installed packages:
===================
Package Version State
Archive_Tar     0.9     stable
Console_Getopt  1.0     stable
DB      1.3     stable
```

You can type *pear install HTML_QuickForm* to automatically install the package in the shell. If your Internet connection was down at the time, you can use the Package browser as follows to retrieve the required package, resolve any dependencies, *HTML_Common* in this case, and install the collection:

```
$ pear install /Path/to/Down➥
load/HTML_Common-1.2.1.tgz
install ok: HTML_Common 1.2.1
$ pear install /Path/to/Down➥
load/HTML_QuickFrom-3.0.tgz
install ok: HTML_QuickForm 3.0
```

Then type *pear list* again to display the installed packages. You can later enter *pear upgrade-all* periodically to update any packages you have installed. *pear uninstall* followed by the package name will uninstall a package. And if you are interested in even more detail on the general Pear configuration, typing *pear config-show* will provide that information.

HTML_QuickForm in Action

The *PEAR::HTML_QuickForm* package – note the typical naming convention with the colon – provides functions for creating, validating, and handling HTML forms. It is easy to manage and at the same time extremely flexible. In addition to simple output facilities, the package also provides interfaces for well-known PHP template engines such as Smarty [5].

Listing 1: *emailformular.php*

```
01 <?php
02 require_once
'HTML/QuickForm.php';
03
04 // Show QuickForm version
05 print 'PEAR::HTML_QuickForm
Version ';
06 print
HTML_QuickForm::apiVersion() .
'<br/><br/>';
07
08 $myForm     = new
HTML_QuickForm('EmailFormular',
'POST');
09 $myForm->addElement('header',
'', 'Personal Data');
10
11 $myForm->addElement('text',
'textName',    'Surname:');
12 $myForm->addElement('text',
'textFirstname', 'First name:');
13 $myForm->addElement('text',
'textEmail',   'Email:');
14 $myForm->addElement('submit',
'submitButton','Submit data');
15
16 $name =& $myForm-
>getElement('textName');
17 $name->setMaxLength(30);
18 $name->setSize(30);
19
20 $vname =& $myForm-
>getElement('textFirstname');
21 $vname->setMaxLength(20);
22 $vname->setSize(30);
23
24 $email =& $myForm-
>getElement('textEmail');
25 $email->setMaxLength(50);
26 $email->setSize(30);
27
28 // Add validation rules
29 $myForm->addRule('textName',
'Please enter surname',
'required');
30 $myForm-
>addRule('textFirstname',' Please
enter first name',   'required');
31 $myForm->addRule('textEmail',
'Please enter email address',
'required');
32 $myForm->addRule('textEmail',
'Email invalid', 'email');
33 $myForm->addRule('textEmail2',
'Please enter email',
'required');
34 $myForm->addRule('textEmail2',
'Email invalid', 'email');
35
36 // Client-side validation
using JavaScript also possible
37 // $myForm-
>addRule('textEmail', 'Email
invalid', 'email', NULL,
'client');
38
39 // Freeze form if validation
OK
40 if ( $myForm->validate() )
41 {
42    print 'Thank you! Your data
is as follows:';
43    $myForm-
>removeElement('submitButton');
44    $myForm->freeze();
45 }
46
47
48 // Display form
49 $myForm->display();
50
51 ?>
```

Listing 1 shows a short example. The Pear package uses an object oriented programming approach, as is typically the case. The sample form prompts the user to input a family name, first name and email address. Validation is performed to prevent faulty data input – everyday programming experience shows that careful validation pays, even for the most simple of structures.

The *require_once 'HTML/QuickForm.php'* construction in line 2 includes the required Pear package. An absolute pathname is not required at this point, as PHP will automatically search the previously configured *include_path*. This again reflects the structure of Pear: *QuickForm* occupies a position below *HTML* in the hierarchy, and *HTML* is a physical directory.

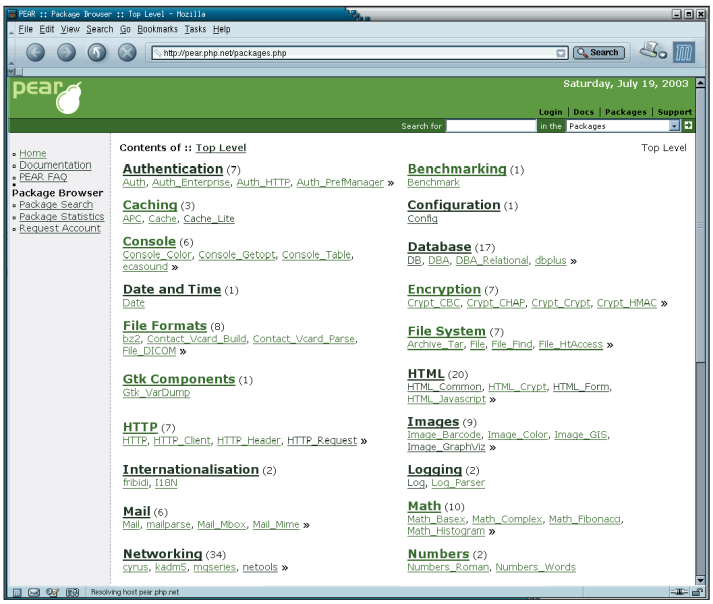


Figure 1: In PHP-speak components are known as packages. The web based package browser shows the packages in a tree view, sorted by application category

A Profusion of Functions

The *QuickForm* API is wide-ranging and extremely functional, especially as of version 3. Thus it makes sense to output the *apiVersion()* in line 6.

This is a so-called static method, something that most developers using an object oriented language will be familiar with: When you use a double colon operator to call a method of this class, no initialization is required.

Two parameters are passed to the constructor in line 8: *EmailFormular* identifies the form and will later be used as the name of the form in the HTML code. *POST* specifies the *HTTP POST* method for transmitting the data.

Other parameters and switches could have been included if required. A third parameter could be used to specify the target document to be generated, for example. As in this case the parameter has been omitted, the current document will be assumed as the target.

advertisement

A Header and Text Boxes

A header, like the one in line 9, is useful to more easily identify the document, and it makes for neater forms. This is followed by three elements of the *text* form type, which any Web browser will render as HTML text boxes. The second parameter identifies the element, whereas the third specifies the text that will appear next to boxes on the Website – this is referred to as a label. The following element types are available in addition to the Submit button shown in line 14:

- checkbox
- image
- hidden
- password
- radio
- select
- text
- textarea

You typically need to specify the length of the fields in an input form. Developers can use the *getElement()*, which returns a reference to each element, and assigns it to the corresponding variable, to do so. In our example, lines 16 through 26 set the visible length of all fields to 30 characters, although the maximum lengths are different.

Table 1: *addRule()* validation types

Entry	Meaning
required	The field must be filled
maxlength	Maximum length of field
minlength	Minimum length of field
email	Valid email address
lettersonly	The entry must comprise only letters
numeric	The entry must comprise only numbers
regex	The entry must match a regular expression

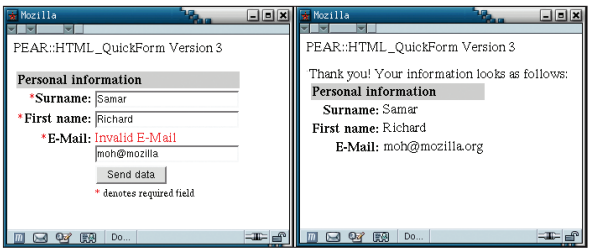


Figure 2: Validating email fields is a simple task, thanks to *QuickForm*. Mandatory fields are specified as such, and erroneous input generates a matching error message

Field Validation On-the-Fly

It is quite simple to validate user input thanks to *QuickForm*. The Method *addRule()* method allows you to assign an arbitrary number of rules to each element. Elements are identified by unique names.

Validation facilities are extremely comprehensive, so the method has a lot of optional parameters. Lines 29 through 34 in Listing 1 demonstrate an extremely simple case, passing the error text as the second, and the validation type as the third parameter. Table 1 shows an overview of major validation types (see Table 1). Depending on the type, the programmer must pass either nothing, a numerical value, or a regular expression to the *addRule()* method, as a fourth parameter.

Our example implements server-side field validation. The comment in line 37 also indicates the possibility to perform client-side validation using Javascript. The *if* instruction then uses *validate()* to check whether each field has been successfully validated. It so, *validate()* returns *TRUE* and freezes the whole

Listing 2: *creditcard.php*

```
01 <?php
02 require_once
'HTML/QuickForm.php';
03
04 // Template for header
05 $headerTemplate  = '<tr><td
style="white-space: nowrap;
background-color: blue;"
align="center" ';
06 $headerTemplate .=
'valign="top" colspan="2"><font
size="5"
color="yellow">{header}</font></t
d></tr>';
07
08 // Template for text fields
(card holder and number)
09 $elTemplate  = '<tr><td
align="right" valign="top">';
10 $elTemplate .= '<!-- BEGIN
required --><font
color="blue"><b>#</b></font><!--
END required --
><b>{label}</b></td>';
11 $elTemplate .= '<td
valign="top" align="left"><!--
BEGIN error --><span
style="color: #ff0000">';
12 $elTemplate .=
'{error}</span><br /><!-- END
error -->{element}</td></tr>';
13
14 // Instantiate form
15 $myForm     = new
HTML_QuickForm('CreditcardForm',
'POST');
16 // new text for mandatory
fields
17 $myForm-
>setRequiredNote('<font
color="blue"><b>#</b></font>
mandatory fields');
18
19 // Add header and set new
template
20 $myForm->addElement('header',
'', 'Creditcarddata');
21 $myForm-
>setHeaderTemplate($headerTemplat
e);
22
23 // Add text fields and set new
template for each
24 $myForm->addElement('text',
'textCardholder',
'Cardholder:');
25 $myForm->addElement('text',
'textCardnumber', 'Cardnumber:');
26 $myForm-
>setElementTemplate($elTemplate,
'textCardholder');
27 $myForm-
>setElementTemplate($elTemplate,
'textCardnumber');
28
29 // Array for credit card types
30 $cardtypes = array( 'visa' =>
'VISA', 'master' => 'EuroCard',
'amex'   => 'American Express',);
31 // Drop-down menu for credit
card types
32 $myForm->addElement('select',
'selectCardtypes', 'cardtype:',
$cardtypes);
33
34 // Arrays for months and years
35 $months = array (
36              '01'   =>
```

form. The *freeze()* feature shows the data entries, without allowing them to be edited.

This is practical after completing data entries, as it allows the user to create a hard copy. The Submit button is removed within the if statement, as it would make no sense to keep displaying the button after data entry has been successfully completed. The most important method finally occurs in line 49: *display()* is required to output the form. Figure 2 shows the results of all this effort, with and without errors.

More Flexibility

The functionality provided by Listing 1 is okay as far as it goes, but the appearance of the form (Figure 2) is definitely nothing to write home about. Additionally, you might prefer to use customized rather than standard messages. This is easily done thanks to the high degree of flexibility that *QuickForm* provides, as another example in Listing 2 shows. This creates a simple form for credit card data input. Developers can use templates to modify the color, form and order of the individual form elements to their liking. These templates are defined in two string variables, for the header and the two text boxes, at the start of Listing 2. The strings in the curly brackets define the text part of the current element and can be placed anywhere. For text fields, {*label*} indicates the name, such as *Cardholder*, and {*element*} the text field itself. Areas enclosed in HTML comments contain the error messages for the mandatory fields, and symbols. In the latter case the red asterisk is replaced by a blue diamond.

As a form is embedded in a table by default, you only have to ensure that the internal table tags are set correctly. However, you might conceivably use a form template without a table. The *setRequiredNote()* method in line 17 replaces the standard messages, with customized messages. Additionally, *setHeaderTemplate()* and *setElementTemplate()* register the defined templates with the elements.

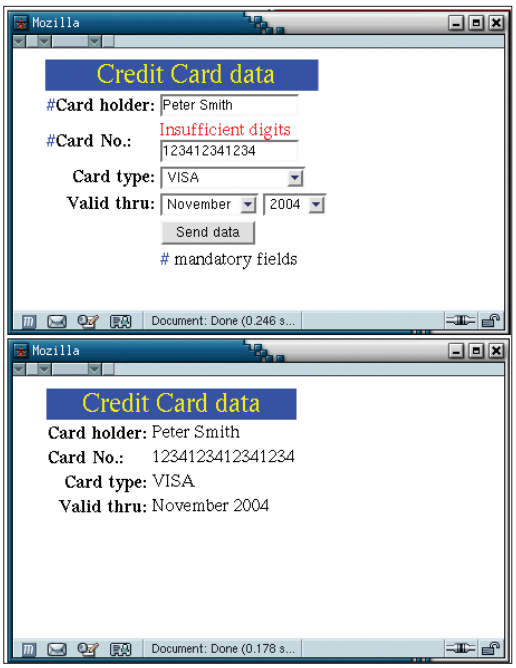


Figure 3: *QuickForm* is extremely flexible when it comes to arranging fields, assigning labels and colors. Templates can be used to fulfill almost any requirements

Dropdown Menu Items Stored in Arrays

Dropdown menus (element type *select*) are defined as values in appropriate

Listing 2: *creditcard.php*

```
'January',  '02' => 'February',
'03'   => 'March',
37              '04'   =>
'April',   '05' => 'May',
'06'   => 'June',
38              '07'   => 'July',
'08' => 'August',  '09'   =>
'September',
39              '10'   =>
'October', '11' =>
'November','12'   => 'December'
40                 );
41 $years = array ( '2003' =>
'2003', '2004' => '2004', '2005'
=> '2005' );
42
43 // Create group elements for
months and years
44 $validTo[] =
&HTML_QuickForm::createElement('s
elect', 'selectValidMonth', NULL,
$months);
45 $validTo[] =
&HTML_QuickForm::createElement('s
elect', 'selectValidYear',  NULL,
$years);
46
47 // group elements create for
months and years
48 $myForm->addGroup($validTo,
'validToGroup', 'Valid to:');
49
50 // Add Submit button
51 $myForm->addElement('submit',
'submitButton','Submit Data');
52
53 // Credit card number have 16
digits
54 $cardnumber =& $myForm-
>getElement('textCardnumber');
55 $cardnumber->setMaxLength(16);
56
57 // Validation rules: The two
text fields must be occupied
58 // Credit card number must be
numeric and comprise 16 digits
59 $myForm-
>addRule('textCardholder',
'Please enter cardholder',
'required');
60 $myForm-
>addRule('textCardnumber',
'Please supply card number',
'required');
61 $myForm-
>addRule('textCardnumber',
'Invalid card number',
'numeric');
62 $myForm-
>addRule('textCardnumber', 'Card
number too short', 'minlength',
16);
63
64 // Freeze form if validation
succeeds
65 if ( $myForm->validate() )
66 {
67     $myForm-
>removeElement('submitButton');
68     $myForm->freeze();
69 }
70
71 // Display form
72 $myForm->display();
73
74 ?>
```

What is Pear, what is PECL?

PHP is an extremely universal language which can easily integrate programming libraries and thus implement interfaces to databases, graphics tools, XML parsers and many other things.

The more popular PHP became, the more work the Community started putting into the development of extensions – this is one of the more obvious advantages of Open Source! But to keep the footprint of the PHP distribution to a reasonable size, only the most important and frequently used extensions are supplied with PHP by default.

areas in lines 30 through 41. The *$cardtypes* area is the fourth parameter of *addElement()* in line 32. This mechanism makes it possible to add new values quickly at a later date without sacrificing readability. The *QuickForm* package also supports groups, where the individual elements are positioned adjacently to one another. In the case of the credit card form (see Figure 3), it makes more sense to organize the validity data – that is the combination of the months and years that form the date – in this way.

To do so, lines 44 and 45 add two dropdown menus to the *$validTo[]* array.

No Need for Scepticism

Developers tend to be finicky and untrusting of other people's code. This is why many developers view Pear with some degree of scepticism. But this kind of resistance to the PHP class library is unjustified. If you trust PHP, you should also be prepared to trust the quality of Pear – at least the stable packages of the Pear Foundation Classes (PFC).

These individual elements do not contain labels, and this is why the third parameter in the static method *createElement()* is *NULL*. A label is specified for the whole group in line 48 using *addGroup()*; the array will be the first parameter passed to it. The remaining instructions are as in Listing 1.

Simple Examples

As there is no need to write client software and a Web browser is available for nearly every operating system, more and more programmers are adopting the server-based Web application approach to application development.

Operating costs are also calculable, as software maintenance costs and the like are lower than for traditional client/server solutions, due to the fact that the software only needs to be updated server-side.

Both PHP examples shown here are quite simple, but they do show that Web applications are both simple to develop and at the same time easy to harden against erroneous input. Workflow management systems are a typical application that involves a mass of forms, and a perfect field of application for *QuickForm*. The library introduced in this article provides a large range of additional functions suited to various fields of application, such as processing file uploads, custom enhancements, specialized callback functions or intelligent filters for form input fields.

But *QuickForm* is not the only library that Pear provides to facilitate the programmer's task. Credit card data transfer, which one could envisage as an extension to the second example, is typically handled by HTTP or Soap. Again there are Pear packages that reduce this process to just a few lines of code, and save developers a lot of effort implementing standard protocols. Thanks to Pear, there is no need to re-invent the wheel for every application field you venture into. ■

Quality with a Capital Q

These high-quality packages are subject to stringent quality control measures. When new features are added, or an implementation needs changing, these changes are discussed by a team of experienced developers using a mailing list.

Pear packages comply with a coding standard that precisely defines how developers should program, which includes how to indent code, assign variable names, and many other aspects. These stringent rules allow both Pear developers and newcomers to the language to quickly grasp the source code.

Also, there is a requirement to provide good documentation for code. Thanks to this standard, API documentation for the package can be created automatically. And it is for this reason that the Pear website at [3] contains a whole bunch of descriptions of this type.

Many Pear package authors are well-known and respected members of the developer community who also contribute to other parts of PHP. To find out how much blood, sweat and tears goes into producing quality code, just read the minutes of the last Pear Meeting [6].

Pears and Pickles for free

With such a large range of extensions available, you have to put some effort into promoting your library. This is why the initiator of Pear, Stig Saether Bakken, distributes pears, and pickles to the attendees of his Pear keynotes and workshops. Although this duo's culinary virtue is debatable, it does draw the audience's attention to the names Pear, and PECL (say pickle). PECL, an acronym for PHP Extension Code Library, comprises a subset of Pear and contains PHP extensions programmed in C.

If you are looking for more exotic extensions, to support the Open Source clone of Microsoft .NET framework, Mono, you should investigate PECL. The installation is handled by the package installer, as is the case for any other Pear package. In the case of PECL packages, ensure that any ensuing system dependencies are resolved. If you are installing the Mono package, you will of course need the Mono Framework itself.

INFO

[1] Listings for this article: *http://www.linux-magazin.de/Service/Listings/2003/09/PHP*

[2] Official PHP Website: *http://www.php.net*

[3] Official Pear Website: *http://pear.php.net*

[4] Pear Package Browser: *http://pear.php.net/packages.php*

[5] Smarty Template Engine: *http://smarty.php.net/*

[6] Protocol of the 2003 Pear Meeting: *http://pear.php.net/news/meeting-2003-summary.php*

THE AUTHOR

Richard Samar, http://richard-samar.de, is a freelance IT consultant who lives near Frankfurt, Germany. He has been a PHP enthusiast for many years, and is well-known in the German PHP Community.