

Chapter 1

Introduction

PEAR and Smarty are two projects that have sprung out of the PHP community within the last few years. The first is a large repository of platform independent well tested library code, while the latter is a complex compiling template engine that brings the Model-View-Controller (MVC) architecture to PHP.

Combined the two projects offer thousands of lines of proven code to the PHP community in much the same way that CPAN has for Perl. This book will cover how to improve both the stability and portability of your own projects by utilizing PEAR for complex error handling, database connectivity, and logging. The book will also show you how to separate your business and data logic by using the Smarty compiling template engine.

What is PEAR?

PEAR stands for PHP Extension and Application Repository and is pronounced just as the fruit is. According to the projects website (<http://pear.php.net>) the project aims to be:

- * A structured library of open-sourced code for PHP users
- * A system for code distribution and package maintenance
- * A standard style for code written in PHP
- * The PHP Foundation Classes (PFC)
- * The PHP Extension Code Library (PECL)
- * A website, mailing lists and download mirrors to support the PHP/PEAR community

PEAR is broken up into packages. Each package has its own maintainer and is, essentially, its own project with its own development team, version numbers, and documentation. Many packages are related to each other through defined relational definitions, such as “Authentication” or “Mail”.

PEAR packages, once cleared by the PEAR community, are placed into a central database where they can be downloaded and quickly installed using the PEAR installer. Each PEAR package comes in a gzip archive with the source, an XML description file and, if you are lucky, an abundance of documentation and examples.

Many have voiced strong opinions against the PEAR project’s strict enforcement of coding standards, however, I believe that large projects, such as PEAR, required a certain

fluidity for ease of reading. The coding standard is only enforced for projects that are included or released through the PEAR project and can be found on the PEAR website.

The PEAR Foundation Classes (PFC) and the PEAR Extension Code Library is what PEAR is really about. PFC and PECL represent thousands of lines of PHP code that provides a ready and able platform and framework to develop large scale projects in PHP with. PFC provides, among other things, the PEAR and PEAR_Error classes which provide destructors and advanced error handling.

PHP Extension Code Library (PECL) is a group of classes and abstracted applications that was built using PFC as their foundations. Such classes include the Mail and Log packages.

Why use PEAR?

PEAR represents the efforts of hundreds of people and hours of development and testing time. The main reason I started using PEAR on my own projects was because I simply did not have the time to design, test and implement code that replaced what PEAR already did.

A common question I hear from those who know that I base my applications on PEAR concerns licensing issues pertaining to proprietary code. I have posted the question to the list and have been assured by the PEAR community that as long as the PHP license is followed there should be no reason why I could not distributed PEAR with a proprietary application.

Furthermore, PEAR currently has over 200 packages, libraries, utilities and applications that span from XML parsers to abstracted email classes to complex networking tools. These packages have been downloaded thousands of times and used by a variety of people on a variety of platforms, which leads to my final point as to why you should use PEAR in your applications; PEAR provides portability for your applications.

To demonstrate how PEAR makes it easier to port applications across platforms I'm going to compare and contrast PHP's default mail function with PEAR's Mail package. Below is a simple example of sending mail via PHP's mail function

```
<?php

$to = 'email@example.com';
$subject = 'Hello';
$msg = 'This is a test message sent with the regular mail function';
$headers = "From: Server <server@example.com>\n".
    "Reply-To: server@example.com";

mail($to,$subject,$msg,$headers);

?>
```

The above example should work on most UNIX servers, but most likely will not work on many Windows servers, which are not always shipped with an SMTP server. Thus the problem arises; what if your web server does not have an SMTP server installed locally on the machine? Many systems administrators opt not to install SMTP servers because of vulnerabilities and package maintenance, which leaves the web developer in the difficult position of being without an avenue to deliver email.

Fear not! PEAR's Mail package allows for multiple avenues of email deliver. For instance, the Mail package supports sending email via PHP's built in mail function, sendmail or raw SMTP. Below is a simple example of how to rewrite the above example using the PEAR Mail package.

```
<?php

/* The PEAR Foundation Classes are included via Mail.php */
require_once('Mail.php');

$email = & Mail::factory('mail');
if(!PEAR::isError($email))
{
    /* The Mail package takes an array of addresses */
    $recipients = array('editor@apress.com');

    /* Create a header array keyed by the header name */
    $headers = array();
    $headers['Subject'] = 'Hello how are you?';
    $headers['From'] = 'Server <server@server.com>';

    /* Our message */
    $msg = 'My message is for you to check the server';

    $result = $email->send($recipients,$headers,$msg);
    if(PEAR::isError($result))
    {
        /* Something went wrong in sending the email */
        die($result->getMessage());
    }
    else
    {
        echo 'Your email has been successfully sent!';
    }
}
else
{
    die($email->getMessage());
}
```

```
}
```

```
?>
```

In the example above you could easily change the factory type from mail to the other two offered by the Mail package: sendmail or SMTP. The reason why this makes life so much easier is because you can port your application quickly without worrying about how mail is sent. For instance, if your web server didn't run an SMTP server and didn't have sendmail installed on it you could easily switch your default Mail factory to SMTP and have the web server send email through your company's main SMTP server. Below is an example of how we can quickly change the above code to work with an SMTP server rather than PHP's mail function.

```
<?php

/* The PEAR Foundation Classes are included via Mail.php */
require_once('Mail.php');

/* Mail's SMTP factory also allows you to define the port and authentication
 * parameters (ie. username and password for your SMTP-AUTH enabled server)
 */
$config['host'] = 'mail.example.com';

$email = & Mail::factory('smtp',$config);
if(!PEAR::isError($email))
{
    /* The Mail package takes an array of addresses */
    $recipients = array('editor@apress.com');

    /* Create a header array keyed by the header name */
    $headers = array();
    $headers['Subject'] = 'Hello how are you?';
    $headers['From'] = 'Server <server@server.com>';

    /* Our message */
    $msg = 'My message is for you to check the server';

    $result = $email->send($recipients,$headers,$msg);
    if(PEAR::isError($result))
    {
        /* Something went wrong in sending the email */
        die($result->getMessage());
    }
    else
    {
```

```
        echo 'Your email has been successfully sent!';
    }
}
else
{
    die($email->getMessage());
}

?>
```

By adding a single line and changing our Mail::factory() call we were able to easily change the behavior of our application to use an SMTP server. The above example perfectly outlines how using the various PEAR packages can make your life much easier when it comes time to port your applications based on client specifications.

What is Smarty?

According to Smarty's website (<http://smarty.php.net>), Smarty is a template engine that "facilitates a manageable way to separate application logic and content from its presentation." More specifically Smarty allows developers to rid their PHP code of HTML and hand the interface design back to the designers without giving the interface designers access to their code.

For instance you could theoretically program your application without adding an ounce of HTML, document which variables you are passing to the Smarty templates and then let your designers loose to do whatever they wish with the documented variables. Below is a simple example of how Smarty works.

```
<?php
    /* Include the Smarty template engine */
    require_once('Smarty.class.php');
    $smarty = new Smarty;

    $a = 5;
    $b = 10;
    $c = ($a + $b);

    /* Assign variables to your template */
    $smarty->assign('a',$a);
    $smarty->assign('b',$b);
    $smarty->assign('c',$c);

    /* Display your template */
    $smarty->display('template.tpl');
```

?>

For many developers reading this text the above code is a thing of beauty in that the logic code contains no HTML whatsoever. However, what makes Smarty so special to me is that once I have created my application I am not responsible for creating the HTML; the designer is. After telling my designer about the variables in my application and what to name his template he could create the template, which might look something like the example below.

```
<html>
<body>
This article has been read {$a} times. Out of the {$a} times it was read {$b} people emailed it to
one of their friends. As a result one could assume that at least {$c} people have read this article.
</body>
</html>
```

As long as my designer has the correct permissions to create and edit template files there should not be a reason for me to write HTML, except initial skeleton templates, and there should not be any reason for my designer to have to go through a middle man to change HTML within my application.

Why use Smarty?

There are quite a few template engines built for PHP currently available on the Internet, however, Smarty has a few things that most do not. Smarty has many advanced features which give it an edge over its competitors.

Smarty has template caching built in. As long as you set up the permissions correctly all of your templates will automatically be cached as static files on your website and refreshed at given intervals.

Smarty is a compiled template engine, meaning that it takes your template file and compiles it into PHP. This feature means that Smarty does not have to do all of the regular expressions and string replacements on your template each time it is requested. Instead Smarty compiles your template into PHP code only after your template files have been updated. The result is a faster template engine.

Smarty has security features that allow programmers to determine which features of the template engine the designers will have access to. This includes the ability to enable and disable template functions as well as the ability to limit access to Smarty's embedded PHP feature.

Smarty is extremely easy to extend through its plugin API. The website is full of documentation on how to build your own variable modifiers, output filters (such as gzip'ing your content after the template has been rendered) and template functions.

At first the proposition of leaving the HTML coding and design work entirely in the hands of your designers seems too good to be true, but a simple example will help expel such worries. Below is a typical example of retrieving data from a table and displaying it in an HTML table.

```
<?php

require_once('DB.php');

$db = & DB::connect('mysql://root:password@localhost/mydb',true);
if(!DB::isError($db))
{
    $sql = "SELECT *
            FROM users
            ORDER BY lname";

    $result = $db->query($sql);
    if(!DB::isError($result) && $result->numRows())
    {
        echo '<table width="100%">';
        echo '<tr>
            <td><b>User ID</b></td>
            <td><b>Last Name</b></td>
            <td><b>First Name</b></td>
            </tr>'. "\n";

        while($row = $result->fetchRow())
        {
            echo '<tr>
                <td>'. $row['userID']. '</td>
                <td>'. $row['lname']. '</td>
                <td>'. $row['fname']. '</td>
                </tr>'. "\n";
            }

            echo '</table>';
        }
    }
    else
    {
        die($db->getMessage());
    }
}
?>
```

In the above example we simply grab the records and put them into the HTML table. After spending an hour building an application such as the one above you submit it to your boss for review, which is then passed to the designer for a final signoff. The designer comes back with a slew of minor font and color changes, which will mean another skipped lunch due to HTML changes. With Smarty you could tell the designer(s) to make the changes themselves. The following example is how you can finally rid yourself of having to spend half your day making minor HTML adjustments.

```
<?php

require_once('DB.php');
require_once('Smarty.class.php');

$db = & DB::connect('mysql://root:password@localhost/mydb',true);
if(!DB::isError($db))
{
    $smarty = new Smarty;

    $sql = "SELECT *
            FROM users
            ORDER BY lname";

    $result = $db->query($sql);
    if(!DB::isError($result) && $result->numRows())
    {
        $users = array();
        while($row = $result->fetchRow())
        {
            $users[] = $row;
        }

        $smarty->assign('users',$users);
    }

    $smarty->display('users.tpl');
}
else
{
    die($db->getMessage());
}

?>
```

The first thing you notice in the above example is that there is no HTML. The second thing you notice is your script isn't displaying anything. Using Smarty does add a small level of complexity in that it requires you essentially have two parts to a single file now: the code and the template file. Below you will find an example template that corresponds to the above code.

```
{if is_array($users) && count($users)}  
<table>  
<tr>  
<td><b>User ID</b></td>  
<td><b>Last Name</b></td>  
<td><b>First Name</b></td>  
</tr>  
{foreach item=row from=$users}  
<tr>  
<td>{$row.userID}</td>  
<td>{$row.lname}</td>  
<td>{$row.fname}</td>  
</tr>  
{/foreach}  
</table>  
{else}  
<b>No users found!</b>  
{/if}
```

The above example would be the file called from Smarty's display function called users.tpl. This file could sit on your server someplace where the designer has write access allowing him to download the file and change the text himself. Smarty's website has an entire section devoted entirely to designers, which explains how to manipulate Smarty variables.

Another hidden benefit of Smarty is the fact that your code is now independent, for the most part, of HTML. For instance you could easily do a browser check in the above example and display a WAP template instead of an HTML template. Other hidden benefits of Smarty include the ability to filter the output through other programs, which could allow you to use gzip compression or even convert your document to a PDF.

Summary

Incorporating PEAR and Smarty into your applications is a great way to enhance your programming platform. Instead of relying on your own database abstraction layer or your own templating engine rely on ones that were written with the help of thousands of people and tested by millions more. PEAR and Smarty have been used on websites which log millions of hits a month and have proven to work well in some of the most hostile

Internet conditions, which is why I chose to use them in my own applications and is why you should too.