www.phparch.com

JUNE 2008

php | architect™

The Magazine For PHP Professionals

EAV Modeling

Not just for medical records after all

A Refactoring Diary: The Story Continues
Our intrepid frameworks investigator finds his spiritual home

Email Verification
That email address is valid—but is it a fake?

Migrating PHP, part II: PHP Code
The pitfalls of upgrading, and how to avoid them

INSIDE

etc/: Beyond Safe Mode
exit(0); Welcome to the Intertuber

Test Pattern
Scripting Integration

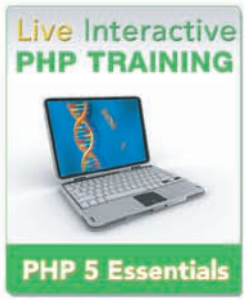
PHP TRAINING SHOULD NOT BE COMPLICATED OR EXPENSIVE

Just ask the 600 students we trained last year.

Training is essential to stay on top of your game—but it can be expensive, cumbersome and inconvenient. Take learning to a new level with our exclusive live online training course and get the best training at the most affordable price!

- Better Than Live™: full classroom experience with the convenience of the Internet
- Experience live training sessions with voice and **screen sharing**
- Interact with your instructor and fellow students in **real time**
- Classes are spread across longer periods to ensure optimum learning
- Includes **free tutoring**

Enjoy affordable prices without sacrificing the quality of your training—try us out today!



PHP ESSENTIALS

Dive into PHP with this great introductory/mid-level course designed for those who are approaching PHP for the first time or are moving to it from another language

CERTIFICATION

This great mid-level course prepares you for the Zend Certification exam—and it comes loaded with over **$500** in **free goodies!**

AJAX & Web 2.0

Master the world of Web 2.0 development with our latest live online training course- **Building Rich Internet Applications with PHP 5 and AJAX!**

PROFESSIONAL

Designed for the seasoned developer who has mastered PHP, this course tackles the most advanced PHP 5 topics, like security, XML, design patterns, and much more.

php | architect
The Magazine For PHP Professionals

Free no-nonsense trial

For more information
http://www.phparch.com/phptraining

php | architect™

CONTENTS

JUNE 2008

FEATURES

6 Email Verification — by Sharon Levy
That email address is valid—but is it a fake?

16 Migrating PHP, part II: PHP Code — by Stefan Priebsch
The pitfalls of upgrading, and how to avoid them

25 EAV Modeling — by Carl Welch
Not just for medical records after all

34 A Refactoring Diary: The Story Continues — by Bart McLeod
For everyone out there who isn't a frameworks guru (yet)

COLUMNS

4 Editorial — by Steph Fox
E_YMMV

44 Test P attern — by Matt Zandstra
Scripting Integration

50 etc/ — by Stuart Herbert
Beyond Safe Mode

57 exit(0); — by Marco Tabini
Welcome to the Intertuber

Download this month's code at: http://www.phparch.com/code/

WRITE FOR US!

If you want to bring a PHP-related topic to the attention of the professional PHP community, whether it is personal research, company software, or anything else, why not write an article for php|architect? If you would like to contribute, contact us and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine. Visit www.phparch.com/writeforus.php or contact our editorial team at write@phparch.com and get started!

E_YMMV

by Steph Fox

As you're probably aware, Phar is now a core extension in PHP 5.3. This is great news for Greg Beaver, the project lead; he has worked incredibly hard to make the extension do all that could possibly be asked of it, and more. Marcus Börger and myself also work on the extension, but we've been left blinking at the speed of development more than once. At the time of writing, Greg had just added OpenSSL signing support to Phar, which means that the only thing left on the immediate TODO is performance optimization.

Oh, and forward compatibility.

Now optimization is definitely more Greg's kind of thing than mine, and Marcus doesn't have much spare time these days, so I took it on myself to focus on that other thing. I went to look at PHP 6.

The good news is that from now, all phars created using PHP 5.2.1 and up should run under PHP 6 and vice versa. The structure's unchanged. Sing hallelujah. The bad news is that a phar with a default stub won't run under versions of PHP prior to PHP 5.2.1, whereas we had PHP 5.1 support this time last week. Blame: **b** and the binary cast. The default stub calls **unpack()**, you see, and PHP 6 needs to be *told* that the second argument to **unpack()** is a binary string and not a Unicode string. The only ways we have of telling PHP 6 this will throw a parse error in PHP 5.2.0 and under. There's no way to have both backward compatibility and forward compatibility: we have to choose.

I approached Andrei Zmievski and asked why we couldn't simply have a fallback implicit conversion from Unicode to binary where the context is known, i.e. in built-in PHP functions that *require* binary strings. He explained that there's no way to be certain of the intended encoding. I argued. After all, the user has control over the encoding through various INI directives and/or through the actual encoding of the physical script; we just needed to know which of those values would be used for implicit conversion. Andrei reiterated that implicit conversion is generally a Bad Thing, due to the lack of certainty.

Muttering 'new INI directive' under my breath, I'm still failing to see why an implicit conversion in that context couldn't be allowed and accompanied by an **E_STRICT** or even an **E_DEPRECATED** advisory message, just to make BC more possible. We never expected to support PHP 4, and thankfully there aren't many PHP 5.0 users out there, but PHP 5.1 is relatively popular and it's pretty hard to swallow losing userland compatibility with it from PHP 5.2.1 up.

I'll be interested to see what Stefan Priebsch has to say about forward compatibility in his migration series, but unfortunately we all have to wait another month to read his take on that. It's still interesting, from the perspective of PHP 5.3.0-dev, to see just how far we've come since PHP 4.

php|architect™

June 2008
Volume 7 - Issue 6

Publisher
Marco Tabini

Editor-in-Chief
Steph Fox

Author Liaison
Cathleen MacIsaac

Editorial Team
Arbi Arzoumani
Steph Fox
write@phparch.com

Graphics & Layout
Arbi Arzoumani

Managing Editor
Arbi Arzoumani

News Editor
Eddie Peloke
Elizabeth Naramore
news@phparch.com

Authors
Stuart Herbert, Sharon Levy, Bart McLeod, Stefan Priebsch, Marco Tabini, Carl Welch, Matt Zandstra

php|architect (ISSN 1709-7169) is published twelve times a year by Marco Tabini & Associates, Inc., 28 Bombay Ave., Toronto, ON M3H1B7, Canada.

Contact Information:

General mailbox: *info@phparch.com*
Editorial: *editors@phparch.com*
Sales & advertising: *sales@phparch.com*

Printed in Canada

e-xact transactions

Payment Processing Technology

MOVING BUSINESS. REAL TIME. REAL SMART

They don't get IT!

- Banks do what banks do.
- Take your money.
- Use your money.
- Charge you to access your money.

We get IT!

- We build software.
- Our code works.
- Download via the Web.
- Bank Agnostic.
- 24 x 7 Test System

You get IT!

- **FREE**
- http:developer.e-xact.com
- Influence our developement

E-xact Transactions (Canada)
Suite 304 - 134 Abbot Street
Vancouver, BC, V6B 2K4
Canada

Contact Information
Toll Free: 1-877-303-9228
Ph: 604.691.1670
Fax: 604.691.1678

Web: www.e-xact.com
Email: comments@exact.com

Trading Symbol: EXZ.U
Listing: (CDNX)

Email Verification

by Sharon Levy

A user submits an email address through a contact page. Next, your PHP script takes over and validates it. You may conclude that you can now move on. But have you really exercised due diligence? If that email address were a fake, how would you know? More importantly, how would your script know? In this article, I will explore some techniques that should prove useful.

There are two issues a developer may come up against from time to time: email validation and email verification. While validation largely concerns the format of an email address—the presence and placement of an @ symbol, a period and other characters—verification involves answering a simple question. Is the email address genuine? If an email address is phony or does not belong to the person providing it, then storing it is a poor idea.

The bonus of email verification is that it can also enhance user experience. Since as humans we are error prone, a number of false email addresses could be the result of typos rather than any malicious intent. Back in January 2006 Daniel Bonniot, of Drupal fame, reported that merely checking whether a domain exists could catch "between 1/2 and 2/3 of typos" (http://drupal.org/project/email_verify). Email verification can thus provide a mechanism to support users, allowing you to give immediate feedback so they can make any necessary corrections.

As for pranksters, we might try a little psychology to discourage them. For example, a Web page might warn that the IP address of the user's computer will be contained in an email sent to the real owner, thereby allowing her to trace the prankster's identity. Of course, this tactic relies on the assumption that most users are unaware that IP addresses can be allocated dynamically.

PHP: *5.2.1+*

O/S: *Any supported by PHP*

Useful/Related Links:

- *CGI 1.1 specification:* http://hoohoo.ncsa.uiuc.edu/cgi/interface.html
- *RFC 821 (SMTP):* http://tools.ietf.org/html/rfc821
- *RFC 2821 (SMTP):* http://tools.ietf.org/html/rfc2821
- *SMTP clarifications:* http://tools.ietf.org/html/rfc1123
- *SMTP overview:* http://nemesis.lonestar.org/site/mail_trouble.html
- *DNS extensions:* http://tools.ietf.org/html/rfc1886
- *DNS records:* http://www.debianhelp.co.uk/dnsrecords.htm
- *DNS and Windows:* http://www.tech-faq.com/understanding-dns.shtml
- *dig tutorial:* http://www.hungrypenguin.net/dig.php
- *dig manpage:* http://linux.die.net/man/1/dig
- *host manpage:* http://linux.die.net/man/1/host
- *nslookup manpage:* http://linux.die.net/man/1/nslookup
- *nslookup.exe tutorial:* http://support.microsoft.com/kb/200525

TO DISCUSS THIS ARTICLE VISIT:
http://c7y-bb.phparchitect.com/viewforum.php?f=10

Clearly, if someone wishes to intentionally provide a fake email address in a "Contact Us" mail form, it will take more than psychology alone to deter such an annoyance. A developer who knows how to verify an email address may eliminate or greatly reduce the spam that a business receives.

There are two approaches to email verification. The more familiar one is where the website sends an email to the newly registered user. If the user responds to the email, within a limited time span, that user's email address is judged to be legitimate. This method has the reputation of being virtually bullet-proof—but is it? Some email programs may consider such correspondence as "spam" or "junk mail". The user needs to think about checking the spam filter and junk mail folder, assuming he even knows what to do. This approach is also limited; sometimes email verification needs to occur within seconds rather than hours. In such cases, email verification needs to be automated. This second approach is well suited for PHP.

Validation First

An important initial step in verifying an email address is to ascertain its validity. This involves checking that it conforms to one of several formats, such as *user@example.com* or *user@php.example.com*. (Multiple subdomains must pass muster, too.) There is an excellent article about email validation in *Security Corner: An In-Depth Look at mail()* by Stefan Esser (see php|architect volume 6, issue 6).

I have browsed several web-based tutorials on email verification. They were all instructive but usually brief by necessity. I wish to take what I've learned along the way, expand on it, and share some code for doing email verification in PHP 5.

Extracting the Domain Name

Since *user@example.com* conforms to a valid email format, let's proceed to verify it. A preliminary step is to split the email address in two. Using PHP's **explode()**, you can cleanly split the address on the **@** symbol , keeping the domain while discarding the user name, as follows:

```
$email = 'user@example.com';
list(, $domain) = explode('@', $email);
```

Performing the split with **explode()** is sensible since we are not dealing with a regular expression. For those situations where a regular expression is needed, **preg_split()** should be your choice over **split()**, according to the official source for all things PHP, **www.php.net**.

Now that I've shown you how to extract the domain name, let's proceed with investigating whether it is genuine. If it proves to be inauthentic, then we are finished. And, I can give my computer a break.

What Is A Domain?

When setting out to determine whether a domain is valid, we need to consider what a domain actually is.

If you set up a website and purchase a name for it, that name is its domain name. For example, when a business seeks a Web presence, it needs to choose a top level domain name (TLD) and a second level domain. The obvious choice for the TLD is frequently **com**, while designating the *second level domain* requires more thought. So, in the case of **phparch.com**, the second level domain is **phparch**. Prepending that with **www** results in the full URL, **www.phparch.com**.

The *Domain Name System* (DNS) is what makes it possible for you to type a URL into the location area of your browser and be directed to the correct site. The way DNS works is that there is a distributed database of IP addresses and corresponding human-readable names across the Internet, spread over millions of computers whose job is to function as *domain name servers*. A domain name server translates a URL into an IP address, a machine-readable string of numbers and dots.

DNS and Email

By means of DNS, we can determine whether a domain name corresponds to a real domain. If you query a domain name server, you can see what, if any, records it has for a specific domain. Such records may include whether there is a mail server, denoted by an MX record. However, if that record type is missing, we should check to see if there is an A record. Such a record indicates the IP address of the domain's host, which may be treated as if it were the mail server (see RFC 2821, Sections 3.6 and 5.). Knowing whether a domain possesses MX or A resource records (RR) informs us about its validity.

At this point, we are still considering whether *user@example.com* contains a valid domain. So, let us use the DNS system and see what we might learn. First, I will show you how do this manually on Linux so that we can

understand how to do the same thing later using PHP.

DNS Resolution

If you're running a standard Linux distro, you may choose to use one of the BIND (Berkeley Internet Name Domain) DNS resolving utilities such as **host** or **dig** (Domain Information Groper). The man pages describe **host** as a simple utility, whereas **dig** is referred to as a flexible tool. The syntax of either utility is straightforward; which of them you choose is a matter of preference.

Using **host** to detect an MX record simply involves typing on the command line:

```
host -t MX example.com
```

You may optionally append a period to the domain. This ensures that you are querying only for DNS records specific to the fully qualified domain name (FQDN) **example.com** and not, for instance, **example.com.whatever.net**. If there are any **MX** records belonging to **example.com**, they will be listed in the output.

I used this command, without the **-t** parameter, on an email domain whose website is now defunct following two mergers and acquisitions. I expected that **host** would by default search for an **A** RR and find nothing. Instead, the output contained the names of five mail exchange servers—all belonging to the previous company that had acquired the domain!

If you prefer not to have to remember to use a switch like **-t**, try using **dig**. At the command prompt, **dig** syntax is simpler than that of **host**:

```
dig [<DNS server>] <domain name> [<type>]
```

If you omit the **type** argument, the default behaviour is for the query to check for the **A** record.

Having dwelled on the mechanics of using **host** and **dig**, we should now have an easier time as we make our way toward accomplishing the same feat using PHP.

"The main advantage of using dig with PHP is that it involves little code, and it's fast."

To The Web

If you set **$domain** to **"gmail.com"**, see what the following code yields:

```
exec("dig " . escapeshellarg($domain). " MX", $ips);
var_dump($ips);
```

On my shared hosting account, the script outputs an empty array because my webhost is thankfully, concerned about security and disallows **exec()** and back ticks. You should have better luck with this snippet, if you have a dedicated server and it's under your own control.

If the string value of **$domain** is derived from user input, it is best to escape it by using **escapeshellarg()**, thereby rendering it harmless in the shell environment. Single quotes will envelop the entire string, and single quotes within the string are themselves quoted.

The results of the call to **dig** will populate **$ips**, which—being an array—we can manipulate. If back ticks are available, you could achieve similar results. It's just a little more work, since you would need to parse the output.

The main advantage of using **dig** with PHP is that it involves little code, and it's fast. The result will quickly

LISTING 1

```
<?php

error_reporting(E_ALL|E_STRICT);

require 'listing2.php';
require 'listing3.php';

$email = array();

// Insert values for 3 email addresses you'd like to test
$emails[0] = '';
$emails[1] = '';
$emails[2] = '';

foreach ($emails as $email) {

    try {
        $host = getMXServer($email);
        if ($host) {
            $mboxChecker = new mailBoxChecker($email, $host);
            if ($mboxChecker) {
                $mboxChecker->checkMailBox();
                $mboxChecker->setActive();
                $mboxChecker->report();
            }
        }
    } catch(Exception $e) {
        echo $e->getMessage()."<br />\n";
    }
}

?>
```

confirm if **$domain** has an MX record. If not, you can easily check if the domain even has an IP address by substituting an **A** for **MX** in this code snippet. Or, try the **ANY** parameter instead, which will tell you whether the domain name has any DNS records.

Obviously, none of the code snippets I've used so far are pure PHP solutions. To implement them, you would need to be running a standard Linux distro with the **dig** and/or **host** utilities installed. In addition, you must have unrestricted use of **exec()**. All in all, you may find an entirely PHP-based solution a bit more practical as well as interesting.

Pure PHP

PHP has an abundance of built-in networking functions, so the immediate question is which of them will provide meaningful information about a domain and its mail servers. To see if there is an MX record associated with the domain, my inclination is to use a combination of **checkdnsrr()** and **getmxrr()**.

checkdnsrr(), as its name implies, checks for the existence of DNS records. Since its only job is to return a Boolean result, it's fast. The result indicates whether the specified type could be found. What surprised me when I ran my application (see Listings 1, 2 and 3) was that valid email addresses were sometimes rejected. I blamed **checkdnsrr()** initially and accused it of returning false results.

Since then, I believe that innocent function is blameless. When I turn on error reporting, I sometimes see a mysterious message when I re-run the application:

```
Notice: fputs() [function.fputs]: send of 6 bytes
failed with errno=32 Broken pipe".
```

Obviously **fputs()** has been unable to finish reading the socket. I tried using **stream_set_blocking()**, which is supposed to help **fputs()** finish its job, but that didn't work—perhaps that function still has a bug, as has been previously reported.

I am getting ahead of myself. So, let's get back to how to obtain an MX record
in PHP. The following code aims to do precisely that for **$domain**:

```
if (checkdnsrr($domain, 'MX')) {
    getmxrr($domain, $mxhosts);
    $host = (count($mxhosts)) ? $mxhosts[0] : null;
}
```

As the code shows, when there is an MX record, we can then obtain it with the aid of **getmxrr()**. When invoked, **$mxhosts** will automatically be filled in for you with an array of data.

Each element in the **$mxhosts** array contains a mail exchange URL. However, mail servers have a priority

LISTING 2

```
<?php

function getMXServer($email)
{
    $host = '';

    if (!$email) {
        throw new Exception('No email address given');
    }

    // split $email to get the domain
    list(, $domain) =   explode('@', $email);

    // a windows solution
    if (!function_exists('checkdnsrr')) {
        $host = getWinRR($domain);
    } else {
        // a linux solution
        if (checkdnsrr($domain, 'MX')) {
            getmxrr($domain, $mxhosts, $weight);
            asort($weight);
            $priority_key = key($weight);
            $host = (count($mxhosts)) ? $mxhosts[ $priority_key ] : null;
        } else {
            if (checkdnsrr($domain, 'A')) {
               $host = $domain;
            }
        }
    }

    if (!$host) {
        throw new Exception("$email appears to be invalid. Did you make
a mistake?");
    }

    return $host;
}

```

LISTING 2: Continued...

```
function getWinRR($domain, $rr='MX')
{
$isMX = false;
$isA = false;
$host = false;

    exec("nslookup -type=$rr $domain", $output);

    foreach ($output as $record) {

        $arr = explode(' ', $record);
        $isMX = (($rr == 'MX') && strstr($record, $rr));

        if (!$isMX) {
            $isA = ( strstr($record, 'Name') );
        }

        if ($isMX ) {
            $arraySize = sizeof($arr) - 1;
            /* Note: this will return an MX at random
               rather than the favoured MX server */
            $host = $arr[ $arraySize ];
            break;
        } else {
            if ($isA) {
                $host = $domain;
            }
        }
    }
    return $host;
}

?>
```

LISTING 3

```
<?php

class mailBoxChecker
{
    const PORT=25;
    const SUCCESS=220;
    const OKAY=250;
    const ALLGOOD=3;

    private $timeout = 30; // number of seconds
    private $mbox_status = 0;
    private $email;
    private $host;
    private $active;
    private $converstion;

    public function __construct($email, $host)
    {
        $this->email = $email;
        $client = (isset($_SERVER['HOSTNAME']))? $_SERVER['HOSTNAME']
: $_SERVER['HTTP_HOST'];

        if (!$client) {
            exit('Unable to determine client host name');
        }

        $this->conversation = array(
                              "HELO $client\r\n",
                              "MAIL FROM: <{$email}>\r\n",
                              "RCPT TO: <{$email}>\r\n",
                              );
        $this->host  = $host;
        $this->fp = @fsockopen($this->host,
                               self::PORT,
                               $errNum,
                               $errMsg,
                               $this->timeout);

        if (!$this->fp){
            throw new Exception('System error - ' .
                                '$errNum $errMsg');
        }
    }

    private function getEmail()
    {
        return $this->email;
    }

    private function setTimeout($num)
    {
        $this->timeout = $num;
    }

    private function getTimeout()
    {
        return $this->timeout;
    }

    private function getFp()
    {
        return $this->fp;
    }

    private function getConversation()
    {
        return $this->conversation;
    }

    private function getMbox_status()
    {
        return $this->mbox_status;
    }

    private function setMbox_status()
    {
        $this->mbox_status++;
    }

    private function getActive()
    {
        return $this->active;
    }

    public function setActive()
    {
        $bool = ($this->getMbox_status() == self::ALLGOOD)? true :
```

rating, and **$mxhosts** may not reflect the priority order. The mail server with the lowest priority rating is the preferred one to actually contact. With a little more work, let's modify the code by inserting an optional third parameter, **$weight**:

```
if (checkdnsrr($domain, 'MX')) {
    getmxrr($domain, $mxhosts, $weight);
    asort($weight);
    $key = key($weight);
    $host = (count($mxhosts)) ? $mxhosts[$key] :
null;
}
```

asort() comes in handy here, as it sorts the **$weight** array in ascending order while still maintaining key associations. Now we can get the key corresponding to the lowest priority value, and so we may contact the preferred server.

When **checkdnsrr()** (or **dns_check_record()** in PHP

LISTING 3: Continued...

```
false;
        $this->active = $bool;
    }

    public function report()
    {
        $email = $this->getEmail();
        $resBln = $this->getActive();
        $result = ($resBln === true)? "true" : "false";
        echo "\n<p>Is $email an active mailbox? $result</p>\n";
    }

    public function checkMailBox()
    {
        $commands = $this->getConversation();
        $this->setTimeout(5);
        $fp = $this->getFp();

        if (!stream_set_timeout($fp, $this->getTimeout())) {
            throw new Exception('Unable to set stream timeout');
        }

        $intVal = intval(fgets($fp));

        if ((self::SUCCESS !== $intVal) && (self::OKAY !== $intVal)) {
            throw new Exception('Server has refused connection');
        }

        // starting the conversation ...
        for ($i = 0, $max = count($commands); $i < $max; $i++) {
            fputs($fp, $commands[$i]);
            $intVal = intval(fgets($fp));

            if (($intVal === self::SUCCESS) || ($intVal === self::OKAY))
{
                $this->setMbox_status();
            }
        }

        fputs($fp, "QUIT\r\n");
        $meta_data = stream_get_meta_data($fp);
        fclose($fp);

        if ($meta_data['timed_out']) {
            throw new Exception('Timeout occurred while reading or
writing data');
        }
    }

} // end class

?>
```

5), returns **FALSE**, we should set the value of its second parameter to **A**. If the result yields **TRUE**, then we can safely consider the domain itself as the mail server.

If we wanted the actual data contained in the A record, i.e. the host IP address for the domain, we could fetch it with **dns_get_record($domain, DNS_A)**. Note that **DNS_A** is a constant, as opposed to the second parameter in **checkdnsrr()** which is a string.

"One question that may arise is what kind of a domain name makes an acceptable argument for HELO."

What About Windows?

If you are running PHP under Windows, the companion functions **checkdnsrr()** and **getmxrr()** are unavailable. Attempting to use either function will result in PHP notifying you about a "Call to undefined function." Fortunately, there is another way you can retrieve the DNS data for a host. One of the BIND utilities that has a Windows version is **nslookup** which, as its name hints, will do a name server lookup. In fact, **nslookup.exe** comes as standard with the Windows operating system.

If you wanted to create a Windows-friendly **checkdnsrr()** function, here's one way to do it using **nslookup**:

```
function getWinRR ($domain, $rr='MX'){
    exec("nslookup -type=$rr $domain", $output);
    foreach($output as $record) {
        if (strstr($record, $rr)) {
            return true;
        }
    }
    return false;
}
```

In contemplating the code for **getWinRR()**, a few thoughts come to mind.

You may wonder why I use **strstr()**, instead of a function like **preg_match()**. My primary reason is speed. Built-in functions that don't require a regex will always be faster than those that do, owing to the nature of the way regex technology works.

At present, **getWinRR()** is a nice starting point but the function could benefit from further enhancement to allow you to obtain the actual DNS data. You may peruse an improved version, which includes parsing the results with PHP, in Listing 2.

Of course, this function depends mightily on a developer's having the ability to use **exec()** in her working environment. An alternative would be to try the PEAR class **Net_DNS**, a port of the PERL module NET::DNS. The class merits your consideration since it uses a socket to make a connection instead of relying on system commands like **exec()**.

Talking With Strangers

Suppose that **example.com** was indeed a valid domain. Whether we are done, will depend on how certain we need to be that *user@example.com* represents an existing mailbox. Let's assume it is worthwhile to go the extra mile.

To achieve this end, we will use SMTP (Simple Mail Transfer Protocol) to have a conversation with the mail server at **example.com**. The dialogue will be initiated as if our intent were to send *user@example.com* an email from itself, no less. This step is important, for it will clarify the status of the mailbox. Having such a dialogue will require making a connection to the server on port 25. Here's the text of how such a conversation might have transpired between another non-existent website named **b.com** and **mailserver.example.com**:

```
Client: HELO www.b.com
Server: 220 system in order
Client: MAIL FROM: <user@example.com>
Server: 250 Hello b.com
Client: RCPT TO: <user@example.com>
Server: 250 OK
Client: QUIT
```

As you can see, **b.com** initiates a brief conversation with the server by issuing a series of commands and awaits a response for each. **HELO** alerts the server that a mail session has started. Although we could try **EHLO** instead, I've found that that gives less reliable results; not every MX server implements support for it. If the server responds with a **220**, meaning **READY**, I know that the server has accepted the SMTP connection and further communication is possible.

One question that may arise is what kind of a domain name makes an acceptable argument for **HELO**. RFC 2821 (Section 2.3.5 Domain) clearly states "A domain name that is not in FQDN form is no more than a local alias. Local aliases MUST NOT appear in any SMTP transaction." However, another techie I've consulted with in

writing this piece firmly believes that many mail servers will accept unqualified domain names, too. At this point, the only thing we both agree on is that most MX servers will not accept **localhost** as an argument to **HELO**.

Returning to the server dialogue, the client **b.com** next tells the server over at **example.com** to accept mail from a specific email address by issuing a **MAIL FROM:** command. Note, the **HELO** argument might be different from the **MAIL FROM** information. SMTP permits email to be sent or relayed from a user to even itself by means of another website. In the dialogue depicted here, the server accepts **b.com** as a fully qualified domain and compliantly agrees to deliver email from *user@example.com* to that mailbox. (Incidentally, silly as it may sound, sending email from yourself to yourself can be beneficial, particularly when doing a lot of multi-tasking.)

RCPT TO instructs the server regarding the recipient of the email. The server is amenable, responding with another **250**. The mailbox appears to be active, so there is nothing more to do except end the session with a **QUIT** command.

It may seem a tad deceptive to lead the server into thinking that **b.com** wishes to send an email, when we only want to know the status of a mailbox. While the SMTP protocol would seem to provide a more straightforward solution with its **VERIFY**/**VRFY** command (see RFC 2821, section 4.1.1.6), many servers disallow it. If given the wildcard * parameter, **VERIFY** provides the email addresses of all the users on a server, making it an unwitting accomplice of spammers.

Returning to the imagined dialogue, we see what should occur when there are no errors. But, what if I had made a simple typo, such as transposing a couple of letters in the user name of the email address? In that case, expect that the server would complain with a **550** message, meaning that the mailbox is unavailable.

Time Out

Now that we understand how to dialogue with a server using a telnet session, let's do it again—only with PHP this time. In order to dialogue with the server—whether from the command line or from a script—we need to establish a connection to the server on port 25. PHP provides a fine function for this purpose: **fsockopen()**. We could establish a connection with code as brief as this:

```
$host = 'mail.example.com';
$port = 25;

$fp = @fsockopen($host, $port);
if (!$fp) {
    exit("Unable to connect at this time... try
again later\n");
}
```

But what if the connection times out? How would we know that?

That actually happened to me after relying on a similar script to connect to a server overseas. When I ran the same script on a domestic server, back in the USA, I became aware of the timeout issue. To get around it, we need to modify the code:

```
$host = 'mail.example.com';
$port = 25;
$timeout = 30; // seconds

$fp = @fsockopen($host, $port, $errNum, $errMsg,
$timeout);
if (!$fp) {
    exit("System error $errNum: $errMsg\n");
}
```

The **$timeout** parameter tells our script how long to keep trying to make a connection before giving up. If an error occurs, **$errNum** and **$errMsg** will both be automatically set. According to the PHP manual, the **timeout** parameter will not necessarily be understood in all environments; but in most situations, setting **$timeout** will prevent your script hanging.

To retrieve the server's response to the connection we need to read from the socket. We're looking for an immediate server greeting of **220**, and we can safely ignore anything else that may follow. The best function to do that is **fgets()**:

```
$success = 220;
$response = fgets($fp);

if ($success == intval($response)) {
    echo "connection accepted\n";
}
```

To obtain the response, we have a variety of functions from which to choose, including those involving regular expressions. Since the server always prefaces a response with a three-digit numeric code, I use **intval()** to detect whether or not the server responds with **220**, meaning success. A response code of **451** would denote failure at this point (see RFC 2821).

If I were actually expecting to receive lines exceeding 8K, the code would make use of the optional second parameter for line length. Allowing PHP to continue reading a line length greater than 8K without passing in the optional parameter is more resource

intensive, and hence inefficient.

Another place where a timeout may occur is when reading from or writing to the pipe that **fsockopen()** creates between your machine and the mail server. You can deal with that issue by using two companion functions from the streams API: **stream_set_timeout()** and **stream_get_meta_data()**. **stream_set_timeout** operates on the stream itself, whereas the **timeout** parameter in **fsockopen()** only operates while making the connection to the server. The first element of the array returned by **stream_get_meta_data()** contains the timeout status for the stream.

> **"Common server variables ... do not always exist, despite what it says in the CGI 1.1 specification."**

A Note About _SERVER

Any server-side dialog script will need to be cautious when it comes to using server variables, particularly in the **HELO** greeting. It's not safe to assume that every environment will have **$_SERVER['HTTP_HOST']**, for example. I once came across a terminal script that consistently timed out whenever there was a brief period of inactivity. The script appeared to work fine except that, for some mysterious reason, email addresses I knew were valid were assessed otherwise.

In attempting to debug that script, I did a **var_dump()** of **$_SERVER** and noticed that common server variables such as **$_SERVER['SERVER_NAME']** did not appear to exist, despite what it says in the CGI 1.1 specification. Since the box was not my own, I assumed that this was due to the configuration of that particular system. However, as the PHP manual has it, if you run a script that uses server variables in a CLI environment, "... few, if any, of these will be available." Providing **HELO** with the correct information when identifying your server is critical, otherwise the results may be inaccurate.

In order to tie all the scripts presented so far together, you will find an email verifier script in Listing 1 which includes the two files whose code is available in Listing 2 and Listing 3. I hope you will enjoy trying them out. To run them, I suggest that you use a regular www-server. Oh, and do keep **error_reporting** on, at least for the first trial run.

On Reliability

There are three variables for email addresses in Listing 1. Set the variables to use email addresses of your choice. When I tested the code I used one valid email address and two that I knew were bogus. One of those was fake because it represented the good one distorted by a deliberate typo. The other one I knew was completely phony. I decided to test the verification program on a Web server, both as a Web application and as a command line script (CLI).

From my US-based box, the program correctly verified the one active mailbox and rejected the other two, whether run as a CLI script or as a Web application. Encouraged by these results, I decided to test the script from a virtual host based in the UK. To my surprise, the results were erratic there, and sometimes produced false positives if I simply refreshed the browser. The only noticeable difference between the two boxes is that one uses anti-spam greylisting, which may possibly explain some false positive results. Or not. Perhaps the discrepancy is due to something else entirely. Owing to networking minutiae that may differ from one domain to another, the seemingly simple task of email verification can produce questionable results.

Of the two steps involved in email verification, the first seems far more reliable than the second; it exposes the validity of the domain with pretty good precision. The second step, however, can be very hit and miss. It can yield false results—positive or negative—as well as providing accurate information. The false negative we can do something about, in that it's possible to fall back on an exchange of email with the user, but the false positive may be beyond one's control.

These results, then, are not absolutes. They could be made more reliable if you only took the second step with mail servers that have proven themselves consistently reliable over time. In other words, you'd need a domain whitelist—and you may find that, even then, changes in your server environment could impact the level of reliability.

From Theory Into Practice

The way you use email verification, and the extent to which you use it, is likely to depend on the context. If a user plans to become a member of your website the most rigorous sort of verification would be in order, which would involve sending the user an email and asking for a response. The site ought to explain this, and should also cover what to do if user does not receive the email, i.e. checking whether the email may have landed in his spam folder. This sort of rigor is necessary to avoid storing a fake email address or worse, one that exists but does not belong to the user.

Another situation might involve a "Send A Friend" feature. This typically has the user provide a friend's email address through a form, and that data may or may not be stored. If neither the sender nor sendee's email will be stored, then simply checking whether their domains are valid may be all that is realistically needed.

A "Contact Us" feature also raises the question of how much email verification there needs to be. If you simply want to collect feedback and have no intention of replying to individuals, the halfway solution of checking the domain may be enough. However, if you intend to actually reply to the user or store the email address, the extra step of the serverside chat may be warranted.

Since there is an uncertainty factor when it comes to that extra step, probably email verification should be relied upon at most as a way to help users avoid typos when providing an email address. You may argue that it's sufficient to require that users enter the email address twice in succession. If the strings do not match, obviously there is a typo. But, what about the case of a user making the same typo twice in succession? Perhaps the domain name is a commonly misspelled word, for example, or perhaps the user has multiple email addresses, with some ending in **.com** and others in **.net** or **.org**. With email verification, that kind of typo can readily be detected.

Finally, it's important how you respond to your users if their email addresses prove unverifiable. An accusatory tone can quickly alienate users and cause them to feel negative towards your site. A helpful approach is to respond to failed verification with an informative message, such as: "There seems to be a typo in the email address you've provided. Are you sure you wish to submit it as-is?" This kind of approach gives the user the opportunity to check whether any correction is necessary—and a website with happy users spells happiness for its developers.

SHARON LEVY is a self-taught PHP developer and evangelist who has been doing Web development professionally since the glory days of the dot-com boom. In its aftermath, she prefers working with open source technologies such as PHP. Sharon applies solutions to companies using Linux or Windows, in industries ranging from avionics to finance and e-commerce. Her formal education includes a B.A. from UCLA and a certificate in C/UNIX.

Python Magazine

And now for something completely different

The first monthly magazine dedicated exclusively to Python.

The first issue is on us

Get a **free PDF copy** of the October issue.
No Purchase & no registration required
(because we know you'll be back anyway).

Print & PDF (1 year, 12 issues)	PDF only (1 year, 12 issues)
US & Canada: $69.99 CAD International: $89.99 CAD	Worldwide: $59.99 CAD

SUBSCRIBE TODAY!

For more info go to:
http://www.pythonmagazine.com

Migrating PHP, part II: PHP Code

by Stefan Priebsch

As PHP 4 support will be discontinued on August 8th, 2008, now is the time to migrate your legacy PHP applications to PHP 5. In last month's issue, I covered some important aspects of the server environment that you should keep in mind when preparing a PHP migration. In this article, I will show you code that is likely to cause problems when migrating and provide you with solutions to make your application work on the target system.

There are three levels of problems that PHP code can cause when migrating.

First off, PHP code that used to work fine may not compile on the target system. This is usually due to naming conflicts, which can be resolved rather easily.

Second, the code may throw new errors, warnings or notices on the target system. It is very important that you configure PHP to output all error classes. In most cases, an **E_WARNING** or **E_NOTICE** is an indicator of a potential problem in your code that could land you in hot water further down the line if you choose to ignore it. To be able to distinguish new error messages from existing ones, you should begin the migration by either fixing all existing error messages or at least documenting them, so that you don't wrongly attribute them to the new environment.

Last but not least, once the code compiles and works without any errors, warnings or notices, you have to make sure that the application's behaviour is still the same. This can be done by comparing test results computed on the original system and the target system.

While most of the issues mentioned in this article are not too serious in themselves, ignoring them often leads to hard-to-find errors that show up somewhere else, leading to long and tedious debugging sessions. By paying attention to the many small details, you can avoid many migration problems—or at least, discover their origin more quickly.

PHP: *5.2.1+*

TO DISCUSS THIS ARTICLE VISIT:
http://c7y-bb.phparchitect.com/viewforum.php?f=10

Case Sensitivity

As we know, operating systems treat case sensitivity differently. Windows, for example, is case insensitive, while Unix is case sensitive. PHP has a very pragmatic approach to case sensitivity, meaning that some aspects are case sensitive while others are case insensitive. Fortunately, this behaviour does not depend on the operating system PHP runs on, except for the file handling, which is always system dependent.

Variable names in PHP are *always* case sensitive. Interestingly, constants are also case sensitive, although by convention they are usually written in upper case letters. When defining a constant, you can use an optional boolean parameter that forces the constant to be case insensitive:

```
<?php

define('TEST', 3, true);
define('test', 4);

var_dump(TEST);
var_dump(test);
```

PHP will issue an **E_NOTICE** when running the above snippet, as you are trying to redefine an already existing constant:

```
Notice: Constant test already defined in test.php on
line 4
int(3)
int(3)
```

Unfortunately, many programs configure PHP so that **E_NOTICE** errors are not displayed. Having received no warning message, you would not necessarily know that the **define()** in line 4 had no effect, causing the script to subsequently run with a wrong value.

PHP 4 has some so-called magic constants, **__LINE__**, **__FILE__**, **__FUNCTION__** and **__CLASS__**. These were joined by **__METHOD__** from PHP 5.0 on. Starting with PHP 5.3, two more magic constants will be available: **__DIR__** and **__NAMESPACE__**. The names of magic constants always begin and end with two underscores. Magic constants are also always case insensitive, as opposed to normal constants.

One catch to watch out for is that **__FUNCTION__** and **__CLASS__** used to return lower case characters in PHP 4, but in PHP 5 will return the names as they were defined. If you need to retain PHP 4 behaviour for string comparisons and so on, use **strtolower()** to convert the result to lower case characters.

As with function and method names, PHP treats class names as case insensitive. This encourages sluggish programming, because you can write class names in varying case while still referring to the same class:

```
class Test
{
    public function __construct()
    {
        var_dump('class Test');
    }
}

$test = new test;
$test = new Test;
$test = new TEST;
```

PHP instantiates the same class three times:

```
string(10) "class Test"
string(10) "class Test"
string(10) "class Test"
```

When you're mapping class names to file names, which is a common approach in PHP OO, case can become a real issue because the underlying operating system may be case sensitive or case insensitive. This can lead to files not being found on Unix, while they are found on Windows.

To avoid such cross-platform issues, you should never use file names that only differ in casing. Although this is possible and perfectly valid on Unix, you will run into trouble if you try to copy your application to Windows. Migrating applications from Unix to Windows is not always a trivial operation, not least because Unix allows more special characters in file names than Windows. Unix basically allows every special character except the directory separator (which is the forward slash /). Windows is more restrictive and disallows a number of special characters: question marks, single quotes, colons and asterisks—and you can't create a file under Windows whose name begins with a period. I would recommend only using alphanumeric file names with underscores to ensure cross-platform portability.

Naming Conflicts

Reserved keywords have a special purpose in PHP. When PHP compiles the source code to executable code, the lexer scans the source and translates the plain text into so-called tokens, in much the same way as **token_get_all()** does. To find its way through the source code, the parser uses a number of reserved keywords as anchor points. If a reserved keyword appears as an identifier (i.e. function or method name, class or constant) in the source code, the parser gets confused and fails to compile the PHP script. Therefore, when migrating PHP code, naming conflicts due to new

reserved keywords on the target system can occur.

Although it is possible to define a constant with a keyword as its name—because the identifier is enclosed in quotes—accessing that constant will not work, as the following example shows:

```
<?php

define('case', 3);
var_dump(case);
```

Depending on your parser, the output will be either:

```
Parse error: syntax error, unexpected T_CASE, ex-
pecting ')' in test.php on line 4
```

or

```
Parse error: parse error, expecting `')'' in test.
php on line 4
```

In either case the error occurs in line 4, where we try to output the constant, and not in line 3, where we define it. Even namespaces, finally available with PHP 5.3, will not protect you from conflicts with reserved keywords.

"Even namespaces, finally available with PHP 5.3, will not protect you from conflicts with reserved keywords."

To avoid naming conflicts with keywords, you should prefix all your class, function and constant names. When it comes to method names, though, using a prefix is not very nice:

```
class fooTest
{
    public function fooNew()
    {
        echo "this is fooTest::fooNew()\n";
    }
}
```

To avoid this, rather than prefixing your method names you should use composite names consisting of at least two words. Using a verb and a noun—**doFoo()**, **registerFoo()**, **importFoo()**—will also help make the purpose of the method clearer while greatly reducing the risk of name conflicts. In future PHP versions, the limitation that keywords may not be used as identifiers may be removed, but currently this is an issue we will have to live with.

In addition to potential naming conflicts with keywords, naming conflicts are also possible with functions, classes, and constants. For obvious reasons, user functions in PHP are not allowed to match the names of built-in PHP functions. The problem is that PHP extensions can register function names too, so you can never really be sure which function names will be taken by a given PHP installation. To give you an idea of the magnitude of the problem, over 1500 functions are defined on my development system, which has a mere handful of PHP extensions loaded. You can output a multi-dimensional array of all the defined functions in your own environment—both **internal** and **user**—by calling **print_r(get_defined_functions())**.

Happily, most PHP extension authors stick to the rule of prefixing their function names with the extension name followed by an underscore. You should do the same with your user functions in the global namespace: use a unique prefix to avoid naming conflicts. Choosing a prefix boils down to some guessing, though, because you can never know which PHP extensions might be activated in the PHP installation your application runs on. Refrain from using library names such as *imap*, *ldap*, *odbc* or *mysql* as prefixes in your own code, as they are very likely to be names of PHP extensions.

Naming conflicts with classes depend on the PHP installation too, since extensions can also register classes, interfaces and exceptions. You can use the following script to list all the class and interface names currently used in your environment. Since exceptions are also objects, they will appear in the list as well:

```
var_dump(implode(', ',
                 array_merge(get_declared_classes(),
                             get_declared_inter-
faces())));
```

On my PHP 5.2 installation, calling that line results in a list of 128 names, some of which have great potential for naming conflicts: **ArrayObject**, **DateTime**, **Directory**, **DOMElement**, **Iterator**, **Serializable**, and (under Windows) **com**. To avoid naming conflicts, prefix all your class names. Either that, or become an early adopter of PHP 5.3 and use namespaces.

Fortunately, naming conflicts are usually rather easy to spot, since they generally result in a fatal error that names both the duplicator and the duplicated (**cannot redeclare...**). There are some cases, though, where class name conflicts can lead to strange runtime errors. This happens when two classes/files on your system have the same name, the wrong file is inadvertently loaded, and you thus try to call a non-existing class method or access a non-existing member.

array_merge()

The built-in function **array_merge()**, as you probably know, merges one or more arrays. While PHP 4 silently converts any non-array arguments to arrays, in PHP 5 **array_merge()** expects all parameters passed to the function to actually *be* arrays:

```
<?php

$a = array(1, 2, 3);
$b = 4;
var_dump(array_merge($a, $b));
```

Under PHP 4, the output from this code is:

```
array(4) {
  [0]=>
  int(1)
  [1]=>
  int(2)
  [2]=>
  int(3)
  [3]=>
  int(4)
}
```

Under all versions of PHP 5, the code triggers a warning and the result is empty:

```
Warning: array_merge(): Argument #2 is not an array
in test.php on line 3
NULL
```

To avoid problems due to this change, you could write a wrapper function and replace all occurrences of **array_merge()** with a call to this wrapper function.

Since **array_merge()** can take a variable amount of parameters, the wrapper function is necessarily a little complicated, as you can see in Listing 1. The other alternative of course would be to make certain that every single argument to **array_merge()** in your legacy code is actually an array.

ip2long()

Another built-in PHP function, **ip2long()**, converts an IP address given in the common format of four numbers separated by a dot to a 32-bit number. On PHP 4, the function returns **-1** if the string passed in was not a valid IP address. PHP 5 returns **FALSE** instead. To fix your code, you should replace any checks for **-1** following a call to **ip2long()** in your application with a type-safe comparison with **FALSE**:

```
if (false === ip2long(...)) ...
```

strrpos() and strripos()

Under PHP 4, **strrpos()** finds the last occurrence of a given character (needle) in a string (haystack) and returns the index value of that character. This

LISTING 1

```
<?php

function array_merge_compat()
{
    $num = func_num_args();

    if (0 == $num) {
        return NULL;
    }

    if (1 == $num) {
        return func_get_arg(0);
    }

    $result = func_get_arg(0);

    for ($i = 1; $i < $num; $i++) {
        $arg = func_get_arg($i);

        if (NULL == $arg) {
            continue;
        }

        if (is_array($arg)) {
            $result = array_merge($result, $arg);
        } else {
            $result = array_merge($result, array($arg));
        }
    }

    return $result;
}

// and test...
$a = array(1, 2, 3);
$b = 4;
var_dump(array_merge_compat($a, $b));

?>
```

functionality has altered in PHP 5. Whereas PHP 4 only searched for a single character—and, if given more than one character as the search string, the only character sought would be the first in the 'needle'—PHP 5 will return the index value of the first character in the matched substring. This can lead to very different results:

```
echo strrpos('elephpant', 'php');
```

Under PHP 4 the output is **5** (matching the final **p** in **elephpant**), while in PHP 5 the output is **3** (matching the start of **php** in **elephpant**). In order to have the same code return the same results under both PHP 4 and 5, you should secure every call to **strrpos()** that concerns more than one 'needle' character with a **substr()** statement returning the first character of the string to search for. Again, we can easily write a wrapper function:

```
function strrpos_compat($haystack, $needle, $offset
= 0) {
    return strrpos($haystack, substr($needle, 0, 1),
$offset);
}
```

The same logic holds true for the case insensitive sister function, **strripos()**.

strtotime()

The function **strtotime()** attempts to parse a date description in English and convert the result to a Unix timestamp. As with **ip2long()**, whereas PHP 4 returned **-1** to indicate an error, PHP 5 will return **FALSE**. Just like with **ip2long()**, you should modify any checks for **-1** in conjunction with calls to **strtotime()** to use a type-safe comparison.

$this

Inside an object instance, **$this** refers to that object instance. In PHP 4, **$this** could also be used as a local variable, although of course it never was recommended usage. In PHP 5, using **$this** outside an object context triggers a fatal error:

```
PHP Fatal error: Cannot re-assign $this in test.php
on line...
```

The solution to this problem is simple: rename your variable to something other than **$this** to avoid the naming conflict.

Older PHP versions also allowed redefining **$this** inside a method, thereby effectively changing the class of the object instance, as in Listing 2. In PHP 4, that approach works and **$test** becomes a **Something** object:

```
object(something)(0) {
}
```

In PHP 5, the same script bails out with a fatal error:

```
Fatal error: Cannot re-assign $this in test.php on
line 10
```

Here, redefining **$this** won't work; you can't overwrite the current object. You need to explicitly create another object with **new** and return a reference to it. There are a number of ways to do this, depending on what you actually hoped to achieve:

```
    public function doTest()
    {
        return new Something;
    }
...
$test = new Test;
var_dump($test->doTest());
```

```
    public function doTest()
    {
        $this-obj = new Something;
    }
...
$test = new Test;
$test->doTest();
var_dump($test->obj);
```

but most likely you'd be looking at a factory approach:

```
      public static function doTest() {
          return new Something;
      }
...
$test = new Test;
$test = Test::doTest();
var_dump($test);
```

LISTING 2

```
1. <?php
2. // PHP 4 code
3.
4. class Something {}
5.
6. class Test
7. {
8.     function doTest()
9.     {
10.        $this = new Something;
11.    }
12. }
13.
14. $test = new Test;
15. $test->doTest();
16. var_dump($test);
17.
18. ?>
19.
```

Comparing Objects

There are two ways to compare objects. When using the **==** operator, PHP will check whether two objects have the same class and the same members. Using the strict comparison operator **===**, PHP will check whether two object references actually point to the same object instance.

The non-strict comparison has been changed in PHP 5.2 to recursively compare all members. As a result, when encountering circular references, PHP can fall into an endless loop. Fortunately, this will be detected:

```
<?php

class Test
{
    var $test;
}

$t1 = new Test;
$t2 = new Test;

$t1->test = $t2;
$t2->test = $t1;

var_dump($t1 == $t2);
```

In PHP 5.2, this script will output the following error message:

```
Fatal error: Nesting level too deep–recursive depen-
dency? in test.php on line 13
```

In older PHP 5 versions, or in PHP 4, the same script will work perfectly and output:

```
bool(false)
```

A simple solution to this problem, if the application logic allows it, is to replace **==** with **===**. If this is not possible, you may have to create a custom **compare()** method that compares the relevant members, but avoids the endless recursion.

Dynamically Loaded Code

Most PHP applications consist of a number of files, not just one. This makes managing the source code easier, since each file can be edited independently from the others. For object-oriented code, it is recommended practice to put each class into one file. This allows for selective loading of classes, for example with an autoload handler.

In PHP 4, you could load function definitions using **include** or **require** multiple times without seeing an error. In PHP 5, any attempt to define a class or function that has already been defined leads to a fatal error. If you encounter this error, you should reorganize your code so that included files only contain code inside functions and classes, and load them using **include_once** or **require_once**. This way, PHP will ensure that every file is indeed loaded only once and there will be no redefines.

Should you work with code in the global scope (outside functions and classes), as is often the case with templates, make sure not to define any functions or classes in the files that contain such code. This will allow you to load the code with **include** or **require** as often as you need, without any danger of a fatal error due to redefined functions or classes. It should be noted, though, that this is not really good programming style. I would recommend encapsulating the code in a function, loading the function once, and then calling it multiple times.

"Non-strict object comparison has been changed in PHP 5.2 to recursively compare all members."

Where PHP code uses interfaces—a new language feature in PHP 5—classes must be defined before they are used. Although PHP scripts are compiled before they are executed, the parser cannot resolve this type of forward reference, as the following example demonstrates:

```
class Test implements Testable {}
interface Testable {}

$test = new Test;
var_dump($test);
```

This program works fine:

```
object(Test)#1 (0) {
}
```

Now try moving the **new** statement so that the parser will 'see' it before it sees the class definition:

```
$test = new Test;
var_dump($test);

class Test implements Testable {}
interface Testable {}
```

Now the program fails:

```
Fatal error: Class 'Test' not found in test.php on
line 2
```

To avoid this error, you must stick to the "define before use" rule. An easy way to do this is to use autoloading, and put each class and interface into an individual file. PHP will then take care of loading all the required code in the correct order.

unset() and Strings

The function **unset()** can delete existing variables or array elements. To delete an array element the key must be specified, e.g. **unset($a[2])**. The same syntax allows read access to specific characters of a string, but it has never been possible to delete characters from a string using **unset()**. Whereas PHP 4 silently ignored the attempt, PHP 5 bails out with a fatal error:

```
<?php

$string = 'hello world';
unset($string[0]);
var_dump($string);
```

Under PHP 4, this snippet outputs "hello world". Under PHP 5, it fails with a fatal error:

```
Fatal error: Cannot unset string offsets in test.php
on line 4
```

To avoid this, just don't use **unset()** on a string offset—it never worked anyway. You could sanely use the following check to avoid inadvertently doing that:

```
if (!is_string($var)) unset($var[0]);
```

Modulo Division

As I expect we all know, a division by zero is undefined.

The modulo operator (%) calculates the remainder of a division. When modulo dividing by zero, the remainder is undefined as well. Older PHP versions simply returned **FALSE**, meaning that hard-to-find follow-up errors could occur. Newer PHP versions will also return **FALSE**, but will issue a warning too:

```
Warning: Division by zero in test.php on line 3
bool(false)
```

To ensure that you do not confuse a numeric **0** with **FALSE**, you should always use type-safe comparison (**===**) for the result.

Type-Converting Integer Values

PHP is a dynamically typed language. That means that, unlike languages with strong typing, PHP does not need to know the type of a variable when creating it. PHP will interpret the variable as a different data type according to the context.

> **"In some cases, automated conversion is not that easy."**

In some cases, automated conversion is not that easy. While it is obvious that the string **"123"** matches the integer value **123**, the question arises whether **" 123"** and **"123 "** should also be interpreted as **123**.

The various PHP versions are not entirely equal when it comes to making these decisions. PHP 4 and PHP 5.0.0 both give a wrong response, but no warning (this assuming the date is evaluated as being within the UNIX date range). Some versions of the PHP 5.0 series will refuse point blank to convert a string with leading or trailing whitespace to an integer value. Current PHP 5 versions are a little more relaxed about the whitespace, but an **E_NOTICE** will be triggered, although the conversion itself is generally successful. An example is the **date()** function, which formats a date given as a Unix timestamp:

```
var_dump(date('d.m.Y', '   120000000000  '));
```

Since PHP 5.1, this snippet will output:

```
Notice: A non well formed numeric value encountered
in test.php on line 3
string(10) "16.10.1961"
```

Should you see notices like this in your application, consider using **trim()** to remove the surplus

whitespace. Keep it in mind that, up until PHP 5.1, the returned value would have been wildly inaccurate, and casting the input string to integer will both silence the notice and give the same wrong result as in PHP 4.

If PHP completely fails to convert the value to an integer when passing a string to **date()**, the following warning will be issued:

```
Warning: date() expects parameter 2 to be long,
string given in test.php on line 3
```

In that case, **date()**—in line with many other native PHP functions—will return a boolean **FALSE** instead of a string.

Empty Objects

The way PHP treats empty objects has changed between PHP 4 and PHP 5. Although an object without properties was considered empty by PHP 4, all versions of PHP 5 will disagree:

```
class Test {}

$test = new Test;
var_dump(empty($test));
```

In PHP 4 this outputs **TRUE**, while in PHP 5 it will output **FALSE**. This difference can be explained by the fact that objects in PHP 4 were basically arrays rather than true objects.

Should your application happen to rely on the PHP 4 behaviour, you could add an **isEmpty()** method to the object that uses the Reflection API to check whether there are any members:

```
class Test
{
    public function isEmpty()
    {
        $refl = new ReflectionClass($this);
        return sizeof($refl->getProperties()) == 0;
    }
}

$test = new Test;
var_dump($test->isEmpty());
```

This will work under all versions of PHP 5.

__toString()

The magic **__toString()** method is called to convert an object to a string. This allows you to print objects directly using **print $object**, instead of having to call a special **render()** or **print()** method.

Older PHP 5 versions used to output the object identifier when implicitly converting an object to a string. Some applications may rely on this behaviour to write log files.

Until PHP 5.2, **__toString()** was only called when the object was directly output by **print** or **echo**. Since then, **__toString()** is called whenever the object is used as a string. The behaviour is unlikely to change again.

If no **__toString()** implementation is present in the object, PHP will not convert the object to a string any more, but will output a new kind of error: a catchable fatal error that can be handled by a custom error handler. It looks like this:

```
Catchable fatal error: Object of class Test could
not be converted to string in test.php on line 16
```

It is forbidden to throw exceptions in a **__toString()** method, and trying to do so will result in a fatal error:

```
Fatal error: Method Test::__toString() must not
throw an exception in test.php on line 18
```

PHP Extensions

Due to the large number of PHP extensions available, it would be impossible to provide you with a full overview of all the changes and potential migration issues. I will therefore only list a few known issues with commonly used core extensions.

The new **Date** extension comes as standard since PHP 5.2.0, providing both an OO API and a much greater range than UNIX timestamps could offer while also retaining back compatibility with the old PHP 4 implementation of date/time support. You shouldn't see any changes when using it, but you need to be aware that you should set an appropriate timezone yourself in your *php.ini*:

```
date.timezone = Europe/London
```

or in your script:

```
ini_set('date.timezone', 'Europe/London');
```

or (again in your script):

```
date_default_timezone_set('Europe/London');
```

Omitting this step will mean that an **E_STRICT** (another new-to-PHP 5 error level) message is thrown every time you use a date or time related PHP function. If you—or your operating system—set an invalid timezone, you will see an **E_NOTICE**.

Since PHP 5.2.1, the **SPLFileObject** method

getFilename() no longer returns the full path of a file, but only the filename. To retrieve the full name, you will need to use the method **getPathname()**. If you only need the path without the filename, use **getPath()**.

The Tidy extension in PHP 5 uses a newer version of Tidy than that used in PHP 4. When migrating from PHP 4, you must adapt your code to the new Tidy API.

As PHP evolves, new parser tokens are frequently added. Sometimes, existing tokens are also removed. **T_ML_COMMENT**, which represents multi-line comments, was available in PHP 4, but was never actually used. It has been removed in PHP 5.

When PHP 4 was released XML was relatively young, and the libraries that offered support for XML were not very sophisticated. As a result, XML support in PHP 4 was never DOM-compliant. For PHP 5, the XML support has been completely rewritten. The XML extension in PHP 5 is based on a different library than the PHP 4 extension—libxml2 rather than expat—so the API is different. The same holds true for the XSL extension in PHP 5. In fact, *all* the new XML libraries in PHP 5 are libxml2-based.

If your application makes use of the old XML or XSL extensions, you will have to adapt the code to the new. Due to the large number of differences, you will probably have to completely rewrite the part of your application that handles XML. If you only need to read XML data, you can consider using the SimpleXML extension. SimpleXML is rather slow, though, and does not support modifying the XML data. There are newer alternatives to explore in the XMLReader and XMLWriter extensions, which may provide better performance.

Summary

In this article, I have covered PHP code that may need to be modified when migrating from PHP 4 to PHP 5. I still have left out some important aspects of a migration, namely error and reference handling. I will cover these in the next issue, together with an overview of how you can start preparing your code for the migration to PHP 5.3 and, eventually, PHP 6.

If you would like to share your PHP migration experiences with me, drop me a line at *stefan.priebsch@e-novative.de*. I'm eager to hear from you.

STEFAN PRIEBSCH has been solving IT problems for over 25 years. He is founder and CEO of e-novative GmbH, one of the first German IT consultancies offering PHP-based solutions. Stefan holds a degree in Computer Science and is an internationally recognized PHP expert, author, trainer and consultant. You can reach him at *stefan.priebsch@e-novative.de*.

EAV Modeling

by Carl Welch

Although best known for its use in the medical records industry, the Entity-Attribute-Value (EAV) model can be a useful technique for designing any database that needs to store a large and/or frequently-changing set of data fields.

Occasionally a Web application needs to track entities with a large or variable set of attributes. Examples might include qualities describing a person, or details describing an event. A simplistic data design might be to create a table for the entity, with one column for every attribute in the set. This is very effective for an entity with a small and rarely-changing set of attributes, but it doesn't scale very well.

A few examples of where this solution is not very effective would include:

- a very large set of attributes, only some of which will apply to an individual entity
- a large set of attributes that tends to change frequently over time
- attributes that can have very long values

PHP: *5.2.1+*

Other Software: *MySQL 4.1 (could be applied to other RDBMS)*

Useful/Related Links:

- *MySQL Indexes:* http://dev.mysql.com/doc/refman/4.1/en/indexes.html
- *MySQL Partitioning:* http://dev.mysql.com/tech-resources/articles/performance-partitioning.html
- *CSRF:* http://shiflett.org/articles/cross-site-request-forgeries

TO DISCUSS THIS ARTICLE VISIT:
http://c7y-bb.phparchitect.com/viewforum.php?f=10

The third point became significant to me when I was working on a project a few years ago. I was maintaining a Web application whose attributes included user responses to textarea-type questions—in other words, free text that had to be stored as data. Occasionally I would end up with database records that taxed the storage engine's table width limits, because some of the site users had written very long responses.

In this article I will introduce the entity-attribute-value (EAV) data model. It is a much more sophisticated (and complicated) solution than the simplistic model described above. The EAV model is sometimes used in the medical records industry, where it fits perfectly: OBGYN data generally isn't relevant to male patients, and the record for a patient who has never been diagnosed with diabetes won't have much use for data regarding insulin prescriptions.

The EAV model can make it much easier to add, remove and alter a wide range of entity attributes managed by a Web application. Making this kind of change can even be as easy as updating the application's configuration metadata. The consequence of the EAV model is typically a greater number—and greater complexity—of SQL statements.

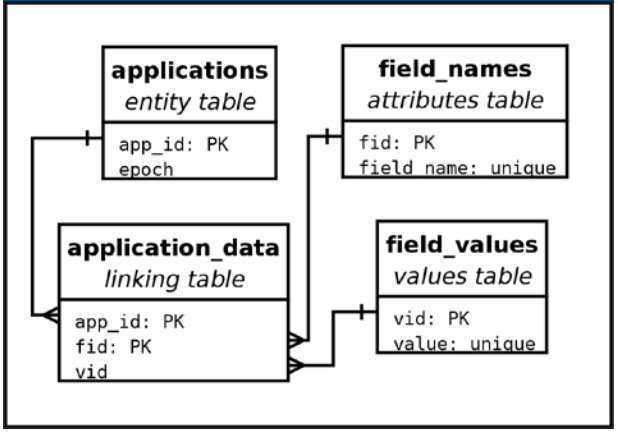
I will develop the idea of the EAV model in this article, using the example of an online scholarship application system to illustrate. I'll provide the SQL statements; I'll try to keep them reasonably generic, but since I use MySQL myself they are bound to show a MySQL bias. This does not, by any means, restrict the model to MySQL. Similarly, although any code samples will naturally be written in PHP, the same concepts could be applied in any programming language. Finally, given that the number of data abstraction layers available rivals the number of application frameworks—not to mention the number of protons in the universe—I'll just stick with the standard PHP *mysql* extension for code samples. I'll also briefly discuss data types, collation issues, and MySQL storage engines.

For the sake of brevity, I will not be checking the return values of the **mysql_*()** functions, but obviously your code should do this. Better yet, you could use an abstraction layer that throws exceptions. And, naturally, if your abstraction layer of choice supports prepared statements, that's a great way to mitigate the risk of SQL injection.

LISTING 1

```
1. CREATE TABLE applications (
2.     app_id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
3.     epoch DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00'
4. );
5.
6. CREATE TABLE field_names (
7.     fid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
8.     field_name VARCHAR(50) NOT NULL DEFAULT '',
9.     UNIQUE KEY (field_name)
10. );
11.
12. CREATE TABLE field_values (
13.     vid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
14.     value VARCHAR(255) NOT NULL DEFAULT '',
15.     UNIQUE KEY (value)
16. );
17.
18. CREATE TABLE application_data (
19.     app_id INT UNSIGNED NOT NULL DEFAULT 0,
20.     fid INT UNSIGNED NOT NULL DEFAULT 0,
21.     vid INT UNSIGNED NOT NULL DEFAULT 0,
22.     PRIMARY KEY (app_id, fid)
23. );
24.
25. INSERT INTO field_names VALUES(1, 'fname');
26. INSERT INTO field_names VALUES(2, 'lname');
27. INSERT INTO field_names VALUES(3, 'justification');
28.
```

FIGURE 1



EAV Basics

In its simplest incarnation, an EAV model would have four tables: one each for the **entities**, **attributes** and **values** of the model's name, and a **linking table** to pull those three together. An example schema appears in the class diagram in Figure 1, and the associated SQL statements in Listing 1. Please take note of the uniqueness constraints in the **field_names** and **field_values** tables. Also, please note that I haven't specified storage engine or collation types. I'll talk about these later in this article.

So, on with our example scholarship application system. Let's start by imagining a page that asks the user for his or her first name, last name, and a short—that's less than 255 bytes—summary of why he or she should be a scholarship recipient.

When someone decides to apply for a scholarship and

fills out our form, the application will do the following:

- create a new row in the **applications** table and store the primary key value of the inserted row in **$app_id**
- insert the applicant's first name into the **field_values** table and store the primary key value of the inserted row in **$vid**
- record the value of **field_names.fid** in **$fid** ('fname' => 1, in this case)
- insert a row into the **application_data** table with an SQL statement resembling: **INSERT INTO application_data VALUES('$app_id', '$fid', '$vid')**
- repeat the previous three steps for the applicant's last name and justification

This implementation is basic, and clearly has scalability issues. I'll try to address those as we go along. It may also appear problematic in another way, at first glance. For example, what happens when the second person named Carl applies for a scholarship? A row containing the value **Carl** will already exist in the **field_values** table...

Well, you could search the **field_values** table for **Carl** and, if it's already stored there, pick up the **field_values.vid** you'll need later on. If **Carl** isn't already listed in **field_values**, you'll need to insert it and then retrieve the primary key of the inserted row. It's not a problem to have two applicants share the same value and primary key in the values table, any more than it is to find two people named Carl in real life. The uniqueness of the entire entry is controlled by the **application_data** table.

Since you would generally stand a good chance of needing to insert the value, and since you would have to ask the database for the primary key anyway, I prefer to insert the row and then search for the inserted row, as you can see in Listing 2.

First we fetch the attribute data from the **field_names** table; I've assumed here that our HTML form fields, passed to the script by a POST request, will share the same names as their equivalents stored in **field_names**. Then we create a new entity record by inserting a row into the **applications** table that contains the current timestamp, and store its id in **$app_id**. Only then can we start packing the values passed to the script from our form (that is, the V part of EAV) into the **field_values** table.

You may recall that we put a uniqueness constraint on **field_values.value**, because we don't want two rows in **field_values** to contain the same value. Thanks to this constraint, attempting to insert a value that is already present will cause MySQL to return an error. For this reason we need to use an **INSERT IGNORE...** query: this downgrades the MySQL error on duplication to a warning, which means that we won't see any output regardless of the outcome. The query will, however, still only insert the value if it's not already there. Following that with a **SELECT** statement will fetch the **vid** associated with that specific value, whether the value was already in the table or not.

Finally, we populate the linking table, **application_data**, with a single record that contains the unique identifiers for the entity, attribute and value records we just inserted (or didn't, as the case may be). As I mentioned earlier, this effectively ties the entity, attribute, and value records together with a unique identifier provided by the **applications** table.

Let's assume that our fictional system will also

LISTING 2

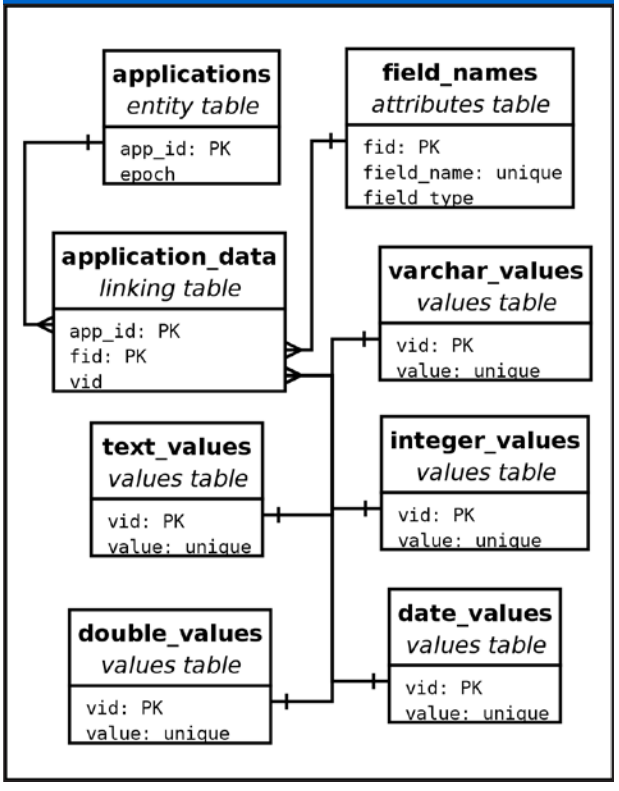
```
<?php

$fid_ref = array();
$vid_ref = array();

$sql = 'SELECT fid, field_name FROM field_names';
$res = mysql_query($sql);

while ($row = mysql_fetch_row($res)) {
    $fid_ref[$row[0]] = $row[1];
}

mysql_free_result($res);

$sql = 'INSERT INTO applications (epoch) VALUES(NOW())';
mysql_query($sql);
$app_id = mysql_insert_id();

foreach ($fid_ref as $fid => $field_name) {
    $e_val = (!empty($_POST[$field_name]) ? mysql_real_escape_string($_POST[$field_name]) : '');
    $sql = "INSERT IGNORE INTO field_values " .
           "SET value='$e_val'";
    mysql_query($sql);

    $sql = "SELECT vid FROM field_values " .
           "WHERE value='$e_val'";
    $res = mysql_query($sql);

    $row = mysql_fetch_row($res);
    $vid_ref[$fid] = $row[0];
    mysql_free_result($res);
}

foreach ($vid_ref as $fid => $vid) {
    $sql = "INSERT INTO application_data " .
           "(app_id, fid, vid) " .
           "VALUES('$app_id', '$fid', '$vid')";
    mysql_query($sql);
}

?>
```

require an administrative interface that allows the scholarship office to view reports of the applications. Imagine a view whereby the scholarship office worker can see a list of recently submitted application hyperlinks and, by clicking a link, send a GET request with the application ID in the query string. Listing 3 illustrates how the administrative interface code would retrieve and display an application submission.

LISTING 3

```
<?php

$app_id = (!empty($_GET['app_id']) ? intval($_GET['app_id']) : 0);
$sql = "SELECT field_name, value " .
       "FROM application_data JOIN field_names " .
       "ON application_data.fid=field_names.fid " .
       "JOIN field_values " .
       "ON application_data.vid=field_values.vid " .
       "WHERE app_id='$app_id' " .
       "ORDER BY application_data.fid";
$res = mysql_query($sql);

while ($row = mysql_fetch_row($res)) {
    print '<p><strong>' . htmlentities($row[0]) .
          ':</strong><br />' .
          nl2br(htmlentities($row[1])) . '</p>';
}

mysql_free_result($res);

?>

```

FIGURE 2



Data Types

What if the scholarship office wants to collect fields that don't really work as **VARCHAR** types? For example, they might be interested in the candidate's grade point average (a **DOUBLE**), or the candidate's date of birth (a **DATE**, **DATETIME** or **INTEGER** [epoch]), or even the size of the candidate's high school graduating class (an INTEGER). Or what if they want to allow more than 255 bytes for responses to the "justification" question, or decide they would like to set further essay-style questions?

Well, you *could* make **field_values.value** a **TEXT** column and treat every piece of input data as a string. That might work, but it would also render the column index just about useless.

A better approach—albeit at the expense of even trickier SQL statements—would be to add a new column, **field_type**, to the **field_names** table. As the name implies, this new column would indicate the data type. There would then need to be a **values** table for each data type supported by your application, in place of the original **field_values** table. So you might have tables named **varchar_values**, **integer_values**, **date_values**, **text_values**, **float_values** and so on. The updated schema for **field_names** and the ***_values** tables appears in Figure 2 and Listing 4.

You may have noticed that I specified an index prefix length of 100 bytes for the uniqueness constraint in

LISTING 4

```
CREATE TABLE field_names (
    fid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    field_name VARCHAR(50) NOT NULL DEFAULT '',
    field_type ENUM('VARCHAR', 'INTEGER', 'DOUBLE',
        'DATE', 'TEXT') NOT NULL DEFAULT 'VARCHAR',
    UNIQUE KEY (field_name)
);
CREATE TABLE varchar_values (
    vid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    value VARCHAR(255) NOT NULL DEFAULT '',
    UNIQUE KEY (value)
);
CREATE TABLE integer_values (
    vid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    value INT(11) NOT NULL DEFAULT 0,
    UNIQUE KEY (value)
);
CREATE TABLE double_values (
    vid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    value DOUBLE NOT NULL DEFAULT 0,
    UNIQUE KEY (value)
);
CREATE TABLE date_values (
    vid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    value DATE NOT NULL DEFAULT '0000-00-00',
    UNIQUE KEY (value)
);
CREATE TABLE text_values (
    vid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    value TEXT NOT NULL DEFAULT '',
    UNIQUE KEY (value(100))
);
```

the **text_values** table (**TEXT** and **BLOB** column indexes both require a prefix length). This approach makes the assumption that no two applicants will type exactly the same 100 bytes at the beginning of their essay responses. An alternative would be to omit the uniqueness constraint in **text_values**; omitting the constraint may be better for performance, anyway, according to the MySQL manual.

The updated PHP code would need to keep track of each field's data type so that it can insert the corresponding value into the appropriate value table. As you can see, I've done precisely this in Listing 5.

The code for the administrative interface will likewise need to be type-aware. When pulling the data fields for an application, the code will need to pull fields from each of the value tables. Although it makes for a long query, a union can work for this, as illustrated in Listing 6.

LISTING 5

```
<?php

$fid_ref = array();
$vid_ref = array();

$sql = 'SELECT fid, field_name, field_type ' .
       'FROM field_names';
$res = mysql_query($sql);

while ($row = mysql_fetch_row($res)) {
    $fid_ref[$row[0]] = array(
        'name' => $row[1],
        'type' => $row[2],
    );
}

mysql_free_result($res);

$sql = 'INSERT INTO applications (epoch) VALUES(NOW())';
mysql_query($sql);
$app_id = mysql_insert_id();

foreach ($fid_ref as $fid => $field_ref) {
    $field_name = $field_ref['name'];
    $field_type = $field_ref['type'];
    $e_val = (!empty($_POST[$field_name]) ? mysql_real_escape_string($_
POST[$field_name]) : '');
    $table_name = $field_type . '_values';

    $sql = "INSERT IGNORE INTO $table_name " .
           "SET value='$e_val'";
    mysql_query($sql);

    $sql = "SELECT vid FROM $table_name " .
           "WHERE value='$e_val'";
    $res = mysql_query($sql);

    $row = mysql_fetch_row($res);
    $vid_ref[$fid] = $row[0];
    mysql_free_result($res);
}

foreach ($vid_ref as $fid => $vid) {
    $sql = "INSERT INTO application_data " .
           "(app_id, fid, vid) " .
           "VALUES('$app_id', '$fid', '$vid')";
    mysql_query($sql);
}

?>
```

Attribute-Related Enhancements

For something like a scholarship application, or anything that asks a large number of questions, you'll probably want to break the form up into several pages. It has to be better than having a single, monstrously long form with a scrollbar two pixels tall.

To achieve this, imagine having other attribute-related items in your attribute table (here, **field_names**). For example, you could have a column named **field_names.page_number**, indicating on which page a particular field should appear. And **field_names.display_order** could be used to determine the order in which the fields should appear on the page, either in the public interface used by the applicants, or in the administrative interface.

The attribute table could also store input validation rules. For example, some of the data fields, such as first and last names, will need to require input, whereas others, such as gender and ethnicity, may well be optional. You'll probably want to impose length limits on some of the fields—no more than 80 characters for an email address, for example—and you may want to enforce regular expression matching for others, such as **/\A\d{2}/\d{2}/\d{4}\z/xms** for the date of birth. Items that could be expressed using a drop-down list,

LISTING 6

```
<?php

$sql = "SELECT field_name, serialized_display_options " .
       "FROM field_names WHERE page_number=1 " .
       "AND active='yes' ORDER BY display_order";
$res = mysql_query($sql);
$form_HTML = '';

while ($row = mysql_fetch_row($res)) {
    $form_HTML .= create_form_field($row[0], $row[1]);
}

mysql_free_result($res);

?>
```

FIGURE 3

Employment History			Dates	
Job Title	Employer	Hours/week	From	To

like marital status, could even have a list of allowed responses.

Another useful column might be **field_names.default_value** (or maybe **field_names.can_be_null**), so that your application will know what to do if a user doesn't care to provide a response to an optional question. And **field_names.active** could be a Boolean flag that you clear when the scholarship office tells you that they no longer want to collect the **mother_maiden_name** field. You wouldn't want to simply delete that row from **field_names**, because you probably already have corresponding rows stored in **application_data**.

In fact, if you set up the attribute table to contain all the information related to displaying and validating your form fields, then adding a new field could be as easy as adding a row to this table. The code for generating the view is in Listing 7, and the code for processing the POST request on that page is in Listing 8.

The attribute table examples offered in this article are fairly unsophisticated. But it can be a good place to store any attribute-related metadata that will be useful to you at runtime.

Hierarchical Fields

When I was doing this for my work, the "scholarship office" threw me an interesting curve that had me stumped for a while. They gave me printouts of how they wanted their application to look, and one of the pages had something like Figure 3.

OK, so at first glance this doesn't look like a big deal. There should clearly be **VARCHAR** fields named **employment_job_title**, **employment_employer**, and **employment_hours_per_week**, and there should be **DATE** fields named **employment_from** and **employment_to**.

The problem is that an applicant might provide zero, one, or more than one employment entries. I'm way too fastidious (anal? dainty?) to do things like **employment_job_title_1**, **employment_job_title_2**, etc. Besides, what if I only go to **employment_job_title_5**, and an applicant has had (and wants to report) six jobs?

The way I chose to approach this was to put another column in the linking table capable of grouping together the five fields for a given employment entry on a given application (see Listing 9).

The scholarship application has a page with a form that contains input boxes for the five data fields (**employment_job_title**, **employment_employer** and so on). When someone submits that form, our scholarship application inserts the data in the same way as in the previous sections, but it now also creates an identifier for this particular employment entry on this particular scholarship application. This identifier is inserted as the **set_id** for those five rows in **application_data**. The application then returns the user to the same form, so that he/she may enter another employment entry at will.

An easy way of doing this would be to use the **epoch** for the **set_id**, with the assumption that a legitimate user wouldn't POST more than one employment record per second. But this introduces a race condition, so a better alternative would be to use some randomly-generated data pushed through a digest function such as MD5 or SHA1. In fact, if you are using anti-CSRF

LISTING 7

```
<?php

$sql = "SELECT field_name, serialized_display_options " .
       "FROM field_names WHERE page_number=1 " .
       "AND active='yes' ORDER BY display_order";
$res = mysql_query($sql);
$form_HTML = '';

while ($row = mysql_fetch_row($res)) {
    $form_HTML .= create_form_field($row[0], $row[1]);
}

mysql_free_result($res);

?>

```

LISTING 8

```
<?php

$sql = "SELECT field_name, serialized_validation_rules " .
       "FROM field_names WHERE page_number=1 " .
       "AND active='yes'";
$res = mysql_query($sql);
$form_is_valid = 1;

while ($row = mysql_fetch_row($res)) {
    $field_name = $row[0];
    $field_is_valid = validate_field($_POST[$field_name],
                                     unserialize($row[1]));
    if (!$field_is_valid) {
        $form_is_valid = 0;
        break;
    }
}

mysql_free_result($res);

?>

```

LISTING 9

```
CREATE TABLE application_data (
    app_id INT UNSIGNED NOT NULL DEFAULT 0,
    fid INT UNSIGNED NOT NULL DEFAULT 0,
    set_id VARCHAR(40) NOT NULL DEFAULT '',
    vid INT UNSIGNED NOT NULL DEFAULT 0,
    PRIMARY KEY (app_id, fid, set_id)
);

```

tokens on your forms (and if you aren't, you should be), you could just use that for the **set_id**.

For fields that aren't part of a set, such as **first_name**, you could simply have an empty string in **application_data.set_id**.

Later, when the administrative interface is retrieving the data for scholarship application #327, in which the applicant provided two employment entries, it'll find ten rows with **app_id=327** and the **application_data.fid** values corresponding to the employment-related fields. Five of those rows will share one value of **set_id** and the other five will have a different value, thus allowing the admin interface to group the fields into the distinct employment entries.

The following sections will describe a few other issues I have encountered in creating EAV systems.

"What if I only go to employment_job_title_5, and an applicant wants to report six jobs?"

Linking Table Inserts/Updates

Depending on the nature of your application, your users may need to update fields for which they have already provided data. In the case of our scholarship application, too, we should allow the applicant to go back and revise the data he/she has provided on a previous page. Another common example would involve allowing your users to update their email address or background color preference.

If so, you may want to consider using the **REPLACE INTO...** syntax for inserting and updating rows in your linking table (**application_data**). The advantage of using **REPLACE INTO...** is that it works for adding new rows into the linking table *and* for updating existing rows, which also means that the syntax will work if you don't know whether you're inserting or updating.

Imagine that you added **cell_phone_number** as a data field two months ago. With **REPLACE INTO...**, you can use the same SQL statement for a new user who needs to insert data for that field, and for a returning user who wants to update the information he or she entered some time last year.

Storage Engines

If you recall, I intentionally omitted specifying the engine type in the **CREATE TABLE...** statements. Some of the database interaction lends itself to database transactions, like creating a new application record. This interaction creates an entry in the entity table, may add several entries in one or more of the value tables, and will add many entries to the linking table. If you're comfortable using a transaction-enabled storage engine such as InnoDB, you could wrap this interaction in a transaction for atomic changes. If you're interested in trying this, your schema statements might look something like those in Listing 10. (Note that I've only shown the **varchar_values** table there; you'd probably want to include all the value tables.)

The foreign key constraints in **application_data** are such that you'll be prevented from deleting a row in the values table if that row references a row in the linking table. A similar arrangement is in place for the attribute table as well, but I've done something a bit different for the entity table. With the **ON DELETE CASCADE** clause, deleting a row in the entity table (**applications**) will automatically delete all the corresponding rows

LISTING 10

```
1. CREATE TABLE applications (
2.     app_id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
3.     epoch DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00'
4. ) ENGINE=InnoDB;
5.
6. CREATE TABLE field_names (
7.     fid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
8.     field_name VARCHAR(50) NOT NULL DEFAULT '',
9.     field_type ENUM('varchar', 'integer', 'double', 'date', 'text')
10.        NOT NULL DEFAULT 'varchar',
11.    UNIQUE KEY (field_name)
12. ) ENGINE=InnoDB;
13.
14. CREATE TABLE varchar_values (
15.    vid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
16.    value VARCHAR(255) NOT NULL DEFAULT '',
17.    UNIQUE KEY (value)
18. ) ENGINE=InnoDB;
19.
20. CREATE TABLE application_data (
21.    app_id INT UNSIGNED NOT NULL DEFAULT 0,
22.    fid INT UNSIGNED NOT NULL DEFAULT 0,
23.    set_id varchar(40) not null default '',
24.    vid INT UNSIGNED NOT NULL DEFAULT 0,
25.    PRIMARY KEY (app_id, fid, set_id),
26.    INDEX app_data_fid_idx (fid),
27.    INDEX app_data_vid_idx (vid),
28.    FOREIGN KEY (app_id) REFERENCES applications (app_id) ON DELETE
    CASCADE,
29.    FOREIGN KEY (fid) REFERENCES field_names (fid) ON DELETE RESTRICT,
30.    FOREIGN KEY (vid) REFERENCES varchar_values (vid) ON DELETE RESTRICT
31. ) ENGINE=InnoDB;
32.
```

in the linking table. In other words, a single **DELETE** command will totally eliminate all traces of a scholarship application, which can be a useful (if somewhat dangerous) short cut.

On the other hand, if the nature of your application is such that you need MySQL full-text searches, you may need to stick with MyISAM—at least for the **varchar_values** and/or **text_values** tables.

> "Depending on the nature of your application, it may be helpful to track changes in a value."

Collation

Another important consideration is collation in your **varchar_values** table. The default character collation in MySQL is case insensitive. Imagine two users whose last names are McMurtry and Mcmurtry (note the difference in capitalization), and pretend that Alice Mcmurtry submits before Bob McMurtry does. With the default collation, **Mcmurtry** will be in **varchar_values** after Alice submits, and Bob's attempt to insert **McMurtry** won't insert a new row because MySQL will consider **Mcmurtry** and **McMurtry** to be the same string. This may be very frustrating to Bob, because he'll see a last name of **Mcmurtry** in the dropdown list but won't be able to change it to **McMurtry**.

By setting the collation of the **varchar_values.value** column to a case sensitive collation, MySQL will consider **Mcmurtry** and **McMurtry** to be distinct strings:

```
CREATE TABLE varchar_values (
    vid INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    value VARCHAR(255) CHARACTER SET latin1
    COLLATE latin1_general_cs NOT NULL DEFAULT '',
    UNIQUE KEY (value)
);
```

Tracking Value History

Depending on the nature of your application, it may be helpful to track changes in a value over time. For example, if your application allows its users to update their phone numbers or email addresses, it might be helpful to keep a record of those changes. You can accomplish this sort of thing by adding a table similar to that in Listing 11.

The **changed_to_id** column is a foreign key to **vid** in one of the value tables. **fid**, a foreign key to the attribute table **field_names**, indicates which value table by way of **field_names.field_type**. So whenever a row in the linking table (**application_data**) is added or updated, you can add a row to **value_history**, and this will create a record of how those values are changed over time.

Notice that I'm using **changed_at** to distinguish between different changes of the same field for a given entity. You could set that using MySQL's **NOW()** function, or with something like **date('Y-m-d H:i:s')** in PHP. Again, though, this creates a possible race condition and may not be appropriate to your particular circumstances.

Trailing Whitespace

Trailing whitespace is another possible gotcha for **VARCHAR** data, something I discovered only very recently when working on another EAV system. This other system, which is not a Web application and which includes the history tracking described above, imports data from a set of text files into a MySQL database with an EAV architecture. These text files are periodically updated by an external process. As the values change, the importer needs to update the data—and create history records.

The interesting thing here is that when you try inserting data into a **VARCHAR** column, MySQL removes any trailing whitespace before inserting it. So if one of the data items to be imported is a character string such as 'hello ' (with trailing whitespace), MySQL will blithely insert **hello** into the table. At the next import runtime, the importer will compare the data in the current input text file with the persistent data stored in the database. Although the data in the text file is

LISTING 11

```
1. CREATE TABLE value_history (
2.     app_id INTEGER NOT NULL DEFAULT 0,
3.     fid INTEGER NOT NULL DEFAULT 0,
4.     changed_to_id INTEGER NOT NULL DEFAULT 0,
5.     changed_at DATETIME NOT NULL
6.                         DEFAULT '0000-00-00 00:00:00',
7.     PRIMARY KEY (app_id, fid, changed_at)
8. );
9.
```

'hello ' on both occasions, the importer will detect a difference because there is trailing whitespace in the current value and none in the value pulled from the database. Fields will therefore be updated unnecessarily, and you end up with a bunch of erroneous historical data. In fact, it will appear that that particular value was changed from **hello** to **hello** at every import runtime.

In such cases, it becomes important that the importer strips the trailing whitespace before comparing it with the persistent data.

Xtreme EAV

Here are a couple of fun ideas for the more ambitious among you.

If you end up running several Web applications with EAV databases, you could try sharing the value tables. The idea would be to have a single database containing the ***_values** tables. Your EAV Web application databases would have their own attribute, entity and linking tables, but no value tables of their own.

For those of you who found that idea entirely too practical, maybe this will appeal to you more...

Some search engines have the useful—or irritating, depending on your point of view—feature of suggesting search items as you type into the search engine input field. If you're into AJAX and want to do something like this, here's a fabulous way to waste lots of bandwidth and CPU cycles. Let's say that you wanted the user to see suggestions as she or he is typing in the **first name** field (you know, just in case he or she is having trouble remembering how to spell it), and you wanted the suggestions to be drawn from the pool of values that other users have entered in the **first name** field.

We'll pretend that you want to suggest values that match the first three characters the user types into the input field, and that you want to display the 10 most popular matching values. If the **field_names.fid** value for **first name** is 1, and the first three letters typed are **car**, then the SQL statement triggered by the AJAX call might be something like this:

```
SELECT COUNT(*) c, value COLLATE latin1_general_ci v
    FROM application_data t1 JOIN varchar_values t2
    ON t1.vid=t2.vid WHERE fid=1
    GROUP BY t1.vid HAVING v LIKE 'car%'
    ORDER BY c desc LIMIT 10;
```

Recall that we stipulated case sensitive collation on **varchar_values.value**, so this query must explicitly tell the engine to use a case *insensitive* collation. Otherwise, **car** would match **carlotta** but not **Carl**.

Again, this kind of thing is a great way to turn your database server into an abacus.

Horizontal Partitioning

One final point I'd like to cover concerns something I haven't actually tried yet myself. If your application is wildly popular, some of your tables—particularly the value and linking tables—may attain alarmingly large row counts. If so, you may wish to explore the horizontal partitioning feature in MySQL. There's a useful article about partitioning on the MySQL developer resource pages (see the links box at the head of this article).

Horizontal partitioning, in theory at least, would for example allow you to split up your **varchar_values** table into two tables: one containing values that begin with lower-case letters, and one containing capitalized values. This particular example would use the **RANGE** partitioning mode, but there are several other modes (and methods) available.

Conclusion

In this article we've looked at the entity-attribute-value (EAV) model of database design. I wrapped the discussion in an example of an online scholarship application Web application, but I hope I've convinced you that this technique is potentially useful in a broad variety of projects. At the expense of some substantially more complicated SQL statements, EAV architecture can offer a system with greater scalability and normalization.

Carl Welch is a Web developer and Linux system administrator. He bears unhealthy addictions to science fiction, RSS feeds, version control and other silliness—evidence of which is documented at http://mbrisby.blogspot.com.

A Refactoring Diary: The Story Continues

by Bart McLeod

This is the final part of my diary recording the attempt to refactor my old legacy CMS, which I would like to replace with a rock solid, framework-based Internet application. I started adapting my CMS to use the Zend Framework last month, and will now conclude the journal. Enjoy!

If you missed last month's php|architect, you should know that I changed my approach quite radically when I started to work with the Zend Framework. One thing I learned from the previous two refactoring projects was that reading about the framework before you start work is invaluable in terms of time saved. Another was that it's a good idea to get each small part working properly before taking the next step.

I struggled a little with the Zend autoloader, until I had the idea to add **_autoload()** to my bootstrap file and call **Zend_Loader::loadClass()** from inside the function. I also had some problems with **mod_rewrite** (again). But the part I'm still working on, 16 'days' into the project, is adapting my legacy CMS to the MVC model.

As before, this is purely a spare-time project. Each 'day' in my diary therefore represents at most a few hours, and sometimes much less.

PHP: *5.2.5*

O/S: *Any supported by PHP and MySQL*

Other Software: *HTTP server, e.g. IIS or Apache*

Useful/Related Links:
- *http://framework.zend.com*

TO DISCUSS THIS ARTICLE VISIT:
http://c7y-bb.phparchitect.com/viewforum.php?f=10

Day 17: The Model

I consulted Cal Evans' ZF book about the Model part of the MVC implementation. It comes down to this: complete freedom. *None*, *Light* or *Heavy* Model. Cal prefers a heavy model, which not only creates, updates and deletes data but also has some knowledge of how to behave. It will for example have ideas about what valid data is. In my case, the heavy model would know that if a given subject already exists, it may not be created again. Instead of letting the error occur and confronting the user with it, my new CMS should use AJAX or a simple **JsHttpRequest** to ask the model something like **subjectExists($myNewSubject)**. This would be part of the clientside validation process, so no invalid form submission would take place to frustrate the user.

I am faced with a few choices now. I could:

- Re-use my auto-query functionality and pass the queries to **Zend_Db**
- Build a heavy model for every table
- Generate a light model for every table and add heavy stuff only where necessary

Option 1 does not allow the intelligent options that I would like to use. Option 2 is going to mean a lot of work, which is not why I chose to adopt a framework. That leaves me with Option 3, which still means some work because it involves the writing of a code generator.

And then, where should I place the model, and how should I talk to it inside the controller?

The classes that belong to the model go into the *models* directory, so I create a sub-directory *cms*. The model for the **subject** table goes in *application/default/models/cms/CmsSubject.php*. (I'll simplify this with a constant, **MODELS_BASE**, to represent the full physical path.) The controller will then include the model:

```
require_once MODELS_BASE . 'cms/CmsSubject.php';
```

Reading Cal's book, I found I'd been working too hard for database insertion. I was building a traditional query and calling **$db->query($query)**. It worked fine, but ZF is simpler than that. You can build an associative array, say **$data**, with the column names as keys, and do:

```
$affected_rows = $db->insert($tablename, $data);
```

Notice that **$data** shares many key-value pairs with the array returned by **$this->getRequest()->getParams()** in my controller, after posting my form. This similarity gave me the extraordinary idea of populating the model class almost automatically. The only difference is that the array provided by the request object contains a few extra key-value pairs. These must be filtered out of the array before it is used in the call to the database.

Filtering is achieved using a whitelist approach. The model has to know which fields are in the table, so it will extract those fields from the request parameters.

Should I add the array of valid fields or columns manually, or add it automatically by querying the database for it? I choose 'automatically', because I'm creating a lot of models. If you only have a few models, or if performance is a big issue, you should type the column list yourself.

Having come this far, it is a logical step to have a **Base** class for the models. In their most primitive form, subclasses will have to override only one property of their parent: the table name. See Listing 1 for an implementation of the **CmsDataObject** base class. Note how extremely simple the code has become using this approach. (Of course, you must realize that this type of model does not offer any security at all.)

The form validation itself is now responsible for offering valid data to the model. It is alright to do this, as long as you guard the server-side form validation process. The **read()** and **delete()** methods are slightly less straightforward then the others, and were not implemented yet. For these two, the model needs to know, or to be told, what the primary keys are.

One way to achieve this would be to pass the primary keys in. Another way would be to determine them in the constructor, while reading all the fields. For read operations, it is clearly easier to just pass them in, since you will need to provide the values anyway. For deletion, this should not be necessary. Deleting a data objects' record is typically an ambiguous operation: what are you going to do with the data object afterwards? It will still exist, but it will also be invalid, since its counterpart in the database is deleted. Perhaps you could ask some kind of cleaning service to garbage collect it. Or does **unset($this)** work inside a class? Maybe deletion should not be made part of the data object itself. You could use a **TableManager** class instead. A **TableManager** could also be made responsible for answering questions like **subjectExists()**... fine. My **read()** method is now implemented, see Listing 2.

If you want validation inside your model, it can

LISTING 1

```
<?php

    class InvalidDataException extends Exception {}

    class CmsDataObject {
        protected $columns = array();
        protected $data = array();
        protected $db;
        protected $tablename = 'subject';
        protected $primaryKeys = array();
        protected $primaryKeyValues = array();

        function __construct($code = null) {

            if (!is_null($code)) {
                $this->read($code);
            }

            $this->db =  Globals::getDb('mrvane');
            $query = "SHOW columns FROM $this->tablename";
            $columns = $this->db->fetchAll($query);

            foreach ($columns as $row) {
                $this->columns[] = $row['Field'];
                if ($row['Key'] == 'PRI') {
                    //add to primary keys
                    $this->primaryKeys[] = $row['Field'];
                }
            }

            $this->initializeDefaults();
        }

        protected function initializeDefaults() {
            //no implementation in the base class
        }

        protected function hasValidData() {
            return true;
        }

        public function populate($array) {

            foreach ($array as $k => $v) {

                if (in_array($k, $this->columns)) {
                        $this->data[$k] = $v;
                }

                if (in_array($k, $this->primaryKeys)) {
                      $this->primaryKeyValues[$k] = $v;
                }
            }
        }

        public function __get($name) {

            if (in_array($name, $this->columns)) {
                return $this->data[$name];
            }

            return null;
        }

        public function update() {

            if (!$this->hasValidData()) {
                throw new InvalidDataException("Invalid data in " .
__CLASS__ . "::" . __FUNCTION__ . "()");
            }

            $keys = array();

            foreach ($this->primaryKeys as $key) {
                    $keys[$key] = $this->data[$key];
            }

            return $this->db->update($this->tablename,
                                     $this->data,
                                     $this->buildCriteria($keys));
        }

        public function delete() {
            return false;
        }

        public function create() {
```

LISTING 1: Continued...

```

            if (!$this->hasValidData()) {
                throw new InvalidDataException("Invalid data in " .
__CLASS__ . "::" . __FUNCTION__ . "()");
            }

            return $this->db->insert($this->tablename,
                                     $this->data);
        }

        public function read($keys = null) {
            return false;
        }

        protected function buildCriteria($keys) {
            $criteria = '';
            $delimiter = '';

            foreach ($keys as $k => $v) {
                $criteria .= $delimiter;
                $criteria .= " $k = '$v' ";
                $delimiter = " AND ";
            }

            return $criteria;
        }

        public static function dateToMySql($dateString) {
            $date = new Zend_Date(date_parse($dateString));
            return $date->get('YYYY-MM-dd');
        }

        public static function dateFromMySql($dateString) {
            $date = new Zend_Date(date_parse($dateString));
            return $date->get('dd-MM-YYYY');
        }

        public function getCount($criteria = "") {
            $query = "SELECT count(*) FROM $this->tablename $criteria";
            return $this->db->fetchOne($query);
        }

        public function getData() {
            $public_data = array();

            foreach ($this->data as $k => $v) {
                $public_data[$k] = $this->__get($k);
            }

            return $public_data;
        }

        public function listRecords($offset = 0, $number = 10, $where) {
           $safe_offset = (int) $offset;
           $safe_number = (int) $number;

           if ($safe_number === 0) {
               $safe_number = 10;
           }

           $query = "SELECT * FROM $this->tablename $where LIMIT $safe_
offset, $safe_number";
            $results = $this->db->fetchAll($query);
            return $results;
        }

        public function exists() {

            if ($this->read($this->primaryKeyValues)) {
                return true;
            }

            return false;
        }
    }

```

easily be accomplished by introducing a **hasValidData()** function. This will be checked before an **INSERT** or **UPDATE** operation, and an **InvalidDataException** can be thrown on failure. Of course, for every subclass of the base data object, you will then have to override **hasValidData()** since it is specific to each table. In the base class, **hasValidData()** will return **TRUE**. Apart from that, it has no implementation.

I now have a basic model in place. Creating this model was easy, straightforward and fast, thanks to the simplicity of ZF. The next step is to make the form act as a database administration page capable of listing the existing subjects and paging them.

course makes no sense. Does it try to set the primary key? Am I using the function the wrong way? Yes, I am. I need to specify a third argument when calling Zend_Db's **update()** method—the **WHERE** clause. Again, it would be nice if the data object had knowledge of its primary keys. I will pass them in once more, and encapsulate the building of the **WHERE** clause inside a function. This works—no bug for ZF after all! I'm not sure if my **WHERE** clause builder is right for multiple keys, but I'll see trouble with the **translations** table later if it isn't.

After this exciting adventure, it's time for more fun. I add a button that brings up a new empty form and add

"The good news is that Zend_Date can handle many date formats and convert them to timestamps internally."

First, I need to refactor the controller so that the building of the form is in a separate function. This is going be the private function **buildForm($data = null)**. If **$data** is an associative array, the values will be displayed in the form fields that correspond to the keys. This way, I will be able to reuse the form for display across different actions.

So far, saving a record created a new one. How about updating? The model will need to know if it already has a counterpart in the database or not. This can be accomplished by setting a flag when reading it from the database. I try using the base data object property **in_db** for this, and the new **createOrUpdate()** method to take care of the decision.

Unfortunately this idea does not work, due to the stateless nature of HTTP. The flag will be gone upon re-posting the form. So, perhaps the form itself should store the flag. This can be done by evaluating the **$data** object passed into the **formBuilder()** function. If it is **NULL**, no flag needs to be placed. I will check for the flag, **exists**, in the **save()** action of the **subjectsController**.

I found a ZF bug! The **update()** method is now called, but complains about a duplicate primary key... which of

default values for the dates, because I'm tired of typing valid date values during testing. Bringing up a new empty form can be done by changing the action, but also simply by using a regular button that changes the **document.location**. This is done in three minutes. Now for the default date values. ZF comes with classes for dates, so I'll look at those first.

Day 18: Dates

Once the default time zone is set in *php.ini*, it's safe to use **Zend_Date**. The default dates are created in my controller, but it's also possible to create a custom class for date elements:

```
$date = new Zend_Date;
$begin = $data->begin ? $data->begin : $date-
>get('dd-MM-YYYY'); //ISO format
$end = $data->end ? $data->end : '31-12-9999';
```

$begin and **$end** are passed into their respective form elements as the values. The form is now fully functional, except there is no way to delete a **subject** yet.

One of the things I hoped to gain from moving to a framework was nice date selectors, but **Zend_Form** doesn't provide them. Even if it did, I need them to

handle Dutch date formatting.

The good news is that **Zend_Date** can handle many date formats and convert them to timestamps internally. So I should be able to use one field and a date selector, display the Dutch date format (or any other) and convert it to MySQL date format on save, or back on retrieval.

The next question is where to put the functionality to convert the dates? Should I create a **convertDate()** method in the base data object, so that it can be used anywhere? Or use a utility class and call it whenever needed? Or does **Zend_Date** itself offer such a utility? Let me check that first:

```
$date = new Zend_Date('10.03.2007 00:00:00', Zend_
Date::ISO_8601, 'nl');
```

Feed it the date and specify the locale, and **Zend_Date** will parse the date string correctly. You then extract the date using format specifiers before offering it to MySQL. It's also possible to pass the constructor a date array as the first argument. This can be obtained by calling a built-in PHP function, **date_parse()**, on an arbitrary date string. So I end up with two methods:

```
public static function dateToMySql($dateString) {
    $date = new Zend_Date(date_parse($dateString));
    return $date->get('YYYY-MM-dd');
}

public static function dateFromMySql($dateString) {
    $date = new Zend_Date(date_parse($dateString));
    return $date->get('dd-MM-YYYY');
}
```

And they do their job! As a side effect, though, my default date values no longer work, and I get the most curious dates from my custom getter when the data object is empty. This is overcome by adding the condition **if (isset($this->data['name']))** to the getter.

Now that the date magic works nicely, it's time to find a selector. I could just steal the date selector from symfony. Except that, on my new laptop, I no longer have symfony installed...

Putting it to work will have to wait till next time.

Day 19: The Date Picker, And More

I started the day by moving the setting of default values from the **buildForm()** method in the controller to the **CmsSubject** class in the model. This not only allows cleaner code in **buildForm()**, but also offers a cleaner separation between model and controller.

Setting up the date picker from symfony was a cinch. The only bottleneck was the paths to the script and stylesheets (paths, again!). Relative paths no longer seem to work, so I had to prefix each path with a slash. The fun part, though, is that you can set the date picker in the template. There's no need to create a specific date input class, it's so easy to set up. There are even localized versions available for download, so I now have a Dutch date picker!

It now becomes apparent that I need to re-use common parts of the templates, since it makes no sense to include the date picker scripts and styles every time. They should be in a common header, and the registration of the **begin** and **end** fields should be in a common footer or a decorator for the form. A simple **include** would do, but what does the manual have to say about it?

```
$this->headScript()->appendFile("/scripts/jquery.
js");
$this->headScript()->appendFile("/scripts/
ui.datepicker.js");
$this->headScript()->appendFile("/scripts/
ui.datepicker-nl.js");
echo $this->headScript();
```

Although this outputs the desired script tags in my html head, it does not solve the include problem. I'll just use a regular **include** for the head. Or maybe I can make a different action use the same template? They do not really differ much anyway! I managed to render the *index.phtml* view script for the **save()** action. But I had to get rather rough with it to accomplish this:

```
echo $this->view->render('subjects/index.phtml');
die();//rude, I think
```

There should be a more graceful way... If I leave out the **die()**, two view scripts are rendered: the one with the name of the action AND *index.phtml*, in that order. So there is a stack of view scripts, but how can I influence the order in which they are rendered? If I can determine the order, it means a solution to my include problem. I can make a *header.php* script and call:

```
echo $this->view->render('header.php');
```

in every action that needs it. But I'll give up on this for now, and use regular **include_once** statements to include *header.php* and *datescript.php*. I had to configure the path to the *views* directory in order to easily include my files in two views. It still doesn't feel really good this way, but it works. Next time I will try form validation, but I still would like to reduce the number of views.

Retrospective comment: I stand corrected here. Instead of having **$this->view->render()** in the

controller, I should have written **$this->render('form')** in order to render *form.phtml* for all actions that use the form. Section 7.7 of the programmers reference guide documents this. The point is you should not call **render()** on the controller's view object, but on the controller itself. Doing this means you no longer need a fully qualified path to your view script. Calling **$this->render('form')** inside your controller action will override the default view to render, so you can simply re-use *subjects/form.phtml* for your **new**, **save**, **edit** and **delete** actions.

Day 20: Form Validation

Form validation is relatively simple. If you build your form elements like I did, you have no doubt passed in an **options** array on creation of the form elements. This array is the place to put your validators:

```
'validators' => array('alpha','myvalidator');
```

so that you may add as many validators as you like for each element. If you create elements explicitly with the **new** keyword, thus yielding a new element instance, you will probably have an **addValidator()** method at your disposal.

To know if your form is valid, you will need to rebuild it after submission. That smells like .NET. On your rebuilt form (which does NOT need to be filled with the submitted values) you should call **isValid($data_array)** to know whether the data is valid. If you use the default decorators you will also see messages around any form elements that have errors, but in my case, I do not see these. My form validates as expected and even gives me nice—albeit English—error messages when I call **$form->getMessages()**.

I also found that, once the form is validated you can use **$form->getValues()**, so there's no need to call upon the request object to get the values. This call will also guarantee filtered values, which is good if you want to hand them over to your model. However, none of this works until you actually have called **isValid()** on the form!

Although it's not overly important, this means I will have to find a way to first build the form and then populate it with the values of the model or the request after validating it. This is just to show the user the actual input values following an attempt to save a record with the form. Or would the population of the form elements be done automagically, after calling **isValid()**? That would be really nice. No, unfortunately, this kind of magic goes beyond ZF.

This simply means that what I do at present is not correct. First, I populate the form with values taken directly from the request. These can break the form if the values contain HTML tags, so I should at least call the **escape()** function on them I guess. Second, after I validate the form, I repopulate the model with the filtered values from the form. Could this be the cause of the decorators not working? After validation, I set the **form** property of the view again from scratch, without a subsequent call to **isValid()**. It's hardly likely that the decorators with errors would show up then, is it? Bingo—I removed the second call to **buildForm()** now, assigned the already-built form to the **form** property of the underlying view, and the error message shows up!

The next issue is that I want to populate the form, not only with the values from the request, but also with filtered values. This means I will have to populate the form with the model after validating the form, so I might need a different mechanism if the form is invalid... I understand if you can't follow me here. Experiment with it yourself, and you'll get the picture.

Now I found a problem with validators. If I try to use a regex validator I get an error, even though I followed the instructions in the manual. Perhaps I mis-typed something? Ah, I found it: the regex itself should be in an array. Setting up a complex element is better done the object oriented way. At least in my opinion:

```
$s = new Zend_Form_Element_Text('sequence_number');
$s->setLabel('volgorde');
$s->setRequired(true);
$s->setValue($data->sequence_number);
$s->addValidator('regex',false,array('/^\d{1,5}$/'));
$form->addElement($s);
```

All the regex validator checks is that **$s** consists of 1 to 5 digits. Unfortunately, the message my users will see when their input does not meet this requirement is hardly human readable: **'111111' does not match against pattern '/^\d{1,5}$/'**. It will be clear to the user that what she typed did not live up to my expectations, but what is she expected to make of the message? I'd better write my own validators for fields like these, which will allow me to provide my own error messages.

On my way there, I found this interesting function : **Zend_Form::persistData()**. This might be a solution to my earlier problems with persisting data on the form. And it works! There is no need to use **POST** or other request data 'manually' before validation—all you have to do is call **persistData()** on the form!

Zend_Form also has a **populate()** method that will

probably do the opposite: populate the form with values retrieved from the database, perhaps after a **save** action, to reflect the data that actually made it into the database. I imagine that **populate()** should take an associative array of key-value pairs where the keys match the form element names. The **data** property of the data object class would be the perfect candidate for form population; I'll just need to add a **getData()** method to the base data object. Yahoo! This works, first time right.

There is a subtle difference now in the way I determine if a form contains data from an existing record. I now add the hidden **exists** element *after* populating the form following an update or creation of a new record.

The more I get to know ZF, the simpler the code becomes.

The only drawback of populating the form using the **$data** array is that my nice Dutch dates (from my custom getters) have turned back into English dates, since they were stored in the English format internally. Let me see if I can change the internal storage, or just modify the values when handing out the **data** property. This seems to work, but I don't trust it; the dots in the date are not replaced with MySQL dashes... Ah, there was an error in my **getData()** method! Here is the corrected version:

```
public function getData(){
    $public_data = array();
    foreach ($this->data as $k=>$v) {
        $public_data[$k] = $this->__get($k);
    }
    return $public_data;
}
```

Note that since the magic getter does not work internally, as in **$this->property**, I have to call it explicitly as **$this->__get('property')**.

Another thing that calls for adjustment is the default values on the dates, which were also provided by the data object. I can set them, using a dummy data object. Defaults are defaults anyway.

On to the custom validators. A quick inspection of the ZF code tells me that most or all validators extend **Zend_Validate_Abstract**. I also found I can use simpler validators for the sequence number. Instead of using a regex to limit the input to 5 digits, I can add two validators: one to check that the input is all digits (the 'digits' validator) and one to checks that the input value is less than 100000 (the 'lessThan' validator). As a bonus, this also gives me error messages that non geeks can understand. However, I wonder that the framework contributors didn't think about translating these simple error messages. Why would they have omitted such an important step? Perhaps there is some kind of entry point where I can translate them; perhaps this was considered preferable to subclassing validators with the single purpose of translating the messages.

The simplest approach would of course be to instantiate a validator explicitly and set a custom (translated) message. This works! Simplicity wins again. All I need to do now is set up labeling and the means of translating the labels. Does ZF provide some kind of labeling toolset? Not that I don't have one, but while I'm at it, I might as well ask ZF first (I can feel this is going to become a habit, asking ZF first).

Zend_Label does not exist, but **Zend_Translate** does. I will not go into it very deeply here, but I will tell you which solution I chose and why.

I chose to use **.csv* files for translations, because they are fast and easy to edit—even by my customers. It's also very easy to convert my existing translation files to the CSV format. The code, then, looks like this:

```
$translate = new Zend_Translate('csv', 'path/to/
mytranslation.csv', 'de');
$translate->addTranslation('path/to/other.csv',
'fr');
```

LISTING 2

```
<?php

class CmsDataObject {

    // ...

    public function read($keys = null) {

        if (is_null($keys)) {
            /* Populate the data object before calling
               this!There is no need to provide the keys
               or even to know which fields they are. */
            $keys = $this->primaryKeyValues;
        }

        if (is_array($keys)) {
            $criteria = ' WHERE ';
            $criteria .= $this->buildCriteria($keys);
        } else {
            throw new Exception("Invalid keys in " . __CLASS__ . "::" .
__FUNCTION__ . "()");
        }

        $query = "SELECT * FROM $this->tablename $criteria LIMIT 1";
        $result = $this->db->fetchRow($query);

        if (is_array($result)) {
            $this->data = $result;
            return true;
        }

        return false;
    }

} // class CmsDataObject

?>
```

```
echo $translate->_('hello');
/* Zend_Translate will automatically detect and use
the users language */
```

The *.csv* file will contain something like **hello;bonjour**—and that's really all there is to it! **Zend_Form** also has a **setTranslator()** method. If it's my lucky day, this means it will translate all labels and possibly all error messages automatically, if translations are present.

Although not the most inspiring of tasks, I have to add translations now or risk this becoming a burden later. I will build only a Dutch language file; the English labels will be in the code itself. I tried to set a translator on the form, but it wants a **Zend_Translate_Adapter**, and I don't know what that does or how to get one that works. Would the book help me here? No, it doesn't. Back to the manual then.

Strangely enough, setting the translation adapter on the form is different from adding a translation file as a **Zend_Translate** instance to the global scope:

```
$form->setTranslator(new Zend_Translate_Adapter_Csv(
                        ROOT.'/languages/nl/cms.
csv','nl'));
```

Now let me put some translations into the file and some English labels into the form. This works instantly. The only word that does not translate is the value on the submit button, which is now **save** (the Dutch version would be **opslaan**). So the form translator only translates labels; the submit button must be translated manually. I wonder if error messages will be translated... yes they are! To find the messages you'll need to translate, just trigger all the errors you can think of and put them into your translation files. You'll probably want messages in the default language to be translated too, because this gives you a clean way to customize them without setting them explicitly on the validator objects. It might be nice to use a factory to hand out translators for the form, allowing the ability to switch between languages. The configured or detected locale could then be used automatically by the factory to determine which translation to hand out.

Unfortunately, while testing my error messages I find that my alphabetic validator doesn't do as I intended. It allows characters like **ê**, while my intention was to allow only **[a-z]**. I will need a regex validator here so that I can use URLs without having to worry about encoding. This also means I will have to set a custom error message, because the key **regexNotMatch** will apply to several other validators and so can't be used for a specific translation. What I find absolutely stunning is that, since using a translator on the form, I no longer get error messages: instead I see keys for the error messages, which will ease both translation and customization.

"To find the messages you'll need to translate, just trigger all the errors you can think of and put them into your translation files."

In this particular case however, I will have to translate the custom message myself. Oops, and even that does not work! It's a minor detail, but still something for the ZF team to look into I think:

```
$validator_abc->setMessage('uh oh','regexNotMatch');
```

If used in conjunction with a translator on the form, this snippet displays the key **regexNotMatch** rather than a custom message. As soon as I remove the translator, the custom message shows up. So if you use a translator, you are *only* able to set custom messages by translating the keys. This is good thinking, in itself, but a regex validator can deal with so many different patterns that you may end up translating only **regexNotMatch**. In that case, there is no way of telling users what kind of input is required for different regexes. For now, I will just translate it as 'invalid input' and let the user guess.

Another issue is that **required** no longer works if I

LISTING 3

```
<?php

class CmsDataObject {

    // ...

    public function delete() {

        if (!$this->hasValidData()) {
            throw new InvalidDataException("Invalid data in " . __CLASS__
. "::" . __FUNCTION__ . "()");
        }

        $keys = array();

        foreach ($this->primaryKeys as $key) {
            $keys[$key] = $this->data[$key];
        }

        return $this->db->delete($this->tablename,
                                 $this->buildCriteria($keys));
        }

} // class CmsDataObject

?>
```

LISTING 4

```
<?php

$form->addElement(
        'submit',
        'delete',
        array('value' => $form->getTranslator()->_('delete'),
              'class' => 'cms_button',
              'onclick' => "document.getElementById('cms_form').action='/
subjects/delete';return true;",
              'disabled'=>'disabled')
              );

$attribs = $form->getElement('delete')->getAttribs();
var_dump($attribs);

$element = array_pop($attribs);
var_dump($element);
var_dump($attribs);

$form->getElement('delete')->setAttribs($attribs);
$attribs = $form->getElement('delete')->getAttribs();
var_dump($attribs);

/* output:
array(3) {
  ["class"]=>
  string(10) "cms_button"
  ["onclick"]=>
  string(74) "document.getElementById('cms_form').action='/subjects/
delete';return true;"
  ["disabled"]=>
  string(8) "disabled"
}
string(8) "disabled"
array(2) {
  ["class"]=>
  string(10) "cms_button"
  ["onclick"]=>
  string(74) "document.getElementById('cms_form').action='/subjects/
delete';return true;"
}
array(3) {
  ["class"]=>
  string(10) "cms_button"
  ["onclick"]=>
  string(74) "document.getElementById('cms_form').action='/subjects/
delete';return true;"
  ["disabled"]=>
  string(8) "disabled"
}
*/

?>
```

set it the OO way. or if I pass it in as an option... Of course, I could change the regex to refuse empty input. The funny thing is, **required** works for the **sequence_number** element. Beats me, altogether.

For better or for worse, everything is translated now. What I have omitted is configuration or detection of the locale. I will leave that to you, because I have some other interesting things to do before this pilot project is complete: deleting and listing subjects.

Retrospective comment: In the final ZF 1.5 release, custom messages are also shown if the form has a translator. The downside is that you have to translate them manually. If you had this issue yourself, you should remove the translation for **regexNotMatch** from your language file, otherwise it will overrule the custom message you set! It would allow for simpler code if a custom message could be translated by the form translator.

Day 21: Deleting Records

I have decided to give the base **CmsDataObject** knowledge of its primary keys. This allows for an easy to use **delete()** method, although it will leave the user with an invalid data object if he does not destroy it. It will also make the **update()** method simpler, since the primary keys don't need to be passed in. You will find the **delete()** method in Listing 3.

Minor bug found: **setAttribs($array)** doesn't unset the attributes. What I am trying to accomplish is to have a disabled delete button on the form, enabled only when an existing record is loaded into the form. If you run the code in Listing 4, you will see that the button remains disabled. As you can see in the listed output, the **$attribs** array no longer has a **disabled** attribute. But after setting those attributes on the element, the element still has it. I will try to fix this.

From inspection of the code, I learned that **setAttrib('disabled', null)** will do the trick. However, for convenience, I would like to propose a **removeAttrib()** method for **Zend_Form_Element**, in line with most of the other classes in **Zend_Form**.

Last Friday I applied to become a Zend Framework contributor, because I wanted to fix some of the issues I encountered. This may take a while, because you need to sign and send in a CLA before you can contribute anything, or even report bugs in the bug tracker. It should take about five days to process my application.

In other news, I am no longer working with the pre-pre-release version. ZF 1.5RC1 is out!

Day 22: Using Modules

I have now started another ZF-based project. This one is not experimental—it has to go live within two weeks. Along the way, reading through the manual, I found a solution to the modular approach. If you read all through my ZF diary, you will remember that I tried in vain to have my CMS run as an admin module, rather than as a frontend application. The key to the solution is in the bootstrap file, where you may specify a whole array of controller directories rather than just one. The frontend is simply the default array key. The path to the **admin** module can be loaded in the **FrontController** by calling:

```
$controller->setControllerDirectory(
                array('admin'=>$path_to_admin,
                      'default'=>$path_to_de-
fault));
```

Obviously, this way you will be able to have as many modules as you wish.

One important thing to keep in mind is that you have to prefix the class names with the module name plus an underscore, e.g. **Admin_IndexController**. However, you should **not** prefix the names of the files in which these classes are located. So **Admin_IndexController** is defined in a file named *IndexController.php*, and located in *APP_ROOT.'/admin/controllers*. Do not forget this, write it down, and put it under your pillow.

Time's Up!

Today I became a contributor to ZF: I received approval by e-mail. It took them ten days, but there you go. Now, what were those issues again?

In Retrospect

Picking up this article after a few weeks I feel I have so much to tell you, but there's simply no room for it all. It is clear by now that I will be using Zend Framework for my future projects wherever it seems to fit. At this point I'm not only refactoring the CMS backend, but also the frontend of an existing website. Why?

- My code is organized into small units that are easier to maintain
- The code is cleaner and shorter, so it's easier to maintain or update
- My code is becoming more reliable
- My programs are now so extensible that it is even making me happy
- I feel I get a lot of support from the community even before I ask questions, simply because others are asking the same questions and the answers are dripping from the forums right into my mailbox
- When I go to code elements myself, I keep discovering ZF already has something for it. Example: **Zend_Registry**. I implemented my super ninja **Globals** class as a registry—no need. Just setting the database connection in the config file and putting that in the registry will do!
- I save so much time coding that there is time left to focus on the users and their tasks. The CMS is improving so much, it promises to be fun to use
- The ZF community is so active, that all the issues I found while writing this article were resolved before I found the time to report them

The list could probably be longer, but I think you get the picture. I found my framework. Now it's your turn to find yours!

BART MCLEOD is a self-employed Web developer and the owner of Space Web, a small company in The Netherlands that builds websites for other small companies (http://spaceweb.nl). Bart can be—and usually is—hired to solve Web and database related programming issues for other companies, focusing on PHP and .NET development. He is also a painter (http://bartmcleod.nl) and a father. You may contact him at mcleod@spaceweb.nl.

Scripting Integration

by Matt Zandstra

Testing as you go can be crucial to a project's success, especially when you really don't think you have enough time to do it—and besides, actual development is so much more interesting... If this sounds like you, automated builds and tests are the way to go. In this month's Test Pattern, I'll introduce you to the joy of scripting the mundane.

Since my last column, I have had to get agile with one of those projects. You know the kind—an unmovable deadline which seems unfeasibly close, fluid requirements, little time for QA. With a vague air of mania hanging over your team, it's tempting to dive in and focus exclusively on churning out code.

With everything having to work more or less on deadline day though, it pays to pay some attention to automated build and testing. You don't want to press the big red button on launch day and not be entirely sure what will happen. Far better for everyone on your team to have their own big red button, and to get them hitting it daily.

You might think that a bunch of unit tests and a PEAR package file are all you need as far as test and installation is concerned, and if you're only producing a library, you're probably right. If, on the other hand, your scope is wider (a Web application, for example) then you probably need more than this. So this month, and in my next column, I'm going to look at some aspects of building the red button. In particular, this month I will focus on automated build: how to make it super easy for a developer to take an untouched development space from a single bootstrap script to a test-ready integration environment. We will build tools to support a fictitious project called **maxithing**.

TO DISCUSS THIS ARTICLE VISIT:
http://c7y-bb.phparchitect.com/viewforum.php?f=10

Scripting the Mundane

We have found that the key to good integration is to remove impedance from our developers—to reduce the annoyances which stand in the way of setting up a development environment, running component

tests, installing and packaging. If you make any of this difficult for your developers, there are two potential outcomes. Either your team will spend valuable coding time on repetitive tasks, to the detriment of morale and productivity, or they will skip some of the tasks to get onto the important business of being brilliant and creative. In the latter case, this can mean nightmares come integration time. Your codebase will be insufficiently tested, and it will be bedded into a number of development environments.

This process of 'bedding in' is insidious, and it happens all the time. A developer will generally download tools and packages whilst he or she is working, and it is easy to forget to add any dependencies that this creates to a build file. It's only when you check out the code somewhere else that the everything begins to fall apart due to dependency failures.

Our solution to these problems is to script the bajeezus out of them. If we find ourselves performing a process with multiple steps more than once, we'll add the steps to a script of some kind. Housekeeping of this sort is often not glamorous, but it can be surprisingly satisfying. Perhaps this is because you get the whole project lifecycle in miniature when creating project management code. You move from identifying need, to scoping solution, to shipping, to seeing your code in operation, to fixes and improvements—sometimes in a matter of hours.

Another nice thing about such scripts is that you can lower the bar somewhat. These are not scripts for end users, they are designed for your team. Since you know, and may even constrain your team's choice of development environment (at least for the purposes of integration) you won't have to worry about every combination of platform and version. You will have the luxury of making assumptions about a developer's environment.

From Bridgehead to Occupation

First things first. You should make it easy for a user to check out a working environment and install it somewhere far away from their inevitably messily customized development space. Let's assume that your team have agreed to use Fedora for development. Ideally we want to be able to give the user a small package or script and say 'run that'. The work involved on her part should be as minimal as possible.

The work environment into which your code installs should default to some kind of base level. What constitutes a base will vary, of course. You may, for example, define your starting point as a particular Fedora distribution with PHP 5.2.5, CVS and MySQL 4.1 already installed. If not, then you will need to include the installation of those base level items in your script. The same is true of configuration—will the environment already have a **$CVSROOT** environment variable populated in a useful way?

It's up to you whether the base environment comes pre-configured with all this stuff, or whether you script it. What you don't want is for your developer to have to install core packages or set up configuration. Remember, we're making things easy.

So, how can developers find access to a clean environment? Virtualization tools such as VMware and Parallels make this pretty easy these days. You can set up a virtual Linux installation and snapshot it in an optimal state.

In Listing 1 you'll find a simple little script, *qbuild-maxi*, that our user might copy into just such a virgin environment. It's a shell script, because at this stage we may not even have access to PHP. The script installs some core tools, and then checks out a project from CVS. Note that the script supports branches. This is crucial, because a serious project is likely to consist

LISTING 1

```
1. sudo yum install -y php
2. sudo yum install -y php-pear
3. sudo yum install -y cvs
4.
5. usage () {
6.     echo "usage: $progname <name> <branch>"
7. }
8. mysys() {
9.     echo "+ $*"
10.    $* || exit 1
11. }
12.
13. progname=`basename $0`
14.
15. if [$# -lt 1]; then
16.     usage
17.     exit 0
18. elif [$# -gt 1]; then
19.     BRANCHSTR=" -r "$2
20. fi
21.
22. NAME=$1
23. sudo pear upgrade pear
24. sudo pear channel-discover pear.phing.info
25. sudo pear install phing/phing
26. sudo pear install PEAR_PackageFileManager
27.
28. mysys cvs checkout $BRANCHSTR $BRANCHSTR -d $NAME projects/maxithing
29.
```

of at least one release branch, with development continuing in HEAD. It may also include any number of temporary development branches. All these should be installed and tested on a regular basis.

Remember, one size won't fit all—this script is customized to a particular Linux distribution, and uses platform-specific tools such as **yum**. Our audience is a select group of developers whose integration environment is a particular snapshot of a Fedora distribution. Of course, as time wears on we'll want to extend our support across a number of platforms so that we can perform our tests in different contexts.

qbuildmaxi may be a little script, but it's a also a big blunt instrument. It might be as well to encourage developers to use it only in a clean virtual environment.

Here's how a developer might run the script:

```
sudo ./qbuildmaxi blah maxithing-release-1_0_0-
branch
```

This should check out the release branch for our project and install its core requirements.

So far, the script is enough to win us a freshly checked out codebase, but not enough to actually install anything or run tests. Before tacking build though, let's take a quick look at what we checked out from CVS. This is the structure of the **maxithing** project.

Organizing Project Files

My team commonly divides our projects' top level directories something like this:

```
pkg/
test/
code/
docs/
archive/
```

In the root directory you will of course create overview documents like *ChangeLog*, *README*, and *RELEASE_NOTES*.

pkg/ is the project control room. It's where we put the highest level scripts. Here you might find scripts for generating documentation, building PEAR packages, installing development applications or setting up execution and development contexts.

Our setup script, here *qbuildmaxi*, might live within *pkg/*, perhaps in *pkg/tools/qbuildmaxi*. It's important that all tools should be checked into CVS and made available to developers. We might also make a centrally hosted package of *qbuildmaxi*, to make it easy to install independently of an existing CVS checkout.

test/ is where we place end-to-end tests. These tests assess the application as an integrated whole. While unit tests assess classes in place, end-to-end tests may assess an installed instance of the application. In order to support this, the test environment may populate a database with sample data and set up any files a system might use or generate during operation.

code/ (also often named *src/*) is where a developer spends most of his or her time. It is where you will find a project's source code. You may also find unit tests, mocks and configuration files here.

docs/ and *archive/* are broadly self-explanatory. *docs/* may contain both user and developer documentation, and *archive/* can store anything old: previous packages, legacy tools and so on.

Now that we know what what our setup script checks out, it's time to do some actual installation.

Phing and PEAR

The simplest build option is to put a *package.xml* file in *src/* and to run the command **pear package** to generate a tarball.

Although we do want to work with PEAR, it may not provide enough functionality for our developers without help. In addition to a simple install, they will likely want to automate other tasks. These tasks might include running integration and unit tests, populating databases with test data, regenerating the PEAR package to include new files, updating the *ChangeLog* file, and any amount of housekeeping.

Such tasks can be handled with a batch of shell scripts, of course, but we are lucky to have a flexible and extensible PHP tool available to us. Phing (Phing Is Not Gnu make) is a port of the excellent Java build tool, Ant. Phing deserves an article, or even a small book to itself, so this can only be a short introduction.

Phing uses an XML file, usually named *build.xml*. This consists of *target* tags, which are invoked like commands by Phing. A particular task can be specified on the command line when calling Phing. Tasks also relate to one another in dependency relationships. So you might have an **install** task which depends upon, and therefore implicitly invokes, a **build** task. Phing also works with *property* tags, which manage data, and *task* tags, which do stuff—like copying files around.

Once we have a project environment in place, Phing can become our project scripting base of operations. Let's kick off by defining a root element (**project**) and setting up a couple of properties:

```
<project name="maxithing"
         default="build"
         basedir=".">
  <property name="VERSION"
            value="1.0.1" />
  <property name="BUILDLOC"
            value="../build/${phing.project.name}-
${VERSION}" />
  <property name="BUILDBALL"
            value="${phing.project.name}-${VERSION}.
tgz" />
```

Of course, over time we'd be adding to these properties. For now, I'll kick off with some basics: a version, a location for building (relative to the *pkg/* directory where we'll save *build.xml*) and a tarball name. Note the way we can refer to existing properties in tag attributes using **${this}** syntax. Also note that Phing provides built-in properties such as **${phing.project.name}**.

Let's look at a target now. Here is **build**, the **default** target as defined in the root **project** tag. It simply sets up two dependencies:

```
<target name="build"
        depends="copyfiles, buildpkg" />
```

copyfiles does just as it says on the tin. It copies files from *src/* to a directory named *build/*. **buildpkg** uses these files to generate a PEAR package file. Here are the two targets:

```
<target name="copyfiles">
  <copy todir="${BUILDLOC}" >
    <fileset dir="../code" />
  </copy>
</target>

<target name="buildpkg">
  <echo>building package</echo>
</target>
```

I lied, of course. **buildpkg** skates over a forthcoming discussion by deploying a simple task: **echo** to output a message. **copyfiles**, though, is the real deal. It uses the **copy** task to move all files and directories from the *src/* directory to the destination I set up in the **BUILDLOC** property. The **fileset** tag here is an example of a *type*. Types are elements that encapsulate special kinds of project data; in this case, a bunch of files and directories. Let's run that code:

```
$ phing build
  Buildfile: /home/mattz/maxibranch/pkg/build.xml
  maxithing > copyfiles:
  maxithing > buildpkg:
  [echo] building package
  maxithing > build:
  BUILD FINISHED

  Total time: 0.4199 seconds
```

Copying files over to a build directory is a good start. We can operate on these files, performing substitutions and other adjustments. We could then use Phing to install the files directly—copying them into target directories on the system. Our end users are unlikely to have Phing, however, and asking people to download a separate build tool just to install our code is a serious barrier to take up. PEAR, on the other hand is often bundled with PHP, and is the standard for easy end-user installation.

We could construct a PEAR build file (*package.xml*) by following the instructions on the PEAR website (*http://pear.php.net/manual/en/guide.developers.package2.php*), and then leave it loose in *src/*. Keeping it up to date with files that have been added and removed from the various packages will be a pain though. Luckily there is a PEAR package named **PEAR_PackageFileManager** and, even more luckily, there is a Phing task named **pearpkg2** that wraps it. Take a look at the code in Listing 2. It may look heavy, but actually most of those elements have direct parallels to elements required for *package.xml*. The **fileset** part at the end is where the magic happens. It causes all files and directories in the build directory to be added to the package file. Now, if we run our build again, Phing tells us that it has created the package:

```
maxithing > buildpkg:
[pearpkg2] Creating [default] package.xml file in
base directory.
```

Now that we can create a PEAR package ,it's time to add a task for installing it. Having run **phing build**, the developer (or the *qbuildmaxi* script) has enough in the *build/* directory to run the **pear install** command. I can easily do this manually:

```
$ cd ../build/maxithing-1.0.1/
$ sudo pear install package.xml
Password:
install ok: channel://pear.php.net/maxithing-1.0.1
```

But how can I achieve this from within the *build.xml* file? There is one imperfect possibility here. The **PhpEval** task allows you to embed PHP code within your build file. We might do something like this:

```
<target name="install2"
        depends="build">
  <php expression="chdir('${BUILDLOC}');
                   system('pear install -f package.
xml');" />
</target>
```

But that's truly horrible, even given my caveat about relaxed standards. We're probably going to want to use

PEAR functionality in other situations (tarball generation springs to mind). Why not build a quick and dirty wrapper of our own, that provides access to PEAR?

Phing's support for custom tasks is what makes it so ultimately useful. If you had to rely only on built-in tasks we'd soon all abandon it for a bunch of arbitrary shell scripts or (shudder) **make**.

Like all good PHP plugins, a custom task must extend a base class. This enforces good structure on your part:

```
require_once "phing/Task.php";

class BasicPear extends Task {

    public function init() {
    }

    public function main() {
    }
}
```

"Phing's support for custom tasks is what makes it so ultimately useful."

Phing calls the **init()** method first, then calls methods corresponding to arguments in the task tag (more on that coming up). Finally it calls **main()**, in which you should perform the real business of your task.

So what's all this about arguments? Well, if you want to accept an argument named **path** in your element, you should create a method that looks like this:

```
public function getPath($value) {
    // do something with $value
}
```

This method is called if the user includes the relevant argument. The **$value** argument of course contains the user-provided input.

So we want to create a task called **basicpear** (so called because it was cobbled together in half an hour and might melt your system if used in the real world). It will accept the following attributes: **cmd**, **arguments**, **options** and **chdir**. Here's the class signature:

```
class BasicPear extends Task {

    private $cmdstr = null;
    private $args = array();
    private $chdir = null;
    private $opts = array();
```

The **getCmd()** method expects a PEAR command (duh):

```
    public function setCmd($str) {
        $this->cmdstr = $str;
    }
```

getArguments() takes a space-delimited list of arguments and saves them as an array:

```
    public function setArguments($str) {
        $this->args = preg_split("/\s+/", $str);
    }
```

getOptions() expects long format options, again space-separated. It, too, breaks its input down into an array:

```
    public function setOptions($str) {
        $options = preg_split("/\s+/", $str);
        foreach ($options as $o) {
            if (preg_match("/^(.*?)=(.*)/", $o, $m))
{
                $this->opts[$m[1]] = $m[2];
            } else {
                $this->opts[$o] = true;
            }
        }
    }
```

setChdir() simply stores its input:

```
    public function setChdir($str) {
        $this->chdir = $str;
    }
```

LISTING 2

```
<target name="buildpkg">
  <taskdef name="basicpear" classname="BasicPear" />
  <delete file="${BUILDLOC}/package.xml" />
  <pearpkg2 name="maxithing" dir="${BUILDLOC}">
    <option name="channel" value="pear.php.net"/>
    <option name="summary" value="my summary"/>
    <option name="description" value="my description"/>
    <option name="apiversion" value="1.0.0"/>
    <option name="apistability" value="stable"/>
    <option name="releaseversion" value="${VERSION}"/>
    <option name="releasestability" value="beta"/>
    <option name="license" value="Apache"/>
    <option name="phpdep" value="5.1.0"/>
    <option name="pearinstallerdep" value="1.4.6"/>
    <option name="packagetype" value="php"/>
    <option name="notes" value="notes"/>
    <mapping name="maintainers">
      <element>
        <element key="handle" value="mattz"/>
        <element key="name" value="matt zandstra"/>
        <element key="email" value="matt@blah.com"/>
        <element key="role" value="lead"/>
      </element>
    </mapping>
    <fileset dir="${BUILDLOC}">
      <include name="**"/>
    </fileset>
  </pearpkg2>
</target>
```

Finally I can work with my collected data in the **main()** method contained in Listing 3. You can see that I store the current working directory for later use, and **chdir()** if the user has set a value for the **$chdir** property. Next I simply pass the command, options and arguments data to various PEAR methods. Note that if things go wrong, I make use of Phing's **BuildException** class. Finally I **chdir()** back to my starting location.

Assuming I save this class in a file named *BasicPear.php* in *pkg/* (next to *build.xml*), how do I get Phing to know about it? Well let's look at the new task in use:

```
<target name="install"
        depends="build">
  <taskdef name="basicpear"
           classname="BasicPear" />
  <basicpear cmd="install"
             options="force"
             arguments="package.xml"
             chdir="${BUILDLOC}" />
</target>
```

LISTING 3

```
<?php

class BasicPear extends Task {

    // ...

    public function main() {
        ob_start();
        $here = getCwd();
        PEAR_Command::setFrontendType('CLI');

        if (!is_null($this->chdir)) {
            chdir($this->chdir);
        }

        $config = PEAR_Config::singleton($pear_user_config,
                                         $pear_system_config);
        $cmd = PEAR_Command::factory($this->cmdstr,
                                     $config);

        if (PEAR::isError($cmd)) {
            $msg = "could not find command '{$this->cmdstr}'";
            throw new BuildException($msg);
        }

        $opts = array();
        $result = $cmd->run($this->cmdstr,
                            $this->opts,
                            $this->args);

        if ($result === false) {
            $msg = "command error '{$this->cmdstr}'";
            throw new BuildException($msg);
        }

        if (PEAR::isError($result)) {
            throw new BuildException($result->getMessage());
        }

        $this->log(ob_get_contents());
        chdir($here);
        ob_end_clean();
    }

} // class BasicPear

?>
```

That's much neater! As you can see, the **taskdef** tag associates the name 'basicpear' with the class **BasicPear**. The way I use the **basicpear** tag itself here should be reasonably self explanatory. It is equivalent to:

```
cd path/to/buildlocation;
pear install package.xml;
cd -;
```

This is what our Phing install now looks like on the command line:

```
$ sudo phing install
  Buildfile: /home/mattz/maxibranch/pkg/build.xml
  maxithing > copyfiles:
  maxithing > buildpkg:
  [delete] Deleting: /home/mattz/maxibranch/build/
maxithing-1.0.1/package.xml
  [pearpkg2] Creating [default] package.xml file in
base directory.
  Analyzing spag/domain/blah.php
  maxithing > build:
  maxithing > install:
  [basicpear] install ok: channel://pear.php.net/
maxithing-1.0.1
  BUILD FINISHED

  Total time: 0.6482 seconds
```

Here, as an added bonus is a target that builds a package file using **basicpear**:

```
<target name="generatePackage" depends="build">
  <taskdef name="basicpear" classname="BasicPear" />
  <basicpear cmd="package" chdir="${BUILDLOC}" />
  <move todir="${BUILDLOC}/packages"
        file="${BUILDLOC}/${BUILDBALL}" />
</target>
```

Just by adding a couple of lines to *qbuildmaxi*, we can now automate both installation and package tarball generation.

Now a developer working on integration can enter a minimal setup environment, and from a small script set up a coding environment checked out either from a branch or HEAD. He or she can also install and make packages using Phing. Next time round, I'll build on this basis to look at some tricks and issues associated with testing.

MATT ZANDSTRA is a developer and writer. He works for Yahoo! in California. Matt is the author of a number of books and articles about PHP including PHP: Objects, Patterns and Practice published by Apress. He writes about code occasionally at http://www.getinstance.com.


Beyond Safe Mode

by Stuart Herbert

We've all heard the rumours about the end of safe_mode, and now it's getting scarily close to becoming a reality. I'd like to use this column to talk about some of the alternative approaches to shared server security.

If you currently rely on PHP's **safe_mode** to secure your Web servers, it's time to start looking for an alternative solution. When PHP 6 is released, it will not include support for **safe_mode**. The PHP core development team have decided—rightly, in my humble opinion—that **safe_mode** doesn't provide the security that is really required. The problem is that **safe_mode** is in the wrong place, architecturally speaking, to solve the problem of securing a shared server.

It simply isn't possible for PHP to provide a 100% cast-iron guarantee that the PHP code of website A cannot look at the MySQL username and password that is part of website B's code. This is because the PHP code for both website A and website B runs as the same user (normally **apache**, **www** or **nobody**). As far as the underlying operating system is concerned, websites A and B are owned by the same user, and therefore they must trust one other, as demonstrated in Figure 1. It is this built-in expectation of trust that **safe_mode** tries to overcome, with limited success. To make a server secure, instead of fighting the operating system, we need to harness it.

PHP: *Any*

O/S: *Linux*

Other Software: *Apache 2*

Useful/Related Links:

- *mpm-itk homepage:* http://mpm-itk.sesse.net/
- *mpm-peruser homepage:* http://www.telana.com/peruser.php
- *suEXEC homepage:* http://httpd.apache.org/docs/2.0/suexec.html
- *suPHP homepage:* http://suphp.org
- *The Web Platform:* http://blog.stuartherbert.com/php/series-the-web-platform/

TO DISCUSS THIS ARTICLE VISIT:
http://c7y-bb.phparchitect.com/viewforum.php?f=10

If website A and website B are owned by different users, and if their PHP code executes as different users, then the underlying operating system can handle the security for us as illustrated in Figure 2. This approach will always be more secure than **safe_mode** ever could be, because the operating system is the right place to be enforcing this sort of security.

But it isn't PHP's job to implement this approach, which is why we must look beyond **safe_mode** for a solution.

PHP's job is to run and execute your code. It's Apache that controls the connection to the user's Web browser, and Apache that decides whether PHP should run or not. To get the right balance between high performance and security, Apache is therefore the correct place to handle the job of switching to the right user before PHP is executed.

I'm going to look at three different ways to change Apache to ensure that your PHP scripts are run as different users. One way involves taking advantage of Apache's existing features, and the other two involve enhancing Apache's capabilities by adding third-party code.

Solution 1: Apache's suexec

Our first option is to have Apache use the built-in suEXEC support to run PHP as a CGI process. Only processes running as the user 'root' can become another user, so Apache uses a setuid binary named **suexec** to temporarily become **root** and then switch to the desired user. This gives us the scenario in Figure 3.

Why use this approach? Well, for a start **suexec** already comes with Apache, which is important if you're not comfortable compiling Apache from source. Chances are that PHP/CGI is also provided as a binary package for your Linux distribution. You can be confident that the Apache developers have made **suexec** very secure, and that they will quickly fix any security problems found in it.

But that's about it on the good-news front. Running PHP via **suexec** is slow slow slow. **suexec** is fiddly to configure, and provides unhelpful log messages if you get it wrong. You also can't use HTTP auth support in your PHP code—that only works under **mod_php**, which can be a problem for some installations of the popular phpMyAdmin application.

FIGURE 1

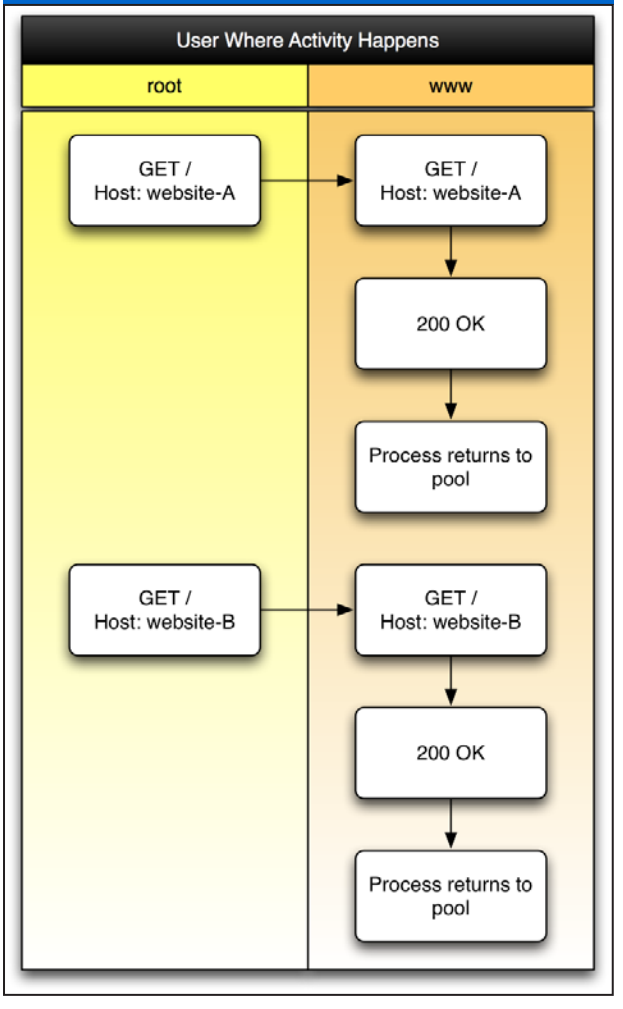
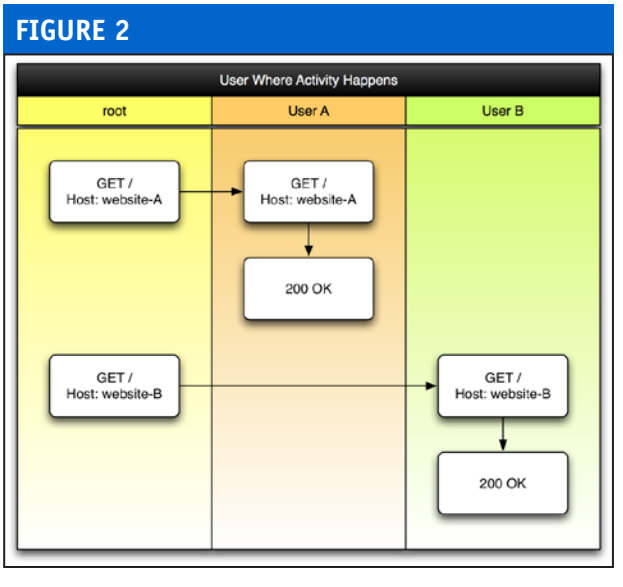


FIGURE 2



Solution 2: mod_suphp

In the quest for a faster solution, **mod_suphp** is our second option. Created by Sebastian Marsching, it provides a much-easier-to-configure alternative to **suexec**. As with **suexec**, you end up running PHP as a CGI process. Each CGI process runs as the user who owns the website, this time through a setuid binary named **suphp**. The model is fairly similar; see Figure 4.

There's no denying that **mod_suphp** is a very popular solution. It has been around for years, and the project provides a helpful mailing list if you run into problems and need help. There are **mod_suphp** packages available for several popular Linux distributions too, which makes it a good choice if you're not comfortable compiling and installing Apache from source code.

Again, the downsides are performance, and the limitations that come from running PHP as a CGI program. The good news is that **mod_suphp** is faster at running PHP scripts than **suexec**, but it's still substantially slower than using **mod_php** with no additional security.

Solution 3: Alternative Apache MPMs

The two solutions we've looked at so far are both mechanisms that Apache's existing processing engines (named MPMs, which is an acronym for multi-processing modules) call. Apache itself remains unchanged; we're just adding extra bits of work for Apache to do. These extra bits of work involve creating new processes, which is computationally expensive. As a result, we get the security that we seek, at the cost of performance and capacity. This can be a bit of a problem in a commercial environment; the business model for shared hosting servers is normally built around cramming as many websites as possible onto a single machine, to keep costs as low as possible.

What if, instead of adding extra work for Apache to do, we could change the work that Apache does?

This is the approach taken by two alternative Apache MPMs: **mpm-peruser** and **mpm-itk**. Both MPMs are based on **mpm-prefork**, the default MPM for Apache 2, which mimics the behaviour of Apache 1.3. This makes it safe to use them with **mod_php**, without having to worry about multi-threading issues. Both of these MPMs fully implement our desired security behaviour from Figure 2.

The main difference between **mpm-peruser** and **mpm-itk** is that the former maintains a per-user pool of Apache processes, whilst **mpm-itk** does not. One of the main performance tricks that Apache uses is to create

"What if, instead of adding extra work for Apache to do, we could change the work that Apache does?"

FIGURE 3

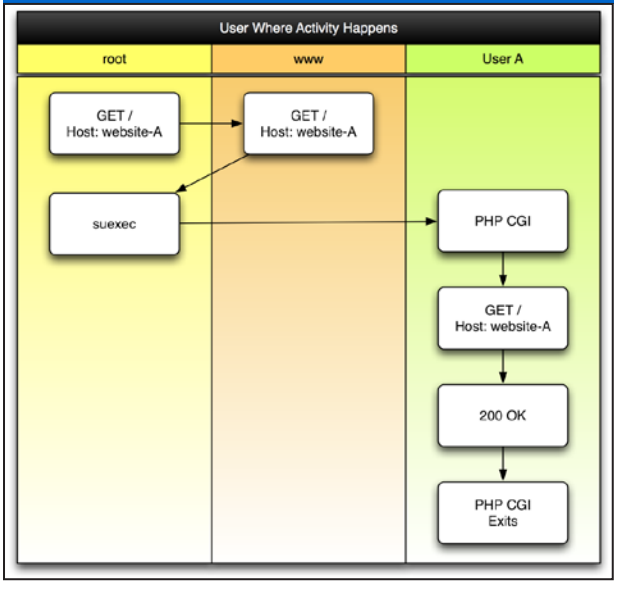
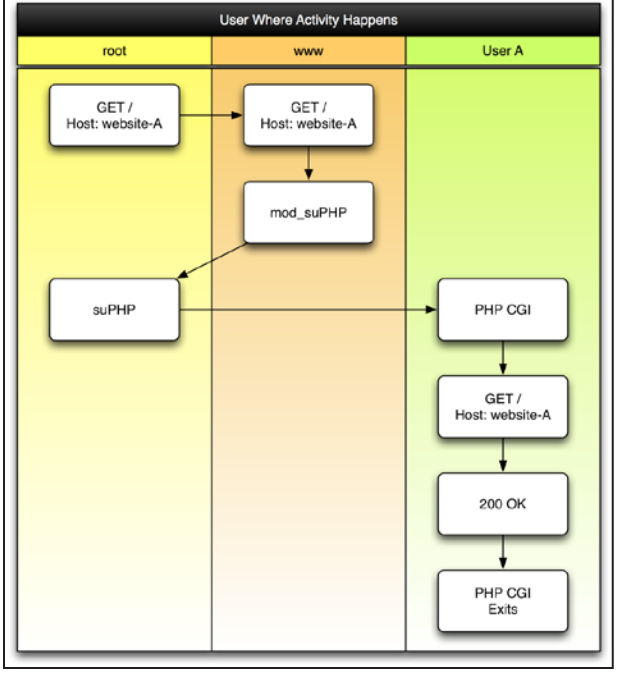


FIGURE 4



a pool of Apache child processes that just hang around until they are needed. After the page request has been served, the Apache child process goes back into the pool ready for the next request. Incidentally, it's this model that makes **mod_php** work so well, and at the same time makes it incredibly difficult for **mod_ruby** to run Ruby on Rails well.

mpm-peruser copies the same trick, except that it creates a pool of Apache processes for each user, as illustrated in Figure 5. Under UNIX-like operating systems, processes cannot change which user they run as once they have dropped their root privileges, so they cannot switch from one pool to another. (This makes UNIX-like systems inherently more secure than Windows, where processes can change which user they run as.)

mpm-itk does not copy this trick; after the Apache process has served the page request the process is terminated, as demonstrated in Figure 6. This makes **mpm-itk** a little bit slower, but simpler and less time-consuming to configure.

mpm-peruser is well suited to servers where you understand the load that the different websites on the server will receive, and where you also have the time to put into tuning the size of the Apache process pools. A good example of this might be your company website, where you want to re-use an existing PHP application such as a blog engine, but you need to run it in a separate virtual host (and as a separate user) to minimize the risk when/if it gets hacked.

By contrast, the simpler **mpm-itk** is well suited to servers where the load across the different websites is unpredictable or just not well understood. It's also a great choice when you simply don't have the time to put into tuning the process pool sizes of **mpm-peruser**. **mpm-itk** is slower than **mpm-peruser** because it doesn't maintain the per-user process pools, and it does spend longer running with root privileges, so it could prove to be a larger security risk.

What About PHP/FastCGI + suexec?

Whenever I talk about securing shared hosting servers, I'm nearly always asked why I don't recommend using

FIGURE 5

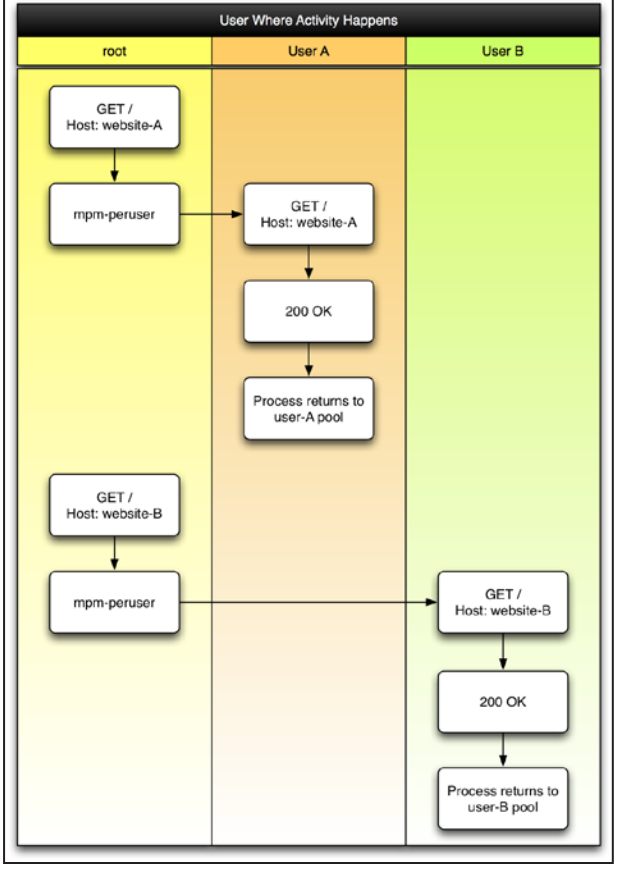
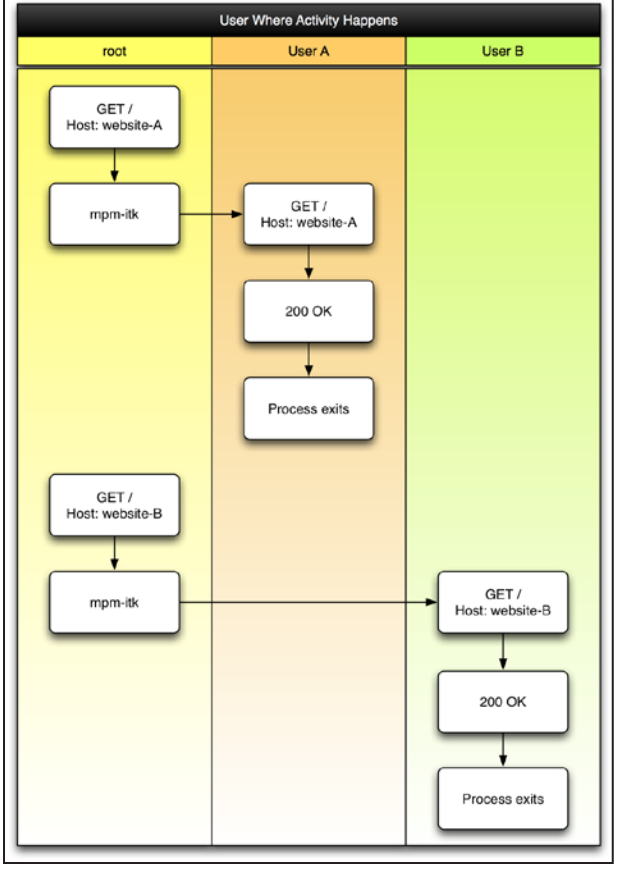


FIGURE 6



PHP/FastCGI + **suexec**, so it's something that I should cover here. My own experience is that PHP/FastCGI provides excellent performance (especially when Apache is compiled to use the multi-threaded **mpm-worker**), but even with PHP 5.2.6, a small number of Web requests fail when they should succeed. That is completely unacceptable to me. If I can't trust my Web server to handle the page requests I believe it should, how can I expect my customers to trust my website?

I'm hoping to find the time to work out exactly where the fault is (PHP? Apache? my testing?) and file a useful bug report, but until this has been done, I can't recommend PHP/FastCGI + **suexec** to anyone who needs their website to be 100% reliable.

Conclusions

Alternative Apache MPMs provide the best solution for securing a shared Web server. If you know how the load will be distributed between the websites (and if you have the time to make the effort), **mpm-peruser** provides the highest performance without compromising security. And if you don't know, then **mpm-itk** provides a simpler approach that offers the security we need.

There are bound to be other approaches out there that I haven't covered today. As I come across those and evaluate them, I'll be posting articles about them as part of The Web Platform series on my website, at *http://blog.stuartherbert.com/php/series-the-web-platform/*. You will also find instructions for installing and/or configuring **suexec**, **suphp**, **mpm-peruser** and **mpm-itk** among the articles in that series.

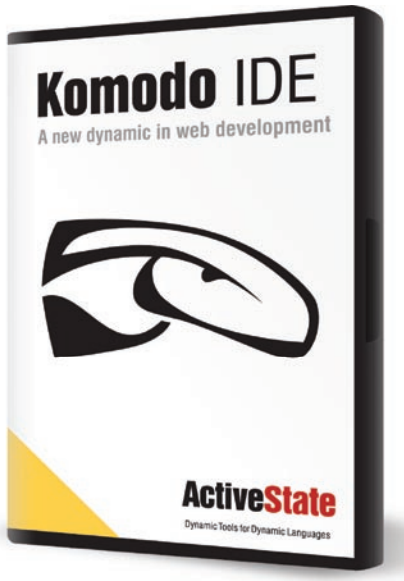
STUART HERBERT is the Technical Manager at Gradwell.com, where he oversees implementing Gradwell's mission to enable the Internet that you can't see. A co-author of the Zend Certification Study Guide for PHP 4, Stuart regularly blogs about The Web Platform—the eco-system that must be created to run your PHP applications. Away from computers, Stuart is a keen photographer and Tai Chi instructor. You can follow his blog at *http://blog.stuartherbert.com/php*.

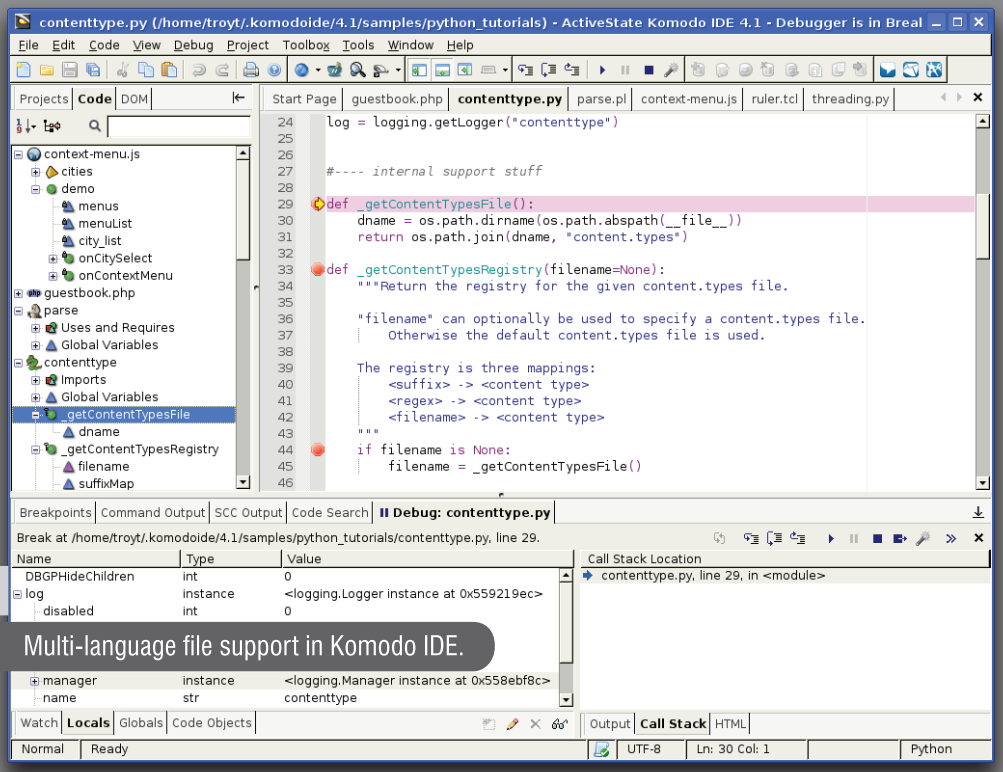
ActiveState

Komodo IDE

All your code. One killer IDE.

Create, test and debug all your sites and applications in a single powerful IDE. Writing great code is easy with multi-language editing for all the languages you use: PHP, HTML, CSS, JavaScript—and many more. Find problems fast with round-trip Ajax and PHP debugging using the open source XDebug extension. Simplify collaboration with a suite of team development tools, and create powerful plug-ins with just a few lines of code.





Multi-language file support in Komodo IDE.

Download your free trial now!

www.activestate.com/php_ide/komodo

ActiveState

Dynamic Tools for Dynamic Languages

Welcome to the Intertuber

by Marco Tabini

In my years navigating the perilous waters of the Internet, I have learned to categorize people into two groups: those who speak, and those who do.

The speaker is brash, outspoken, verbally diarrheic and, generally speaking, the least dangerous of the two. After all, he's too busy telling the world how beautiful his (or her) latest creation is that takes advantage of Google Maps to create a Web 2.0-powered representation of the migration of fire ants by means of multimedia SMS messages posted to Flickr via Yahoo! Messenger, to be capable of producing something that is *really* useful.

The doer, on the other hand, is the real disruptor. A doer has two distinguishing characteristics: he dislikes useless technology beyond the realm of personal research, and is lazy enough that he doesn't want to reinvent the wheel every time he needs to get something done—because, you see, the doer actually needs to get things done.

In the world of the Intertubes, everybody is a bit speaker and a bit doer. After all, it is difficult to create without sharing, and to share without creating. Mashups are a great example of this attitude—the vast majority of the experiments in Web-two-point-oh-ness you see are completely, utterly pointless. Do you really need to map the path of ice cream vans throughout your neighborhood? Probably not.

On the other hand, mashups are also a great example of the opportunities that technology offers us today, and that would have been very hard to reproduce just a few years ago. For better or for worse, we have settled on a number of widely-implemented communication standards that make it possible for almost any two (or three, or four) technologies to talk to each other; for the doer who is looking to solve a problem, having this Rosetta-like capability is like striking gold and finding the ore is mixed with a diamond vein.

"Do you really need to map the path of ice cream vans throughout your neighborhood?"

As a case in point, let me give you a little overview of how the registration system we have built for our conferences works. Lots of people seem to be curious about the way we do this, and I've seen other companies spend considerable amounts of money buying an off-the-shelf solution that practically does the same thing.

The entire attendee management process at any of our conferences is completely automated—a person signs up online through our website, and their information is passed along by several services until it eventually makes its way onto a badge. There are several advantages to this: first, human interaction is kept to a minimum—which also means that the opportunity for making mistakes is also kept well in check. This is particularly important when you're dealing with attendees coming from all over the world and their hotel reservations. It's easy to misspell a name when you can't even read it, and I'd rather not have to tell a person who has just flown for ten hours to come to an event he trusted me to organize that his hotel room is, well, not there.

Approximately one week before the beginning of a conference, our backend system "mashes up" with an online fax provider to send out confirmation faxes to those people who have requested them. We don't always use this feature, but the capability is there. The final confirmations, which we also send via e-mail, contain a unique identifier for each attendee as well; this comes in handy later on in the process.

Once we get on the scene at a conference, we make a copy of our

user database on a local server that sits on a secure, private network to which only our machines have access. The data resides on a MySQL database, on top of which we have built a PHP-based system that acts a simple Web service capable of performing three actions: finding an attendee by any of a number of search parameters, marking an attendee as checked in and determining whether a given attendee has already checked in. The service is exposed through a very simple JSON-based interface—the very same interface, in fact, that we use for most of our Web properties.

On the frontend, a GUI application written using Adobe AIR (and very appropriately named *Badger*) communicates with the on-site backend through the Web service and allows any number of operators to sign in attendees at a high speed, thanks to that unique ID that appeared on their final confirmation e-mail or fax. A third PHP system interfaces with a simple, run-of-the-mill label printer either through AppleScript (if the server happens to run on OSX) or COM (if it's running on Windows) and provides badge-printing capabilities.

Because a client installation of Badger can interface independently with both the server that provides user-management functionality and the server that handles the printing of labels, this system can scale to an essentially limitless level at the cost of nothing more than a $150 printer, $100 in labels and two people for every thousand or so attendees per day. And, since the technology is capable of determining whether a user has already checked in, the process can be completely decentralized with minimal risks. In fact, it would be relatively easy to integrate it with a simple kiosk capable of allowing attendees to check themselves in.

So, here you have it: a mash-up system built entirely out of glue that holds together five different technologies and takes maximum advantage of their individual strong points to minimize costs and improve productivity. It's not glamorous, but I'm happy to leave the ant-tracking, Google-mapping goodness to others while I actually solve problems.

Want **YOUR** 15 minutes of

?

php | architect
wants to hear from you!

If you want to bring a PHP-related topic – be it your personal research, company software, or anything else the professional PHP community might be interested in – why not write an article for php|architect?

If you would like to contribute, contact us and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit www.phparch.com/writeforus.php
or contact our editorial team at write@phparch.com
and get started!

NEXCESS.NET INTERNET SOLUTIONS

NEXCESS.NET Internet Solutions
304 1/2 S. State St.
Ann Arbor, MI 48104-2445
http://nexcess.net

PHP / MySQL SPECIALISTS!

Simple, Affordable, Reliable **PHP / MySQL Web Hosting Solutions**

POPULAR SHARED HOSTING PACKAGES

MINI-ME $6.95/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 $21.95/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 $16.95/mo

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

NEXRESELL 2 $59.95/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

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

php 4 — MySQL 3.x/4.0.x

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 FOR DETAILS

Dedicated & Managed Dedicated server solutions also available

Serving the web since Y2K