PHP 101 (part 13): The Trashman Cometh – Part 1

Waiting to Exhale

Maybe you've heard the term **GIGO** before.

If you haven't, it stands for **G**arbage **I**n, **G**arbage **O**ut, and it's a basic fact of computer programming: if you feed your program bad input, you're almost certainly going to get bad output. And no matter which way you cut it, bad output is not a Good Thing for a programmer who wants to get noticed.

In case you think I'm exaggerating, let me give you a simple example. Consider an online loan calculator that allows a user to input a desired loan amount, finance term and interest rate.

Let's assume that the application doesn't include any error checks, and that the user decides to enter that magic number, 0, into the *Term* field.

You can imagine the result. After a few internal calculations the application will end up attempting to divide the total amount payable by zero. The slew of ugly error messages that follow don't really bear discussion, but it's worth noting that they could have been avoided had the developer had the foresight to include an **input validation routine** when designing the application.

The moral of this story? If you're serious about using PHP for web development,

one of the most important things you must learn is how to validate user input and deal with potentially unsafe data. Such input verification is one of the most important safeguards
a developer can build into an application, and a failure to do this can snowball into serious problems, or even cause your application to break when it encounters invalid
or corrupt data.

That's where this edition of PHP 101 comes in. Over the next few paragraphs, I'm going to show you some basic tricks to validate user input, catch "bad" data before it corrupts your calculations and databases, and provide user notification in a gentle, understandable and non-threatening way. To prepare for this exercise, I suggest you spin up a CD of John
Lennon singing '*Imagine*', fill your heart with peace and goodwill towards all men, and take a few deep, calming breaths. Once you've exhaled, we can get going.

An Empty Vessel...

This tutorial assumes that the user input to be validated arrives through a web form. This
is not the only way a PHP script can get user data; however, it is the most common way. If
your PHP application needs to validate command-line input, I'd recommend you read my
article on the PEAR `Console_Getopt` class, available for your perusal at
http://www.zend.com/pear/tutorials/Console-Getopt.php.

It's common practice to use client-side scripting languages such as JavaScript or VBScript

for client-side form validation. However, this type of client-side validation is not foolproof. You're not in control of the client, so if a user turns off JavaScript in his or her browser, all your efforts to ensure that the user does not enter irrelevant
data become – well – irrelevant. That's why most experienced developers use both
client-side and server-side validation. Server-side validation involves checking the values submitted to the server through a PHP script, and taking appropriate action when the input is incorrect.

Let's begin with **the most commonly found input error**: a required form field that is missing its value. Take a look at this example:

```
<html>
<head></head>
<body>
<?php
if (!isset($_POST['submit'])) {
?>
   <form action = '<?php $_SERVER['PHP_SELF'] ?>' method = 'post'>
   Which sandwich filling would you like?
      <br />
   <input type = 'text' name = 'filling'>
   <br />
   <input type = 'submit' name = 'submit' value = 'Save'>
```

```
    </form>
<?php
}
else {
    // set database variables
        $host = 'localhost';
    $user = 'user';
    $pass = 'secret';
    $db = 'sandwiches';
        // get user input
    $filling = mysql_escape_string($_POST['filling']);
        // open connection
    $connection = mysql_connect($host, $user, $pass) or die('Unable to connect!');
        // select database
    mysql_select_db($db) or die('Unable to select database!');
        // create query
    $query = 'INSERT INTO orders (filling) VALUES ("$filling")';
        // execute query
    $result = mysql_query($query) or die("Error in query: $query. ".mysql_error());
        // close connection
        mysql_close($connection);
        // display message
    echo "Your {$_POST['filling']} sandwich is coming right up!";
}
?>
</body>
```

```
</html>
```

It’s clear from the example above that submitting the form without entering any data

will result in an empty record being added to the database (assuming no `NOT NULL`

constraints on the target table). To avoid this, it’s important to verify that the form does, in fact, contain valid data, and only then perform the `INSERT` query.

Here’s how:

```
<html>
<head></head>
<body>
<?php
if (!isset($_POST['submit'])) {
?>
    <form action = '<?php $_SERVER['PHP_SELF'] ?>' method = 'post'>
        Which sandwich filling would you like?
    <br />
    <input type = 'text' name = 'filling'>
    <br />
    <input type = 'submit' name = 'submit' value = 'Save'>
    </form>
```

```
<?php
}
else {
    // check for required data
    // die if absent
    if (!isset($_POST['filling']) || trim($_POST['filling']) == '') {
        die("ERROR: You can't have a sandwich without a filling!");
    }
    else {
        $filling = mysql_escape_string(trim($_POST['filling']));
    }
    // set database variables
    $host = 'localhost';
    $user = 'user';
    $pass = 'secret';
    $db = 'sandwiches';
    // open connection
    $connection = mysql_connect($host, $user, $pass) or die('Unable to connect!');
    // select database
    mysql_select_db($db) or die('Unable to select database!');
    // create query
    $query = 'INSERT INTO orders (filling) VALUES ("$filling")';
    // execute query
    $result = mysql_query($query) or die("Error in query: $query. ".mysql_error());
    // close connection
    mysql_close($connection);
```

```
    // display message
    echo "Your {$_POST['filling']} sandwich is coming right up!";
}
?>
</body>
</html>
```

The error check here is both simple and logical: the `trim()` function is used to trim leading and trailing spaces from the field value, which is then compared with an empty string. If the match is true, the field was submitted empty, and the script dies with an error message before MySQL comes into the picture.

A common mistake, especially among newbies, is to replace the `isset()` and `trim()` combination with a call to PHP's `empty()` function, which tells you if a variable is empty. This isn't usually a good idea, because `empty()` has a fatal flaw: it'll return true even if a variable contains the number 0. The following simple example illustrates this:

```
<?php
// no data, returns empty
$data = '';
echo empty($data) ? "$data is empty" : "$data is not empty";
echo "<br />\n";
// some data, returns not empty
```

```
$data = '1';
echo empty($data) ? "$data is empty" : "$data is not empty";
echo "<br />\n";
// some data, returns empty
$data = '0';
echo empty($data) ? "$data is empty" : "$data is not empty";
?>
```

So, if your form field is only allowed to hold non-empty, non-zero data, `empty()` is a good choice for validating it. But if the range of valid values for your field includes the number 0, stick with the `isset()` and `trim()` combination instead.

Not My Type

So now you know how to catch the most basic error – missing data – and stop script processing before any damage takes place. But what if the data's present, but of the wrong type or size? Your 'missing values' test won't be triggered, but your calculations and database could still be affected. Obviously, then, you need to add a further layer of security, wherein the data type of the user input is also verified.

Here's an example which illustrates:

```
<html>
<head></head>
```

```
<body>
<?php
if (!isset($_POST['submit'])) {
?>
    <form action = '<?php $_SERVER['PHP_SELF']?>' method = 'post'>
        How many sandwiches would you like? (min 1, max 9)
    <br />
    <input type = 'text' name = 'quantity'>
    <br />
    <input type = 'submit' name = 'submit' value = 'Save'>
    </form>
<?php
}
else {
        // check for required data
    // die if absent
    if (!isset($_POST['quantity']) || trim($_POST['quantity']) == '') {
            die ("ERROR: Can't make 'em if you don't say how many!");
    }
        // check if input is a number
    if (!is_numeric($_POST['quantity'])) {
            die ("ERROR: Whatever you just said isn't a number!");
    }
        // check if input is an integer
    if (intval($_POST['quantity']) != $_POST['quantity']) {
```

```
            die ("ERROR: Can't do halves, quarters or thirds... I'd lose my job!");
    }
    // check if input is in the range 1-9
    if (($_POST['quantity'] < 1) || ($_POST['quantity'] > 9)) {
            die ('ERROR: I can only make between 1 and 9 sandwiches per order!');
    }
    // process the data
    echo "I'm making you {$_POST['quantity']} sandwiches. Hope you can eat them all!";
}
?>
</body>
</html>
```

Notice that once I've established that the field contains some data, I've added a bunch of tests to make sure it meets data type and range constraints. First, I've checked

if the value is numeric, with the `is_numeric()` function. This function tests a string to see if it is a *numeric string* – that is, a string consisting only of numbers.

Assuming what you've got is a number, the next step is to make sure it's an integer value

between 1 and 9. To test if it's an integer, I've used the `intval()` function to extract the integer part of the value, and tested it against the value itself. Float values (such as 2.5) will fail this test; integer values will pass it. The final step

before

green-lighting the value is to see if it falls between 1 and 9. This is easy to accomplish with a couple of inequality tests.

Whilst on the topic, it's also worth mentioning the `strlen()` function, which returns the length of a string. This can come in handy to make sure that form input

doesn't exceed a particular length. The following example shows how:

```
<html>
<head></head>
<body>
<?php
if (!isset($_POST['submit'])) {
?>
   <form action = '<?php $_SERVER['PHP_SELF']?>' method = 'post'>
      Enter a nickname 6-10 characters long:
   <br />
   <input type = 'text' name = 'nick'>
   <br />
   <input type = 'submit' name = 'submit' value = 'Save'>
   </form>
<?php
}
```

```
else {
        // check for required data
    // die if absent
    if (!isset($_POST['nick']) || trim($_POST['nick']) == '') {
            die ('ERROR: Come on, surely you can think of a nickname! How about
Pooky?');
    }
        // check if input is of the right length
    if (!(strlen($_POST['nick']) >= 6 && strlen($_POST['nick']) <= 10)) {
            die ("ERROR: That's either too long or too short!");
    }
        // process the data
    echo "I'll accept the nickname {$_POST['nick']}, seeing as it's you!";
}
?>
</body>
</html>
```

Here, the `strlen()` function is used to verify that the string input is neither too long nor too short. It’s also a handy way to make sure that input data satisfies the field length constraints of your database. For example, if you have a MySQL `VARCHAR(10)` field, strings over 10 characters in length will be truncated. The `strlen()` function can serve as an early warning system in such cases, notifying the user of the length mismatch and avoiding data corruption.

The Dating Game

Validating dates is another important aspect of input validation. It's all too easy, given a series of drop-down list boxes or free-form text fields, for a user to select a date like `29-Feb-2005` or `31-Apr-2005`, neither of which is valid. Therefore, it's important to check that date values provided by the user are valid before using them in a calculation.

In PHP, this task is significantly simpler than in other languages, because of the `checkdate()` function. This function accepts three arguments – month, day and year – and returns a Boolean value indicating whether or not the date is valid. The following example demonstrates it in action:

```
<html>
<head></head>
<body>
<?php
if (!isset($_POST['submit'])) {
?>
   <form action = '<?php $_SERVER['PHP_SELF']?>' method = 'post'>
   Enter your date of birth:
   <br /><br />
   <select name = 'day'>
```

```
    <?php
    // generate day numbers
    for ($x = 1; $x <= 31; $x++) {
        echo "<option value = $x>$x</option>";
    }
    ?>
    </select>
    <select name = 'month'>
    <?php
    // generate month names
    for ($x = 1; $x <= 12; $x++) {
            echo "<option
value=$x>".date('F', mktime(0, 0, 0, $x, 1,1)).'</option>';
    }
    ?>
    </select>
    <select name = 'year'>
    <?php
    // generate year values
    for ($x = 1950; $x <= 2005; $x++) {
            echo "<option value=$x>$x</option>";
    }
    ?>
    </select>
```

```
    <br /><br />
        <input type = 'submit' name = 'submit' value = 'Save'>
    </form>
<?php
}
else {
    // check if date is valid
    if (!checkdate($_POST['month'], $_POST['day'], $_POST['year'])) {
            die("ERROR: The date {$_POST['day']}-{$_POST['month']}-
{$_POST['year']} doesn't exist!");
    }
    // process the data
    echo "You entered {$_POST['day']}-{$_POST['month']}-{$_POST['year']} -
which is a valid date.";
}
?>
</body>
</html>
```

Try entering an invalid date, and see how PHP calls you on it. Ain't that cool?

If you're storing date input in a MySQL table, it's interesting to note that MySQL does *not* perform any rigorous date verification of its own before accepting a `DATE`, `DATETIME` or `TIMESTAMP` value. Instead, it expects the developer to build date verification into the application itself. The most that MySQL will do, if it encounters an obviously illegal value, is convert the date

to a

zero value – not very helpful at all! Read more about this at

http://dev.mysql.com/doc/mysql/en/datetime.html.

While we're on the topic, let's talk a little bit more about **multiple-choice form elements** like drop-down list boxes and radio buttons. In cases where it's mandatory to

make a choice, a developer must verify that at least one of the available options has

been selected by the user. This mainly involves clever use of the `isset()`

and – for multi-select list boxes – the `is_array()` and `sizeof()`

functions. The next example illustrates this:

```
<html>
<head></head>
<body>
<?php
if (!isset($_POST['submit'])) {
?>
   <form action = '<?php $_SERVER['PHP_SELF'] ?>' method = 'post'>
   Pizza base:
   <br />
   <input type = 'radio' name = 'base' value = 'thin and crispy'>Thin and crispy
```

```
    <input type = 'radio' name = 'base' value = 'deep-dish'>Deep-dish
  <br />
Cheese:
  <br />
  <select name = 'cheese'>
    <option value = 'mozzarella'>Mozzarella</option>
    <option value = 'parmesan'>Parmesan</option>
        <option value = 'gruyere'>Gruyere</option>
  </select>
  <br />
  Toppings:
  <br />
  <select multiple name = 'toppings[]'>
    <option value = 'tomatoes'>Tomatoes</option>
        <option value = 'olives'>Olives</option>
    <option value = 'pepperoni'>Pepperoni</option>
    <option value = 'onions'>Onions</option>
    <option value = 'peppers'>Peppers</option>
    <option value = 'sausage'>Sausage</option>
        <option value = 'anchovies'>Anchovies</option>
  </select>
  <br />
```

```
    <input type = 'submit' name = 'submit' value = 'Save'>
    </form>
<?php
}
else {
        // check radio button
    if (!isset($_POST['base'])) {
        die('You must select a base for the pizza');
    }
        // check list box
        if (!isset($_POST['cheese'])) {
        die('You must select a cheese for the pizza');
    }
        // check multi-select box
    if (!is_array($_POST['toppings']) || sizeof($_POST['toppings']) < 1) {
                die('You must select at least one topping for the pizza');
    }
        // process the data
    echo "One {$_POST['base']} {$_POST['cheese']} pizza with ";
    foreach ($_POST['toppings'] as $topping) echo $topping.", ";
        echo "coming up!";
}
?>
</body>
</html>
```

Nothing to tax your brain too much here – the `isset()` function merely checks to see if at least one of a set of options has been selected, and prints an error message if this is not the case. Notice how the multi-select list box is validated: when the form is submitted, selections made here are placed in an array, and PHP's `is_array()` and `sizeof()` functions are used to test that array and ensure that it contains at least one element.