



# By The Disenchanted Developer

# Table of Contents

# Cherchez La Femme

So there I was, minding my own business, working on a piece of code I had to deliver that evening, when the pretty dark−haired girl who sits in the cubicle behind me popped her head over and asked for my help.

"Look", she said, "I need your help with something. Can you write me a little piece of code that keeps track of Web site URLs and tells me when they change?"

"Huh?", was my first reaction...

"It's like this", she explained, "As part of a content update contract, I'm in charge of tracking changes to about thirty different Web sites for a customer, and sending out a bulletin with those changes. Every day, I spend the morning visiting each site and checking to see if it's changed. It's very tedious, and it really screws up my day. Do you think you can write something to automate it for me?"

Now, she's a pretty girl...and the problem intrigued me. So I agreed.

# A Little Research

The problem, of course, appeared when I actually started work on her request. I had a vague idea how this might work: all I had to do, I reasoned, was write a little script that woke up each morning, scanned her list of URLs, downloaded the contents of each, compared those contents with the versions downloaded previously, and sent out an email alert if there was a change.

Seemed simple − but how hard would it be to implement? I didn't really like the thought of downloading and saving different versions of each page on a daily basis, or of creating a comparison algorithm to test Web pages against each other.

I thought there ought to be an easier way. Maybe the Web server had a way of telling me if a Web page had been modified recently − and all I had to do was read that data and use it in a script. Accordingly, my first step was to hit the W3C Web site, download a copy of the HTTP protocol specification, from ftp://ftp.isi.edu/in−notes/rfc2616.txt, and print it out for a little bedside reading. Here's what I found, halfway through:

```
The Last−Modified entity−header field indicates the date and
time at
which the origin server believes the variant was last
modified.
```

There we go, I thought − the guys who came up with the protocol obviously anticipated this requirement and built it into the protocol headers. Now to see if it worked...

The next day at work, I fired up my trusty telnet client and tried to connect to our intranet Web server and request a page. Here's the session dump:

```
$ telnet darkstar 80
Trying 192.168.0.10...
Connected to darkstar.melonfire.com.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 18 Oct 2002 08:47:57 GMT
Server: Apache/1.3.26 (Unix) PHP/4.2.2
Last−Modified: Wed, 09 Oct 2002 11:27:23 GMT
Accept−Ranges: bytes
Content−Length: 1446
Connection: close
Content−Type: text/html

Connection closed by foreign host.
```

As you can see, the Web server returned a "Last−Modified" header indicating the date of last change of the requested file. So far so good.

# Code Poet

With the theory out of the way, I was just about ready to make my first stab at the code. Since I was told that there are a large number of URLs to be monitored, I decided to use a MySQL database table to store them, in addition to a brief description of each URL. The table I came up with is pretty simple − here's what it looked like:

```
CREATE TABLE urls (
id tinyint(3) unsigned NOT NULL auto_increment,
url text NOT NULL,
dsc varchar(255) NOT NULL default '',
date datetime default NULL,
email varchar(255) NOT NULL default '',
PRIMARY KEY (id)
);
```

And here's a sample of the data within it:

```
mysql> select * from urls;
+----+--------------------------------+---------------+------+--------
+----+--------------------------------+---------------+------+----
------+
| id | url | dsc | date | email
|
+----+--------------------------------+---------------+------+--------
+----+--------------------------------+---------------+------+----
------+
| 1 | http://www.melonfire.com/ | Melonfire.com | NULL |
user@some.domain |
| 2 | http://www.yahoo.com/ | Yahoo.com | NULL |
user@some.domain |
| 3 | http://www.devshed.com/ | Devshed.com | NULL |
user@some.domain |
+----+--------------------------------+---------------+------+--------
+----+--------------------------------+---------------+------+----
------+
3 rows in set (0.00 sec)
```

Next up, I needed a script that would iterate through this database table, connect to each of the URLs listed within it, and obtain the value of the "Last−Modified" header − basically, replicate what I did with my telnet client, as many times as there were URLs. Here's what I put together:

```
<?php
// DB connection parameters
$db_host="localhost";
$db_user="joe";
$db_pass="65h49";
$db_name="db167";

// open database connection
$connection = mysql_connect($db_host, $db_user, $db_pass) or
die
("Unable to connect!"); mysql_select_db($db_name);

// generate and execute query
$query1 = "SELECT id, url, date, dsc, email FROM urls";
$result1 =
mysql_query($query1, $connection) or die ("Error in query:
$query1 . " .
mysql_error());

// if rows exist
if (mysql_num_rows($result1) > 0)
{
// iterate through resultset
while(list($id, $url, $date, $desc, $email) =
mysql_fetch_row($result1))
{
$response = "";

// parse URL into component parts
$arr = parse_url($url);

// open a client connection
$fp = fsockopen ($arr['host'], 80);

// send HEAD request and read response
$request = "HEAD /" . $arr['path'] . "
HTTP/1.0\r\n\r\n";
fputs ($fp, $request);
while (!feof($fp))
{
$response .= fgets ($fp, 500);
}
fclose ($fp);

// split response into lines
$lines = explode("\r\n", $response);

// scan lines for "Last-Modified" header
foreach($lines as $l)
```

```
{
if (ereg("^Last-Modified:", $l))
{
// split into variable-value component
$arr2 = explode(": ", $l);
$newDate = gmdate("Y-m-d H:i:s",
strtotime($arr2[1]));

// if date has changed from
last-recorded date
if ($date != $newDate)
{
// send mail to owner
mail($email, "$desc has
changed!", "This is an automated message to inform you that
the URL
\r\n\r\n $url \r\n\r\nhas changed since it was last checked.
Please
visit the URL to view the changes.", "From: The Web Watcher
<nobody@some.domain>") or die ("Could not send mail!");

// update table with new date
$query2 = "UPDATE urls SET date
= '" . $newDate . "' WHERE id = '" . $id . "'";
$result2 = mysql_query($query2,
$connection) or die ("Error in query: $query2 . " .
mysql_error());
}
}
}
}
}

// close database connection
mysql_close($connection);
?>
```

How does this work? Let's look at that next.

# Digging Deep

The first step in my script is to connect to the MySQL database and run a query to get a list of all the URLS to be checked.

```
// open database connection
$connection = mysql_connect($db_host, $db_user, $db_pass) or
die
("Unable to connect!"); mysql_select_db($db_name);

// generate and execute query
$query1 = "SELECT id, url, date, dsc, email FROM urls2";
$result1 =
mysql_query($query1, $connection) or die ("Error in query:
$query1 . " .
mysql_error());
```

Assuming the query returns one or more rows, the next step is to iterate through the resultset and process each record:

```
// if rows exist
if (mysql_num_rows($result1) > 0)
{
// iterate through resultset
while(list($id, $url, $date, $desc, $email) =
mysql_fetch_row($result1))
{

// processing code here

}
}
```

For each URL found, I need to extract the host name and the file path on the server − this is extremely easy with PHP's very cool parse_url() function, which returns an associative array containing the various constituent elements of the URL.

```
// parse URL into component parts
$arr = parse_url($url);
```

This data can then be used to open a socket connection to the host Web server, send an HTTP HEAD request, and place the response in a PHP variable.

```
// open a client connection
$fp = fsockopen ($arr['host'], 80);

// send HEAD request and read response
$request = "HEAD /" . $arr['path'] . " HTTP/1.0\r\n\r\n";
fputs ($fp, $request);
while (!feof($fp))
{
$response .= fgets ($fp, 500);
}
fclose ($fp);
```

This response is then broken up into individual lines, and each line is scanned for the "Last−Modified" header − note my use of the ereg() function to accomplish this task. The corresponding date is then converted into a UNIX−compliant timestamp with the strtotime() function, and that timestamp is again converted into a MySQL−compliant DATETIME data type, suitable for entry into the MySQL table.

```
// split response into lines
$lines = explode("\r\n", $response);

// scan lines for "Last−Modified" header
foreach($lines as $l)
{
if (ereg("^Last−Modified:", $l))
{
// split into variable−value component
$arr2 = explode(": ", $l);
$newDate = gmdate("Y−m−d H:i:s", strtotime($arr2[1]));

// snip

}
}
```

The date retrieved from the "Last−Modified" HTTP header is then compared with the date previously recorded for that URL in the database. If the dates are the same, it implies that the page located at that URL has not been modified since it was last checked. If they're different, it implies that a change has taken place and the user should be alerted to it. The database also needs to be updated with the new modification date, so as to provide an accurate benchmark for the next run of the script.

```
// if date has changed from last−recorded date
if ($date != $newDate)
{
```

```
// send mail to owner
mail($email, "$desc has changed!", "This is an automated
message
to inform you that the URL \r\n\r\n $url \r\n\r\nhas changed
since it
was last checked. Please visit the URL to view the changes.",
"From: The
Web Watcher
<nobody@some.domain>") or die ("Could not send mail!");

// update table with new date
$query2 = "UPDATE urls SET date = '" . $newDate . "' WHERE id
=
'" . $id . "'";
$result2 = mysql_query($query2, $connection) or die ("Error in
query: $query2 . " . mysql_error());
}
```

It might look complicated − but it's actually pretty straightforward. Will it work?

# Backtracking

So far, it looks like everything's hunky−dory − but being the suspicious character I am, I thought it might be worth trying the code out against a few servers before accepting the script above as a reliable tool. And that's when I hit my first roadblock − as it turned out, some servers didn't return the "Last−Modified" header, which meant that the script couldn't determine when the page had been last modified.

I thought this was pretty strange, as it seemed to be a violation of the rules laid down in the HTTP protocol. Back to the specification, then, to see if I could resolve this apparent conflict...

A little close reading, and the reason for the discrepancy became clear:

```
HTTP/1.1 servers SHOULD send Last−Modified whenever feasible.
```

In other words − they don't *have to*. And there's many a slip betwixt the cup and the lip...

OK, maybe I should have read the fine print before writing that script. Still, better late than never.

Back to the drawing board, then. After a little thought and a few carefully−posed questions to the PHP mailing lists, it seemed that my initial plan was still the most reliable − download and store the contents of each URL, and compare those contents against the previous version to see if there was any change. This wasn't the most efficient way to do it − but it didn't look like I had any alternatives.

# Plan B

My next step, therefore, was to redesign my database table to support my new design. Here's the updated schema:

```
CREATE TABLE urls (
id tinyint(3) unsigned NOT NULL auto_increment,
url text NOT NULL,
dsc varchar(255) NOT NULL default '',
md5 varchar(255) NOT NULL default '',
email varchar(255) NOT NULL default '',
PRIMARY KEY (id)
);
```

Notice that I've replaced the original "date" column with one that holds the MD5 checksum for the page being monitored. Why? I figured that I could save myself a little disk space (and the time spent on designing a comparison algorithm) by using the MD5 checksum features built into PHP.

With the database schema updated, the next step is to update the PHP script that does all the work:

```
<?php
// set up database access parameters
$db_host="localhost";
$db_user="joe";
$db_pass="65h49";
$db_name="db167";

// open connection to database
$connection = mysql_connect($db_host, $db_user, $db_pass) or
die
("Unable to connect!"); mysql_select_db($db_name);

// generate and execute query
$query1 = "SELECT id, url, dsc, md5, email FROM urls";
$result1 = mysql_query($query1, $connection) or die ("Error in
query:
$query1 . " . mysql_error());

// if rows exist
if (mysql_num_rows($result1) > 0)
{
// iterate through resultset
while(list($id, $url, $desc, $csum1, $email) =
mysql_fetch_row($result1))
{
```

```
// read page contents into a string
$contents = join ('', file ($url));

// calculate MD5 value
$csum2 = md5($contents);

// compare with earlier value
if ($csum1 != $csum2)
{
// send mail to owner
mail($email, "$desc has changed!", "This is an
automated message to inform you that the URL \r\n\r\n $url
\r\n\r\nhas
changed since it was last checked. Please visit the URL to
view the
changes.", "From: The Web Watcher
<nobody@some.domain>") or die ("Could not send mail!");

// update database with new checksum if changed
$query2 = "UPDATE urls SET md5 = '$csum2' WHERE
id = '" . $id . "'";
$result2 = mysql_query($query2, $connection) or
die ("Error in query: $query2 . " . mysql_error());

}
}
}

// close database connection
mysql_close($connection);
?>
```

What's the difference between this script and the one I wrote earlier? This one retrieves the complete contents of the URL provided, using PHP's file() method, concatenates it into a single string, and creates a unique MD5 checksum to represent that string. This checksum is then compared to the checksum stored in the database from the last run; if they match, it implies that the URL has not changed at all.

In case you're wondering what MD5 is, nope, it's not James Bond's employer, the following extract from http://www.faqs.org/rfcs/rfc1321.html might be enlightening: "The [MD5] algorithm takes as input a message of arbitrary length and produces as output a 128−bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. The MD5 algorithm is intended for digital signature applications."

Any change to the Web page located at that URL will result in a new checksum being generated the next time the script runs; this new checksum can be compared with the previous one and the results emailed to the concerned person, with the database simultaneously updated to reflect the new checksum.

Here's the relevant section of code:

```
// read page contents into a string
$contents = join ('', file ($url));

// calculate MD5 value
$csum2 = md5($contents);

// compare with earlier value
if ($csum1 != $csum2)
{
// send mail to owner
mail($email, "$desc has changed!", "This is an automated
message
to inform you that the URL \r\n\r\n $url \r\n\r\nhas changed
since it
was last checked. Please visit the URL to view the changes.",
"From: The
Web Watcher
<nobody@some.domain>") or die ("Could not send mail!");

// update database with new checksum if changed
$query2 = "UPDATE urls SET md5 = '$csum2' WHERE id = '" . $id
.
"'";
$result2 = mysql_query($query2, $connection) or die ("Error in
query: $query2 . " . mysql_error());
}
```

Since this system does not depend on the presence or absence of specific headers in the HTTP response, it is far more reliable − though not as efficient − than the previous technique.

All that's left is to put this script into the server "crontab", so that it runs once a day:

```
5 0 * * * /usr/local/bin/php −q webwatcher.php > /dev/null
2>&1
```

Here's a sample message generated by the system:

```
Date: Fri, 18 Oct 2002 15:12:52 +0530
Subject: Melonfire.com has changed!
To: user@some.domain
From: The Web Watcher <nobody@some.domain>
```

```
This is an automated message to inform you that the URL

http://www.melonfire.com/

has changed since it was last checked. Please visit the URL to
view the
changes.
```

I have to warn you, though, that this system can substantially degrade performance on your server if you feed it a large number of URls to monitor. Even though MD5 is a pretty efficient algorithm, the time taken to connect to each URL, retrieve the contents, create a checksum and process the results can eat up quite a few processor cycles (one of the reasons why I'm running it at midnight via "cron"). If you're planning to use this in your own organization, limit the number of URLs to around thirty and try and give the script relatively−smaller−sized Web pages to track...or else you might find yourself rapidly scratched off your network administrator's Christmas card list.

# Closing Time

And that's about it. I set up the database for the pretty dark−haired girl, restarted the "cron" daemon on our intranet server, and went back to work. This utility was a pleasant detour from my daily development tasks − I learnt a little bit about HTTP, socket connections, the MD5 algorithm and reading files over HTTP − and I hope you enjoyed reading about it as much as I enjoyed developing it.

In case you'd like to read more about the techniques discussed in this article, here are a few links that I found helpful:

The HTTP 1.1 specification, at ftp://ftp.isi.edu/in−notes/rfc2616.txt

The MD5 Message−Digest Algorithms, at http://www.faqs.org/rfcs/rfc1321.html

The PHP mail() function, at http://www.php.net/manual/en/function.mail.php

The PHP md5() function, at http://www.php.net/manual/en/function.md5.php

Using remote files with PHP, at http://www.php.net/manual/en/features.remote−files.php

Using PHP from the command line, at http://www.php.net/manual/en/features.commandline.php

Until next time...ciao!

Note: Examples are illustrative only, and are not meant for a production or professional services environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!