O'REILLY
ONLamp.com
LAMP: THE OPEN SOURCE WEB PLATFORM

Published on **ONLamp.com** (http://www.onlamp.com/)
See this if you're having trouble printing code examples

Using MySQL to Stop Editing Web Pages

by Russell Dyer
02/19/2004

Although there's much that can be done with web design, sometimes I find it to be extremely boring. When I'm deep into a Perl project, the last thing I want is to meet with other department managers to discuss changes in the text on the corporate web site. It's not a good (or interesting) use of my time. As a result, over the last few years I've developed CGI scripts for sites in Perl and databases in MySQL so that non-technical staff can manage and update site content with little help from me.

Concepts and Layout

When most technical people think of web pages, I suspect that they think of static pages. They know that web pages may be dynamically generated, but the CGI code that I've seen usually has static HTML buried in it. It's true that at the core there will always be text. Nevertheless, non-technical people cannot edit content text contained in a flat HTML page or in a CGI script. A technical person must make any changes for them.

The goal of developing dynamic web pages is not only to reduce code to a minimum, but also to reduce the involvement of technical staff in maintaining the data, especially data that belongs to other departments. With this concept in mind, you should eliminate all flat HTML pages and lift out of CGI scripts all text that is displayed to the user. This should be your goal initially, and your rule eventually, if you want to get out of the text-management business and instead focus on what you enjoy and what you were hired to do: develop software.

If it's to become accessible to non-technical staff, content text belongs in a database like MySQL. All content should be stored in tables for easy retrieval and editing. On a simple web page there, is typically a page heading and a short paragraph introducing the page. Below that it's common to have sub-headings with paragraphs of related text. There are often hyperlinks to other sections and pages in the left margin. Each of these components should be separated and stored in MySQL. This may seem tedious and unnecessary at first, but it can be very useful and save you enormous amounts of time. It also will allow those who are concerned with site content to make changes on their own.

Content Text

We'll start by creating a database and a table in MySQL that will store the page headings and the introductory text for every page of a fictitious web site. It's good form to include a record number and the record creation date. We also need a column for section, so that our CGI scripts can select the correct record for the section of the site to be displayed. Finally, we will need a field for the headings and another for text to be posted. Here's what we'll enter in the `mysql` client:

```
CREATE DATABASE db1;

    USE db1;

    CREATE TABLE postings (
        rec_id    INT AUTO_INCREMENT  PRIMARY KEY,
        post_date DATE,
```

```
        section   INT,
        heading   VARCHAR(50),
        post      TEXT
    );
```

The first two lines create the database called `db1` and then switch the MySQL session to it. The next set of lines create our first table, `postings`. The record identification column is automatically incremented so that we're assured of unique numbers for each row. The column `section` is an integer data type so that it can be linked to another table that we'll create to house data on each section — that table will have a `rec_id` column like this table. We've limited the heading to fifty characters; that should be enough. Finally, the `post` column will store the page's introductory text. We've made this column's data type `TEXT`, which will hold up to 64K of text, which is plenty.

That takes care of our page headings and opening paragraphs. We could create a separate table in MySQL to store the sub-text of each page. Instead, we'll just use the `postings` table with some modifications. We're already pushing our new policies, I know.

```
ALTER TABLE postings
    ADD COLUMN type     ENUM('top', 'sub') AFTER section,
    ADD COLUMN sequence INT                AFTER section,
    ADD COLUMN status   ENUM('ac', 'in');
```

We've added an enumerated column called `type` with choices to designate the difference between postings for the top of web pages and for sub-postings. Before that, we've added a column to allow a non-technical administrator to dictate the sequence in which pages will be displayed. The last column added will allow postings to be created in advance with a setting of inactive (`in`) so that they're not displayed until they're activated (`ac`). Of course, they could also be deactivated later, if desired. This all makes it possible to use a `SELECT` statement like the following to retrieve and display the headings and related text:

```
sub postings {
    my ($q, $section, $type) = @_;
    my $sql_stmnt = "SELECT heading, post
                     FROM   postings
                     WHERE  section=?
                     AND    type=?
                     AND    status='ac'
                     ORDER BY sequence";
    my $dbh = DBI->connect("DBI:mysql:db1:localhost", "user", "password")
              || die "Could not connect:" . DBI->errstr;
    my $sth = $dbh->prepare($sql_stmnt);
    $sth->execute( $section, $type );

    my $postings = $sth->fetchall_arrayref();
    $sth->finish();
    $dbh->disconnect();

    foreach my $row (@$postings) {
        my($heading, $text) = @$row;
        print $q->h3("$heading"), "\n",
        $q->p("$text"), "\n";
    }
}
```

Before explaining this function, let's look at how we might call it from a Perl CGI script:

```
#!/usr/bin/perl -w

use strict;

require 'library.pl';

use DBI;
use CGI;

my $q       = CGI->new();
```

```
my $section = $q->param("section");

...  # start HTML
postings($q,$section,'top');
postings($q,$section,'sub');

...
```

In the script excerpt just above, we're requiring or merging *library.pl* into our script — that's the script that contains the function `postings`. Next we're calling *CGI.pm* and setting up a new CGI object (`$q`). We then capture the parameter labeled `section` that's sent to the script by the user (e.g., `index.cgi?section=1001`). After our initial HTML items, we'll call the function `postings`, passing it the CGI object and the section number. Going back to the function, we parse out the section number and store it in `$section` and the posting type and store it in `$type` for our local use. We next insert these values into our SQL statement. We then connect to MySQL and fetch the matching rows from the table all at once and store the results in memory. Finally, we loop through the array of rows of data and print each heading and paragraph.

Incidentally, the variable `$links` is actually a reference to an array containing the rows of data retrieved. This is why you see the perhaps odd notations (e.g., `@$links`). To learn more about references, read Randal Schwartz' excellent new book Learning Perl Objects, References & Modules.

Margin Links

The next table we'll need to create will keep track of the hyperlinks displayed in the left margin of each page. For this table, we'll have SQL statements in our CGI scripts that will retrieve certain rows based on which section of the site the user is viewing.

Therefore, the table `links` will not only need columns related to the hyperlink, but also a column for the section number:

```
CREATE TABLE links (
    rec_id      INT          AUTO_INCREMENT PRIMARY KEY,
    name        VARCHAR(75),
    description VARCHAR(50),
    type        ENUM('section', 'sub-section', 'all', 'offsite'),
    address     VARCHAR(100),
    target      ENUM('_blank', '_self'),
    section     INT,
```

```
    sequence    INT,
    status      ENUM('ac','in'));
```

The column called `name` contains the text displayed to the user in the margin of the web page. The `description` column provides alternate or title text for the hyperlink. The column labeled `type` determines the type of link. This can allow us to organize links in more meaningful ways. The `address` column contains the site address of the link. This could be an internal or an external address.

The column `target` provides the `target` property for the link: `_blank` will open the link in a new browser window. We'll use this for offsite links so that users don't forget from where they came. The `section` column, of course, sets the pages on which the links will display; `sequence` is for their ordering. Finally, `status` will allow the site administrator to disable a link temporarily.

Once we load our `links` table with the data for linking each section together and for allowing the user to reach each page within a section, we can set up our CGI scripts to retrieve the data:

```
my $main_links = "SELECT name, address, target
                  FROM links
                  WHERE type='section' AND status='ac'
                  ORDER BY sequence";

my $section_links = "SELECT name, address, target
                     FROM links
                     WHERE type='sub-section' AND section=?
                     AND status='ac'
                     OR type='all' AND status='ac'
                     ORDER BY sequence";

my $offsite_links = "SELECT name, address, target
                     FROM links
                     WHERE type='offsite' AND section='$section'
                     AND status='ac'
                     ORDER BY sequence";
```

The first SQL statement retrieves the main links for all active sections of the site. The second SQL statement will retrieve active sub-section links only for the section of the site the user happens to be viewing, based on the value of the variable `$section`. It will also retrieve links for all pages of the site, regardless of the section. The third SQL statement retrieves offsite links for the section. All three of these SQL statements order the rows based on `sequence` number as set by the site administrator.

Of course, beware that interpolating `$section` directly into a SQL statement can leave you vulnerable to SQL injection attacks. Be sure that you use DBI's `quote()` method to escape hazardous characters or, better yet, use DBI placeholders.

In a CGI script, we would perhaps print a "Sections" heading in the left margin and then use the first SQL statement above to retrieve the links to other sections and print them. To do this in Perl, we might have a function that looks like this:

```
sub margin {
    my ($q, $link_kind) = @_;
    my $sql_stmnt;

    if ($link_kind eq 'main') {
        $sql_stmnt = $main_links;
    }
    elsif ($link_kind eq 'section') {
        $sql_stmnt = $section_links;
    }
    else {
        $sql_stmnt = $offsite_links;
    }
```

```
    my $dbh = DBI->connect("DBI:mysql:db1:localhost", $user, $password)
              || die "Could not connect: " . DBI->errstr;

    my $sth = $dbh->prepare($sql_stmnt);
    $sth->execute();

    my $links = $sth->fetchall_arrayref();

    $sth->finish();
    $dbh->disconnect();

    foreach my $row (@$links) {
        my ($name, $address, $target) = @$row;

        print $q->a({ -href => $address, -target=> $target }, $name),
            $q->br, "\n";
    }
}
```

This Perl function requires the use of the CGI Perl module. Call it like this:

```
...

print "Sections", $q->br, "\n";
margin($q, 'main');

print "Links", $q->br, "\n";
margin($q, 'section');
margin($q, 'offsite');

...
```

Each line that calls the function `margin` passes the CGI object (`$q`) and instructions as to which SQL statement to use. The function will pick up the instruction, running it through the `if/elsif/else` statement to determine the SQL statement to use, and storing it in `$sql_stmnt` for its own use. Next, the function connects to the MySQL server, queries the appropriate database with the SQL statement, stores the data in memory (referenced by `$links`), and then disconnects. Finally, it loops through the complex data structure with a `foreach` statement and prints out a hyperlink for each row.

Administrative Forms

We need to develop some online forms so that our non-technical, site-content administrators can modify and add data on their own. I usually set up a separate, password-protected web page with links to, and brief descriptions for, each administrative form. There are several forms that can be created. Since content administrators will require training and learning time, you may want to develop just a few basic forms initially and add more as they learn. Start with a page to list the basic data in the table `postings` and link each record to another form to edit, add, and delete postings.

```
... # Initial set up

my %types = (
    top => 'Page Heading & Paragraph',
    sub => 'Sub-Heading & Paragraphs',
);

my %sections = (
    1000 => 'Main',
    1001 => 'Products',
    1002 => 'Service',
    1003 => 'Shipping',
);

my $sql_stmnt = "SELECT rec_id, post_date,
                 type, heading, section
                 FROM postings
```

```
                    ORDER BY section, type, post_date";

my $dbh = DBI->connect("DBI:mysql:db1:localhost", $user, $password)
          || die "Could not connect: " . DBI->errstr;

my $sth = $dbh->prepare($sql_stmnt);
$sth->execute();

my $results = $sth->fetchall_arrayref();
$sth->finish();
$dbh->disconnect();

... # Start HTML page

foreach my $row (@$results) {
    my ($rec_id,$post_date,$type,$heading,$section) = @$row;
    print
      "$sections{$section} - $types{$type} ",
      $q->a({ href=> "post-edit.cgi?rec_id=$rec_id" }, $heading),
      " ($post_date)", $q->br, "\n";
}

... # End HTML and exit
```

Here again, we're using the *CGI.pm*, which precedes this excerpt. We set up a hash listing our sections of the web site (`%sections`) and another hash for the types of posts (`%types`). Next, we put together our SQL statement to retrieve some key columns from our table. We then connect to MySQL, run the SQL statement, store a reference to the results in the variable `$results`, and then disconnect from MySQL. We loop through the array of rows of data and print each row within a hyperlink that will connect the user to the CGI script *post-edit.cgi* with the `rec_id`. That script will allow the user to edit the data.

This brings us to our form for editing and adding postings.

```
... # Initial set up

my $script = 'post-save.cgi';

print
   $q->header(-type => 'text/html'), "\n",
   $q->start_html, "\n",
   $q->start_form(-method => 'post', -action => $script), "\n",
   $q->h3("WebPage Posting Form"), "\n",

   "Select Section:  ",
   $q->popup_menu(-name => 'section', -values =>[sort keys %sections],
                  -labels =>\%sections),
   $q->br, "\n",

   "Post Type:  ",
   $q->radio_group(-name => 'type', -values =>[sort keys %types],
                   -default => 'sub', -labels =>\%types),
   $q->br, "\n",

   $q->b("Heading:  "),
   $q->textfield(-name => 'heading', -value => $heading, -size => '10'),
   $q->br, "\n",

   $q->b("Body:  "), $q->br,
   $q->textarea(-name => 'post', -value => $post,
                -rows => '25', -columns => '60'),
   $q->br, "\n",

   $q->submit('Save'), "\n",
   $q->end_form,
   $q->end_html;

...
```

If you're not familiar with *CGI.pm* notation, this excerpt especially may look a little cryptic. Basically, we're starting an HTML page and form that will post the data that the user enters into the form here to the proper Perl script (*post-save.cgi*). On the next line, we display the page heading (`h3`). We then instruct the user to select a section of the site from a pull-down list (a pop-up menu) to display the posting that is about to be entered. This list comes from the hash `%sections`, which is not shown in this excerpt. After that, the user is allowed to choose the type of posting. Next, the script provides a text box to enter a heading for the posting and a large multi-line box to enter paragraphs of text to be posted. Incidentally, we'll need to convert double hard-returns entered by the user to HTML paragraph tags. The module `HTML::FromText` can help with this task. Finally, we end with a submit button and end the form and the web page.

There are several other forms to develop. We will need a form for the `links` table as well as forms for other tables that will hold a list of sections, and possibly text, that may appear at the bottom of each page. However, this one example gives you the idea of how to proceed on your own.

Conclusion

Once you embrace the concept that a site's content does not belong in flat HTML pages or buried in CGI scripts, but in accessible MySQL tables, then the process of handing over content administration to other people can begin. Once you finish the conversion of a site so that non-technical administrators can manage it, then the content of the site will improve greatly and quickly, and your involvement will be strictly as a developer, not just a typist. The result will be better web sites, more satisfied co-workers or clients, and higher job satisfaction for you.

Russell Dyer has worked full-time for several years as a free-lance writer of computer articles, primarily on MySQL.

Return to ONLamp.com.