Building a Content Management System

Alexander Shepherd

Abstract

I have built a utility that allows users to edit the content of their web sites. Here I talk about some of the problems encountered and how these were solved. These include how the system decides which features are accessible and which are protected and how to improve the efficiency and speed of the utility. I used various software structures, and made the utility easy to use, while also provided for the extension of this project in the future, by myself or others.

1. First Steps

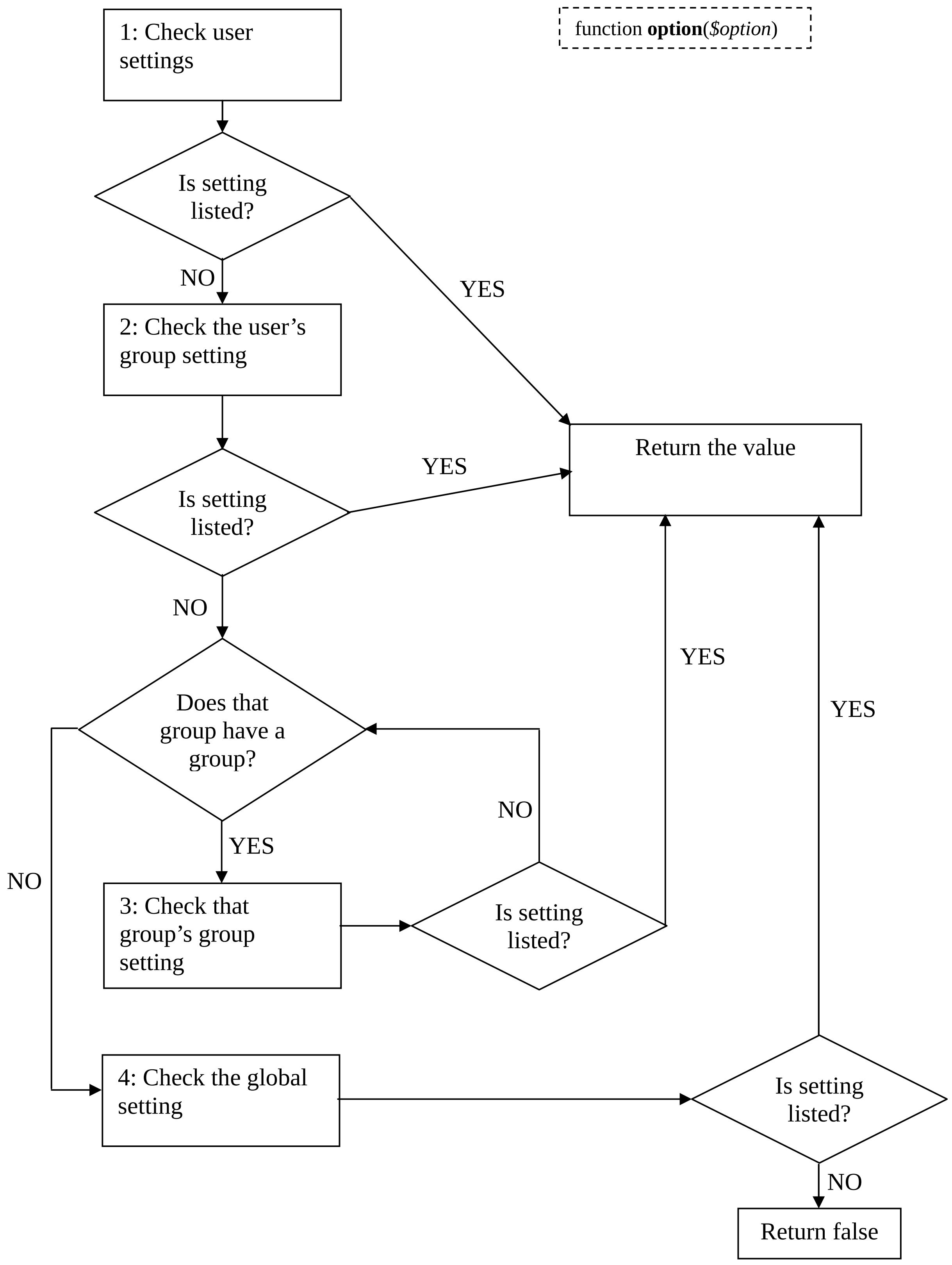
1.1 **A useful challenge** - For my project, I wanted to make something that would be useful to my existing web site clients, that was technically challenging for me and that could be further developed by others.

1.2 **Content Management** - During the past eighteen months, I have been building web sites for various people and organisations. I wanted to give my clients an easy way for them to edit their content, without requiring any technical knowledge on their part, thus avoiding the need to pay a web developer to do it for them. For non-IT-literate people, to safely edit and upload content, it is crucial to protect them from inadvertently damaging the existing structure of the site. My CMS should also ideally be deployable for sites built by anyone – not just me.

2. User Authentication

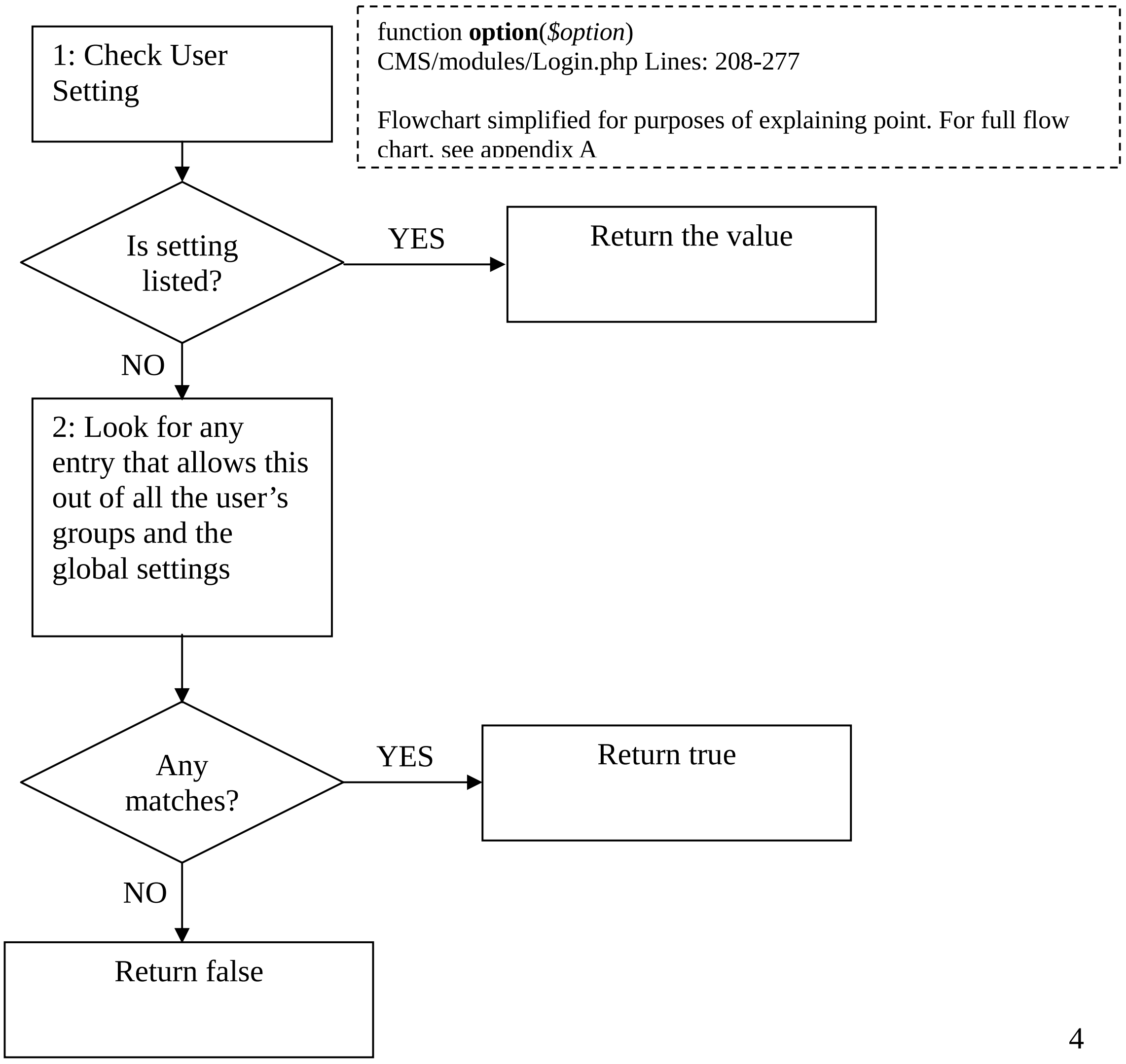
2.1 **Deciding what a user can and cannot do** - To handle various users I built a standard login form, which compares the username and password to ones stored in the database and then remembers who is logged in using the $_SESSION array. However, granting different users different privileges - at a global level, a user level or a user group level - is more complex. Originally, I intended to allow a user to be a member of one group only, with multiple groups allowed to be a member of one parent group, and so on. So the process for checking a user setting (e.g. to allow access to a file) would be as shown:

Fig 1: The option() function:


function **option**(*$option*)
1: Check user settings
Is setting listed?
NO
YES
2: Check the user's group setting
Is setting listed?
YES
Return the value
NO
Does that group have a group?
YES
NO
3: Check that group's group setting
Is setting listed?
NO
YES
4: Check the global setting
Is setting listed?
YES
NO
Return false

2.2 **Introducing more advanced user groups** - However, it soon became apparent that this would not work if users needed to be members of groups that had no relation to each other. For example, I wanted my own user account to be a member of both “Beta Testers” and “Admin Users”, neither of which is a child of the other; I did not want all beta testers to be granted admin privileges nor did I want all admin users to be granted beta tester privileges. So, to give more flexibility and require fewer steps, I adopted the technique of assuming that everything was denied, then taking the most lenient setting available to the user:

Fig 2: The option() function rethought:



function **option**(*$option*)
CMS/modules/Login.php Lines: 208-277
Flowchart simplified for purposes of explaining point. For full flow chart, see appendix A
1: Check User Setting
Is setting listed?
YES
Return the value
NO
2: Look for any entry that allows this out of all the user’s groups and the global settings
Any matches?
YES
Return true
NO
Return false

2.3 **Reducing repetitive code** - Improving the efficiency of this function was my next challenge. For example, if I wanted to check if a user had set the CMS to strip file extensions, assuming the user is a member of the following groups:

“Admin Users” (gid: 1)

“Beta Testers” (gid: 2)

“Non-techie Users” (gid: 5)

the MySQL query that would be generated for step 2 (fig. 2) would be:

```
SELECT * from `cms_options` WHERE (`owner`='GLOBAL' OR
`owner`='g:1' OR `owner`='g:2' OR `owner`='g:5') AND `allowed`=1
AND `option`='stripExtension'
```

(The option that deals with this is “stripExtension”)

In order to generate this string, the computer needs to loop through an array of the user’s groups using the following code:

```
$mysql_query = "SELECT * from `cms_options` WHERE
(`owner`='GLOBAL'";
foreach ($groups as $group) {
      $mysql_query .= " OR `owner`='g:" . $group . "'";
}
$mysql_query .= ")";
```

Simplified from: set_user() CMS/modules/Login.php Lines: 151-173
Full code for set_user() is included in appendix B

When I first built the CMS, the above code was executed every time an option was checked: this worked perfectly well but, given that a large amount of options needed to be checked, this is inefficient. So I moved the above code so that:

a) it was executed as soon as the user logged on and

b) the resulting SQL string was stored in the $USER['allowed_query'] variable. The addition of:

“ AND `allowed`=1 AND `option`=’stripExtension’”

makes the SQL specific to a particular setting lookup.

2.4 **Preventing time-consuming server database requests** - It became clear that some options were often checked more than once; in particular “stripExtension” was checked for every single file that was listed in the screen’s side bar. Given that each check requires one or more SQL queries, I wanted to limit this where ever possible. A first thought would be to simply edit the function that generated the XHTML for the side bar, such that it stored whether or not “stripExtension” was allowed. However, it then occurred to me that other functions were bound to be making the same repetitive, inefficient demands on the system. Furthermore, even if I took the time to ensure that all of my code did not repeat checks, third party developers could not be relied upon to do the same. This can be resolved by adding a global settings array, called $OPTIONS. The first time option(“stripExtension”) is called, the retrieved value is saved to the $OPTIONS array and before performing any more mysql queries, the $OPTIONS array is checked to see if the value has already been obtained.

3. Speed

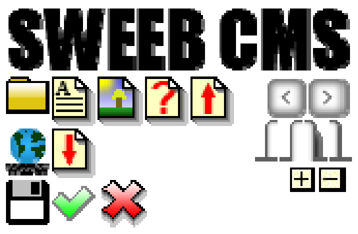
3.1 **Reducing loading time** - Another objective was to improve the speed of the loading of images. It is easy to forget how many images are loaded in a page, as many are used for subtle background effects that the user may not consciously notice as an ‘image’. However, in order for the CMS to render correctly, about 22 images needed to be loaded (3 for tabs, 3 for tabs when clicked, the CMS logo, the plus / dash symbols for folders, the fade for the button bar, the fade for divider between the side bar and main area and all the file / folder / save icons). The delay caused by loading all these images could be noticeable as each image, regardless of size, requires a (relatively) slow HTTP request before it can be displayed on the client’s screen. (Such a delay is evident in Microsoft Office Outlook Web View, which requires about 35 images, most of which often load noticeably slower than the rest of the page). I also wanted to keep in mind that this problem may become even more severe because:

a) The WYSIWYG editor also requires a lot of images.

b) There is a strong possibility that more file icons would be added later (for files like PHP, CSS, JavaScript, DOC, PDF, etc.).

c) Third party developers may want to use images themselves.

This led me to try using CSS sprites (see Appendix C); I had not had experience with this technique before.



CMS/images/main_sprite.png

3.2 **Limiting the amount of data traffic** - To reduce the sending of unnecessary characters, JavaScript and CSS files have been compressed[1, 2]. Compressing the JavaScript reduced its size from 28KB to 15KB (a reduction of 46%), as well as cutting the number of files (and therefore HTTP requests) from three to one. The compression of the CSS trimmed its size from 7KB to 5KB (a reduction of 29%).

4. AJAX

4.1 **Providing a more interactive, multi-dimensional user experience** - Using AJAX offers several improvements to the user interface. Firstly, it avoids the unprofessional look of a page completely reloading for every request, since the blink of white whilst the page loads is sometimes noticeable. Secondly, it allows the contents of the side bar to be independent of the main area (an effect I could only replicate without AJAX by saving a lot of values to either cookies or the $_SESSION variable). Finally, it keeps the sent data to a minimum. The CMS, as opposed to normal “brochure” web sites, requires an almost constant communication between the server and client. To minimise this, the server should send only task specific information and not the whole page for every single request. Server side, the AJAX communication is achieved by the CMS’s functions saving their responses to the $RESPONSE array, instead of directly printing them. The respond() function converts the $RESPONSE array into JSON[3] and prints it. (The contents of the $RESPONSE array are shown in Appendix D).

5. Object Orientation

5.1 **Hiding the true location of files from the client** - To protect the privacy of the web server, the CMS never sends real paths to the client. Instead, JavaScript deals entirely with virtual paths, which replace the site root with the site's name. For example, if we have a site named "The Leys" and its root is at:

```
/var/www/theleys.net/@/htdocs/
```

And we want to deal with its home page:

```
/var/www/theleys.net/@/htdocs/index.php
```

The virtual path that client JavaScript would receive is:

```
The Leys/index.php
```

5.2 **Thinking of a path as an object instead of a string** - Originally, I used two functions to convert virtual paths to real paths and vice versa: get_path() and put_path(). The main drawback to this was that it created confusion between functions that use virtual paths and functions that use real paths. It also meant that paths were converted far more often than necessary, which is inefficient. To solve this, I decided to stop using strings to store paths, and instead build my own path object. The path class contains both the real path and the virtual path, as well as some information about the path, such as:

a) its parent directory,

b) whether or not it is a directory and

c) its file extension.

5.3 **Using the path object to increase security** - The constructor function of the path class also performs checks to ensure it does not contain illegal characters:

> ' < ? * : \ "

It also stops users navigating to areas they should not by using "..". Although important, these security checks are extremely easy for the programmer to forget. Doing them automatically prevents accidental slips in security. The path object also contains a function used to require that the path exists; it will throw an error if not. This is useful to check, before attempting to perform file system tasks, so avoiding PHP's non-JSON error messages.

5. 4 **Using the path object to aid in debugging** - I have also made the path object's real path variable private, forcing developers to do all file system tasks using the path object's own functions. This is useful because all the necessary checks are performed on a file before an action is attempted. For example, it always checks if the file exists before trying to rename it. Performing these checks means that, if something were to fail during a file system task, the developer can find out the source of the problem instead of only knowing that the task failed. Using the path object in this way also limits the amount of information third party developers have about the file system, without limiting the functionality available to them.

5.5 **Creating an extendable system** - Centralising file system tasks also provides a convenient platform on which to develop more advanced file functionality, such as distinguishing between read / write permissions and allowing files appearing in the same virtual folder to be stored in different real folders (useful for building in functionality to allow users to edit files before publishing them to a public web site).

6. Templates

6.1 **Giving users control over content, without compromising page design**

- Since the main focus of the CMS is ease of use for non technically literate users, I used CKEDITOR[4] which allows WYSIWYG editing of HTML files. However, WYSIWYG has two drawbacks for a professional-looking site. Firstly, if the user has the ability to edit the entire page, they may accidently damage the page structure if they hit backspace too many times (a problem I have faced with other whole page editors, such as Microsoft SharePoint's page editor). Secondly, if the site has repeating regions, such as a calendar of events, it is easy for the user (or multiple users) to be inconsistent with formatting. In response to these problems I have included the ability to use templates. When the user requests to edit a file, the editor() function checks to see if it is a template file by searching for "<!--CMS_Template-->". Given that a desired feature of the CMS is for it to create pages that could stand alone, it is important that the template mark-up does not affect the HTML. To achieve this, I used HTML comments in the form:

```
<!--CMS:TAG;ATTRIBUTE;ATTRIBUTE-->
Text that can be edited
<!--CMS:End-->
```

The following tags are understood:

“Edit”	Small WYSIWYG area
“Text”	Textbox
“Date”	A date
“Picture”	Replaceable picture
“Repeat”	Region that takes repeating units
“Repeat_Part”	Repeating unit
“Editable”	Full page WYSIWYG area

The “Editable” tag is used to make entire pages editable except for the top and bottom. For example, for http://ertl.sweeb.net/ the XHTML that opens and closes the page are set, but the content region is all editable.

The other tags would be used for a page such as http://www.theleys.net/calendar/index.html in which news items are a repeating unit, each with a news title textbox and news text as a small WYSIWYG area. So that page’s (simplified) mark-up would be:

```
<html>
<!--CMS_Template-->
<body>
<h1>News</h1>
<!--CMS:Repeat-->
  <!--CMS:Repeat_Part-->
    <b><!--CMS:Text-->Leys gets Green Flag<!--CMS:End--></b>
    <!--CMS:Edit-->
      <p>We'll raise it at Speech Day!</p>
    <!--CMS:End-->
    <hr />
  <!--CMS:End-->
<!--CMS:End-->
</body>
</html>
```

7. Future Development

7.1 **Beta testing** - The site is now currently being beta tested by some of my clients. Over the next few weeks I shall be implementing some of their suggestions and attempting to solve any issues that are raised. Beta testing is an important part of the process of building anything, as the developer will always see the system in a different light to the end user and so will not always be able to appreciate which features are not obvious to use.

7.2 **Improving the home page** - The home page also needs work, as at the moment the text is a constant so content cannot be dynamically added. I would like to develop this to allow plug-ins to add items to the home page, such as a help section, or an interactive walkthrough, helpful for first time, non-computer-literate users of the CMS.

7.3 **Removing all need for a developer** - Although it can be used by people with no experience with HTML or PHP, etc, to set it up requires an administrator with direct access to the database, and knowledge of both MySQL (or at least know how to use phpMyAdmin[5] competently) and the internal running of the CMS. However, this requirement can be overcome by building some settings pages. These will be straight forward for me to program but will enormously improve the usability of the CMS.

8. Conclusion

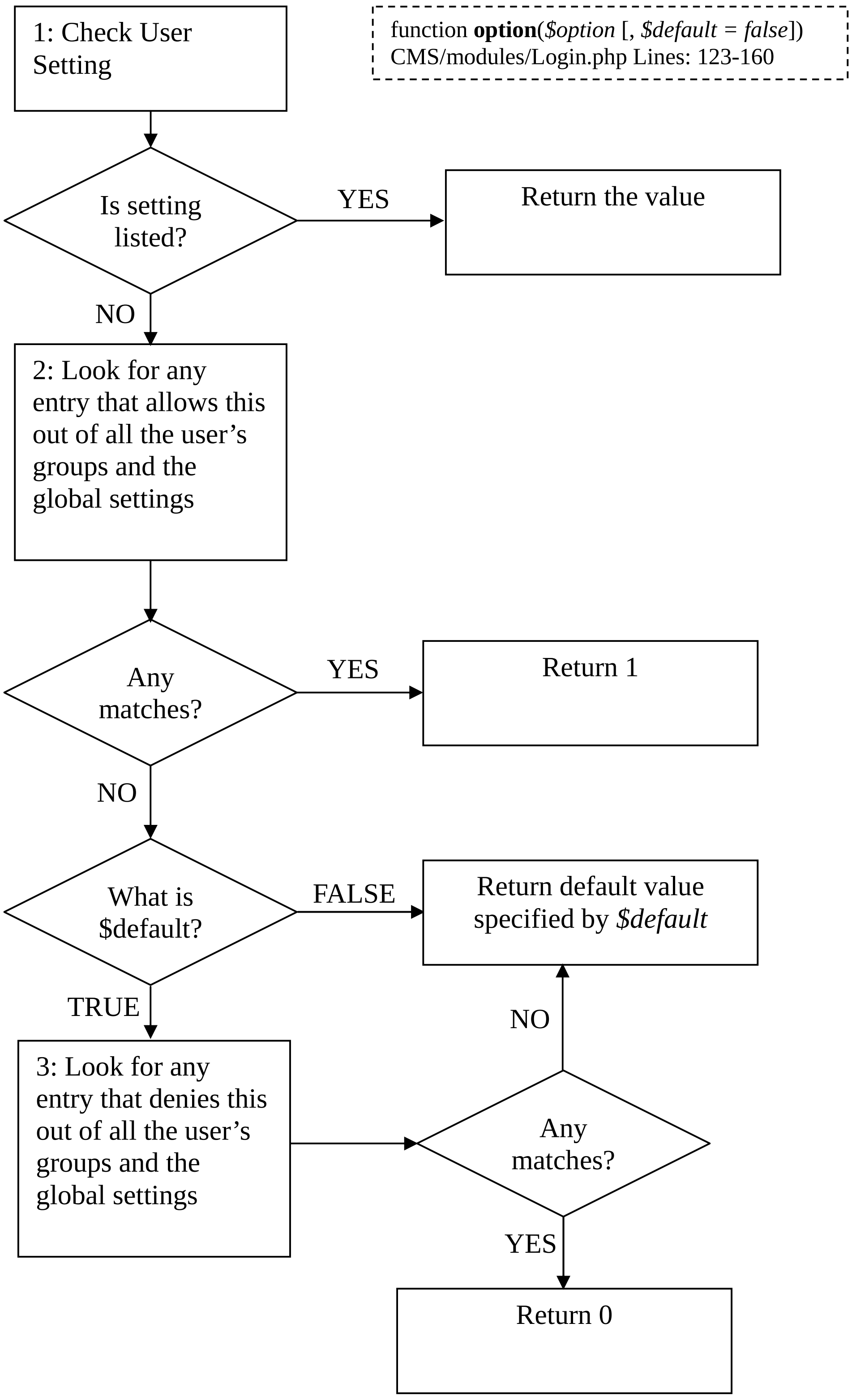
8.1 **Success in a limited time** - I have not spent as much time as I would have liked on this project because I was away in London for six weeks while working as a programmer for Music Jobs Ltd. However, I do believe that I have built a product of worth, which will be of benefit to my clients by allowing them to keep their site's content up to date and professional-looking, without requiring them to hire a web developer to do it for them.

8.2 **Continuous learning** - As with all my software projects, from the simple game I wrote and sold for the Lenten appeal in year 8, to the multi-vehicle stopwatch built for my PDA, I have learned a lot during this project. I am planning to use my new found knowledge of CSS sprites and object orientation for my next project: Easy Ticket, which will build on the ticketing system that I have developed for the school, but expanding it to serve any amateur drama / event group.

Appendix

Appendix A

The full option() function, with explanation:



function **option**(*$option* [, *$default* = *false*])
CMS/modules/Login.php Lines: 123-160
1: Check User Setting
Is setting listed?
YES
Return the value
NO
2: Look for any entry that allows this out of all the user's groups and the global settings
Any matches?
YES
Return 1
NO
What is $default?
FALSE
Return default value specified by *$default*
TRUE
3: Look for any entry that denies this out of all the user's groups and the global settings
Any matches?
NO
YES
Return 0

Step 3 must be added if $default = true because by this point we only know the following:

- The user has no settings regarding this
- None of the user's groups allow it

So, we can only assume that the option is allowed if we also check that none of the user's groups are denying that option.

This function returns mixed values; sometimes it returns true or false (as a Boolean) and sometimes it returns 1 or 0 (as an Integer). The reason for this is that normal logical expressions will work the same for both:

"if ($result == 0)" is the same as "if ($result == false)"

"if ($result == 1)" is the same as "if ($result == true)"

However, when using a more specific logical expression, the difference in type allows the calling function to determine whether the result was retrieved from the database or if it was just the default:

"if ($result === 0)" is not the same as "if ($result === false)"

"if ($result === 1)" is not the same as "if ($result === true)"

"==" checks if they are equal in value whereas "===" checks if they equal in value <u>and</u> type.

For example, the function that checks if the current path is allowed sometimes needs to check the parent directory if the current path is not referred to in the database:

```php
//Check the database for this path
$check = option("File:" . $path->getPath(), true);

//If default value, see if we need to check parent directory
if ($check===true)
{
    if ($up===false)
    {
        return true;
    }
    else
    {
        //If there is no up, the file is denied
        if ($path->pathInfo["dirname"]==".")
        {
            return false;
        }
        else
        {
            //Do this again for the parent directory
            return $this->checkAllowed($path->up());
        }
    }
}
else if ($check===1)
{
    //Database said yes
    return true;
}
else
{
    //Database said no
    return false;
}
```

Section from: Path::checkAllowed() CMS/modules/Path/PathSafety.class.php Lines:86-119

Appendix B

The full set_user() function:

```php
function set_user($mysql_assoc)
{
    GLOBAL $USER;
    $USER = $mysql_assoc;

    $mysql_query = "SELECT * from `" . SQL_PREFIX . "options` "
                 . "WHERE (`owner`='GLOBAL'";

    if ($mysql_assoc["groups"]!='')
    {
        $groups = explode(';', $mysql_assoc["groups"]);
        foreach ($groups as $group)
        {
            $mysql_query .= " OR `owner`='g:" . $group . "'";
        }
    }

    $mysql_query .= ")";

    $USER["groups"] = $groups;
    $USER["allowed_query"] = $mysql_query;

}
```

set_user() CMS/modules/Login.php Lines: 151-173

Appendix C

What are CSS Sprites and how can they be of benefit?:

Normally in (X)HTML, images are included using the image tag:

<img src=”path/to/image.jpg” alt=”” />

However, loading a lot of images can take time because an individual HTTP request has to be made for each one. As HTTP requests are the slowest part of loading a page, it is common practise to try to minimise them. We can do this by grouping all images into one file, and then taking advantage of CSS's “background-position” property to control which section of the sprite is in view:

```
#my_logo {
        background-image: url(‘path/to/sprite.jpg’);
        background-position: 0px 0px;
        height: 10px;
        width: 10px;
}
#my_other_logo {
        background-image: url(‘path/to/sprite.jpg’);
        background-position: -10px 0px;
        height: 10px;
        width: 10px;
}
```

This will style two elements such that “my_logo” looks through a 10px by 10px ‘window’ at the first half of the sprite and “my_other_logo” looks through a 10px by 10px window at the second half of the sprite when the background is shifted to the left by 10px with “`background-position: -10px 0px;`”. This reduces the number of HTTP requests because the browser needs to request only one image (sprite.jpg) instead of two (my_logo.jpg and my_other_logo.jpg)

Appendix D

The format of the $RESPONSE array, which is turned into a JSON array before being sent from the server to the client:

“Text”	The XHTML to be displayed
“Element”	The element to put the XHTML in
“Append”	If set to true, the XHMTL will be added to the end of the element. If false, all current text in the element is replaced
“Code”	A status code; at the moment this almost always is set to 200, the HTTP status code for “OK”, regardless of whether or not the operation is successful
“Status”	Human readable form of “Code”
“Script”	JavaScript to run
“nextCall”	More JavaScript; used when the response asks JavaScript to perform two more AJAX requests (something I aim to phase out to reduce unnecessary HTTP requests) in order to space them out, as two requests cannot be sent at the same time

Appendix E

Notes about the CMS' compatibility and reliability:

I have built the CMS to be W3C[6] compliant:

I have tested the CMS on the following browsers:

- Microsoft Internet Explorer 8
- Google Chrome 5
- Safari 5
- Mozilla Firefox 3

This covers over 80% of users. The remaining 20% are almost certain to have a perfect experience because the CMS is W3C compliant. (An exception is browsers that do not support JSON but I am currently working to fix this).

These are two practises that unfortunately many web developers do not do, although they most definitely should. Code that is W3C compliant is code that is correct. Non-compliant code may render correctly, but developers can only guarantee this if the site has been tested on all browsers. This is sloppy habit to get into.

Testing on a wide range of browsers is also important to ensure all users have a satisfactory experience. If developers only test on Internet Explorer, for example, they may produce code that is dependant on Internet Explorer's incorrect HTML rendering, which displays poorly on other W3C compliant browsers. This is a serious flaw because Internet Explorer currently has about 50% of the browser share (and dropping), which means other users could have a sub-standard experience.

Valid XHTML 1.0 Transitional:

http://validator.w3.org/check?uri=http%3A%2F%2Fcms.sweeb.net%2F

Valid CSS 2.0:

http://jigsaw.w3.org/css-validator/validator?uri=
http%3A%2F%2Fcms.sweeb.net%2F

References

1: JavaScript compression by ckpackager:

http://svn.fckeditor.net/CKPackager/trunk/

2: CSS compression by my own PHP / CSS compression tool:

http://cms.sweeb.net/Compressor/

3: JSON - JavaScript Object Notation:

http://www.json.org/

4: CKEDITOR – JavaScript WYSIWYG editor:

http://www.ckeditor.com/

5: phpMyAdmin - Tool for administrating databases. Used by most developers:

http://www.phpmyadmin.net/

6: W3C - World Wide Web Consortium

http://www.w3.org/

Links

PHP reference:

http://www.php.net

Wordpress – A popular CMS:

http://www.wordpress.org/

Joomla! – Another CMS. I looked at this for inspiration on template mark-up:

http://www.joomla.org/

Drupal – Another CMS. I looked at this for ideas about plug-ins:

http://drupal.org/

MySQL - Database:

http://www.mysql.com/

W3C (X)HTML validator:

http://validator.w3.org/

W3C CSS validator:

http://jigsaw.w3.org/css-validator/