

SCMS - A Static Content Management System

Richard Whitty

September 2008

1 Reasons for Development

1.1 Aims

On starting this project, I had a good idea of what it should accomplish, namely (in no particular order):

- Require no additional server-side processing, so the created sites are not limited to hosting which supports specific scripting mechanisms.
- Provide an easily updated blog and project listing.
- Not be limited to a certain type of site.
- Provide as much functionality as possible to mimic server-side content management systems.
- Be extensible.
- Allow the generation of standards-compliant XHTML.
- Not be dependant on client side scripting (e.g. DHTML/Javascript) due to my personal dislike for pages based on these technologies. Nearly anything you should need to do is possible with plain CSS and XHTML.

2 Introduction

This piece of software allows you to create professional-quality web sites with most of the advantages of a dynamically generated site, however it produces a set of plain XHTML documents which can be uploaded to any server and do not require server-side scripting (e.g. PHP, ASP) or database support.

This has the advantage of reducing load on the server (because it does not need to run a program for every page view), hence making your site faster and more responsive. It also means that you can easily test the site off-line without having to worry about setting up a test database/web server.

All formatting is controlled by CSS, which are user-provided, however I offer a number of sample 'themes' to act as a starting point.

There are a number of disadvantages, such as the inability of users to directly comment on news/blog posts or create accounts. However for many sites this functionality is not required, but forced on them by their use of a full-featured content management system.

This software is designed for people who know how to use XHTML and CSS to get what they want, but make it easier to do so, and facilitate reuse of markup.

3 Page Types

There are currently four types of page which can be incorporated into web sites. These are abstracted away from the core functionality so adding more is relatively straightforward.

plainpage A plain page with text. These tend to make up the bulk of a typical site.

blog a blog-like news feed, which provides an ATOM feed and an archive of posts.

projects a page class created for my own web site, which provides information about any number of projects. It is geared towards software based items, but does not have to be used this way.

links a set of links to either internal or external sites, with optional icon and description.

Using even plainpage on it's own, it is possible to construct quality web sites (example <http://www.safetyassessment.co.uk>) that cater to the needs of the individual client.

4 Creating a Site

The first step to creating a site is to get hold of the CMS source package. You can find it at <http://www.csc.liv.ac.uk/~cs6rlw/projects.html#scms> (or will be able to when the login server comes online).

Next, untar the package into the directory you are going to use for building your web site. Edit the file CONFIG and set the relevant options.

You can create sections as described under subsequent sections. Once you have done so (or after making any changes to the source material) you must rebuild the site.

```
$ source cms/maker
$ make_site
```

This will load the CMS' functions and create the XHTML files in the current directory. All required media for the site should be listed in assets.txt (e.g. images, additional pages, source code).

You can check this by running the function `make_dist`, which will create a directory 'dist' that should contain everything required by the site. If there is something missing that is in the list of assets, you will get an error message and the process will halt. To correct this, locate the relevant file or files and place them in the current directory. Repeat.

Once this is complete you can verify that all required assets are included by viewing the site with your browser of choice (either copy to a local web server, or just point the browser at the 'dist' directory).

5 Global Configuration

There are a number of global configuration variables that need to be set.

DESTINATION the path to sync to as gets passed to scp.

PROJECT_DIRECTORY the directory containing the project subdirectories.

CSS_FILE The stylesheet to use.

BLOG_SHOW_POSTS the number of blog posts to show on the main blog page, any others are relegated to the archives.

FIRST_NAME first name of the site's owner.

LAST_NAME last name of the site's owner.

EMAIL_ADDRESS contact email address for the site.

SITE_TITLE the title of the site.

SECTION_DIR the directory containing the section definitions for the site.

6 Adding Pages

Each page is given a directory in the `$SECTION_DIR` directory. This contains the files which define how the page should behave. If a link to the page should appear in the site navigation part of the site, then the name of the directory should be added to the file `section_list` (in the position you would like it to appear in the list).

While each class or page provides different features, there is some common functionality:

- Text files are run through a paragraphing function. This means that all the text will be enclosed in `<p>` tags, starting a new paragraph on a blank line. If you need to include other things, such as lists or preformatted segments, then it will not generate valid XHTML (but will still work in all the browsers I've tried).
- For these situations many of these filenames can be suffixed with `_raw`, in which case the contents of the file will simply be output as-is to the relevant part of the page - without being run through the paragrapher. This means you need to provide your own paragraphing or formatting tags. `_raw` files take precedence over plain ones.
- Each section has a directory in `SECTION_DIR`. This should contain a file 'type' which in turn contains one of the page class names, nothing else.

6.1 Plain page

content/content_raw what to display on the page.

filename what the resulting page should be called (for links etc)

header what to put in the `<title></title>` tag and in a header on the page.

6.2 Blog/news page

Each post gets a subdirectory containing these files:

content/content_raw the actual post.

link the link to attach to the title. (optional)

name the title of the post

summary the summary or subtitle of the post for the atom feed. (optional)

You should also place a file 'header' in the section directory containing the blurb for the top of the page. Each individual post is contained within a "blogpost" `<div>` section, so you can format them in a specific way if desired.

Additionally an atom feed is generated in the site root with the filename being the name of the blog with the extension '.atom'.

6.3 Project listing

Each project gets a subdirectory in the section, containing the following files:

brief_description short blurb about the project. Should adequately explain the purpose or function of the project.

long_description the rest of the description.

short_name the name of the directory (todo: is this actually needed then?).

long_name full name of the project.

link_paper the url of the paper about this project. (optional)

link_source URL of source code download. (optional)

README the README for the project - not just a link but the actual file. (optional)

image_url URL for the screenshot/logo of the project. (optional)

Additionally, there should be a file header in the section directory containing the blurb for the top of the page.

6.4 Links page

The site can display a number of links in the sidebar. Each gets a subdirectory in 'links' containing the following files:

name The name of the link (or the alt text for the image)

link the URL to link to.

icon path to image to display for the link, the name is then used as the alt/title text. (optional)

description longer blurb that is featured on the links page, but not the menu.

7 Controlling the Appearance of your Site

You should use CSS to control the display of the pages. The overall layout is divided using `<div>` tags. The relationship between id and content is given below:

header This is the content of `site_header`. Typically displayed at the top of the page.

footer This is the content of `site_footer`. Typically displayed at the bottom of each page.

navsite This is the navigation menu. A (possibly nested) unordered list of hyperlinks.

content This is the main content pane. The various page types control what is displayed here.

8 Technical Information

8.1 Implementation

GNU Bash was chosen as the implementation language, because of it's practical ubiquity. Even if it is not on a system by default (e.g *BSD, Solaris) it is either in the ports tree or easily built by hand.

The rest of the tools used are standard POSIX tools (`cat`, `stat`, `date`), or `bash-builtins`.

The main module is called 'maker'. This houses both common functionality and the functions to generate and publish the site. It is relatively well commented.

The plugin interface consists of two functions which are expected to be available in each of the modules:

make_relevant_pages this creates all the pages related to the section. It is assumed that it will also add them (and any other required files) as requisites.

do_menu_entry A section is responsible for it's own menu item. This is so complex pages may have more than one entry on the page, possibly pointing to anchors (as in the projects module), or possibly to add an extra page (like the blog archive).

Both are passed a single parameter, which is the name of the directory in `SECTION_DIR` that the section resides. Modules are advised not to clobber the core functions, though it is possible (it is conceivable you may want to do this, however it is likely to be a bad idea).

8.2 Known Problems

Here is a list of problems with the CMS or the behaviour of generated sites, either in a specific browser or in general.

8.2.1 Internet Explorer indents the first item of some lists

Don't know how to work around this, searching the internet brought up some nasty CSS workarounds (not an option as CSS is user-provided) or javascript-based solutions (not an option because of the original design spec). It would seem to be a bug in Internet Explorer. (Tested versions: 6)

8.3 Workarounds

Although I would generally discourage the use of workarounds for specific browsers, the reality of the web is such that most people will have just one browser they use. They also expect any site to work as intended. As such there are some methods I was required to employ in order to make this a reality.

Workarounds are only an option when they do not cause the resulting page to deviate from the relevant standards, or are likely to cause maintenance issues in future.

8.3.1 Gecko and self closing tags

Gecko-based browsers (Seamonkey, Firefox, Epiphany) do not treat self-closing html tags correctly. This would appear to be due to how they rewrite the HTML when parsing it. For example the anchor tag

```
<a name="top" />
```

is treated as though it remains open until a corresponding `` is found (for example after the first link following the anchor). This causes problems with the W3C core styles, among others.

As a result, this is worked around by separating the closing tag for such elements, so for the example above, the markup produced by the CMS would be:

```
<a name="top"> </a>
```