SCMS Manual

Jim Finnis

January 2010 -

Contents

1 Introduction

SCMS was designed as a system for creating simple websites of a few, mostly static pages, although it can be used for larger systems. Most large CMSs use an SQL database to store their data, but the main rationale for SCMS is that it requires no such database — all the data is contained in files. This means that it can be developed and maintained on a test server, rather than on the live site, with files being transferred to the live site using a revision control system such as Subversion.

SCMS is a self-contained PHP system, requiring no other installable modules. Like most CMSs, SCMS uses a template system so that many pages can be made up of the same HTML and CSS elements, with only the actual content changing. In addition, SCMS handles multilingual sites elegantly: there is a default set of data for each template and page which can be overriden by data in a language subdirectory.

2 Building a Site

This chapter outlines the process of building a site from scratch, and gives references to other chapters to help you.

2.1 Building the starting site

Your artist should have produced a static HTML site for you, consisting of one HTML page with some placeholder content. Our first step is to turn that into an SCMS site, using the static HTML as a template.

2.1.1 Install SCMS

Naturally, if you haven't installed SCMS you'll need to do that — typically into a directory *outside* the public HTML directory, which you then create links to. This allows several sites to share the same SCMS installation. This is described on page 19.

2.1.2 Create the directories for your site

You'll also need to create the public HTML directories for each site you're going to set up. This is described more fully in section 3.1, but it basically consists of:

- creating the public HTML directory if it's not already there — usually the hosting service sets one up — from now on I'll refer to this as the *root directory*;
- creating a link to the SCMS `core` directory inside the root directory;
- creating a link to `core/idx.php` named `index.php` inside the root directory;
- running `scms/createdirs` from inside the root directory;

- copying `scms/default_config.php` into the `site` directory and renaming it `config.php`;
- editing `config.php` as described on page 20 (chances are you won't need to do anything;)
- copying the language files required into the the `site/languages` directory. These can be found in `NLSfiles` in the SCMS distribution. Note that if you ever add a new language, you'll have to clear the cache by running the `clearcache` script in the SCMS directory, from your root directory.

You'll now have the following things in your root directory:

- `site` is where your site's content lives;
- `site/languages` contains the definition files for the languages your site understands;
- `site/pages` contains files for each individual page;
- `site/templates` contains a directory for each template your site uses;
- `site/global` contains global tag definitions (often this is empty;)
- `tmp` is a temporary directory;
- `tmp/cache` contains server-side caching data;
- `config.php` is the configuration file.

2.1.3 Creating the site template

Your web designer should have created a static HTML and CSS page which you can use as a template, consisting of the elements common to all sites with some placeholder content. This should consist of a single HTML file, one or more CSS files, some images, and maybe some Javascript files.

Copy these files into a `templates/default` directory (which you'll need to create,) renaming the HTML file as `main.html`.

Edit `main.html` so that the names of all external files — images, CSS and Javascript — are changed to

```
{{templateroot}}myfilename.ext
```

The `templateroot` tag will be changed to the root directory of the current template, i.e. `templates/default/`. Don't worry about the placeholder content yet.

2.1.4 Creating a dummy page and navigation file

Create a file called `home` in the `site/pages` subdirectory, with the following content:

```
name=The Home Page
```

Now create a file called `navigation` in the `site` directory, with a list of our pages. Naturally, it's just

```
home
```

for now. The process of installing SCMS and setting up a site, along with the structure of the site and how the URLs work, is described in more detail in the overview, section 3.1.

2.1.5 Test it!

You should now be available, and you should see the site your artist designed.

2.2 Templating

The trick now is to add different content for each page using templating. As an example, edit the `main.html` file changing all occurrences of the page's title to the string

```
{{page:name}}
```

Now when you load your page, you'll see that string is changed to

```
The Home Page
```

which is the value of the `page:name` tag for that page, as set in the page's file.

What you can do now is replace the placeholder content in `main.html` with `{{page:content}}`, and then create some content inside `site/pages/home` using a multi-line tag definition:

```
name=The Home Page
content=[[
    <p>This is the home page. Please put something in here
    which makes more sense.</p>
]]
```

2.2.1 Template dumping for debugging

For fun, change the definition of your `home` page to this:

```
name=The Home Page
content={{dump}}
```

If you reload the page, you'll get a full listing of all the tags currently defined[1]. Most of these will be internal to SCMS, but two will be yours — scan down to see them. Each entry in the list shows:

- the tag name,
- the tag type (typically text or a function, with the name of the PHP function which performs the tag if the latter),
- an optional comment,
- where the tag is defined: `(internal)` if SCMS defines it, or a tag definition file name if you've defined it in your site.

2.2.2 Function tags

As noted above, some tags are functions. When using a function tag, the arguments are separated from each other, and from the tag name, using the vertical bar character |. Try changing your `home` page file to this:

```
name=The Home Page
content=[[
    Your wibble tag was : {{httpgetparam|wibble}}
]]
```

Now, if you append `?wibble=foo` to your browser URL when you load your site, you'll get a page telling you what the value of the `wibble` HTTP GET value was. We can also check that a value was set:

[1]assuming your template's CSS can handle wide tables well!

```
name=The Home Page
content=[[
    Your wibble tag was : {{ifnotempty|{{httpgetparam|wibble}}|
                                {{httpgetparam|wibble}}|
                                <i>(not defined)</i>
                          }}
]]
```

Here, the `ifnotempty` tag takes three arguments — the first is the string we're checking, the second is the string to return if the first argument is longer than zero length, and the third is the string to return otherwise. It should be obvious what the code does and how it works. Note that the spacing and newlines aren't important, since they'll expand to HTML spaces which are generally ignored.

There's a lot more about templating in chapter 4 on page 25, which describes how the Template Definition Language works in detail. Also, chapter 5 on page 35 will explain more about the structure of template directories, and chapter 6 on page 39 will explain how pages work.

We can now create several pages by adding entries for each page to the `navigation` file and creating new page files inside `site/pages`, each of which has the same name as the entry in the navigation file and defines the `page:` tags expected by the template in a different way.

2.2.3 Page URLS

The URLs for these new non-default pages are in the following form:

```
http://mysite.blah.com/index.php/pagename
```

In other words, the page name is bolted onto the URL after `index.php`. We could write `A HREF` tags by hand to create intra-site links, but there is a better way.

2.2.4 Links between pages

To create links between pages, you can use the `url` tag to generate a full URL for a given page. For example, a link to a page called `misc` can be created by:

```
<a href="{{url|misc}}">Miscellaneous</a>
```

2.3 Navigation

This still isn't ideal – we don't want to have to create separate navigation menus on every page. Instead, we use the templating system's complex collection handling to create and render a navigation menu according to our requirements.

Replace your home page with this code:

```
name=The Home Page

menu=[[
    {{treeprefix|default|<ul>}}
    {{treesuffix|default|</ul>}}
    {{treeunselnode|default|<li>
        <a href="{{a:url}}">{{a:name}}</a>|</li>}}
    {{treeselnode|default|<li>{{a:name}}|</li>}}
    {{marktree|{{navtree}}|spec|{{spec}}}}
    {{rendertree|{{navtree}}|a}}
]]

content=[[
{{page:menu}}
<p>My test content</p>
]]
```

This is the simplest possible navigation menu, which renders as a simple list of items. All pages are rendered as links, apart from the current one. Don't worry about how it works internally — all it does is set up a bunch of templates which `rendertree` uses to draw a tree of page navigation nodes.

What you should notice is that we've set up a `menu` tag inside the page, and invoked it from the page's content using `page:menu`. All tags defined in page files are prefixed automatically with `page:`.

2.3.1 Putting the menu definition in the template

We don't want to have to use this code for every page we create — it should really be part of the template. We can do this by adding a new *tag file* to the `templates/default` directory. Let's call it `menu.tags` and give it the following content:

```
menu=[[
    {{treeprefix|default|<ul>}}
```

```
        {{treesuffix|default|</ul>}}
        {{treeunselnode|default|<li>
            <a href="{{a:url}}">{{a:name}}</a>|</li>}}
        {{treeselnode|default|<li>{{a:name}}|</li>}}
        {{marktree|{{navtree}}|spec|{{spec}}}}
        {{rendertree|{{navtree}}|a}}
    ]]
```

In other words, just the menu code. Any file in the current template's directory which ends with `.tags` will automatically get loaded and its tags defined.

We can now replace our page file's content with:

```
name=The Home Page

content=[[
{{template:menu}}
<p>My test content</p>
]]
```

Note that every tag defined in a template's `.tags` file is prefixed with `template:`.

2.3.2 Putting the whole menu in the template

We can do even better than this, of course. What we really should do is put `{{template:menu}}` somewhere inside our `main.html` and remove all references to the menu from our page files. That way, we define the menu once and never have to worry about it again.

Your web designer will probably have put some kind of navigation menu into his static HTML file — you'll need to work out how to convert this into a menu definition like the one above, which can be quite complex. Chapter 9 will give you some help with this.

2.4 More!

There's a lot more in SCMS, including:

- hierarchical navigation trees (see chapter 9,)
- multiple templates for different pages (just add a `template=` line to your page file, see chapter 6,)

- multilingual support (see chapter 11,)
- complex collection handling (see chapters 7 and 8) — you can use this in modules to create lists or hierarchies of nodes for your page to render the same way as navigations are rendered;
- a standard system for adding new tags and GET and POST handlers (see chapter 12,)
- a 404 handling system (see chapter 14,)
- styles for different devices (see chapter 10,)
- site caching, so the system doesn't regenerate a page's content unless it's been changed (see chapter 15.)

3 Overview

This is the *real* documentation, going into a lot of detail. Please read the previous chapter first.

3.1 Installing SCMS

You will need the following:

- Some degree of knowledge of PHP, but not too much.
- A server running PHP5 (I'm using 5.2 to test with).
- Shell access on that server.

And here are the steps to follow:

- First, check SCMS out of the repository, or unpack the SCMS archive, into a directory called `scms` on your server, but not your public HTML directory.
- Link the `core` directory into your public HTML directory, where you want your new site to be. If you can't link, it's OK to copy it, but make sure you copy the whole core directory structure (`cp -R` is your friend.)
- Link the `core/idx.php` file to `index.php` in your public HTML directory. Again, it's OK to copy if you have an operating system that doesn't do easy linking.
- While in the destination directory, run the `createdirs` script to create the temporary directories and set the correct permissions. You may find the permissions somewhat over-permissive; feel free to modify them — SCMS must be able to create and write files in the `tmp` and `tmp/cache` directories.

- Copy (not link) the `default_config.php` directory into the new directory `site` inside your destination directory, and call it `config.php`.
- Link or copy the files for the languages you require from `NLSFiles`[1] into the new `site/languages` directory. For example, if you need English and Welsh, use `en_US.nls.php` and `cy_GB.nls.php`. If your language isn't included, it's easy to create a new NLS file — see the files that are there for examples. Note that if you ever add a new language, you'll have to clear the cache by running the `clearcache` script in the SCMS directory, from your root directory.
- Copy or link at least the `default` template directory from the sample templates into the new `site/templates` directory. The default template is minimal, you'll be either editing it or replacing it with another template.
- Create a default home page inside `site/pages`. You can copy the `testhome` file as a starting point.
- Create a `site/navigation` file indexing this page. This can just consist of the word `home` on a line by itself, indicating that the site has one page, called 'home.'
- Modify the `site/config.php` file you copied earlier. See below for notes on how to modify it.
- You may wish to create a 404 template and page, and add modules, but we'll deal with those later. At this point your site should work, showing the default page successfully.
- Now you can build the rest of your site by modifying or creating new templates and pages.

3.1.1 Modifying the config.php file

This is a PHP file containing several values you may need to change, though many sites won't require it to start with. Later on, you will need to change the caching data:

[1] It stands for National/Native Language System

- **ROOTPATH** gives the absolute path of the root directory of the site in the server's filesystem — i.e. where your index.php, site and core directories are. It's generated automatically from server variables (i.e. those in `$_SERVER[]`) but you can set it by hand if this fails for some reason.

- **ROOTURL** gives the root URL of the site, **without** the index.php. It's generated automatically from server variables, but again you can set it by hand.

- **DEFAULTLANG** is the code for the default language. You must have the corresponding NLS file in your `site/languages` directory. For example, if you set DEFAULTLANG to `en_GB`, you must have `en_GB.nls.php` in there. By default, it's set to `en_US`: US English.

- **DEFAULTPAGE** is the name of the default page to use when no pagename is specified. By default, it's 'home.'

- **CACHING** is used to enable page caching. Set it to 0 when testing and 1 when your site is ready.

- **BROWSERCACHING**, when turned on, makes sure most pages are cached on the browser. See chapter 15 for details. Set to 0 when testing and 1 when your site is ready.

- **$cache_exceptions** is an array of strings, giving the names of pages which should not be cached, either on the server or in the browser. It's empty by default, but you should add to it the names of pages which provide interactive content.

Certain modules may add variables to the configuration file — see each module you use for documentation.

3.2 The SCMS directory

Here's the top level of a typical SCMS directory. It's this directory which will contain the `index.php` file, through which all SCMS pages are accessed. This is usually a link to `core/idx.php`. Those entries in *`italics`* are directories, the rest are files.

```
clearcache   core   index.php
modules      site
```

- **clearcache** is a short script to clear the page cache and temporary files. You should run this after modifying the navigation or language data (see below).
- **core** is a directory containing the core files of SCMS. Usually, it's actually a link to a shared directory for all sites managed on this server.
- **index.php** is the main SCMS PHP source file. You shouldn't need to alter it after installation. In fact, it's usually a link to a file in the *core* directory.
- **modules** is a directory containing all the modules available with SCMS by default. In general, you link to modules you require in this directory (see below). Again, it's often just a link into the standard SCMS distribution (but see section 3.3.1.)
- **site** is a directory which contains all the information specific to your site. This is the only directory in which you should need to change anything.

3.3 The site directory

This directory, automatically created along with its subdirectories with the `createdirs` script, is where you create your site files. It's the only directory whose contents you modify after the initial installation. It looks like this:

```
config.php  global      languages
modules     navigation  pages
templates
```

This breaks down thus:

- `config.php` is the site configuration file, which you must modify.
- `global` contains global tag definition files[2] which are loaded for all pages, and which define tags which can be used in all templates and pages.
- `languages` contains the NLS files for each language the site can use.
- `modules` contains a directory for each module which can be used in the site. These are typically copied or linked from the installation `modules` directory (but see section 3.3.1.)
- `navigation` is a file describing the structure of the site as a whole: how the pages are related to each other. It's used to generate the navigation menus.
- `pages` contains a single tag definition file for each page in the site. Each file has the same name as the page name as it appears in the URL.
- `templates` contains a directory for each template. Each page either specifies the template it uses, or the `default` template is used.

3.3.1 Alternate module structure

If a site only uses standard modules, there may be no `modules` directory at the top level. Instead, `site/modules` may link directly to the SCMS distribution's `modules` directory, which contains all the standard modules.

[2]files which define tags which are expanded by the templating system — see below.

3.4 Anatomy of an SCMS URL

The URL of page on an SCMS site is divided into two parts: the **root URL** and the **page specifier**. The root URL is simply the URL of the site as a whole, up to 'index.php.' for example

```
http://foo.com/index.php
```

or

```
http://foo.com/blog/index.php
```

Note that `index.php` cannot be omitted unless you're accessing the home page (the first page on the top level of the navigation file.)[3]

The **page specifier** is the rest of the URL after `index.php`. It consists of the page name and any GET query details: `home` or `prods/chair?q=hello&f=bar`. You can have slashes in the page specifier, and this will imply a directory structure in your pages directory (see below.) If there is no page specifier, the default page (set in `config.php`) is assumed.

3.4.1 The canonical URL

Most links created on other sites will be in the form given above, with or without the `index.php` in the case of the home page. However, an internal link within the site will be in the *canonical form*, which incorporates the language code. This is the same as the normal URL, but with the language code between the root URL and the page specifier. For example `http://foo.com/index.php/en_GB/home`.

3.4.2 Example

Here's an example of a SCMS URL, and a canonical form of the same URL, given that the default language is US English:

```
http://foo.com/index.php/rooms/room1
http://foo.com/index.php/en_US/rooms/room1
```

This means SCMS should serve the page whose specifier is `rooms/room1`.

[3]Note, however, that the `ROOTURL` value defined or generated in `config.php` is the URL *without* the `index.php`

4 Templates and Tags

4.1 Introduction

SCMS uses a template system to produce its output. In such a system, you create an HTML *template* — a block of HTML with some sections replaced by *tags* — and SCMS will replace these tags with their value. For example, if we had the template:

```
<html>
<body>
    <h1>{{title}}</h1>
    {{content}}
</body
</html>
```

and the following tag definition file:

```
title=Home Page
content=[[
This is some example content. It's just placeholder text
for now.
]]
```

SCMS would give the following output:

```
<html>
<body>
    <h1>Home Page</h1>
        This is some example content. It's just placeholder text
for now.

</body>
</html>
```

As you can see, each tag (marked by double curly brackets) has been replaced with the value specified in the tag definition file for that tag. You can infer most of the syntax of the definition file, but I'll describe it below.

The whole of SCMS is based on this system. Many tags are defined by the system, for example {{`root`}} which gives the root URL. Some tags are functions: rather than just replacing with a fixed string, they run a function and use the output as a replacement text. It's a very powerful system, as you will see.

4.2 Tag Definition Language

Most of the files you create will be written in *tag definition language*, a sample of which can be seen above. Here's a slightly more detailed description of the language, although I'm not going to discuss things like collections — for details on those, see that chapter.

4.2.1 Single line definitions

A line containing an equals sign will define a tag whose name is on the left of the sign, and whose value is on the right of the sign. Whitespace on either side of the entire line and around the equals sign will be stripped before processing.

```
name=value of the tag
```

defines the tag {{`name`}} has having the value 'value of the tag' as do the definitions:

```
name  = value of the tag
      name=value of the tag
                 name =    value of the tag
```

4.2.2 Multiline definitions

If the right of the sign consists of the string '[[', the tag has a multiline value which is terminated by the string ']]'. Anything on the same line after the terminating string will be ignored:

```
name = [[This a multiline definition. Everything up to the
double square bracket is the tag value]] but this is ignored.
```

Note that a definition like

```
name =
[[ This just won't work,
Sorry.]]
```

won't work — the opening '[[' has to be on the same line as the equals sign.

4.2.3 Ignored lines and comments

Outside a multiline tag definition, a line without an equals sign will be ignored, unless it starts with '!' in which case it is a special command (see below.) All lines starting with '##' will be ignored, even inside a multiline definition[1].

```
name = value
This line is completely ignored.
name2 = [[a multiline definition
of which this line is a part
##but this line is ignored
and this is the last line]]
```

4.2.4 Tags used in tag definitions

Tag definitions can contain other tags, which will be expanded. The process of expansion is repeated until no more tags remain:

```
datestring=The current date is {{date}}
```

A tag cannot, however, contain an reference to itself, even as an argument of a conditional function tag (see below):

[1]It's two hash signs, because a single hash introduces an ID selector in CSS, so it can be quite common at the beginnings of lines.

```
name=Using the tag {{name}} here causes an error.
name={{if|{{testval}}|{{name}}|}} As does this.
```

(See section 4.4 below for details on tags which take arguments, but suffice it to say that the arguments are separated from each other and the name of the tag by vertical bar characters: "|".)

The process of expansion is done when the tag is used to create ('render') the page, *not* when the tag is defined.

4.2.5 Redefining tags

It's perfectly valid to redefine a tag – in fact, system relies on it. The most recently read definition will replace the previous definition. An example:

```
name=An old version
name=A new version, replaces the previous definition
```

4.2.6 Special commands

Special commands appear outside multiline definitions, and start with a '!' character. They take arguments separated from the command name by spaces, and separated from each other by commas..

- **!delim xx,yy** changes the multiline delimiters — '[[' and ']]' by default — to the strings `xx` and `yy`. Note that the new delimiters must also be 2 characters long!

4.2.7 Using the {{ and }} in templates

It's possible to use these characters by escaping them with a backslash – `\{` is replaced by `{` and `\}` is replaced by `}`.

4.2.8 Suppressing 'unknown tag' errors

If a tag is encountered which is not defined, a fatal error usually results. You can avoid this by prepending the tag name with a '@' character. For example, putting `{{@ivegotnolegs}}` into a template will just result in that tag being expanded as the empty string if it is undefined[2].

[2]which is pretty likely, honestly - `ivegotnolegs` is a very odd name for a tag.

4.2.9 Tag names containing tags

Normally, if you try to do something like

```
{{foo{{bar}}|1}}
```

you'll get an error saying that the tag `foo{{bar}}` doesn't exist. This is because the tag name is not passed through the templating system, so it cannot contain tags. This is mainly for speed. However, you can circumvent this and force the tag name to be expanded as a template by preceding it with '!':

```
{{!foo{{bar}}|1}}
```

will work as you expect. You can follow the exclamation mark with '@' if you want to suppress an uknown tag error.

4.3 Tag name prefixes

If you try to use some of the examples above in your home page, you'll find that they won't work. This is because of an added complication to the tag definition process. During the process of tag definition, a prefix is added to each tag's name as it is defined depending on the context in which it was defined.

- If the tag definition file is part of a template (i.e. the file is below the `site/templates` directory) the tag name is prefixed with `template:`
- If it's the template for a page (i.e. it's in the `site/pages` directory) then it's prefixed with `page:`
- If it's a global template (i.e. it's in the `site/global` directory) it's prefixed with `global:`

Only templates defined by the system or by the `set` function tag (see below)[3] can have 'bare' template names, without a colon in them. As an example, if the following tag definition file were for a page:

```
name=My Home Page
content=[[This is my home page. It's very much under construction
at the moment.]]
```

the tags {{`page:name`}} and {{`page:content`}} would be defined.

[3]or by modules, although this is bad practice — see Modules, below.

4.4 Function Tags

As explained above, a tag inside a template is in the form `{{tagname}}`. This will cause the template system to look up the tag and replace it with the tag's value. Tags values can be simple strings such as those you define in a tag definition file, but they can also be functions which return the replacement string.

In this case, the tags can take arguments. Inside a template, the syntax for a function tag which takes arguments is:

```
{{tagname|arg1|arg2|arg3...}}
```

Different tags obviously take different numbers of arguments. In most cases, the arguments are themselves processed by the template system prior to use, but not always. The cases where this isn't done are relatively few, and will be documented under the tag in question.

Here's an example of the sort of expansion that takes place. Consider the following tags:

- **ifnotempty** returns[4] its second argument if its first argument is a string longer than 0 in length. Otherwise it returns its third argument. In short: `{{ifnotempty|string|useiflong|useifempty}}`.
- **httpgetparam** evaluates to the value of `$_GET[argument 0]`, or to the empty string if no such HTTP GET parameter exists.

Now consider this bit of tag definition language:

```
test=[[
 {{ifnotempty|{{httpgetparam|wibble}}|
    There's a wibble, it's {{httpgetparam|wibble}}|
    Sorry, no wibble. Ah well.
}}]]
```

Assuming this is in the page tag definition file, this tag will be defined under the name `page:test`. When expanded, the `ifnotempty` tag will run, reading its arguments in. It will then expand its first argument, which will cause the `httpgetparam` tag to run, returning the value of `$_GET['wibble']`. If this is not zero length, `ifnotempty` will expand its second argument, which will involve

[4]i.e. replaces with

calling `httpgetparam` again to get the GET parameter. This will be returned as the expansion of the `ifnotempty` tag, and hence the whole `page:test` tag. If the GET parameter was zero length or didn't exist, then the third argument to `ifnotempty` will be expanded (which does nothing as it contains no tags) and returned.

Note that the tag arguments can contain large, multilined strings — entire blocks of HTML if you so wish — and can be nested quite deeply inside other tag arguments. It's this side of processing that can slow down SCMS' output, however.

Function tags are very powerful, and SCMS contains an awful lot of them. They're all described in the reference section at the end of this document, along with all the standard string tags.

4.5 Arguments in user defined tags

It's also possible to use arguments in your own tags, defined in Tag Definition Language. You refer to argument *n* within the tag's definition using `{{$n}}` (where argument numbers start at 1.) Here's an example:

```
text = The word {{page:makered|RED}} should appear in red.
makered = <span color="#ff0000">{{$1}}</span>
```

Here, the `page:makered` tag takes its first argument and wraps it in an HTML span which turns it red. When the tag `page:text` is processed, it calls `page:makered` with the word RED, wrapping it in the span.

Note that as each argument is fetched in the parsing of the tag, it is processed through the templating system. For example, in the above example, the string `RED` which is the first argument of the `page:makered` tag is passed through the template processor and all its tag expanded before being stored in tag `{{$1}}`.

4.5.1 Undefined user arguments

Using this system may cause odd problems if you try to access undefined arguments. For example, consider

```
printlist = [[
<ul>
  <li>{{$1}}</li>
  <li>{{$2}}</li>
  <li>{{$3}}</li>
</ul>
```

If you use the following:

```
{{printlist|one|two|three}}
```

everything will work as expected : you'll get a bulleted list of 'one,' 'two' and 'three.' However, if you just do

```
{{printlist|one|two}}
```

the `$3` tag will be left undefined. This will result in one of two things:

- The last user argument of that number defined for any tag, or
- an unknown tag exception if no argument of that number has ever been used.

We could modify the code to make this behaviour more predictable, but it would slow down template processing.

4.6 Using 'set' to set temporary tags

When programming, you may often want to set temporary variables to hold complex expressions. You can do a similar thing in tag definition files using the `set` tag. This takes two arguments: the name of the tag to set, and the value to set it to:

```
{{set|test|{{ifnotempty|{{httpgetparam|wibble}}|
    There's a wibble, it's {{httpgetparam|wibble}}|
    Sorry, no wibble. Ah well.}}}}
```

This sets the tag `test` *without a prefix* to expand to the string given in the second argument. Note that this is stored directly as a string: is is not itself expanded. Consider the following snippet:

```
{{set|temporary|foo}}
{{set|test|{{temporary}}}}
{{set|temporary|bar}}
{{test}}
{{set|temporary|baz}}
{{test}}
```

This will display the string 'bar baz'. This is because the following sequence of events occurs:

- The string `foo` is stored in `temporary`
- The string `{{temporary}}` is stored in `test`
- The string `bar` is stored in `temporary`, overwriting `foo`
- The tag `test` is expanded to `{{temporary}}`
- The string `{{temporary}}` is expanded to `bar` and output
- The string `baz` is stored in `temporary`, overwriting `bar`
- The tag `test` is expanded to `{{temporary}}`
- The string `{{temporary}}` is expanded to `baz` and output

This is good behaviour but it can result in lots of code being executed to expand tags over and over again, such as the following code:

```
{{set|test|{{ifnotempty|{{httpgetparam|wibble}}|
    There's a wibble, it's {{httpgetparam|wibble}}|
    Sorry, no wibble. Ah well.}}}}
{{test}}
{{test}}
{{test}}
{{test}}
{{test}}
```

This would run the `ifnotempty` and associated tags five times, giving the same result. If you know the result of the expansion isn't going to change, you can put the string `p` as a third argument, telling SCMS to `process` the string through the template system before storing it. The snippet

```
{{set|temporary|foo}}
{{set|test|{{temporary}}|p}}
{{set|temporary|bar}}
{{test}}
{{set|temporary|baz}}
{{test}}
```

will display the string 'foo foo' because of the following sequence of events:

- The string `foo` is stored in `temporary`
- The string `{{temporary}}` is expanded to `foo`, which is stored in `test`
- The string `bar` is stored in `temporary`, overwriting `foo`
- The tag `test` is expanded to `foo` and output
- The string `baz` is stored in `temporary`, overwriting `bar`
- The tag `test` is expanded to `foo` and output

4.7 Debugging

There are two main tools available to you during debugging: template dumps and exceptions. When an exception occurs, you'll get both a PHP trace and a traceback of all the tags being expanded and their arguments.

A template dump is done by putting the `dump` tag into your page or template. It generates, in HTML table form, a list of all the tags currently defined.

5 Template Directories

When you create your site, the first thing you will need to do is create a template. SCMS does provide a default template, which will be used for any page which doesn't explicitly specify one, but it's extremely minimal – you should replace it with your own as soon as possible.

Each template is contained its own directory in the `site/templates` directory, the name of which is the name of the template. It contains at least the file `main.html`, and probably a few `.tags` files. It should also contain any other files used by that template — CSS and ECMAScript files, images and so forth. These can be in subdirectories if you like.

5.1 How templates work

As you'll see later, each page is actually a file in tag definition language, each tag defined being automatically prefixed by '`page:`' as we mentioned earlier. The page says which template it should be rendered with, and the template defines what the overall 'shape' of the page is. This is how it works:

After the page file is read in, the `page:template` tag is checked. If it doesn't exist, we'll use the `default` template, otherwise we'll use the template named.

We then read any `.tags` files in the template's directory, parsing them to define their tags. Such tags will be prefixed with '`template:`'

Finally we read in the `main.html` template file, and put it through the template processor. This will use the tags defined in both the template and the page to generate HTML output. The output is then fed to the client browser[1].

5.2 The main template file

This file, called `main.html`, is the starting point for any page output. It's a plain HTML file containing template tags which are replaced to generate the

[1] Naturally I'm missing out a lot here: for example how `globals` is processed, and how modules and caching work.

page. The output produced is then fed to the client browser.

5.2.1 An example `main.html`

Here's an example of such a file:

```
<html>
 {{scmstag}}
 <head>
  <title>{{page:name}}</title>
  <link rel="stylesheet" href="{{templateroot}}site.css"/>
  {{navlinks}}
 </head>
 <body>
   <h2>{{page:name}}</h2>
   {{template:langmenu}}
   <h3>Navigation</h3>
   {{template:navmenu}}
   <p>Time created: {{ftime|%c}}</p>
   {{@page:content}}
 </body>
</html>
```

The following tags are used in this template:

- `scmstag` outputs an HTML comment which says that this HTML was generated by SCMS, and which version. It typically looks like this:

  ```
  <!-- Generated by SCMS, $Rev: 305 $ $Date: 2010-08-24
      15:30:17 +0100 (Tue, 24 Aug 2010) $ (of functpls) -->
  ```

- `page:name` is the actual name of the page, defined in page's tag definition file.

- `templateroot` produces the URL for the current template directory (followed by a slash), which is where we would expect to find data for the template, like images and CSS files.

- `navlinks` is a convenient tag defined by the system which produces accessibility `<link rel>` HTML tags defining the page's relationship to others in the site — see Chapter 9: Navigation, for more details.

- `template:langmenu` is a tag defined in one of the template's own `.tags` files, describing how to produce the languages menu.
- `template:navmenu` is a tag defined in one of the template's `.tags` files, describing how to produce the navigation menu.
- `ftime` is a system tag to output the current time (i.e. the time at which the page is generated) using the C library's `strftime()` function. The argument is the format to use - `%c` means the form 'Tue Jan 5 18:39:44 2010.' You can use this to determine how old a page is in the cache, if caching is enabled.
- `page:content` is a tag defined by the page, which is the actual content of the page. The value of this tag is likely to be a large, multiline string which may contain other tags to be expanded. The '@' means that if the tag doesn't exist, an empty string should be used instead of giving an error.

5.3 The .tags files

As noted above, SCMS scans the template directory for any tag definition files ending in `.tags` and reads them in, defining the tags and prepending their names with the `template:` string. Tags thus defined are called *template tags.*

These tags are mainly used in the `main.html` file, defining such things as how the page renders its menus. Here's an example which fits with the example file above. Don't worry too much about it for now, it'll be explained by Chapter 8: Trees and Menus.

```
navmenu=[[
{{treeprefix|default|<ul>}}
{{treesuffix|default|</ul>}}
{{treeselnode|default|
    <li class="currentpage">
    <h3>{{item:name}}</h3>|</li>}}
{{treeunselnode|default|
    <li><a href="{{item:url}}" title="{{item:name}}">
        {{item:name}}</a>|</li>}}
{{marktree|{{navtree}}|spec|{{spec}}}}
{{rendertree|{{navtree}}|item}}
]]
```

```
langmenu=[[
{{treeprefix|default|<small>}}
{{treesuffix|default|</small>}}
{{treeselnode|default|{{a:endonym}}|}}
{{treeunselnode|default|<a href="{{a:url}}">{{a:endonym}} </a>|}}
{{marktree|{{langtree}}|code|{{langcode}}}}
{{rendertree|{{langtree}}|a}}
]]
```

This file defines two tags: `template:navmenu` and `template:langmenu`, which render the navigation and language menus respectively. To understand what the various tags do, you'll need to look at Chapter, 8: Trees and Menus. One thing that's worth noting is that a lot of the tags — `treeselnode` and so forth — don't actually produce any text. Instead, they just set strings inside SCMS' menu system that are used when the `rendertree` tag runs. There are quite a few tags which set internal states rather than producing text.

6 Pages

Each page in a site has a single file in the `site/pages` directory, written in tag definition language. This file defines the tags used by the template to render the page's content — tags which are created with the `page:` prefix. You can call this file almost anything you like, provided it's valid as both a filename and as part of a URL. The page specifier will be the path of the file relative to `sites/pages`. One small caveat: the file cannot be called `defaults`. This name is reserved for the defaults file in a page directory, containing tags loaded in by all pages in or below that directory.

Consider our example template, consisting of the `main.html` file described in section 5.2.1. We see that the template requires the tag `page:content` to be defined. We also define `page:template` so that SCMS will know which template to use for this page[1], and `page:name` which is the page's name in navigation menus (and is also used by the template). We can do this by putting the following in the file `site/pages/home`:

```
name=The Home Page
template=default
content=[[
  <p>This is a test page, which doesn't really do very
  much at all.</p>
]]
```

To display this page, we'd request the URL `http://../index.php/home`. SCMS will then do the following:

- Open the page file, `site/pages/home`.
- Define the tags as specified in the file: `page:template`, `page:name` and `page:content`.
- Read the template specified in `page:template` — in this case, all the data will come from the `default` template directory.

[1] If we don't give the template name, the system will use the `default` template.

- Process the template's `main.html` file, resolving the tags.
- Output the concatenated result.

6.1 Special page tags

These are page tags to which SCMS ascribes special meaning. All other tags only have the meanings you give them by using them elsewhere in your templates and pages.

- `page:name` is a short string (preferably a single word) used as the name of the page in navigation. Examples are "Home Page" and "News."
- `page:template` specifies which template we should use. Templates consist of a set of files inside a template directory, and are used to define the overall look of the whole site or a group of pages — see page 35, Template Directories, for more details. If no template is specified, the template `default` will be used.

6.2 Redirection

In some cases, a page redirects to an external site. To do this, create a page file which specifies the name and the tag `page:redir`, which gives the external URL. For example,

```
name=BBC News
redir=http://news.bbc.co.uk/
```

All other tags will be ignored if you use `redir`.

6.3 Hierarchies

Hierarchies of pages can be created inside the `site/pages` directory, with the page specifiers being the filenames of the pages relative to that directory. For example, if there's a page in `site/pages/foo/bar`, the page specifier will be `foo/bar` and the URL will be something like `http://.../index.php/foo/bar`. Loading these pages is not, however, as simple as just reading that file and defining the tags. First we have to read and define tags in all the `defaults` files on the way down to the file.

6.3.1 The defaults file

Before a page file is loaded in, any files called `defaults` that are in that file's directory and each directory above it (up to `site/pages`) are loaded in and processed, in descending order, so that tags defined in deeper files will be read later and so supercede those in files higher up the hierarchy. If the page specifier references a directory rather than a file, the `defaults` file of that directory is the last file read.

This mechanism provides a way to set a large number of tags which are used in a lot of (but not all) pages, although ideally this job is left to templates. One thing it is useful for, perhaps, is setting the `page:template` tag for all the pages in a directory. It really comes into its own in hierarchies — imagine a pages directory containing the following files and directories (marked in *italics*):

`home` *`rooms`* `contact`

Now the *rooms* directory:

`defaults` *`upper`* *`lower`*

And finally the *upper* directory:

`room1` `room2` `room3`

Now consider viewing pages at each level:

- Opening `home` : here, we'll try to open `site/pages/defaults`, but it doesn't exist. We'll just open `site/pages/home` and use the values in it.

- Opening `rooms` : First, we'll try to open `site/pages/defaults` as before, failing. Then we'll try to open `site/pages/rooms/defaults` and succeed. We'll leave it there because this is a directory, not a file.

- Opening `rooms/upper` : We perform the two steps in the last example, and then try to open `rooms/upper/defaults`. This fails, so we stop there (since this is a directory).

- Opening `rooms/upper/room1` : The same as the previous example — failing to read `site/pages/defaults`, reading `site/pages/rooms/defaults`, failing to read `site/pages/rooms/upper/defaults`. Finally we read `site/pages/rooms/upper/room1`.

6.3.2 Marking defaults files as standalone

A danger here, though, is that we might end up with valid pages containing no content: some defaults files might be opened containing a few tags, with no actual content tags set. In order to avoid this, if a `defaults` file contains enough information to make a page by itself, it must contain the `page:standalone` tag. It doesn't matter what you set it to. Note also that it has to be in the `defaults` file of the directory itself — this tag is not inherited from parent `defaults` files.

Therefore, in the second and third cases above (`rooms` and `rooms/upper`) we we'll get a 404 page not found error unless we put this tag into `rooms/defaults` or `rooms/upper/defaults` respectively.

Note also that we can't reference the `defaults` files as pages directly. For example, we can't use `rooms/upper/defaults` as a specifier, even if `defaults` has a standalone tag. If we want to reference just that file, we should use `rooms/upper`.

6.4 The page file hierarchy is not the page navigation hierarchy

Note that the hierarchy of page files in the pages has nothing necessarily to do with the hierarchy of pages as shown in the navigation menu — the two may be similar, but the latter is defined by the `navigation` file, not the page directory hierarchy!

7 Collections

A collection is an array of data structures, with a fixed set of fields. For example, a collection of news items could have the fields 'date' and 'text.'

7.1 Simple collections, and iterating with foreach

Simple collections only have one field, whose name is 'value'. To create such a collection in Tag Definition Language, define a tag using '+=' instead of just an equals sign. If a collection of that name does not exist, it is created. A new item is then created with the string specified as its 'value' field, and added to the collection. Naturally, if a tag already exists with that name which is not a collection, an error occurs. Here's an example, using the `foreach` tag to iterate over the collection:

```
    mycollection += first item
    mycollection += second item
    mycollection += [[And this is the third item, which is
      a multiline item]]

    ## here we use foreach, so that {{page:listofitems}} expands to
    ## an unordered HTML list of the items

    listofitems = [[
        <ul>
        {{foreach|{{page:mycollection}}|a|
            <li>{{a:value}}</li>
        }}
        </ul>
   ]]
```

In this example, the `foreach` tag repeats its third argument (the *template*) for each element in the collection, with the `a:value` tag set to the element's string each time. The 'a' prefix comes from the second argument, which is a string to prefix each field's tag name with inside the template. The `foreach` also sets up an `a:index` tag giving the index (starting at zero) of each element. There's a more complete explanation of `foreach` below in section 7.4.

7.1.1 Accessing elements by index

You can also access each element in a collection by using the zero-based index of the element you want as an argument: `{{mycollection|1}}` would return the string 'second item' in the above example. This means you could access two collections in parallel, iterating over one collection and using the index to access the other collection:

```
newsdate += 1st January 1990
news += Stuff has happened!

newsdate += 2nd January 1990
news += Oh no, it's happened again!

newsdate += 3rd January 1990
news += And for a third time, events have occurred!

listofitems = [[
   {{foreach|{{page:newsdate}}|a|
        <div class=newsdate>{{a:value}}</div>
        <div class=newsitem>{{page:news|{{a:index}}}}</div>
    }}
   </ul>
]]
```

You wouldn't actually do anything like that in real code — if you wanted to store multiple fields in a collection, you'd actually use a complex collection.

7.2 Complex collections

The collection we created above just has one field: 'value.' If you want your collection items to have multiple fields, you must use a *complex collection.* To define a complex collection and add the first item, start a line of tag definition language with the '++' marker followed by the collection name, followed by the field names separated by commas:

```
++mycollection:field1,field2,field3
```

All subsequent lines will now set fields inside the new collection's first item, rather than creating new tags. To start creating another item, use '++' followed by the collection name. You *must exit this mode* by using '++' on a line by itself in order to define new tags. Here's an example:

```
++news:date,text
date=1st January 1990
text=Stuff has happened!

++news
date=2nd January 1990
text=Oh no, it's happened again!

++news
date=3rd January 1990
text=And for a third time, events have occurred!

++
```

The first line defines the collection and specifies the fields which are available, and creates the first item. Each '++' line after than adds a new item to the collection. The equals lines then set fields in the item just created. At the end, the '++' without a name indicates that we're switching back into normal mode, where we define tags rather than fields in a collection.

Finally, it's possible to leave fields undefined, in which case they'll produce errors if you try to expand them. It is, however, possible to test if a field is defined using the `iffieldexists` tag.

You can iterate through a collection and access items within it in exactly the same way as with simple collections, using `foreach`. Instead of just assigning the *`prefix`*`:value` field, each field in each item will be assigned for each iteration of the loop.

Here's an example of the above news tag system written using complex collections:

```
++news:date,text
date=1st January 2010
text=someone somewhere
++news
date=2nd January 2011
text=is having a toffee crisp
++news
date=3rd january 2012
text=[[and now he's finished eating the toffee crisp,
and he's working on a mars bar. Lovely!]]
++

content=[[
    {{foreach|{{page:news}}|a|
        <div class=newsdate>{{a:date}}</div>
        <div class=newsitem>{{a:text}}</div>
    }}
]]
```

7.2.1 Accessing complex collections by index

As we saw above, we can access a simple collection by index by doing something like `{{coll|0}}`. We can also specify the field name so that this can be extended to complex collections:

```
{{coll|0|myfield}}
```

will get the value of `myfield` in item 0 of the collection.

7.3 Collection handles

Whenever SCMS creates a collection, it creates a handle for it: a string in the form `coll:`*`integer`*. When we define a tag as referring to a collection using the '+=' or '++' notation, we create the collection and store the collection's handle in the tag. This can then be accessed using the tag with no arguments. For example, in our news example above, if we used the `page:news` tag with no arguments we would get the string `coll:0` or something similar in our output — the handle string.

We do this because the templating system works entirely on text strings, and therefore tag arguments have to be text strings. However, we sometimes need to be able to have collections as tag arguments (such as to `foreach` above). Substituting a collection handle for an actual collection allows us to do that. In addition, it lets us have collections inside collections — as we'll see later on with trees and menus.

7.4 More collection iteration

As noted above, we iterate over a collection using the `foreach` tag. This takes the collection handle, a tag prefix, and a template as its arguments. The tag operates by doing the following for each item:

- For each field in the item, define a tag ***prefix*** `:`***fieldname*** with the value of the field.
- Define the field ***prefix*** `:index` containing the index of the item.
- Define the field ***prefix*** `:handle` containing the node handle of the item[1].
- Process the template, expanding all tags within it.
- Concatenate the result onto the result string (which starts empty).

[1]see Chapter 8 for details on node handles, you don't need them for most collections.

It's possible to nest foreach loops. For example, you can place one collection inside another and iterate over them like this:

```
## Create two simple collections, page:inner1 and page:inner2

inner1+=inner collection 1, item 1
inner1+=inner collection 1, item 2
inner1+=inner collection 1, item 3
inner2+=inner collection 2, item 1
inner2+=inner collection 2, item 2
inner2+=inner collection 2, item 3

## Create a complex collection, page:outer. It has two fields:
## title is the name of an item, collection is a collection handle.

++outer:title,collection
title=Item 1
collection={{page:inner1}}

++outer
title=Item 2
collection={{page:inner2}}

++

## Now iterate through the outer collection, and within that,
## through the collection indicated by the 'collection' field of
## the current item.

content=[[
    <ul>
    {{foreach|{{page:outer}}|outer|
        <li>Outer collection - {{outer:title}}
        <ul>
            {{foreach|{{outer:collection}}|inner|
                <li>{{inner:value}}</li>}}
        </ul>}}
    </ul>
]]
```

7.5 Creating collections from strings

It's possible to split a string into a collection using the `split` tag. This takes two arguments: the string to split, and the delimiter. It returns the collection handle. Here's an example:

```
{{foreach|{{split|foo/bar/baz|/}}|f|
    <p>{{f:value}}</p>}}
```

This will print separate paragraphs for the three strings `foo`, `bar` and `baz`.

7.6 Debugging

It's possible to dump a collection or tree to an HTML table using the `dumpcoll` tag:

```
{{dumpcoll|{{navtree}}}}
```

will dump the top level of the navigation tree.

7.7 Implementation of collections

Not Yet Documented

Not yet written, but it's fairly straightforward — see `createlangtree()` in `langmenu.php` for an example of how to create a simple list (although it's set up as a tree, for which see the next chapter) and `navmenu.php` for a much more complex example.

8 Trees and Menus

As we have already seen, complex collections can contain fields which are handles to other collections. It's therefore possible to build trees using collections — in fact, section 7.4 contains an example of a sort of tree built with a collection.

SCMS' Tag Definition Language doesn't[1] have the syntax to create these trees easily yourself, although it's easy to create them from inside PHP[2]. It does, however, provide two very useful 'built in' trees: the navigation tree and the language tree[3]. It also provides powerful mechanisms for walking and otherwise navigating a tree hierarchy.

8.1 Tree handles

In the same way as collections are referenced by collection handles, trees are referenced by *tree handles*. Trees are effectively a subclass of collections: a tree handle is also a collection handle, and can be used as such, but a collection handle cannot be used in any of the tags which are for tree handles only. Tree handles are strings of the form `tree:`*n*.

8.2 Structure of a tree node

Trees in SCMS are a little unusual, because the fundamental concept they are built around is not a tree node, but a collection of nodes. For instance, the root level of the tree is a collection which (as we already know) is made up of an array of items. Some of these items have children, which are also collections.

From a nomenclature standpoint, I'll use the word *node* to refer to what in a collection is normally referred to as an *item*. They're fundamentally the same

[1] yet

[2] see Chapter 7.7: Collection Implementation

[3] As we'll see later, the language tree isn't really a tree since it only has one level, but it does have the selection properties of a tree and so uses tree code.

thing, but a node is part of a tree rather than a plain collection, and always has a `child` field (and a few others as specified below).

If a node has an empty string for the child field, or it is unset, the node doesn't have a child tree. If it does have children, the `child` field will hold the tree handle for the subtree. Tree collections differ from normal collections in that they also contain a handle to the parent collection (called `parentc`,) and the index number of the parent node within that collection (called `parenti`.) Note that these are not fields within nodes: they are properties of a tree collection as a whole, and there are special tags for accessing them.

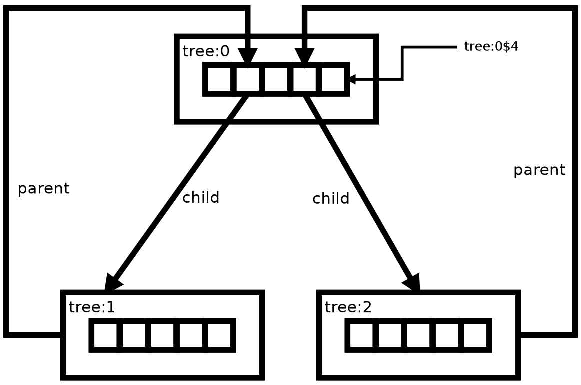

Figure 8.1: Tree structure

Figure 8.1 shows three tree collections making up a two level tree. The collection with handle `tree:0` is the root. Collection `tree:1` is the child of the second node in `tree:0`, while `tree:2` is the child of the fourth node. We've also shown a *node handle*, which points to the fifth node in `tree:0`. This will come in handy later when we need a single entity to refer to a single node in a tree.

In terms of field values, this means that:

- The `child` field of node 1[4] in `tree:0` has the string value `tree:1`.
- The `child` field of node 3 in `tree:0` has the string value `tree:2`.
- The `child` fields of all the other nodes are either unset or zero length strings.
- Collection `tree:1` has a `parentc` of `tree:0` and a `parenti` of 1.
- Collection `tree:2` has a `parentc` of `tree:0` and a `parenti` of 3.
- Collection `tree:0` has an unset or zero length `parentc` value, because it is the root.

8.2.1 Other fields

As well as the `child` field, items (nodes) in a tree collection contain the all the fields of a collection item and some extra fields:

- `ishome` is nonzero if the node is the first in the root level.
- `isintrail` is true if the node is in the trail down to the selected node after a `marktree` (see below).
- `issel` is true if the node is the selected node after a `marktree` (see below).

- `index` is the index of the current node within the current collection. It's not actually a field, being set dynamically during the render, but it behaves like one.
- `level` is the level of the current node within the current collection. Like `index`, it's not actually a field.
- `handle` is the node handle of the current node within the current collection. Another 'pseudofield.'

[4] i.e. the second node: all indices are zero based

8.3 The navigation tree

This tree is created from the `site/navigation` file, whose format is fully explained in Chapter 9: Navigation. Briefly, it describes a hierarchy of pages which can have more than one root (although the first page at the top level is always the home page) using hyphens to show indent level:

```
home
-page1
-page2
--page22
--page23
-other/page3
other/help
```

Here, we can see that the two pages `home` and `other/help` are at the top, root level; while pages `page1`, `page2` and `other/page3` are under the `home` page. Finally, `page2` contains two pages: `page22` and `page23`. The names are the page specifiers — i.e. the files under `site/pages` which contain the page's data. Note, as we pointed out in the Pages chapter, the hierarchy we are specifying here is *not the same* as the file structure!

The navigation tree's handle can be obtained from the `navtree` tag. Since a tree is essentially a collection, we can iterate over the top level with `foreach`:

```
{{foreach|{{navtree}}|node|
    {{node:name}}<br>
    }}
```

We'll just get the page names for `home` and `other/help` written out, since `foreach` doesn't know how to walk a full tree.

8.3.1 Additional fields

As well as the default tree fields listed above, nodes in the navigation tree also have the following fields:

- `spec` is the page specifier of the page, as listed in the `site/navigation` file.
- `url` is the URL of the page.
- `name` is the name of the page, as taken from the `page:name` tag.

- `type` is the type of the page, a single letter with values 'N' for normal, 'I' for invisible, and 'H' for heading only. These are set using special punctuation in the navigation menu – see Chapter 9: Navigation for more details.

8.3.2 Loading and Caching

The navigation tree is created when it's first requested. It's actually cached, held in the `tmp` directory with a separate file for each language. This is to avoid a lot of page file reads for large sites: if we created the navigation tree afresh on each run we'd have to load in every page file to get the `page:name` values. However, you shouldn't usually need to worry about this because if the file doesn't exist, or is old, it will automatically be rebuilt. If there is a problem, you may want to consider running the `clearcache` script from your site's root directory. This will delete the cached files.

8.4 Marking the tree

Once you've loaded or created a tree, such as the navigation tree, you'll probably need to mark the currently selected node and all those nodes in the trail down from the root to the selected one. To do this, we use the `marktree` tag. You need to have the tree handle, the name of a field giving a unique ID for each node, and the value of that field in the selected node. In the navigation menu, the unique ID is the `spec` field (the page specifier); and the value of that field in the selected node will be the current page's page specifier, which is returned from the `spec` tag:

```
{{marktree|{{navtree}}|spec|{{spec}}}}
```

The `marktree` tag itself produces no output, it just clears any marking on the tree and then walks it, setting the appopriate fields in the nodes.

8.5 Rendering a tree

To output a tree into your page, you need to walk the tree using the `rendertree` tag. Before you do that, however, you need to use various tags to define the *node rendering templates* which set how the tree should be rendered. These tags output no data themselves. The tags you'll need to use are:

- `{{treeprefix|level|prefix}}` sets a string to put before starting a new tree or a new level of a multilevel tree. For example, this might be a `<ul>` tag to open a list of nodes. See below for information on levels, but the 'level' value is often `default`, indicating that the same string is to be used for all levels.

- `{{treesuffix|level|suffix}}` similarly sets a string to render after a particular tree or subtree, such as `</ul>`.

- `{{treeunselnode|level|strpre|strpost}}` sets two template strings to use to render each unselected tree node. Again, this can be set differently for each level of the tree. The templates will make use of the the fields in the tree nodes, prefixed with the prefix set in `rendertree`. The 'strpre' string will be used before any subtrees are rendered, and the 'strpost' string will appear after them. You can think of the pre and post strings as a single string, broken in two. Any subtree output is inserted between the two.

- `{{treetrailnode|level|strpre|strpost}}` sets template strings to use to render nodes marked as being in the trail from the root down to the selected node.

- `{{treeselnode|level|strpre|strpost}}` sets template strings to use to render the selected node.

Once these tags are all set up, you can render the tree using the `rendertree` tag. This takes two arguments: the tree handle, and the prefix for the node field tags in the templates. Here's an example:

```
## Each level of the menu is an HTML unordered list,
## and so has <ul>.. </ul> around it.

{{treeprefix|default|<ul>}}
{{treesuffix|default|</ul>}}

## unselected nodes are rendered as list items containing
## a link to the named page

{{treeunselnode|default|
    <li><a href="{{a:url}}"{{a:name}}</a>|</li>
    ##         subtrees output goes here ^
}}

## trail nodes have links, but in bold

{{treetrailnode|default|
    <li><b><a href="{{a:url}}"{{a:name}}</a></b>|</li>
}}

## the selected node is rendered as just the name,
## since there's no point having a link to a page
## you're on.

{{treeselnode|default|
    <li>{{a:name}}|</li>
}}


## Now we can render the tree.

{{rendertree|{{navtree}}|a}}
```

8.5.1 If node selection templates are undefined

If you haven't used one of `treeunselnode`, `treeselnode` or `treetrailnode` for the current tree the system will fall back to templates you have defined, using the following rules:

- If `treeselnode` is not defined, the system will try to use `treetrailnode`. If that's not defined either, it will try to use `treeunselnode`.
- If `treetrailnode` is not defined, it will first try `treeunselnode`, then `treeselnode`.
- If `treeunselnode` is not defined, it will first try `treetrailnode`, then `treeselnode`.

You can find a more complex example of all of these tags in use in Chapter 9, Navigation.

8.5.2 Automatic clearing

Note that all the tree rendering templates are immediately cleared after `rendertree` is called.

8.5.3 Levels

As noted above, the five template setting tags take a level argument, which is `default` in all our examples. This argument can allow each level of the menu to be rendered using a different style. The level is a number starting at zero for the root level, or the word `default` which indicates that we are defining a template for use in levels where no specific level template has been defined. Here's an example:

```
{{treeprefix|0|<ul class="toplevel">}}
{{treeprefix|1|<ul class="2ndlevel">}}
{{treesuffix|default|</ul>}}
```

This will make the top level of the tree render using the CSS class `toplevel`, and the second level render with the class `2ndlevel`. The suffix for both levels of the menu will be the same, because `default` was specified.

8.6 Navigating the tree

It's sometimes useful to navigate the tree in more complex ways, and there are tags to allow this.

8.6.1 Setting temporary collection tags

You may find that you need to store a collection handle in a tag, and then get fields from this tag using the syntax in sections 7.1.1 and 7.2.1. You might be tempted to do something like this:

```
## get the child collection of the first node in the
## navtree
{{set|foo|{{navtree|0|child}}}}

## print the name of its first node
{{foo|0|name}}
```

This won't work – `foo` is a tag name, not a tree handle. However, you know that it *contains* a tree name, so you might be tempted to try

```
{{{{foo}}|0|name}}
```

This still won't work: you can't use `{{foo}}` as the name of a tag because tag names aren't processed through the templater. It wouldn't help even if they were, because you'd be trying to use a tree handle as if it were a tag name; they're two completely different namespaces.

What you can do is create a *collection tag*. In the same way as when you created collections in Chapter 7, you can create a new tag which contains a collection handle, and which the templater knows contains a collection handle rather than just a string. You can do this using the `setcoll` tag:

```
## get the child collection of the first node in the
## navtree and store it in a collection tag
{{setcoll|foo|{{navtree|0|child}}}}

## print the name of its first node
{{foo|0|name}}
```

This will now work, because SCMS knows that `foo` refers to a collection.

8.6.2 Finding the parent tree collection of a node

The tag `findcollection` can, given a field name and a value, search a tree for a given node. It will then return the tree handle of the collection which contains that node, so it can be walked with `foreach`, `rendertree` and so on. Syntactically it's very like `marktree` — so to find the collection containing the current page in the navigation tree, you would use

```
{{findcollection|spec|{{spec}}}}
```

8.6.3 Finding the parent node of a tree collection

To find the parent node of a collection given the collection handle, there are two tags: `parentc` will return the collection handle of the parent, and `parenti` will return the parent node index within that collection:

```
## set a temporary tag, tmp, to the collection
## the current page is in
{{setcoll|tmp|
    {{findcollection|{{navtree}}|spec|{{spec}}}}}}

## set a tag, pc,  to contain the handle
## for the parent collection of the collection
## in tmp
{{setcoll|pc|{{parentc|{{tmp}}}}}}

## and set pi to contain the index of tmp in
## that parent collection
{{set|pi|{{parenti|{{tmp}}}}}}

## Now show the name of that node, if the parent
## collection is not null
{{ifnotempty|{{pc}}|
    {{pc|{{pi}}|name}}
    |No parent node
}}f
```

This will get the collection the current page's is in, find the parent node of that node, and show its name. In other words, it will show the name of the parent page in a hierarchical navigation structure. It's a little long-winded, so there's a better way to do it using *node handles*.

8.6.4 Node handles

A node handle is a tree handle with a suffix referencing a node in the tree collection. Take a look at the diagram at the start of this chapter. You'll see that the fifth node in the collection `tree:0` has the handle `tree:0$4`. The handle consists of the tree handle, followed by a dollar sign, followed by the node's index.

8.6.5 Finding a node

Without node handles, we couldn't represent nodes by a single value and so had no way of searching for a node in the tree. The best we could do was use `findcollection` to find the collection which contained that node. Now we can do better, using `findnode`. This has the same arguments — field to search and search string — but returns the node handle of the node in which the given field equals the given string:

```
## print the node handle of the current page's
## node in the navigation tree
{{findnode|{{navtree}}|spec|{{spec}}}}
```

Incidentally, while you can do this — search for the current node in the navigation tree using `findnode` — the navigation menu system has a shortcut: `curnavnode` contains a handle to the current page's node in the navigation tree.

8.6.6 Finding the handle of the parent node of a node or tree

The `parent` tag, which takes either a tree handle or a node handle, returns the node handle of the parent:

```
## find the node, setting a temporary
{{set|tmp|
    {{findnode|{{navtree}}|spec|{{spec}}}}}}

## print out the handle of the parent if there is one.
## Otherwise, an empty string.
{{parent|{{tmp}}}}
```

8.6.7 Using node handles in place of tree handles

The last tag illustrates a general principle: you can use a node handle anywhere you can use a tree or collection handle, and it will act as the handle for the tree or collection which contains that node. For example, this will iterate over the nodes at the same level in the tree as the current page:

```
{{foreach|{{findnode|{{navtree}}|spec|{{spec}}}}|a|
    {{a:name}}
}}
```

Those of you with your eyes on the ball will note that this principle of being able to use node handles where tree handles can be used, when combined with `findnode`, means that `findcollection` is redundant. It's left in mainly to illustrate the usage of `parentc` and `parenti`, and for legacy templates.

8.6.8 Using fields in the node handle

As yet we can't do much with the node handle itself; we'll now see how to access the fields in a node represented by a node handle. There are several different ways to do this.

Using `getfield` to directly get the field values

The simplest way is to use the `getfield` tag, which takes the node handle and the name of a field:

```
{{set|tmp|
    {{findnode|{{navtree}}|spec|{{spec}}}}}}

{{getfield|{{tmp}}|name}}
```

This will print the name of the current page; a little pointless given that it's just the same as doing `{{spec}}`.

Also, if the node handle is stored with the correct type using `setnode` (see below), you can use the field name as an argument:

```
{{setnode|tmp|
    {{findnode|{{navtree}}|spec|{{spec}}}}}}

{{tmp|name}}
```

Creating temporary node tags with setnode

Alternatively, we can use a similar system to the collection tags in section 8.6.1 above, and create a temporary *node tag.* We can do this by using the `setnode` function tag instead of a plain `set`:

```
{{setnode|tmp|
    {{findnode|{{navtree}}|spec|{{spec}}}}}}

## use the node tag
{{p|name}}
```

The tag `setitem` is also available; it is just an alternative name for `setnode`.

Using `withnode` to access fields

Finally, it's possible to use the `withnode` tag. This takes a node handle, a tag name prefix, and a template, and defines all the tags with the values of all the fields in the node and runs the template with those tag definitions. In other words, it works rather like `foreach` but for a single node:

```
{{withnode|{{findnode|{{navtree}}|spec|{{spec}}}}|a|
    {{a:name}} has the url {{a:url}}
}}
```

8.6.9 Finding the parent collection of a node

Note that this doesn't mean the collection a node is in — since node handles can be used where tree handles are used, we can just use the node handle itself for that. The 'parent collection' here means the tree collection which *contains* the tree collection the node is in. Again, since node handles can be used where tree handles are accepted, we can use the `parentc` tag to find the parent collection of a node. We can therefore iterate over all the nodes in the level above a given node using:

```
## find the current page's node
{{setnode|tmp|
    {{findnode|{{navtree}}|spec|{{spec}}}}}}

## find the parent collection of the
## collection that node is in
```

```
{{setnode|tmp|{{parentc|{{tmp}}}}}}

## if it’s not empty (i.e. we weren’t at root)
## iterate over it
{{ifnotempty|{{tmp}}|
    {{foreach|{{tmp}}|a|
        {{a:name}}
    }}
|}}
```

8.6.10 Finding the index of a node

To do this, just use the `indexof` tag with the node handle:

```
{{setnode|tmp|
    {{findnode|{{navtree}}|spec|{{spec}}}}}}

I am number {{indexof|{{tmp}}}} in my level.
```

8.6.11 Node handles in iterators

As well as setting the fields and the `index` and `level` values etc., iterators like `foreach` and `rendertree` will also set a `handle` value which contains the node handle of the node currently being output:

```
{{foreach|{{navtree}}|a|
## a:handle is defined as a node tag, so
    {{a:handle|name}}
## is the same as
    {{a:name}}
}}
```

8.6.12 Finding the child of a node

If a node has a child, its `child` field will contain the tree handle of the child subtree. You could, therefore, iterate over the top two levels of the navigation tree producing a cartesian product using:

```
{{foreach|{{navtree}}|a|
   {{ifnotempty|{{a:child}}|
       {{foreach|{{a:child}}|b|
            {{a:name}}/{{b:name}}}}
   |}}
}}
```

8.6.13 Checking whether a field is set

The above code could also be written in the following way, using `iffieldset` to test if there is a child:

```
{{foreach|{{navtree}}|a|
   {{iffieldset|{{a:handle}}|child|
       {{foreach|{{a:child}}|b|
            {{a:name}}/{{b:name}}}}
   |}}
}}
```

9 Navigation

In this chapter we'll discuss navigation menus in more detail, describing the different types of navigation item which can be created and how more complex menus can be built. It's worth going into this in some detail because menu generation is probably the most complex part of writing tag definition files.

9.1 The navigation file

As described in Chapter 8: Trees and Menus, the navigation menu is a tree collection created automatically from the file `site/navigation`. This file is a text file which you create, which describes the page organisation of your site[1]. Here's an example:

```
others/home
products
-chairs
--chairs/chair1
--chairs/chair2
--chairs/chair3
-tables
--tables/table1
--tables/table2
--tables/table3
-desks
--tables/desks/desk1
--tables/desks/desk2
--tables/desks/desk3
--tables/desks/desk4
-recentprods
--tables/table2
--tables/desks/desk3
--tables/desks/desk4
```

[1] *not* the directory organisation, as I may have mentioned a few times before.

```
--chairs/chair2
others/contact
others/help
```

Each entry is the name of a page (i.e. a file in the `site/pages` directory.) This will give us a home page, a products page, a contact page, and a help page all at the top level. Note that some of the page data is kept in subdirectories - the contact and help page files are kept in the `others` subdirectory although they appear at the top level in the menu hierarchy. This navigation file will automatically generate the `navtree` tag's tree collection, which we can iterate over to render the menus as described in Chapter 8: Trees and Menus.

Note that the products menu contains four submenus. Three are for different kinds of products and one is for recent products. This recent products menu contains items which are already in some of the other menus: *pages can be referenced in a navigation menu in more than one place.*

9.1.1 Directory items

In the above hierarchy, the chairs, tables and desks entries are actually pages in their own right. However, we can see that those entries refer to directories (since they contain other files for the actual product pages.) To make this work, we have to create a `defaults` file in each of these directories giving the page content when the directory is specified, which contains a line giving a value for the `standalone` tag. For example:

```
name=Tables
standalone=yes
content=[[
    <p>This page describes the various tables we have available.</p>
]]
```

As described in Chapter 6: Pages, this allows a directory to be a page. If one of the subpages (such as `tables/table1`) is loaded, the `table/defaults` tag will still be loaded but its tags will be overriden by those in the page file. If we don't create `defaults` files with `standalone` tags in all three items which are directories, the result will be a 404 when we try to open them.

9.2 Heading-only items

Another alternative is to stop those items from being pages at all — just have them as section headings in the menu. This requires two changes: marking the items as heading-only, and rewriting the menu tags to process that information. We do the first part by simply putting square brackets around the heading-only items. The `site/navigation` file therefore becomes:

```
others/home
products
-[chairs]
--chairs/chair1
--chairs/chair2
--chairs/chair3
-[tables]
--tables/table1
--tables/table2
--tables/table3
-[desks]
--tables/desks/desk1
--tables/desks/desk2
--tables/desks/desk3
--tables/desks/desk4
others/contact
others/help
```

What this does is set the `type` field of the item in the menu tree to `H` rather than the normal `N`. We now need to modify the menu code to stop rendering links for these items. We'll assume we're writing these tags in a `.tags` file in the template's directory, so all tags we define are prefixed with `template:`. First, we'll define a couple of tags which will render an item in the menu with or without a link:

```
## don't show a link
nolinknode={{a:name}}

## show a link
linknode = <a href="{{a:url}}">{{a:name}}</a>
```

This is pretty straightforward. The `nolinknode` tag will render the current tree node without a link, as just a list item; while the `linknode` tag will render it with a link. These two tags are stored directly into the template system without being expanded until they're used, which will be from inside a `rendertree` tag, so the `a:...` items will be in the right context. Now we add a third tag, which will show the node as a link only if the node's `type` is not `H` – i.e., it's not a heading. We do this using the `streq` tag, which takes four arguments — the two strings to compare, the string to use if the strings are equal, and the string to use if they are not equal:

```
## show a link unless this is a heading-only node

linknodeifnotheading=[[
        {{streq|{{a:type}}|H|
            {{template:nolinknode}}|
            {{template:linknode}}
        }}
]]
```

Now we're ready to set up the menu using the menu tags, and render it. This we do from inside a tag definition, since that's the only place we can use tags. The tag we'll define is `navmenu`, and it'll set up the tree rendering environment and then render it:

```
navmenu=[[
    ## first define the prefix and suffix: we'll render
    ## the menu as an unordered list, and each level within
    ## it as another list. Each level, therefore, gets
    ## wrapped in <ul></ul>

    {{treeprefix|default|<ul>}}
    {{treesuffix|default|</ul>}}
```

The next step is to use the `treeunselnode` tag to tell SCMS how to render an unselected tree node. In this case, we call the `linknodeifnotheading` tag we defined before to render a link if the item isn't a heading:

```
    ## unselected nodes have a link unless they're
    ## heading only

    {{treeunselnode|default|<li>
        {{template:linknodeifnotheading}}|</li>}}
```

Now we use `treeselnode` to set up how to render a selected node. This will just render the node without a link, since we don't need a link when we're already viewing this page:

```
  ## selected node renders with no link
  {{treeselnode|default|<li>
      {{template:nolinkitem}}|</li>}}
```

Finally, we need to render items in the trail — i.e. on the path down to the selected item — in the same way as unselected nodes. This is easy: we just don't specify the `treetrailnode` template, and it will fall back to using the template specified by the `treeunselnode` tag as described in section 8.5.1.

All we need to do now is mark the tree, so it knows which items are selected, and render it:

```
  {{marktree|{{navtree}}|spec|{{spec}}}}
  {{rendertree|{{navtree}}|a}}
]]
```

Of course, there's nothing to stop us writing the whole thing without defining the extra templates:

```
{{treeprefix|default|<ul>}}
{{treesuffix|default|</ul>}}

{{treeunselnode|default|<li>
  {{streq|{{a:type}}|H|
    {{a:name}}|
    <a href="{{a:url}}">{{a:name}}</a>
  </li>}}
}}
{{treeselnode|default|<li>
    {{a:name}}|</li>}}

{{marktree|{{navtree}}|spec|{{spec}}}}
{{rendertree|{{navtree}}|a}}
```

and that's perfectly clear (or should be), but note that we're duplicating the 'render without a link' template: I've marked it in *italics* in the code above. It appears as the selected node template, and also in the unselected or trail node template as the 'do if true' part of the 'is this a heading only?' condition. If we wanted to change it later, we'd have to change it in both places.

9.3 Invisible items

In a similar way to heading-only items, items can be marked as ‘invisible.’ This is done by putting them in round brackets in the navigation file:

```
others/home
products
-[tables]
--tables/table1
--tables/table2
--tables/table3
-[desks]
--tables/desks/desk1
--tables/desks/desk2
--tables/desks/desk3
--tables/desks/desk4
(adminlogin)
others/contact
others/help
```

would add an `adminlogin` page which should not be shown by the menu system. As with heading-only pages, the system does nothing with these except change the `type` field in the navigation tree nodes, in this case to `I`. It’s up to you to program your menus to implement this. Here’s an example extending the previous example and using a `switch` tag. Actually, we’ve used two such tags, one for the each part of the node’s template. We don’t need to change the selected node templates, because a selected node can’t be invisible or heading-only:

```
navmenu=[[
    {{treeprefix|default|<ul>}}
    {{treesuffix|default|</ul>}}

    {{treeunselnode|default|
        {{switch|{{a:type}}|
            H|<li>{{a:name}}|
            I||
            default|<li>{{template:link}}{{a:name}}</a>}}|
        {{switch|{{a:type}}|
            H|</li>|
            I||
            default|</li>}}
    }}
    {{treeselnode|default|
            {{streq|{{a:type}}|I||
                <li>{{a:name}}|</li>}}}}

    {{marktree|{{navtree}}|spec|{{spec}}}}
    {{rendertree|{{navtree}}|a}}
```

The `switch` tag takes an input to compare, and then any number of string/output pairs:

```
{{switch|input|str1|output1|str2|output2...|default|defaultoutput}}
```

If the input matches a string, the corresponding output is produced. If no string is matched by the time we get to `default`, the default output is produced. If there's no default item we return the empty string.

So in our code above, the unselected node template works by using a three-way switch: if the type is `H`, it's a heading and we output just the name with no link; if the type is `I`, it's invisible and we output nothing; otherwise we output the link. The selected node works using the `streq` tag, outputting nothing if the link is invisible and just the name otherwise.

9.4 Menus with different levels

In this section we'll discuss common styles of menu which require different levels of the menu to be rendered in different ways, and show some examples of how this can be done.

9.4.1 Simple menu with different levels

This is the simplest, and perhaps most common case: rendering the menu as different classes of HTML unordered list at different levels. We can do this very simply by using different tree prefixes for the different levels:

```
navmenu=[[
{{treeprefix|0|<ul class=level1>}}
{{treeprefix|1|<ul class=level2>}}
{{treeprefix|2|<ul class=level3>}}
{{treesuffix|default|</ul>}}

{{treeunselnode|default|<li>
{{switch|{{a:type}}|
    H|{{a:name}}|
    I||
    default|<a href="{{a:url}}">{{a:name}}</a>
    }}|
    </li>
}}

{{treeselnode|default|<li><b>{{a:name}}</b>|</li>}}

{{marktree|{{navtree}}|spec|{{spec}}}}
{{rendertree|{{navtree}}|a}}
]]
```

Note that I've used `switch` to deal with invisible and heading-only nodes, and that trail nodes are the same as unselected nodes. I've only used one switch in that first case, so invisible nodes will show up as empty list items! I leave fixing it as an exercise.

9.4.2 Single level menu at all levels

In this system, we just show a single level of the menu, and the name of the parent node to allow us to move back up the hierarchy. For example, if we were on the home page, our example system would show something like

```
Home   Products   Contact   Help
```

If we clicked on 'Products', we'd get

`Home` `Chairs` `Tables` `Desks`

Clicking on 'Chairs', we'd get

`Products` `chair1` `chair2` `chair3`

and finally clicking on 'chair1' would give us

`Chairs`

the only link available being one to return us to the previous level. In other words, we show the name of the parent node and the children of the current node. Instead of using tree rendering, we use the tags from section 8.6, Navigating the Tree.

```
navmenu=[[
    {{ifnotempty|{{parent|{{curnavnode}}}}|
        {{withnode|{{parent|{{curnavnode}}}}|a|
            <i><a href="{{a:url}}">{{a:name}}</a></i>}}
        |}}
    {{withnode|{{curnavnode}}|a|
        {{iftagexists|a:child|
            {{foreach|{{a:child}}|b|
                <a href="{{b:url}}">{{b:name}}</a>,
            }}
        |}}
    }}
]]
```

This uses `curnavnode`, which contains a node handle to the node in the navigation tree corresponding to the current page. Then get the node handle of this node's parent, and if it's not empty (i.e. this is not the root node) we use withnode to render it[2]. We then use another withnode to access the current page's node itself. Inside this, we check to see if the node has a child. If it does, we iterate over that child, outputting the url and name tags of the subpages.

9.4.3 Breadcrumbs

A breadcrumb trail is a list of links to all the ancestor pages of the current page. We can create one by rendering the navigation tree with empty templates for unselected pages, so we only see the selected (current) page and those pages in the trail:

```
breadcrumbs=[[
{{treeprefix|default|}}
{{treesuffix|default|}}
{{treeunselnode|default||}}
{{treeselnode|default|{{a:name}}|}}
{{treetrailnode|default|<a href="{{a:url}}">{{a:name}}</a> |}}
{{marktree|{{navtree}}|spec|{{spec}}}}
{{rendertree|{{navtree}}|a}}
]]
```

We've specified that the trail pages are rendered by links, while the selected pages doesn't have a link — after all, you're already viewing that page.

9.4.4 A complex example using `loadpage` to read data from subpages

This is a menu where the two levels are utterly different, and rendered using separate code. The system I'll describe here is one we use on the Broadsword website, where a single level main menu appears at the top of all pages, but there is a special 'apps' menu which appears as part of the apps page content in which the main menu is suppressed, and information about each subpage of apps is shown by reading the subpage data itself.

Here's the navigation file we're using:

```
home
services
news
```

[2]I'm using the <i> tag here because it's easy, not because I think it's a good idea. You should, of course, use a <div> or <span>.

```
apps
-a/myfirstapp
-a/anotherapp
contact
support
```

We can see that there is mainly one level, except for the children of `apps`. That page is dealt with specially, as we mentioned above.

The main menu is essentially straightforward, using `foreach` to iterate through the current level of the navigation tree. However, the requirements of the page layout mean there are a few little complexities. I'll deal with the code bit by bit.

```
navmenuwidth=450
navmenuleft={{sub|512|{{div|{{template:navmenuwidth}}|2}}}}
```

This calculates the dimensions of the menu div for setting in CSS. We set the menu div to be 450 pixels wide, and then given that the parent div in which it will appear is 1024 pixels wide, the left hand edge of the menu div can be set to $512 - \frac{navmenuwidth}{2}$. With this information we can now code the `template:navmenu` tag itself.

```
navmenu=[[
    {{set|curcoll|{{findcollection|{{navtree}}|spec|{spec}}}}}
```

Here we're starting the menu tag, and storing the current level in a variable called `curcoll`. This is a slightly different approach from that used in Section 9.4.2, using `findcollection` to find the the current node's *collection* rather than `findnode` to find the node itself.

```
{{iftagexists|page:nomenu||
    {{marktree|{{navtree}}|spec|{{spec}}}}
```

First we check to see if the current page has a tag called `nomenu`. If so, we don't process the rest of the tag — you'll see from the rest of the code that the rest of the tag is inside the false part of the `iftagexists` condition and so only runs if there's no `nomenu`. Note that double bar: the true part of the condition is the empty string.

```
    <div id="navmenu" style="width:{{template:navmenuwidth}}px;
                             left:{{template:navmenuleft}}px;">
```

This starts the navigation menu div, setting the position and width to the values we worked out earlier.

```
        {{foreach|{{curcoll}}|a|
            {{streq|{{a:type}}|I||
```

Now we use `foreach` to iterate through each item in the current menu level, only showing those for which the `type` field is not `I` – i.e. which are not hidden.

```
            {{iftagtrue|a:issel|
                <span>{{a:name}}</span>|
```

If the node's `issel` field is true – i.e. it is selected – we just render the name of the page as a span. This is picked up by the CSS and rendered accordingly, with no link.

```
                <a href="{{a:url}}"
                    {{streq|{{a:name}}|apps|
                        onmouseout="P2H_StartClock();"
                        onmouseover="P2H_Menu('PMgamesbutton',0,0);"
                    |}}>
                {{a:name}}</a>}}
```

However, if the item is not selected, we render it as a link. There's another complication here - if the name of the current node is `apps` we modify the link to provide scripted events. The idea is that stuff appears while we're hovering over the apps link.

```
            }}
        }}
        </div>
    }}

    ]]
```

Finally, we close off the tags, the div and the definition of `navmenu`.

Reading in data from other pages

Now for the second level of the menu, which appears in the `apps` page file. This is rather clever, in that it will not only display information which the navigation menu knows about the pages, but it will also open each page and read the tags it defines so we can show some additional information about each app. Consider that each subpage of `apps` looks something like this:

```
name=My Application
shorttext=A short description to appear in the list of apps
imgcode=myimage.png
content=[[<p>A much longer text describing the content
of the app’s page in full, which we’ll only see
when we view the page itself.</p>]]
```

We will be able to use some of the tags defined by the subpages inside the `apps` page which lists them.

First, we’re going to define the content of the page:

```
content=[[
{{ifnotempty|{{curnavnode|child}}|
```

This uses `curnavnode`, which we know always contains a pointer to the current node in the navigation tree. Here, we only run the following code if the `child` field in `curnavnode` is a non-empty string: i.e., if the current page has child pages. This uses the syntax in Section 8.6.8.

```
    <div id="listcontainer">
        {{count|i|start}}
```

Now we have created a list div and inside it, set up a *counter* using the `count` tag. This is a value which can be incremented, and used for various purposes. We’re going to use it to set up ‘zebra striping’ so alternate entries in the list are different styles.

```
        {{foreach|{{curnavnode|child}}|a|
            {{loadpage|{{a:spec}}|app}}
            {{page:appentry}}
        }}
    </div>
|}}
]]
```

Now we can start iterating through the collection of the current node’s children. In this case these are the pages which are subpages of the `app` page. For each page, we call the `loadpage` tag. This is a tag which will read the page file whose specifier is given in the first argument, but instead of prefixing each tag defined therein with `page:`, it will use the prefix given in the second argument. For example, `app:name` will be defined after the tag has run, as will all the other tags in the page loaded. Once the subpage we’re currently iterating is loaded, we expand the tag `page:appentry`, which we’ll define shortly.

So to recap, this code:

- Finds the current page's node
- Checks that it has children, and if so...
- Starts a counter called 'i'
- For each child of the current page
  - Load that page's page file, and define all its tags but prefixed with 'app' instead of 'page'
  - Expand `page:appentry` which will use the information we read out of the subpage's file to display some information about the subpage.

Now let's look at that `page:appentry` tag:

```
appentry=[[
    <div class="{{count|i|alt|listitemeven|listitemodd}}"
        style="background-image:url({{imgroot}}{{app:imgcode}})">
        <h1><a href="{{a:url}}">{{app:name}}</a></h1>
        {{app:shorttext}}
    </div>
]]
```

The first line creates a class, and demonstrates how the `count` tag works: we created the counter 'i' at the start of the page, and each time round we read and increment it in `alt` mode. In this mode, the tag will return either its third or fourth item, depending on whether the counter is even or odd. This means that we'll get divs in alternate classes. If we set the background colour slightly differently in each one, we'll see the 'zebra striping' effect we want. The remaining lines use various tags defined inside the subpage's file and loaded in using `loadpage`: `app:name` is the page's name, `app:shorttext` is a tag we add to each app's page to give a brief description for use in this list, and `app:imgcode` is the name of an image file in the image root directory (set to be `site/images/`, the value of `imgroot`).

9.4.5 Two-level horizontal menu

This style of menu is used on sites like *The Register.* It features two horizontal menus — the top level menu, and the current submenu. For example, consider the navigation file

```
(home)
-h/alerts
-h/newsletters
-h/reviews
hardware
-hw/pcs
-hw/servers
-hw/hpc
software
-sw/os
-sw/apps
-sw/dev
```

which is a truncated version of the Register's structure. There would also be additional pages below each sublevel for the individial news stories The home page is special, in that there is no menu entry for it: you can always get there by clicking on the logo. So on the home page, the menus that appear are:

```
top line:    hardware   software
bottom line: alerts newsletters reviews
```

In Software we get:

```
top line:    hardware   software
bottom line: os apps dev
```

If we go into apps or any page below it, we get:

```
top line:    hardware   software
bottom line: os apps dev
```

Clearly the highlighting can be done by using the existing trail mechanism. Let's look at an implementation. First, we'll mark the tree as usual, and then render the root menu directly using a `foreach`:

```
navmenu=[[
    <div class="topmenu">
```

```
        {{marktree|{{navtree}}|spec|{{spec}}}}
        {{foreach|{{navtree}}|a|
            {{if|{{a:isintrail}}|
                {{setnode|lev2node|{{a:handle}}}}|}}
            {{streq|{{a:type}}|I||
                {{if|{{a:isintrail}}|
                    <b>{{a:name}}</b>
                    |
                    <a href="{{a:url}}">{{a:name}}</a>
                }}
            }}
        }}
        </div>
```

This is straightforward, except for the `setnode` lines:

```
            {{if|{{a:isintrail}}|
                {{setnode|lev2node|{{a:handle}}}}|}}
```

if the node is in the trail, store its handle in `lev2node`. What this does is stores which node in the top level menu is in our trail, because this is the menu we want to render for our second level. You might think it would be a good idea to put this `setnode` inside the later in-trail test, rather than doing two separate tests, but if you did that the system wouldn't work properly for hidden pages, like the home page in our example: the set would never run for hidden nodes.

Our second level code follows on immediately afterwards:

```
        <div class="2ndmenu">
        {{iftagexists|lev2node|
            {{iffieldexists|{{lev2node}}|child|
                {{foreach|{{lev2node|child}}|a|
                    {{streq|{{a:type}}|I||
                        {{if|{{a:isintrail}}|
                            <b>{{a:name}}</b>|
                            <a href="{{a:url}}">{{a:name}}</a>
                        }}
                    }}
                }}
            |}}
        |}}
        </div>
    ]]
```

This checks to see if the node we stashed away earlier was actually stored, and that it has a child — another menu. If so, we do exactly the same iteration to render the items in that menu, using the syntax in section 8.6.8 to get the value of the child field of the node, which contains the collection handle of the menu we want.

9.4.6 Pulldown menus

This example is actually more about CSS and JavaScript than SCMS — the SCMS code is very similar to that which generates the simple menu with multiple levels, but the scripts make them look like a pulldown. In a pulldown menu there are two levels: the root level, in which the items are organised in a horizontal line across the page; and the second level, in which items appear vertically below their parent. These items are normally invisible until the parent is hovered over.

As I mentioned before, we can pretty much do this in CSS and JavaScript, given a hierarchy of unordered list (UL) items. First we need to produce our menu system as a test site in HTML, CSS and JavaScript with no clever SCMS stuff. I'm going to steal some code from the Web here[3] and use it as our basis. The HTML is a straightforward nested UL list, which appears in the sample as this:

```
<ul id="nav">
        <li><a href="#">Percoidei</a>
                <ul>
                        <li><a href="#">Remoras</a></li>
                        <li><a href="#">Tilefishes</a></li>
                        <li><a href="#">Bluefishes</a></li>
                        <li><a href="#">Tigerfishes</a></li>
                </ul>
        </li>

        <li><a href="#">Anabantoidei</a>
                <ul>
                        <li><a href="#">Climbing perches</a></li>
                        <li><a href="#">Labyrinthfishes</a></li>
                        <li><a href="#">Kissing gouramis</a></li>
                        <li><a href="#">Pike-heads</a></li>
                        <li><a href="#">Giant gouramis</a></li>
                </ul>
        </li>
</ul>
```

I'm using a tiny font now because there's a lot of code here. Now we add some clever CSS:

```
#nav, #nav ul { /* all lists */
        padding: 0;
        margin: 0;
```

[3]http://htmldog.com/articles/suckerfish/dropdowns/

```
        list-style: none;
        line-height: 1;
}

#nav a {
        display: block;
        width: 10em;
}

#nav li { /* all list items */
        float: left;
        width: 10em; /* width needed or else Opera goes nuts */
}

#nav li ul { /* second-level lists */
        position: absolute;
        background: orange;
        width: 10em;
        /* using left instead of display to hide menus because
        display: none isn't read by screen readers */
        left: -999em;
}

#nav li:hover ul, #nav li.sfhover ul { /* lists nested under hovered list items */
        left: auto;
}
```

And now we load this CSS from inside `main.html` in the template, along with a little snippet of javascript for dumb browsers which don't support the `hover` attribute. We use the `templateroot` tag to get the template directory:

```
<link rel="stylesheet" href="{{templateroot}}site.css"/>
<script type="text/javascript">
<!--
sfHover = function() {
        var sfEls = document.getElementById("nav").getElementsByTagName("LI");
        for (var i=0; i<sfEls.length; i++) {
                sfEls[i].onmouseover=function() {
                        this.className+=" sfhover";
                }
                sfEls[i].onmouseout=function() {
                        this.className=this.className.replace(new RegExp(" sfhover\\b"), "");
                }
        }
}
if (window.attachEvent) window.attachEvent("onload", sfHover);
-->
</script>
```

Finally, all we need is a very simple tag defined in the template's `menus.tags` or similar:

```
navmenupulldown=[[
{{treeprefix|default|<ul>}}
{{treesuffix|default|</ul>}}
{{treeprefix|0|<ul id="nav">}}

{{treeunselnode|default|<li>
    {{streq|{{a:type}}|H|
        {{a:name}}|
        <a href="{{a:url}}">{{a:name}}</a>
    }}|</li>
}}

{{treeselnode|default|<li><b>{{a:name}}</b>|</li>}}

{{marktree|{{navtree}}|spec|{{spec}}}}
{{rendertree|{{navtree}}|a}}
]]
```

This is the same as our previous system, but we've embedded things a bit more – nesting some if the conditionals – and we've put the `nav` CSS ID on the outermost `UL`. It's very easy to use this as a basis for prettier menus, for instance using different CSS classes for different levels of the menu.

9.5 Accessibility

The navigation of a site's structure can difficult for users with some disabilities. Most of these problems can be mitigated with a good HTML template design, and there are many resources on the Internet to help with this. A good place to start is the W3C WCAG guidelines page at `http://www.w3.org/TR/WCAG/`.

Two of the facilities which help accessibility are *document relationship links* and *access keys*. SCMS provides methods for producing these.

9.5.1 Document relationship links (LINK REL tags)

These are links which appear in the HEAD block of an HTML file, and have the following form:

```
<link rel="start" title="Home Page" href="http://foo.org/index.php/home"/>
<link rel="prev" title="Page 1" href="http://foo.org/index.php/page1"/>
<link rel="next" title="Page 3" href="http://foo.org/index.php/page3"/>
```

If you put the tag

```
{{navlinks}}
```

in the HEAD block of your template, SCMS will insert a set of LINK REL tags at that point which correctly match the structure of the navigation menu and the current page's position within it. There's a lot of customisation which can and should be done, which we'll learn about later.

9.5.2 Access keys

Some sites rely on access keys to help users get to common pages. You can support this in your menu by checking the `key` field in each node as you render it. This value is set from the `navigation` file, by putting

```
key=keyvalue
```

after the navigation item, separated by a space. For example, if you wanted access key 1 on some page[4] within your structure you would have a line like:

```
---page1 key=1
```

If the `key` field is set, a special field called `accesskeyattr` will also be set inside the node. Normally set to the empty string, it will now contain a string like

```
accesskey="h"
```

You can use this field in your menu's tree rendering templates, as an attribute inside the `A` tag. For example,

```
{{treeunselnode|default|<li>
  <a href="{{a:url}}" {{a:accesskeyattr}}>{{a:name}}</a>
  |</li>}}
```

Access keys for document relationship links (start, next, previous)

In section 9.5.1, we defined document relationship links. We can specify access keys for these using `navlinks` tag, by passing in extra arguments — one for each key: start, next and previous. We also need to pass in a template string, which is used to render the title text for the link, along with a prefix for the tags defined – the link will have all the standard node fields available to it, from the node that will be the target of the link. For example, if we wanted to use the 1 key for the home page, the N key for the next page, and the P key for the previous page, we could use

```
{{navlinks|a|{{a:name}}, shortcut key {{a:lkey}}|1|N|P}}
```

Going through the arguments in order, we have:

- The prefix `a`, used to prefix the target node's fields when they're set up as tags, in the same manner as a `foreach` or `withnode`.
- The actual template used to render the title of the link, given that the target node (i.e. the start, next or previous node) has had tags set up with the given prefix. Note that the `lkey` field is now usable, giving the access key for the link (it stands for *link key.*)

[4]The actual key combination pressed will be different for different browsers. In Firefox, access key 1 would actually be ALT+SHIFT+1. In Internet Explorer, ALT+1 would focus on the link, but then you'd have to press ENTER.

- The access key for the first node.
- The access key for the next node, if any.
- The access key for the previous node, if any.

How this works is rather complex. Firstly, it causes the `LINK REL` tags output by `navlinks` to have title strings rendered from the template, so we can do effective localisation — particularly of the "shortcut key=xxx" part (the default title is the name of the link.) Secondly, it adds extra `accesskey` attributes to the `accesskeyattr` fields of the appropriate nodes, so that these nodes will have access keys added to their links. These new keys will work alongside any keys you may have already defined with the `key` option in the navigation file.

Unused access keys

If any access key is left empty, it won't be added. For example,

```
{{navlinks|a|{{a:name}}, shortcut key {{a:lkey}}||N|P}}
```

does not have a start key assigned, so no start key will be set up.

An example

Here's the UK government's mandated scheme for access keys, applied to a navigation file:

```
home key=1
-whatsnew key=2
-search key=4
-help key=6
--sitemap key=3
--faq key=5
--complaints key=7
--termsandconds key=8
--feedback key=9
--accesskeydata key=0
```

And here's a simple menu renderer, using the pulldown menu system of section 9.4.6:

```
navmenupulldown=[[
{{treeprefix|default|<ul>}}
{{treesuffix|default|</ul>}}
{{treeprefix|0|<ul id="nav">}}

{{treeunselnode|default|<li>
    {{streq|{{a:type}}|H|
        {{a:name}}|
        <a href="{{a:url}}" {{a:accesskeyattr}}>{{a:name}}</a>
    }}|</li>
}}

{{treeselnode|default|<li><b>{{a:name}}</b>|</li>}}

{{marktree|{{navtree}}|spec|{{spec}}}}
{{rendertree|{{navtree}}|a}}
]]
```

And here's what we'll put into the main template. Note that we're not using a start page access key, because that's already dealt with by the navigation file's explicit key assignment in the first line:

```
{{navlinks|a|{{a:name}}, shortcut key {{a:lkey}}||N|P}}
```

9.6 Creating links within the site

To specify links to other pages which aren't part of menus but in the body of the text, use the `url` tag to generate a URL given a page specification:

```
<a href="{{url|contact}}">Contact page</a>
<a href="{{url|{{defaultpage}}}}">Home page</a>
```

Note the use of `defaultpage` to get the specification of the default page (i.e. the home page.)

An optional second argument is the *style name*, if you're using styles (see chapter 10.)

If you're using HTTP GET parameters, these will be added on to any URLs created with the `url` tag. If you don't want this, use the `urlnoget` tag instead, which discards all of these. You will then have to hand on any tags you want to keep explicitly.

The `link` tag is also useful. It takes the page specifier, a link text and an optional style; and generates a full `A HREF` link tag.

10 Styles

A style is similar to a CSS media type — it's a code SCMS uses to identify classes of browser so you can change how you render sites to fit that browser or user preference. In SCMS you can specify the styles yourself, or rely on SCMS to do it for you.

10.1 Getting the style

The style is determined in one of three ways - an HTTP GET parameter, a user-written PHP hook, or SCMS' own function.

10.1.1 HTTP GET style

If the HTTP GET parameter `style` exists, its value is used as the current style. The other methods aren't used. Therefore the URL

```
http://mysite.com/index.php/home?style=print
```

will always set the `print` style.

10.1.2 The user-written hook

If there isn't an HTTP GET parameter called 'style', the system looks for the file `site/getstylename.php` to determine the style. This is a snippet of PHP in which you can check the values of the `$_SERVER` array, and set the value of `$stylename` accordingly. It should return 'default' as the default style.

As an example, there is a PHP library for mobile platform detection available from `http://detectmobilebrowsers.mobi/` which is free for non-commercial use only. Once included, you get a function which returns true for mobile browsers. You could implement a mobile style by putting the following into `site/getstylename.php`:

```
include('mobile_device_detect.php');

if(mobile_device_detect(true,true,true,true,true,true,false,false))
   $stylename='mobile';
else
   $stylename='default';
```

10.1.3 The default system

If there is no GET parameter and no user hook file, SCMS will fall back to the standard method of determining the style. This will produce the following style strings:

iphone	iPhone, iPad and iPod touch
textmode	Browsers such as Lynx
opera	Opera
firefox	Firefox
Safari	Safari on a desktop/laptop
ie8	MSIE 8
ie7	MSIE 7
ie6	MSIE 6
ie-old	Any older Internet Explorer
default	Anything else

10.2 Using the style: the `stylemap` file

The style name is available to SCMS via the `style` tag, but it's best to use it to change which template is used. This is done via the `site/stylemap` file.

Each line in the file has the form:

```
style/template => commands...
```

where `style` is the style to match, template is the template to match (or * to match any template) and commands is a list of commands as described below. When SCMS finds it's rendering a page with the given template and the style is set to the given style, the commands will be run.

The commands are:

- `template` *`name`* will change the template directory to be used
- `mainfile` *`file.html`* will change the main template file to be used

For example, if we have a page which contains the line

```
template=biglist
```

it will normally be rendered using the template in `site/templates/biglist`. If we want to change how it is rendered by an iPhone, the style map line

```
iphone/biglist => template iphonebiglist
```

will say that if the current style is `iphone` and we are going to use the template directory `biglist`, we should change the template to `iphonebiglist` before we proceed any further and use that directory instead.

Another example:

```
print/* => mainfile print.html
```

This says that whenever we have the `print` style, the main file should be changed from `main.html` to `print.html`. This will happen for every template, so every template directory must contain that new file.

10.3 Linking to another style

It's possible to generate a link from a page to another or the same page in a different style using the `link` or `url` tag. They're similar, but the former generates an entire `<A HREF>` HTML tag, while the latter generates just the URL (useful for image links). The `link` tag takes the page specifier, the link text and the style; while the `url` tag takes just the specifier and the style:

```
{{link|home|Home page in Print style|print}}
{{url|home|print}}
```

both create links to the home page in a print-friendly version (provided you've set up a style map and written a printer-friendly template!)

If you want to generate links to the current page in another style, you use the `spec` tag to fetch the current page's specifier:

```
{{link|{{spec}}|printer friendly|print}}
{{url|{{spec}}|print}}
```

You can leave out the style if you like, in which case they become handy tags for generating links around your site.

11 Support for Multilingual Sites

11.1 Setting up language support

To create a multilingual site, you'll need to copy — or better, link — the NLS files for the languages you require from the distribution's `NLSFiles` directory into your site's `languages` directory. You're site now supports those languages, but will use the default text when no language-specific data is given. In the examples below, we're going to add Welsh support — so we need to copy `cy_GB.nls.php` into the languages. The part before `.nls.php` is the *language ID,* in our case `cy_GB`.

Note that if you ever add a new language, you'll have to clear the cache by running the `clearcache` script in the SCMS directory, from your root directory.

11.2 Creating language-specific data

Multilingual versions of sites are created by writing special *language specific files,* which contain tag definitions overriding the default tag definitions provided in the default page and template data.

For example, consider a simple page file, `pages/home`, like this:

```
name=Home Page

content=[[
<img src={{imgroot}}foo.img/>
<p>{{page:text}}</p>
]]

text=[[
This is the page content.
]]
```

We can see that the page will consist of an image followed by some text — the text will come from the `page:text` tag, which is defined in the page.

We could make a Welsh language version of this page by creating a special *language directory* in your `pages` directory, with the same name as the language ID. For our example home page, that would be `pages/cy_GB/home`. This is a tag definition file containing any tags defined in the page which you want to be different in Welsh:

```
name=Tudalen Gartref
text=[[
Dyma'r cynnwys y dudalen.
]]
```

We don't need to write a definition for `content`, because we don't want to override that — the default version isn't language specific.

Now, when SCMS reads in that page, it will read in the default version (in `pages/home`) and then try to open the current language's version (`pages/cy_GB/home`). It will then read the tags from that file, which will supersede any in the default file. If no file is found, the default version is used in its entirety.

11.2.1 Language-specific template data

The same system works for template data too. Each of the `.tags` files inside the template directory can have some or all of its tag definitions overridden inside by a file of the same name in a language-specific subdirectory.

This works for the main file too: we might have a `main.html` file containing the overall structure of our side with some text embedded in it — perhaps for global links. We could create a language subdirectory in our template's directory, and put a language-specific `main.html` in there, and it would replace the default file when the given language was being requested[1].

11.3 Requesting a given language

When you go to the site's default URL you'll get all the pages in the default language — the language in which they are written in the top-level `pages` and `templates` directories. To access the pages in the new languages you'll need to add a *language navigation menu*.

[1]Of course, it's not a good idea to put blocks of text inside your main file, especially when they might require translation. Put the text inside `.tags` files and refer to them from inside the main file.

This is created from a tree collection, similar to a navigation menu (see Chapter 9) but much simpler, since it's not actually a tree – just a flat collection, where each node is an available language. The tree is generated automatically on demand from the NLS files in the `NLSFiles directory.` Here's an example of some code to render this menu:

```
langmenu=[[
{{treeprefix|default|<small>}}
{{treesuffix|default|</small>}}
{{treeselnode|default|{{a:endonym}} |}}
{{treeunselnode|default|<a href="{{a:url}}">{{a:endonym}} </a>|}}
{{treetrailnode|default|<a href="{{a:url}}">{{a:endonym}} </a>|}}
{{marktree|{{langtree}}|code|{{langcode}}}}
{{rendertree|{{langtree}}|a}}
]]
```

Let's unpick this:

- The `treeprefix` and `treesuffix` lines tell SCMS to render the entire menu in a small font. Typically you'll use a `div` here.

- The `treeselnode` line says that the currently selected node should be rendered as plain text, as just the *endonym* of the language: the name for that language in the language. Examples are "English," "Cymraeg," and "Deutsch."

- The `treeunsel` and `treetrailnode` lines say that the unselected nodes – that is, those nodes for the languages we aren't currently in – should be rendered as links whose text is the endonym, and whose URL is the URL of the list node. This will link to a version of the current page in the appropriate language for the node. The trail line probably isn't necessary but we add it for completeness.

- The `marktree` line say how the tree's nodes should be marked as selected, unselected or trail. This is done by looking for the node whose `code` field is equal to the currently selected language's code (specified in `langcode`.)

- Finally, `rendertree` renders the language tree (whose collection handle is returned from `langtree`) using the prefix `a`, which you'll note is used to access the fields in the other definitions.

11.3.1 Fields in a language tree node

The fields set up in a language tree node are:

- `url` is a URL which links to a version of the current page in the node's language.
- `code` is the language ID, such as `cy_GB`.
- `endonym` is the language's own name for itself, e.g. "Cymraeg," "Deutsch," or "English."
- `exonym` is the language's name in English[2], e.g. "Welsh," "German," or "English."
- `encoding` is the encoding required by the language (e.g. `UTF-8`.)

11.3.2 Using other encodings

For languages which require another encoding scheme, it's possible to get the encoding of the current language using the `langencoding` tag. It's also part of the tree node, although possibly not much use there.

[2]or the site's default language

12 Modules

Modules let you create extensions to SCMS using PHP. They can do two things:

- Register new tags, which can be strings, functions or collections.
- Provide PHP code for responding to HTTP GET and POST requests, which run code before the rest of the page is generated.

Naturally, you'll need to be fairly confident in PHP!

12.1 The modules directory

Modules reside in the `site/modules` directory, which you'll need to create if you want to use modules. Each module has its own directory named after the module. There are several modules provided in the main SCMS directory which you should link into your site's modules directory as you require them.

12.2 Loading modules

To find which modules to load, SCMS checks both the `page:modules` and `template:modules` tags and loads the modules specified in both. Each tag should be defined as a comma-separated list of dispositions and modules. The disposition is either 'post, 'get' or 'all'. If the disposition is 'post' or 'get' , the module is only loaded if it is requested via the POST or GET interface by setting the request parameter 'mod' to the module name. Modules marked 'all' are always loaded for the page or template. Each module file contains the definition of a module object, which it adds to Modules object.
An example:

```
all:foo,post:bar
```

This will always load the 'foo' module, but only load the 'bar' module if it is specifically requested by a POST setting `$_POST['mod']` to 'bar'.

12.3 Creating a module

To create a module, you need to create a directory for it and create a `main.php` file inside it. You can — and should, for large modules — create other files and `require` them if you wish. The main file should contain either or both:

- code to register tags,
- a subclass of `Module` which responds to POST and GET, and code to instantiate and register it with the module system.

If you're just going to register some new tags you don't need a Module object.

12.4 Registering tags

Here's a very simple module, which we'll call `ShowGenDate`. We create a directory of that name in the site modules directory, and write the `main.php` file:

```
<?php

$date = date(DATE_COOKIE);
Template::regstring('ShowGenDate:GenDate',$date,"date of page build");
```

And that's all: we have registered the tag `ShowGenDate:GenDate` to hold the string containing the time at which the code was run — i.e. when the page was last generated. The third argument is a description of the tag. Now, we can use this tag in any page:

```
content=[[
    Hello. Page content last built on
            {{ShowGenDate:GenDate}}
]]
```

Note the important convention of naming the tags defined by a module as

```
    ModuleName:TagName
```

This prevents namespace clashes, but it is only a convention! When you register tags in a module, you're using exactly the same functions the core system uses. In fact, reading through `core/functpls.php` can be useful for figuring out how to write tags — it defines the majority of the standard tags.

12.5 Function tags

Creating tags which actually do something is very easy too. Let's create a tag which just concatenates two strings together:

```
function MyModule_concat($t)
{
  $t->assertargc(2);
  $a = $t->argv[0]);
  $b = $t->argv[1]);
  return $a.$b;
}
Template::regfunc('MyModule:concat','join two strings','str1|str2',
    MyModule_concat);
```

This is how the function works:

- A `TagInText` structure is passed in, containing the arguments and the tag name. You can see more details of this in `core/template.php`.
- The argument count is checked, and the page render will abort if it isn't equal to 2. You can use the `argc` member directly if you want to do something smarter, such as variable arguments.
- Arguments 0 and 1 of the tag are read in.
- The arguments are joined together and returned.

If we create a `MyConcat` module and add the above code to its `main.php` we can now do

```
{{MyModule:concat|foo|bar}}
```

and get the correct answer

```
foobar
```

12.5.1 Processing tag arguments

However, there's a major problem:

```
{{MyModule:concat|{{langcode}}|{{langencoding}}}}
```

will give us

```
{{langcode}}{{langencoding}}
```

instead of the expected

```
en_USUTF-1
```

What's happened is that the arguments to our tag function have not, themselves, been processed by the templating system to expand any tags within them. Most tag functions (but not all) do this, so we'll modify our code to do it:

```
function MyModule_concat($t)
{
  $t->assertargc(2);
  $a = Template::process($t->argv[0]));
  $b = Template::process($t->argv[1]));
  return $a.$b;
}
Template::regfunc('MyModule:concat','join two strings','str1|str2',
    MyModule_concat);
```

Now it'll work.

One of the function tags which don't process its arguments is `treeselnode` (and its siblings.) Can you see why?

Not Yet Documented

`Module` subclasses which can respond to `GET` and `POST`

Not Yet Documented

special modules with no main (c.f. captchas)

Not Yet Documented

predefined modules

13 Other Tags

This chapter deals with tags which haven't been mentioned elsewhere, but which are often useful.

13.1 Numeric operations

Tags are available for performing simple arithmetic. If we call the arguments a, b and so on, and assume they are strings which evaluate to integers when passed through the template system, the operations are:

Tag	**Output**
`add|`a`|`b	$a + b$
`sub|`a`|`b	$a - b$
`mul|`a`|`b	$a * b$
`div|`a`|`b	a/b
`mod|`a`|`b	$a \bmod b$
`int|`a	a as integer, or zero if not parseable

13.2 Comparisons and conditions

The following tags are available for performing numeric and string comparisons, and for conditional code. They all have the same basic form: some arguments defining the condition, then two arguments for the outcome — the first is used if the condition is true, the second if the condition is false[1]. Here are the available conditionals:

[1]See page 70 for examples with `streq`.

Tag	**Result**
`streq`|a|b|c|d	c if a=b, otherwise d
`strieq`|a|b|c|d	c if lowercase(a)=lowercase(b), otherwise d
`cmpn`|`eq`|a|b|c|d	c if int(a)=int(b), otherwise d
`cmpn`|`neq`|a|b|c|d	c if int(a)$\neq$int(b), otherwise d
`cmpn`|`gt`|a|b|c|d	c if int(a)>int(b), otherwise d
`cmpn`|`lt`|a|b|c|d	c if int(a)<int(b), otherwise d
`cmpn`|`gte`|a|b|c|d	c if int(a)$\geq$int(b), otherwise d
`cmpn`|`lte`|a|b|c|d	c if int(a)$\leq$int(b), otherwise d
`if`|a|b|c	b if a is a non-zero integer, otherwise c
`ifnotempty`|a|b|c	b if a is not an empty string, otherwise c
`iftagexists`|a|b|c	b if a is a valid tag, otherwise c
`iftagtrue`|a|b|c	b if value of tag a is a non-zero integer, otherwise c
`useifexists`|a|b	value of a if a is a valid tag, otherwise b
`contains`|a|b|c|d	c if string a contains b, else d

Here's an example.

```
{{streq|foo|bar|equal|notequal}}
```

will display `notequal` because the first two strings are not equal. In all cases, all the arguments may contain tags. In the case of the outcomes, only the outcome actually used will be expanded. That means you can do things like:

```
{{streq|{{httpgetparam|wibble}}|yes|{{loadpage|foo|page}}|}}
```

This will cause the page `foo` to be loaded, all its definitions overloading the current definitions in the page[2], but only if the HTTP GET parameter 'wibble' has the value `yes`.

[2]because we're using the `page` prefix, which is the default prefix for page tags.

13.3 Logical operations

Typically for use with the `if` tag above, the following logical operations are available:

Tag	Result
`and`|a|b	$a \wedge b$
`or`|a|b	$a \vee b$
`not`|a	$\neg a$

13.3.1 Switch

The `switch` tag is SCMS' equivalent to other languages' *switch* or *case* construct. It uses a string comparison between a test string and several candidates, each of which is paired with an output string. If the test string matches a candidate, the corresponding output string is expanded and returned. Here's the syntax:

```
{{switch|teststring|
    candidate1|output1
    candidate2|output2
    candidate3|output3
    default|defaultoutput}}
```

The special condition string `default` matches anything, and is used to provide the default case. It must be the last item. An example of switch usage is on page 72, in the section dealing with invisible items in the navigation tree.

13.4 Numeric for loops

Numeric for loops, like those in BASIC, are done using `for` tag. They have the syntax:

```
{{for|counter tag name|start|end|
    template}}
```

SCMS will set the counter tag to the index and output the processed template for every value of the index between *start* and *end*. An example:

```
<ul>
{{for|i|1|100|
    <li>{{i}}</li>}}
</ul>
```

will produce a bulleted list of all the numbers between 1 and 100.

13.5 Text operations

13.5.1 Trimming whitespace

The `trim` tag is used to trim any whitespace from around a string:

```
<pre>Hello {{trim|    World   }}!</pre>
```

will print

```
Hello World!
```

Naturally whitespace is effectively trimmed automatically in HTML text, so if the `pre` HTML tag wasn't in the above example you would see the above output even if the `trim` tag wasn't there. Therefore, you might consider turning trimming on all the time for the output of all tag processing. To do this, use the `settrim` tag with a non-zero numeric argument. This will cause the PHP `trim` function to be called on all values returned from tags. Turn it off again by using `settrim` with zero as the argument.

13.5.2 Splitting text

The `splitstring` tag allows you to split a text into several substrings using a delimiter, put the substrings into different templates, and concatenate the results together. It's not to be confused with `split`, which also splits a string, but puts the results into a collection — see page 49 for more details.) The syntax is:

```
{{splitstring|mode|delimiter|string to split|
    template1|
    template2|
    template3|...}}
```

This is probably best expressed with an example:

```
{{splitstring|wrap| |foo bar baz blibble|
    {{splitvalue}} is the first string|
    {{splitvalue}} is the second string|
    {{splitvalue}} is the third string}}
```

will output

```
foo is the first string bar is the second string baz is the third string
blibble is the first string
```

For each substring, the tag `splitvalue` is set to the string and the tag `splitindex` is set to the (zero-based) index. The next template is picked from the list of templates and expanded with those tag values, and concatenated onto the output.

The mode determines what happens when the system runs out of templates for the substrings:

- In **wrap** mode, the system cycles back to the first template. This is shown in the example above, when the 'blibble' substring is output with the first template.
- In **clip** mode, the system uses the last template for all subsequent items. This would have resulted in

  ```
  foo is the first string bar is the second string
  baz is the third string blibble is the third string
  ```

- In **stop** mode, the system will not use any subsequent items. This would have resulted in the entry for 'blibble' not being added at all. This is the default mode, so any mode strings other than 'wrap' or 'stop' will cause this behaviour.

We can use `splitstring` to perform the same action on all substrings:

```
<ul>{{splitstring|clip| |My dog's got no nose|
    <li>{{splitindex}}: {{splitvalue}}</li>}}</ul>
```

will result in a bulleted list of the words in the phrase "My dog's got no nose" being output, each word being numbered from zero. Note that we cannot currently split on regular expressions. Using alternate templates is also easy:

```
{{splitstring|wrap| |My dog's got no nose|
    <b>{{splitvalue}} </b>|
    {{splitvalue}} }}
```

will result in every other word being in bold.

13.5.3 Encoding characters into HTML entities

The common task of turning special characters into HTML entities can be done using the `htmlentities` tag. For example,

```
{{htmlentities|"&<>}}
```

becomes

```
&quot;&amp;&lt;&gt;
```

13.6 Tag manipulation

13.6.1 The registry stack

All tag values are kept in the *tag registry.* You can save the state of the registry — and therefore the values of all tags — on to a stack by using the `push` tag, and then restore them by using the `pop` tag.

13.6.2 Loading tags from files

There are two tags provided for doing this. The first, `loadpage`, is used to load a page file (i.e. a tag definition file in the `site/pages` directory.) The syntax is:

```
{{loadpage|pagespec|prefix}}
```

The arguments are straightforward: *pagespec* is the page specification of the file to be loaded, and *prefix* is the prefix to be used in defining the tags, instead of the usual `page` used by the default page loader.

Note, however, that any `defaults` file will be loaded too: this is a full page load. This can cause odd behaviour. For example, if you define collections in your defaults page, they might end up being loaded twice, resulting in a repeated set of items being added[3].

A full example of this tag is given on page 76.

If you want to load tags from an arbitrary file, and not a page file, you can use the `load` tag:

```
{{load|prefix|dir|file}}
```

This will load a file from the given directory (relative to the `site` directory) into the template registry, prefixing all tags with the given prefix.

[3]It seems like an odd case, but I've just had it happen to me.

13.6.3 Debugging tag file loads

The `debugloads` tag sets whether tag file load debugging is on. For example,

```
{{debugloads|1}}
```

will turn it on. When it's on, a `<pre>` block will be written out to the page for every load, indicating the file loaded and the type of load.

13.7 URL and HTTP manipulation

13.7.1 Retrieving HTTP variables

This is done using the tags `httpgetparam` and `httppostparam`, which return GET and POST parameters respectively. They each take a single argument: the name of the required parameter.

13.7.2 URL construction and analysis

Several tags exist which help in analysing the URL by which the page was referenced, and to help to construct other URLs:

- `root` is a URL pointing to the root directory — the directory which contains `site`, `core` etc. This is useful for generating URLs to assets kept in subdirectories outside the normal SCMS tree.
- `imgroot` is a URL pointing to the `site/images` directory, so that commonly used images can be accessed by referring to them as (for example):

  ```
  <img src="{{imgroot}}image.jpg"/>
  ```

- `thispage` returns a URL which references the current page.
- `url|`*page*`|`*style* returns a URL for a page, given the page specifier and optionally the style. Useful for links within the site.
- `link|`*page*`|`*text*`|`*style* creates a link to a page, given the page specifier, the link text, and optionally the style.
- `defaultpage` is the page specification for the home page (as given in `config.php`.) The URL for the home page can be specification.

13.8 Changing the default time zone

It may be necessary to change the default time zone for SCMS. This is done by using the `settimezone` tag with a time zone string:

```
{{settimezone|Europe/London}}
```

SCMS uses PHP's `date_default_timezone_set()` for this functionality — see the documentation for that function to find out what the valid time zone strings are.

14 Handling 404s

If the user enters a request for a page which doesn't exist, a default 404 page will be returned. This is very brief, and rather ugly.

You can customise this by providing a file called 404 in the pages directory, just like any other page. Most users find it a good idea to create a 404 template as well, so that the 404 page has its own style.

In either case, whether you create a 404 page or use the standard one, the HTTP headers will be set correctly by SCMS to indicate the 404 status.

14.1 Example

Here's an minimal example of a 404 page, which would go into `site/pages/404`:

```
name=Page Not Found
template=404

message=[[
<p>
We're sorry, but the page you requested -
<b>{{thispage}}</b> - does not exist.
<p>The home page can be found <a href="{{defaultpage}}"
    accesskey=1>here</a>.
</p>
]]
```

Note that the template is set to `404`, which we'll need to provide.

And here's a template file, which would go into `site/templates/404/main.html`:

```
<html>
<title>{{page:name}}</title>
<link rel="start" title="Home Page, shortcut key=1"
    href="{{defaultpage}}">
</head>

<body>
<h1>{{page:name}}</h1>
{{page:message}}
</body></html>
```

Note that the `page:name` tag still has the name of the page which was requested, not that of the 404 page which was returned.

15 Page Caching

SCMS uses two caching strategies:

- In *server-side caching* SCMS stores each page it generates in a file, and returns the file if the page is requested again, rather than regenerating using the template engine. This process is very, very fast, reducing CPU usage; but has no effect on network usage since the same data is sent to the browser.
- In *client-side caching* SCMS forces the user's browser to store the page, so that if they request it again the system doesn't access the network at all, reducing network usage.

Neither of these options is any use for dynamic pages, and client-side caching may cause changes to pages to not be immediately reflected in what users see unless they explicitly refresh the page.

15.1 Enabling caching

Both kinds of caching are disabled by default, and can be enabled by modifying the following lines in `site/config.php`:

```
define('CACHING',0); // SERVER CACHING MODE
define('BROWSERCACHING',0); // CLIENT CACHING MODE
```

To enable server-side caching, set `CACHING` to 1. To enable client-side caching, set `BROWSERCACHING` to 1.

15.2 Server-side caching

As mentioned above, SCMS' server side caching involves storing the generated HTML for each page if it isn't in the cache, and serving that for subsequent requests.

First, a cache key is generated, using an hash based on the following items in the HTTP request:

- page name
- `page` GET variable
- `action` GET variable
- `c` GET variable
- `p` GET variable
- `feed` GET variable
- `tagid` GET variable
- `pagenumber` GET variable

15.2.1 Extra cache keys

You may need to add to this list — to do this, create a `$cacheitemlist` array in your `config.php` file:

```
$cacheitemlist = array('key','anotherkey','etc');
```

The cache is checked for a file whose name is the cache key. If this doesn't exist, the page contents are generated as usual, and stored in a file of that name before being returned to the client. If it does exist, the contents of that file are returned.

15.2.2 Clearing the cache

If you have server-side caching enabled you *must* clear the cache whenever you change a page to ensure that the updated version of the page is served. This is done by deleting all the cache files. A script called `clearcache` is provided to do this, which must be run from the top level of your SCMS installation. It will also clear the navigation and language cached data.

15.3 Client-side caching

Setting client-side caching on will cause the browser to cache pages it receives for up to a day. If the user requests the page again, they'll just get the version their browser has in its cache unless they do a refresh (i.e. use the F5 key in most browsers.) The disadvantage of this is that if you change a page and clear all your caches, the user isn't guaranteed to see it until a day has passed even if you turn caching off straight away. The advantage is that your network usage may be reduced.

15.4 Cache exceptions

You may have some pages which are dynamic, and therefore should never be cached by either server or client. The specifiers[1] of these pages can be added to the *cache exceptions list* which is in defined in `config.php`:

```
$cache_exceptions = array();
```

For example, you could change this line to:

```
$cache_exceptions = array('news','logs/log','admin','search');
```

to have those pages never be cached on the server or the client.

[1] i.e. the name of the page's file within `site/pages`, see Chapter 6.

16 Reference

16.1 Tag Definition File Syntax

This is a description of the syntax of tag files in BNF, using the following conventions:

- Optional items are enclosed in `[square brackets]`.
- Repeating items (i.e. $1..n$ copies) are enclosed in {`curly brackets`}.
- An optional repeating item (i.e. $0..n$ copies) is enclosed in both types of bracket.
- Terminals (i.e. literal strings) are enclosed in 'single quotes'
- `<string>` is a valid ASCII string containing tags (see Section 16.2, Tag Syntax)
- `<whitespace>` is a whitespace character, i.e. a space or tab

```
<file> ::= {[<definition>]}

<definition> ::= <multilinedef>|<singlelinedef>

<singlelinedef> ::= <tagname> {[<whitespace>]} '='
                    {[<whitespace>]} <string> '\n'

<multilinedef> ::= <tagname> {[<whitespace>]} '='
                    {[<whitespace>]} '[[' {[<whitespace>}] '\n'
                    {<string>} ']]' {[<whitespace>]} '\n'
```

16.2 Tag Syntax

Tags in strings have the following syntax (using the same BNF conventions as the previous section):

```
<tag> ::= '{' <tagname> [{ '|' <argument> }] '}'
<tagname> ::= {<alphanumeric character>}
<argument> ::= <string>
```

16.3 Tags

In the list below, any function tag arguments which are *not* processed through the templating system and so do *not* have any embedded tags expanded are marked with * .

Optional arguments are in brackets.

16.3.1 top level

add	add two numbers	**0**: val1 **1**: val2
and	logical AND operation:	**0**: val1 **1**: val2
cmpn	binary numeric comparision	**0**: condition **1**: val1 **2**: val2 **3**: use if true **4**: use if false
collsize	return the size of a collection	**0**: coll handle
contains	test if str1 contains str2	**0**: str1 **1**: str2 **2**: resultiftrue **3**: resultiffalse
count	counter function	**0**: prefix ***1**: command: start,ct,alt or tog **2**: options (see docs)
curnavnode	get the current node in the navigation tree	
debugloads	set whether load debugging is on	***0**: 0 or 1
defaultpage	the spec of default start page	
div	divide two numbers	**0**: val1 **1**: val2

dumpcoll	dump a collection as an HTML table	**0**: coll handle
dump	dump all templates	
findcollection	find collection in tree containing specified node	**0**: tree **1**: name of field **2**: value of field in required node
findnode	find a node given a field and string	**0**: tree **1**: name of field **2**: value of field in required node
foreach	iterate a collection	**0**: collection handle ***1**: prefix **2**: template
for	numeric for loop	**0**: var **1**: start **2**: end **3**: template
ftime	format the current time with strftime	***0**: format string
getfield	get value of a field in a node	**0**: node handle **1**: field name
htmlentities	encode html entities	**0**: string to encode
httpgetparam	get an HTTP GET parameter	**0**: parameter name
httppostparam	get an HTTP POST parameter	**0**: parameter name
iffieldexists	check a collection field exists	**0**: coll handle **1**: fieldname **2**: do this if true **3**: do this if false
iffieldset	test if a field has been set with a value	**0**: node **1**: field name **2**: useifset **3**: useifnotset
if	if arg0 is nonzero, use arg1 else use arg2	**0**: integer value **1**: useiftrue **2**: useiffalse
ifnotempty	returns argv[1] if the string argv[0] is longer than 0, else argv[2]	**0**: string **1**: use if longer than zero length **2**: use if zero length
iftagexists	is tag defined	**0**: name **1**: iftrue **2**: iffalse
iftagtrue	is tag exists and is a nonzero integer	***0**: name **1**: useiftrue **2**: useiffalse

imgroot	the URL for images (site/images)	
indexof	find the index of a tree node	**0**: node handle
int	return int value	**0**: string
langcode	current language code	
langencoding	current language encoding (e.g. UTF-8)	
langtree	get the language collection pseudotree	
link	a link to the specified page	**0**: spec **1**: linktext **(2)**: style
load	load a TDL file, defining templates with a given prefix	**0**: prefix **1**: dir with trailing / **2**: filename
loadpage	load a page definition file, using the tag prefix given to define the tags	**0**: page specifier **1**: prefix
marktree	set fields a hierarchy collection indicating selection	**0**: tree handle **1**: name of field **2**: value of field in selected item
mod	modulo two numbers	**0**: val1 **1**: val2
module:gethandled		
module:ifgethandledby	check a GET request was handled by a given module	**0**: module name **1**: use if get handled by this module **2**: use if not
module:ifposthandledby	check a POST request was handled by a given module	**0**: module name **1**: use if post handled by this module **2**: use if not
module:posthandled		
mul	mult two numbers	**0**: val1 **1**: val2
navlinks	accessibility link rel tag generator	**(0)**: startkey **(1)**: nextkey **(2)**: prevkey
navlinktemplate:accesskeyattr	collection value	
navlinktemplate:child	collection value	
navlinktemplate:coll	collection handle	

navlinktemplate:handle	collection item alias in walk	
navlinktemplate:index	collection item index	
navlinktemplate:key	collection value	
navlinktemplate:level	collection item level	
navlinktemplate:name	collection value	
navlinktemplate:parentc	collection parent handle	
navlinktemplate:parenti	collection parent node index	
navlinktemplate:spec	collection value	
navlinktemplate:type	collection value	
navlinktemplate:url	collection value	
navtree	get the navigation menu tree	
nl	newline	
not	NOT operation	**0**: value
or	logical OR operation	**0**: val1 **1**: val2
parentc	find parent collection of tree node	**0**: node handle
parent	find parent node of a tree node	**0**: node handle
parenti	find index of node of parent collection of tree node	**0**: node handle
pop	pop the template registry	
push	push the template registry	
rendertree	walk a tree with current walk renderer params	**0**: tree handle ***1**: prefix
root	the root URL	
scmstag		
setcoll	set a tag to alias to a collection	**0**: tagname **1**: coll handle
setitem	set a tag to alias to a collection item	**0**: tagname **1**: item handle
setnode	set a tag to alias to a collection item	**0**: tagname **1**: node handle
set	set a template value	**0**: name **1**: val **(*2)**: p

settimezone	set the time zone	**0**: time zone string
settrim	set whether whitespace is trimmed from tags	***0**: 0 or 1
spec	the specifier for the current page	
split	split a string by delimiter character, return a collection of strings	**0**: string ***1**: delimiter
splittext	split a string by delimiter and process each substring with different templates, concatenating the results	**0**: see definition
streq	case-dependent string compare	**0**: str1 **1**: str2 **2**: resultifsame **3**: resultifdifferent
strieq	case-independent string compare	**0**: str1 **1**: str2 **2**: resultifsame **3**: resultifdifferent
style	the current style (or empty for none)	
sub	subtract two numbers	**0**: val1 **1**: val2
switch	switch statement - compare a string with a set of values, substituting a corresponding string if a value matches	**0**: 0
templateroot	the URL for the current template	
thispage	the URL for the current page	
treeprefix	set the walk prefix for a level	***0**: level ***1**: text
treeselnode	set the selected walk node renderer for a level	***0**: level ***1**: pre-text ***2**: post-text
treesuffix	set the walk prefix for a level	***0**: level ***1**: text
treetrailnode	set the trail walk node renderer for a level	***0**: level ***1**: pre-text ***2**: post-text
treeunselnode	set the unselected walk node renderer for a level:	***0**: level ***1**: pre-text ***2**: post-text

trim	trim whitespace from a string	**0**: string
url	a URL to the specified page, preserving GET vars	**0**: spec **(1)**: style
urlnoget	a URL to the specified page, discarding GET vars	**0**: spec **(1)**: style
useifexists	if tag is defined, use it, otherwise an alternative string	**0**: tagname **1**: string to use if tag not defined
withnode	render a single node in a tree with a given template	**0**: node handle ***1**: prefix for field values ***2**: template to use

Glossary

`.tags` **file**	a tag definition language file in a template directory, so called because it's suffixed with `.tags` — these are read automatically, 37
`getstylename.php`	a file in `site` containing a hook which gets the style name, superceded SCMS' own facility, 91
`site`	the directory containing the site-specific files, 19, 23
`stylemap`	a file containing directions for converting a style name into changes to the template directory or main file, 93
breadcrumbs	a list of links to all the ancestor pages of the current page, telling the user where they are in the site, 76
canonical URL	A URL for a page containing the language code, e.g. `http://mysite.com/index.php/en_US/home`, 24
client caching	caching performed using the Expires and Cache-Control HTTP tags, managed by the browser, 113
collection	an array of data structures which can be created with tag definition language and accessed with tags, 43
complex collection	a collection containing any number of fields, any collection which is not a simple collection (q.v.), 45
config.php	A file containing definitions of site-specific values, 20
core directory	The directory containing all the SCMS source files, 19
createdirs	A UNIX shell script to create the `tmp` directory and set the permissions correctly, 19
function tag	A tag which invokes a function to get its replacement string, 30
handle	a string which refers to a collection or a tree, used in function tags which access collections and trees, 46
hook	a small snippet of user-written PHP which, if present, supercedes or adds to built in functionality, 91
item	a member of a collection, 43
main template file	the file suffixed `.html` in the current template directory which is used as the basis for the entire page. Sometimes called the *main file*, 35
navigation file	the `site/navigation` file containing a text representation of the site's structure, used to generate the navigation tree, 53, 67
navigation tree	a tree built automatically from the navigation file and used in producing the navigation menu, 53, 67
NLS file	A National Language System file describing a given language, 19
node	a member of a tree collection, 51
node rendering templates	template strings defined by special tags which define how a tree should be rendered, 56

page defaults files	files called `defaults` in the pages directory hierarchy, read automatically when a peer page file in the same directory or a descendant page file is read, 40
page file	a tag definition file in the `pages` directory defining tags giving the content of the current page which are used in the template, 39
page name	the tag `page:name`, defined in the page file, 40
page specifier	The part of the URL specifying the site page, e.g. `products/desks` in `http://mysite.com/index.php/products/desks`, 24
page tag	a tag defined in a page file; prefixed with `page:`, 39
redirection	using the `page:redir` tag to make a navigation menu link redirect to an external site, 40
server caching	caching done by SCMS to stop it from re-rendering a page unless it has actually changed, 113
style	a value such as `print` or `iphone` specifying device or user rendering preferences, 91
tag	A string of the form `{{foo}},`which can be expanded into a stored string value or the result of running a function associated with the tag, 25
tag definition language	A language which allows the user to define the values of tags, some of which can themselves be templates, 26
template	A string containing tags which can be expanded, **OR** a set of tags defined by files inside a template directory which define how a set of pages should be rendered and which is selected by the `page:template` tag in the page file, 25
template directory	a directory containing a set of tag definition files and a main HTML file, which together define how a set of pages should be rendered, 35
template tag	a tag defined in a `.tags` file in a template directory; prefixed with `template:`, 37
trail	the path from the root of a tree down to the currently selected node, 53, 55, 71
tree	a structure made up of collections which contain handles referring to sub-collections, and have a superset of a certain predefined set of fields., 51
tree collection	a single collection within a tree, 51
zebra striping	showing alternate lines of a list onn slightly different background colours for legibility, done using a `count` tag, 79

An index of tags and fields