1
CMS Core Design

This chapter is an overview of how a CMS is put together.

In the chapter we will discuss topics such as:

- How a CMS's publicly visible part (the "front-end") works
- Various ways that the administration part (the "admin area") can be created
- Discussion of files and database layout
- Overview of how plugins work

We will also build enough of the basics that we can view a "hello world" page, and detect missing pages as well.

This chapter will focus more on discussion than on practical examples, although we'll build a simple practical example at the end.

The "core" of a CMS is its architecture. Just as the motherboard of a computer is its most important component, without which the CPU, screen, RAM, and other parts cannot come together, the CMS core is the "backbone" of the CMS. It's what connects the database, browser interactions, and plugins together.

In this chapter, we will describe the various parts of that core, and over the next few chapters we will build up that core until we have a stable piece of software, upon which we can then start developing extensions (plugins).

If you don't want to type out the code to test it, you can download an archive of the completed project from the Packt website at `http://www.packtpub.com/support`.

This book's CMS is based on a previously written one called **WebME** (**Website Management Engine**), which has many more plugins written for it than are described in this book—you can download that version of the project here: `https://code.google.com/p/webworks-webme/`.

The CMS's private and public areas

A CMS consists of the management area (admin area), and the publicly visible area (front-end).

The front-end

One very interesting difference between CMS and non-CMS sites is their treatment of a "web page".

In a non-CMS website, when you request a certain URL from the web server, the web server sees if the requested file exists, and if it does, it returns it. Very simple.

This is because there is a very clear definition of what is a web page, and that is tied explicitly to the URL. `http://example.com/page1.html` and `http://example.com/page2.html` are two distinct web pages, and they correspond to the files `page1.html` and `page2.html` in the websites document root.

In a CMS, the definition might be a bit blurred. Imagine you are in a news section of the site at `http://example.com/news`, and this shows an overview of all news snippets on the website. This might be defined as a page.

Now let's say you "filter" the news. Let's say there are 60 news items, and only 20 are shown on the `/news` page. To view the next 20, you might go to `/news?page=2`.

Is that a different page? In a non-CMS site, certainly it would be, but in a database-backed CMS, the definition of a page can be a little more blurred.

In a CMS, the URLs `/news` and `/news?page=2` may not correspond exactly to two files on the server.

Because a CMS is database-backed, it is not necessary to have a separate physical source file for every page. For example, there is no need to have a `/news` file at all if the content of that page can be served through the root `/index.php` file instead.

When we create a new page in the administration area, there is a choice for the engine to either write a physical file that it corresponds to, or simply save it in the database.

A CMS should only be able to write to files that are in the public webspace under the strictest circumstances.

Instead of creating web pages as files, it is better to use a "controller" to read from a database, based on what the URL was. This reduces the need for the CMS to have write-permissions for the publicly visible part of the site, therefore increasing security.

There is a popular programming pattern called **MVC** (**Model-View-Controller**), which is very similar in principle to what a CMS of this type does.

In MVC, a "controller" is sent a request. This request is then parsed by the controller, and any required "model" is initialized and run with any provided data. When the model is finished, the returned data is passed through a "view" to render it in some fashion, which is then returned to the requester.

The CMS version of this is: The website is sent a HTTP request. This request is parsed by the CMS engine, and any required plugins are initialized and run with the HTTP parameters. Then the plugins are finished, they return their results to the CMS, which then renders the results using an HTML template, and sends the result of that back to the browser.

And a real-life example: The CMS is asked for `/news?page=2`. The CMS realizes `/news` uses the "news" plugin and starts that up, passing it the "page=2" parameter. The plugin grabs the information it needs from the database and sends its result back to the CMS. The CMS then creates HTML by passing it all through the template, and sends that back to the browser.

This, in a nutshell, is exactly how the public side (the front-end) of our CMS will work.

So, to rewrite this as an actual process, here is what a CMS does when it receives a request from a browser:

1. The web server sends the request to the CMS.
2. The CMS breaks the request down into its components—the requested page and any parameters.
3. The page is retrieved from the database or a cache.
4. If the page uses any plugins, then those plugins are run, passing them the page content and the request parameters.
5. The resulting data is then rendered into an HTML page through the template.
6. The browser is then sent the HTML.

This will need to be expanded on in order to develop an actual working demonstration. In the final part of this chapter, we will demonstrate the receiving of a request, retrieval of the page from the database, and sending that page to the browser. This will be expanded further in later chapters when we discuss templates and plugins.

The admin area

There are a number of ways that administration of the CMS's database can be done:

1. Pages could be edited "in-place". This means that the admin would log into the public side of the site, and be presented with an only slightly different view than the normal reader. This would allow the admin to add or edit pages, all from the front-end.
2. Administration can be done from an entirely separate domain (`admin.example.com`, for example), to allow the administration to be isolated from the public site.
3. Administration can be done from a directory within the site, protected such that only logged-in users with the right privileges can enter any of the pages.
4. The site can be administrated from a dedicated external program, such as a program running on the desktop of the administrator.

The method most popular CMSs opt for is to administrate the site from a protected directory in the application (option **3** in the previous list).

The choice of which method you use is a personal one. There is no single standard that states you must do it in any particular way. I opt for choice 3 because in my opinion, it has a number of advantages over the others:

1. Upgrading and installing the front-end and admin area are both done as part of one single software upgrade/installation. In options **2** and **4**, the admin area is totally separate from the front-end, and upgrades will need to be coordinated.
2. Keeping the admin area separate from the front-end allows you to have a navigation structure or page layout which is not dependent on the front-end template's design. Option **1** suffers if the template is constrained in any way.
3. Because the admin area is within the directory structure of the site itself, it is accessible from anywhere that the website itself is accessible. This means that you can administrate your website from anywhere that you have Internet access.

In this book, we will discuss how a CMS is built with the administration kept in a directory on the site.

For consistency, even though it is possible to write multiple administrative methods, such as administration remotely through an RPC API as well as locally with the directory-based administration area, it makes sense to concentrate on a single method. This allows you to develop new features quicker, as you don't need to write administrative functions twice or more, and it also removes problems where a change in an API might be corrected in one place but not another.

Plugins

Plugins are the real power behind how a CMS does its thing. Because every site is different, it is not practical to write a single monolithic CMS which would handle absolutely everything, and the administration area of any site using such a CMS would be daunting—you would have extremely complex editing areas for even the most simple sites, to cater for all possible use cases.

Instead, the way we handle differences between sites is by using a very simple core, and extending this with plugins.

The plugins handle anything that the core doesn't handle, and add their own administration forms.

We will discuss how plugins work later on, but for now, let's just take a quick overview.

There are a number of types of plugins that a site can use. The most visible are those which change a page's "type".

A "default" or "normal" page type is one where you enter some text in the admin area, and that is displayed exactly as entered, on the front-end.

An example of how this might be changed with a plugin is if you have a "gallery" plugin, where you choose a directory of images in the admin area, and those images are displayed nicely on the front-end.

In this case, the admin area should look very different from the front end.

How this case is handled in the admin area is that you open up the gallery page, the CMS sees that the page type is "gallery" and knows that the gallery plugin has an admin form which can be used for this page (some plugins don't), and so that form is displayed instead of the normal page form.

On the front-end, similarly, the CMS sees that the page requested is a "gallery" type page, and the gallery plugin has a handler for showing page data a certain way, and so instead of simply printing the normal body text, the CMS asks the plugin what to do and does that instead (which then displays a nice gallery of images).

A less obvious plugin might be something like a logger. In this case, the log plugin would have a number of "triggers", each of which runs a function in the log plugin's included files. For example, the `onstart` trigger might take a note of the start time of the page load, and the `onfinish` trigger might then record data such as how long it took to load the page (on the server-side), how much memory was used, how large the page's HTML was, and so on.

Download from Wow! eBook <www.wowebook.com>

Another word for **trigger** is **event**. The words are interchangeable. An event is a well-established word in JavaScript. It is equivalent to the idea of triggers in database terminology. I chose to use the word trigger in this book, but they are essentially the same.

With this in mind, we know that the 6-point page load flow that we looked at in the *The front-end* section is simplistic—in truth, it's full of little trigger-checks to see when or if plugins should be called.

Files and databases

In this section, we will discuss how the CMS files and database tables should be laid out and named.

Directory structure

Earlier in the chapter, I gave an example URL for a news page, `http://example.com/news`. One thing to note about this is that there is no "dot" in it. The non-CMS examples all ended in `.html`, but there's no "**.whatever**" in this one.

One reason this is very good is that it is human-readable. Saying "*www dot my site dot com slash news slash the space book*" is a lot easier than saying something like "*www dot my site dot com slash index dot p h p question-mark page equals 437*".

It's also useful, in that if you decide to change your site in a few years to use a totally different CMS or even programming language, it's easier to reconcile `/news` on the old system with `/news` on the new one than to reconcile `/index.php?id=437` with `/default.asp?pageid=437`—especially if there are external sites that link to the old page URL.

In the CMS we are building, we have two file reference types:

1. References such as `/news` or `/my/page` are references to pages, and will be displayed through the CMS's front controller. They do not exist as actual files on the system, but as entries in the database.
2. Anything with a dot in it is a reference to an actual real file, and will not be passed to the front controller. For example, something like `/f/images/test.jpg` or `/j/the-script.js`.

This is managed by using a web server module called `mod_rewrite` to take all HTTP requests and parse them before they're sent to the PHP engine.

In our CMS, we will keep the admin area in a directory called `/ww.admin`. The reason for this is that the dot in the directory name indicates to the web server that everything in that directory is to be referenced as an actual file, and not to be passed to the front controller. The "ww" stands for "Webworks WebME", the name of the original CMS that this book's project is based on. You could change this to whatever you want. WordPress' admin area, for example, is at `/wp-admin`.

If the directory was just called `/admin` and you had a page in your CMS also called "admin", then this would cause ambiguity that we really don't want.

Similarly, if you have a page called "about-applicationname-3.0", this would cause a problem because the application would believe you are looking for a file.

The simplest solution is to ban all page names that have dots in them, and to ensure that any files specific to your CMS all have dots in them. This keeps the two strictly apart.

Another strategy is to not allow page names which are three characters or less in length. This then allows you to use short names for your own purposes. For example, using "`/j/`" to hold all your JavaScript files. Single-letter directories can be considered bad-practice, as it can be hard to remember their purpose when there is more than one or two of them, so whether you use `/j` and `/f`, or `/ww.javascript` and `/ww.files` is up to you.

So, application-specific root directories in the CMS should have a dot in the name, or should be three characters or less in length, so they are easily distinguishable from page names.

The directory structure that I use from the web root is as follows:

```
/               # web root
/.private       # configuration directory
/ww.admin       # admin area
/ww.cache       # CMS caches
/f              # admin-uploaded file resources
/i              # CMS images
/ww.incs        # CMS function libraries
/j              # CMS JavaScript files
/ww.php_classes # CMS PHP class files
/ww.plugins     # CMS plugins
/ww.skins       # templates
```

There are only two files kept in the web root. All others are kept in whichever directory makes the most sense for them.

The two files in the web root are:

- `index.php`—this file is the front-end controller. All page requests are passed through this file, and it then loads up libraries, plugins, and so on, as needed.
- `.htaccess`—this file contains the `mod_rewrite` rules that tell the web server how to parse HTTP requests, redirecting through `index.php` (or other controllers, as we'll see later) or directly to the web server, depending on the request.

The reason I chose to use short names for `/f`, `/i`, and `/j`, is historical—up until recently, broadband was not widely available. Every byte counted. Therefore, it made sense to use short names for things whenever possible. It's a very minor optimization. The savings may seem tiny, but when you consider that "smartphones" are becoming popular, and their bandwidth tends to be Edge or 3G, which is much slower than standard broadband, it still makes sense to have a habit of being concise in your directory naming schemes.

Database structure

The database structure of a simple CMS core contains only a few tables.

You need to record information about the pages of the website, and information about users such as administrators.

If any plugins require tables, they will install those tables themselves, but the core of a CMS should only have a few tables.

Here's what we will use for our initial table list:

- pages—this table holds information about each page, such as name, id, creation date, and so on.
- user_accounts—data about users, such as e-mail address, password, and so on.
- groups—the groups that users can be assigned to. The only one that we will absolutely need is "_administrator", but there are uses for this which we'll discuss later.

For optimization purposes, we should try to make as few database queries as possible. This will become obvious when we discuss site navigation in *Chapter 3, Page Management – Part One*, where there are quite a lot of queries needed for complex menus.

Some CMSes record their active plugins and other settings in the database, but it is a waste to use a database to retrieve a setting that is not likely to change very often at all, and yet is needed on every page.

Instead, we will record details of active plugins in the config file.

The configuration file

A **configuration file** (**config file**) is needed so that the CMS knows how to connect to the database, where the site resources are kept, and other little snippets of information needed to "bootstrap" the site.

The config file also keeps track of little bits of information which need to be used on every page, such as what plugins are active, what the site theme is, and other info that is rarely changed (if ever) and yet is referred to a lot.

The config file in our CMS is kept in a directory named `/.private`, which has a `.htaccess` file in it preventing the web server from allowing any access to it from a browser.

The reason the directory has a dot at the front, instead of the usual "ww." prefix, is that we don't want people (even developers!) editing anything in it by hand, and files with a dot at the front are usually hidden from normal viewing by FTP clients, terminal views, and so on.

It's really more of a deterrent than anything else, and if you really feel the need to edit it, you can just go right in and do that (if you have access rights, and so on).

There are two ways a configuration file can be written:

- **Parse-able format**. In this form, the configuration file is opened, and any configuration variables are extracted from it by running a script which reads it.
- **Executable format**. In this form, the configuration file is an actual PHP script, and is loaded using `include()` or `require()`.

Using a parseable file, the CMS will be able to read the file and if there is something wrong with it, will be able to display an error on-screen. It has the disadvantage that it will be re-parsed every time it is loaded, whereas the executable PHP form can be compiled and cached by an engine such as Zend, or any other accelerator you might have installed..

The second form, executable, needs to be written correctly or the engine will break, but it has the advantages that it doesn't need to be parsed every time, if an accelerator is used, and also it allows for alternative configuration settings to be chosen based on arbitrary conditions (for example, setting the theme to a test one if you add `?theme=test` to the URL).

Hello World

We've discussed the basics behind how a CMS's core works. Now let's build a simple example.

We will not bother with the admin area yet. Instead, let's quickly build up a visible example of the front-end.

I'm not going to go very in-depth into how to create a test site—as a developer, you've probably done it many times, so this is just a quick reminder.

Setup

First, create a database. In my example, I will use the name "cmsdb" for the database, with the username "cmsuser" and the password "cmspass".

You can use phpMyAdmin or some other similar tool to create the database. I prefer to do it using the MySQL console itself.

```
mysql> create database cmsdb;
Query OK, 1 row affected (0.00 sec)

mysql> grant all on cmsdb.* to cmsuser@localhost identified by
'cmspass';
Query OK, 0 rows affected (0.00 sec)

mysql> flush privileges;
Query OK, 0 rows affected (0.00 sec)
```

Now, let's set up the web environment.

Create a directory where your CMS will live. In my case, I'm using a Linux machine, and the directory that I'm using is `/home/kae/websites/cms/`. In your case, it could be `/Users/~yourname/httpd/site` or `D:/wwwroot/cms/`, or whatever you end up using. In any case, create the directory. We'll call that directory the "web root" when referencing it in the future.

Add the site to your Apache server's `httpd.conf` file. In my case, I use virtual hosts so that I can have a number of sites on the same machine. I'm naming this one "cms":

```
<VirtualHost *:80>
    ServerName cms
    DocumentRoot /home/kae/websites/cms
</VirtualHost>
```

Restart the web server after adding the domain.

Note that we will be adding to the `httpd.conf` later in this chapter. I prefer to show things in pieces, as it is easier to explain them as they are shown.

And now, make sure that your machine knows how to reach the domain. This is easy if you're using a proper full domain like "myexample.cms.com", but for test sites, I generally use one-word domain names and then tell the machine that the domain is located on the machine itself.

To do this in Linux, simply add the following line to the `/etc/hosts` file on your laptop or desktop machine:

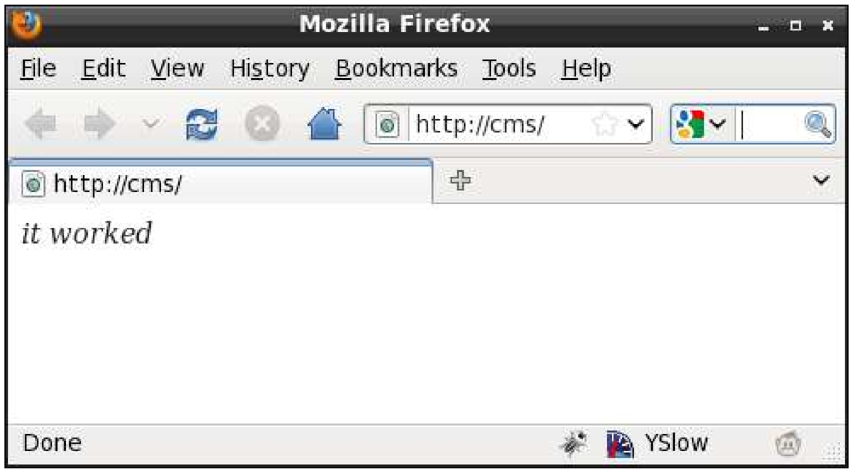
```
127.0.0.1 cms
```

Note that this will only work if the test server is running on the machine you are testing from (for example, I run my test server on my laptop, therefore `127.0.0.1` is correct). If your test server is not the machine you are browsing on, you need to change `127.0.0.1` to whatever the machine's IP address is.

To test this, create an `index.html` file in the web root, and view it in your browser:

```
<html>
  <body>
    <em>it worked<em>
  </body>
</html>
```

And here is how it looks:



it worked

If you have all of this done, then it's time to create the Hello World example.

We'll discuss writing an installer in the final chapter. This chapter is more about "bootstrapping" your first CMS. In the meantime, we will do all of this manually.

In your web root, create a directory and call it `.private`. This directory will hold the config file.

Create the file `.private/config.php` and add a basic config (tailored to your own settings):

```
<?php
$DBVARS=array(
  'username'=>'cmsuser',
  'password '=>'cmspass',
  'hostname'=>'localhost',
  'db_name' =>'cmsdb'
);
```

This will be expanded throughout the book as we add new capabilities to the system. For now, we only need database access.

Note that I didn't put a closing `?>` in that file. A common problem with PHP (and other server-side web languages) happens if you accidentally output empty space to the browser before you are finished outputting the headers. As we are building a templated CMS, all output should happen right at the end of the PHP script, when we're sure we're done compiling the output.

If you place `?>` terminators at the ends of your files, it's easy to accidentally also place invisible break-lines (`\n`, `\r`) as well. Removing the `?>` removes that problem as well. There is no right or wrong here. PHP is perfectly happy with files that end or don't end with `?>`, so it is up to you whether you do so.

We don't want people looking into the `.private` directory at all, so we will add a file, `.private/.htaccess`, to deny read-access to everyone:

```
order allow,deny
deny from all
```

Note that in order for `.htaccess` files to work, you must enable them in your web-server's configuration.

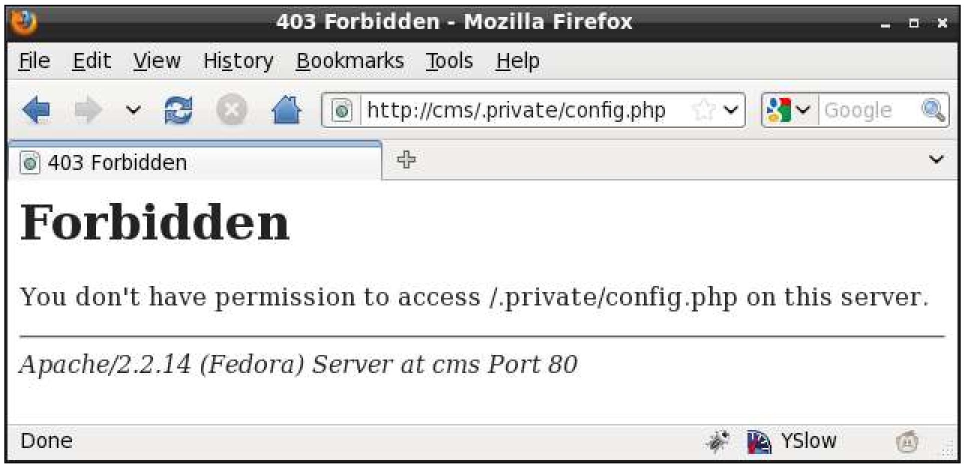
The simplest way to do this is to set `AllowOverride` to `all` in your Apache configuration file for the web directory, then restart the server.

An example using my own setup is as follows:

```
<Directory "/home/kae/websites">
    Options All
    AllowOverride All
    Order allow,deny
    Allow from all
</Directory>
```

You can tune this to your own needs by reading the Apache manual online.

After doing this and restarting your web server, you will find that you can load up `http://cms/` but you can't load up `http://cms/.private/config.php`.


403 Forbidden - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://cms/.private/config.php
Google
403 Forbidden
Done
YSlow

Forbidden

You don't have permission to access /.private/config.php on this server.

Apache/2.2.14 (Fedora) Server at cms Port 80

Next, let's start on the front controller.

Front controller

If you remember from what we discussed earlier, when a page is requested from the CMS, it will go through the front-end controller, which will figure out what kind of page it is, and render the appropriate HTML to the browser.

Note that although we are using a front controller, we are not using true MVC. True MVC is very strict about the separation of the content, the model, and the view.

This is easy enough to manage in small coding segments, but when combining HTML, JavaScript, PHP, and CSS, it's a lot more tricky.

Throughout the book, we will try to keep the various parts separate, but given the choice between complex or verbose code and simple or short code, we will opt for the simple or short route.

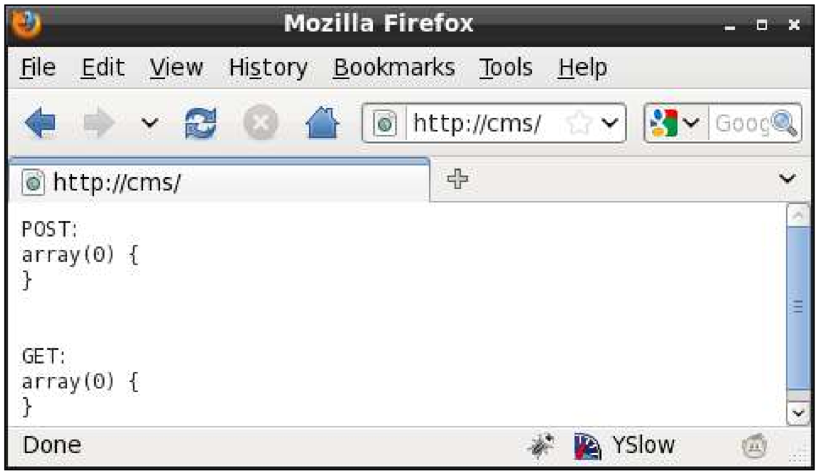
Some CMSes prefer to use URLs such as `http://cms/index.php?page=432`, but that's ugly and unreadable to the casual viewer.

We will do something similar, but disguise it such that the end-user doesn't realize that's basically what's happening.

First off, delete the test `index.html`, and create this file as `index.php`:

```
<?php
header('Content-type: text/plain');
echo "POST:\n";
var_dump($_POST);
echo "\n\nGET:\n";
var_dump($_GET);
```

That displays any details that are sent to the server through `POST` or `GET`:



```
POST:
array(0) {
}


GET:
array(0) {
}
```

Now, let's do the redirect magic.

Create a `.htaccess` file in the web root:

```
<IfModule mod_deflate.c>
  SetOutputFilter DEFLATE
</IfModule>

php_flag magic_quotes_gpc off

RewriteEngine on
RewriteRule ^([^./]{3}[^.]*)$ /index.php?page=$1 [QSA,L]
```

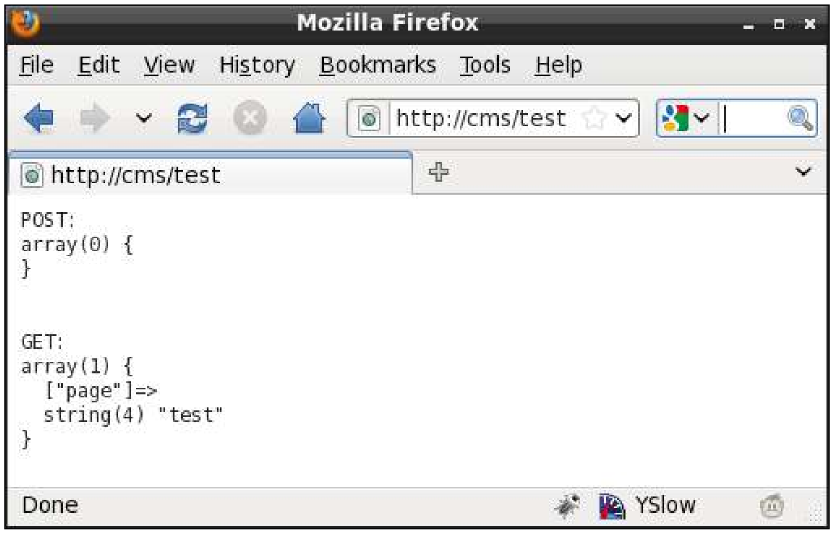
The `mod_deflate` bit compresses data as it is sent (if `mod_deflate` is installed).

We turn off "magic quotes" if they're enabled. Magic quotes are an old deprecated trick used by early PHP versions to allow HTTP data to be used in strings on the server without needing to properly escape them. This causes more problems than it solves, so it is being removed from later PHP versions.

The rewrite section takes any page name requests which are three or more characters in length and do not contain dots, and redirects those to `index.php`. The `QSA` part tells Apache to also forward any query-string parts, and the `L` tells Apache that if this rule matches, then don't process any more.

You can test that now.

Open your browser and go to `http://cms/test`, and you should see the following output:

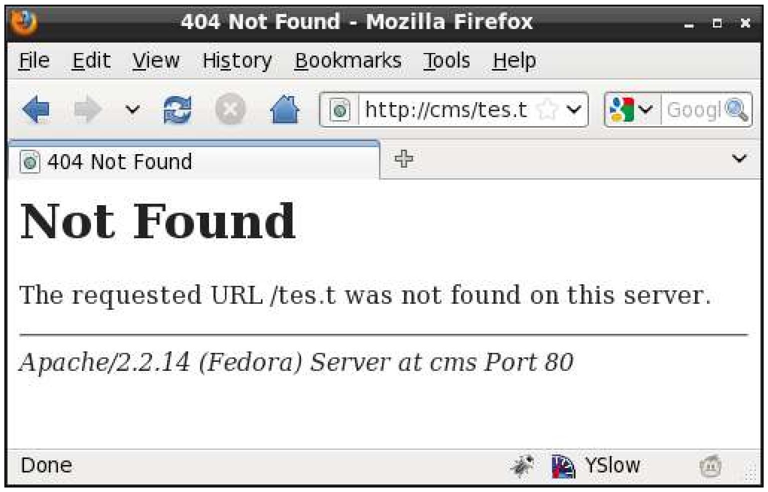


```
POST:
array(0) {
}


GET:
array(1) {
  ["page"]=>
  string(4) "test"
}
```

Notice the `GET` array now has the page name, which we can use in the next section to retrieve data from the database.

And if you put in a dot, you should get a standard **404** message:


404 Not Found - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://cms/tes.t
Googl
404 Not Found
Done
YSlow

Not Found

The requested URL /tes.t was not found on this server.

Apache/2.2.14 (Fedora) Server at cms Port 80

We will discuss proper handling of 404 pages in *Chapter 3, Page Management – Part One*.

Reading page data from the database

Okay—now that we can tell the CMS what page we're looking for, we need to write code that will use that information and retrieve the right data from the database.

First, let's create the "pages" table in the database. Use your MySQL console or phpMyAdmin to run the following:

```
CREATE TABLE `pages` (
  `id` int(11) NOT NULL auto_increment,
  `name` text,
  `body` mediumtext,
  `parent` int(11) default '0',
  `ord` int(11) NOT NULL default '0',
  `cdate` datetime default NULL ,
  `special` bigint(20) default NULL,
  `edate` datetime default NULL,
  `title` text,
  `template` text,
  `type` varchar(64) default NULL,
  `keywords` text,
  `description` text,
  `associated_date` date default NULL,
    `vars` text,
    PRIMARY KEY  (`id`)
  )  DEFAULT CHARSET=utf8;
```

This is the most important table of the database. The various parts of it are:

Name	Description
`id`	The ID of the page in the database. Must be unique. This is an internal reference.
`name`	When a URL `http://cms/page_name` is called, 'page_name' is what's searched for in the database.
`body`	This is the main HTML of the page.
`parent`	In a hierarchical site, this references the 'parent' of the page. For example, in the URL `http://cms/deep/page`, the 'page' entry's parent field will be equal to the 'deep' entry's ID.
`ord`	When pages are listed, in what position of the list will this page be shown.
`cdate`	Date that the page was created on.
`special`	This is used to indicate various 'special' aspects about a page—such as whether the page is the site's home page, or is a site map, or is a 404 handler, and so on. These are details that are important enough that they should be built into the core instead of as a plugin.
`edate`	Date that the page was last edited on.
`title`	This is shown in the browser's window header. When you search online and find pages titled "Untitled Document", it's because the author didn't bother changing this.
`template`	Which template (of the site skin) should this page use. We'll see how this is used in a later chapter.
`type`	Type of page is this. For now, we won't use this, but it becomes important once we start using plugins.
`keywords`	This is used by search engines.
`description`	Again, used by search engines.
`associated_date`	Pages sometimes need to have a date associated with them. An example is a news page, where the associated date may not be the created or last-edited date.
`vars`	This is a 'miscellaneous' field, where plugins that need to add values to the page can add them as a JSON object.

We'll discuss these further throughout the book. For now, we are more concerned with simply installing a single page.

Download from Wow! eBook <www.wowebook.com>

Insert two rows into the database:

```
mysql> insert into pages (name,body,special,type)
  values('Home','<p>Hello World</p>',1,0);
Query OK, 1 row affected (0.00 sec)

mysql> insert into pages (name,body,special,type)
  values('Second Page','<p>A Second Page</p>',0,0);
Query OK, 1 row affected (0.00 sec)
```

For the purposes of this test, we install two pages. The first one, "Home", has its `special` field set to `1`, which means "this is the home page". This means that if the website is called without any particular page requested, then this page will be used (in other words, we want `http://cms/` to equate to `http://cms/Home`).

In both cases, we set the `type` field to `0`, meaning "normal". When we add plugins later, this field will become important.

There are four files involved in displaying the pages.

- `/index.php`: This is the front-end controller. It receives the request, loads up any required files, and then displays the result.
- `/ww.incs/common.php`: This is a list of common functions for displaying pages. For this demo, all it will do is load `basics.php`.
- `/ww.incs/basics.php`: A list of functions common to all CMS actions. Includes database access and the setting up of basic variables.
- `/ww.php_classes/Page.php`: The `Page` class loads up page data from the database.

The process flow is as follows:

1. `index.php` is called by the `mod_rewrite` script.
2. `index.php` then loads up `common.php` which also loads `basics.php`.
3. `index.php` initializes the page, causing `Page.php` to be loaded.
4. `index.php` then displays the body of the loaded page.

Create this file as `index.php` in the web root:

```
<?php
// { common variables and functions
include_once('ww.incs/common.php');
$page=isset($_REQUEST['page'])?$_REQUEST['page']:'';
$id=isset($_REQUEST['id'])?(int)$_REQUEST['id']:0;
// }
// { get current page id
if(!$id){
  if($page){ // load by name
    $r=Page::getInstanceByName($page);
    if($r && isset($r->id))$id=$r->id;
    unset($r);
  }
  if(!$id){ // else load by special
    $special=1;
    if(!$page){
      $r=Page::getInstanceBySpecial($special);
      if($r && isset($r->id))$id=$r->id;
      unset($r);
    }
  }
}
// }
// { load page data
if($id){
  $PAGEDATA=(isset($r) && $r)? $r : Page::getInstance($id);
}
else{
  echo '404 thing goes here';
  exit;
}
// }

echo $PAGEDATA->body;
```

This is a simplified version of what we'll have later on. Basically, we check to see if the page ID is mentioned in the URL. If not, we load up the page using its name (through the `Page` object) to figure out the ID.

When we have the page data imported into the `$PAGEDATA` variable, we simply render it to the screen.

The `ww.incs/common.php` file is pretty bare at the moment:

```
<?php
require dirname(__FILE__).'/basics.php';
```

That will include common functions to do with page display. For now, all it does is load up the `ww.incs/basics.php` file:

```
<?php
session_start();
```

```
function __autoload($name) {
  require $name . '.php';
}
function dbInit(){
  if(isset($GLOBALS['db']))return $GLOBALS['db'];
  global $DBVARS;
  $db=new PDO('mysql:host='.$DBVARS['hostname']
    .';dbname='.$DBVARS['db_name'],
    $DBVARS['username'],
    $DBVARS['password']
  );
  $db->query('SET NAMES utf8');
  $db->num_queries=0;
  $GLOBALS['db']=$db;
  return $db;
}
function dbQuery($query){
  $db=dbInit();
  $q=$db->query($query);
  $db->num_queries++;
  return $q;
}
function dbRow($query) {
  $q = dbQuery($query);
  return $q->fetch(PDO::FETCH_ASSOC);
}
define('SCRIPTBASE', $_SERVER['DOCUMENT_ROOT'] . '/');
require SCRIPTBASE . '.private/config.php';
if(!defined('CONFIG_FILE'))
  define('CONFIG_FILE',SCRIPTBASE.'.private/config.php');
set_include_path(SCRIPTBASE.'ww.php_classes'
  .PATH_SEPARATOR.get_include_path());
```

First, we start off a session to record any data which may need to be passed from page to page.

Next, we set an auto-load function so that we can use objects without explicitly needing to `require()` their files. You can see that in action in the `index.php` where we used the `Page` object despite it not being explicitly included.

Next, we have three helper functions for databases. Because connecting to a database takes up precious resources, it is a waste of time to connect to the database upon every single request to the server. And so we connect only when the first database request is called, and cache that connection for the rest of the script.

Next, we define a few constants:

- `SCRIPTBASE`: This is the directory that the CMS is located in
- `CONFIG_FILE`: This is the location of the configuration file

There will be a few more constants later when we get to themes and uploadable files.

Finally, we have the `ww.php_classes/Page.php` class file:

```
<?php
class Page{
  static $instances         = array();
  static $instancesByName   = array();
  static $instancesBySpecial= array();
  function __construct($v,$byField=0,$fromRow=0,$pvq=0){
    # byField: 0=ID; 1=Name; 3=special
    if (!$byField && is_numeric($v)){ // by ID
      $r=$fromRow?
        $fromRow:
        ($v?
          dbRow("select * from pages where id=$v limit 1"):
          array()
        );
    }
    else if ($byField == 1){ // by name
      $name=strtolower(str_replace('-','_',$v));
      $fname='page_by_name_'.md5($name);
      $r=dbRow("select * from pages where name like '"
        .addslashes($name)."' limit 1");
    }
    else if ($byField == 3 && is_numeric($v)){ // by special
      $fname='page_by_special_'.$v;
      $r=dbRow(
        "select * from pages where special&$v limit 1");
    }
    else return false;
    if(!count($r || !is_array($r)))return false;
    if(!isset($r['id']))$r['id']=0;
    if(!isset($r['type']))$r['type']=0;
    if(!isset($r['special']))$r['special']=0;
    if(!isset($r['name']))$r['name']='NO NAME SUPPLIED';
    foreach ($r as $k=>$v) $this->{$k}=$v;
    $this->urlname=$r['name'];
    $this->dbVals=$r;
    self::$instances[$this->id] =& $this;
```

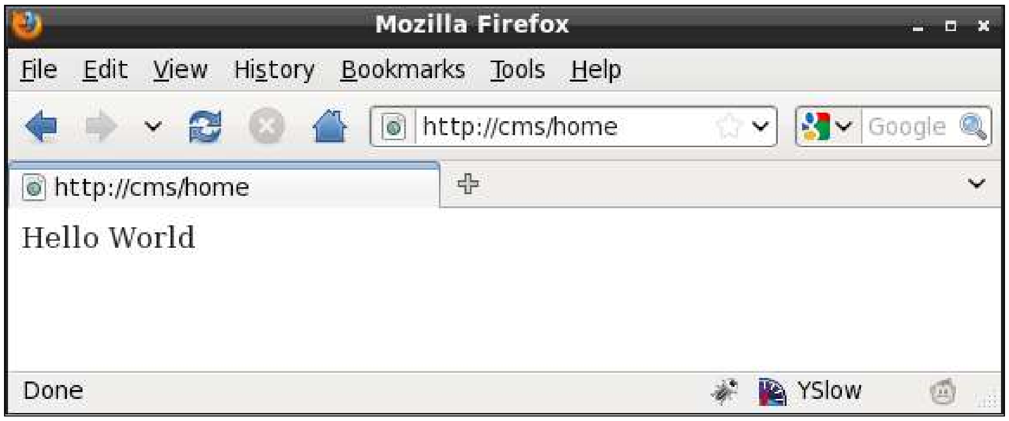
```
      self::$instancesByName[preg_replace(
        '/[^a-z0-9]/','-',strtolower($this->urlname)
      )] =& $this;
      self::$instancesBySpecial[$this->special] =& $this;
      if(!$this->vars)$this->vars='{}';
      $this->vars=json_decode($this->vars);
    }
    function getInstance($id=0,$fromRow=false,$pvq=false){
      if (!is_numeric($id)) return false;
      if (!@array_key_exists($id,self::$instances))
        self::$instances[$id]=new Page($id,0,$fromRow,$pvq);
      return self::$instances[$id];
    }
    function getInstanceByName($name=''){
      $name=strtolower($name);
      $nameIndex=preg_replace('#[^a-z0-9/]#','-',$name);
      if(@array_key_exists($nameIndex,self::$instancesByName))
        return self::$instancesByName[$nameIndex];
      self::$instancesByName[$nameIndex]=new Page($name,1);
      return self::$instancesByName[$nameIndex];
    }
    function getInstanceBySpecial($sp=0){
      if (!is_numeric($sp)) return false;
      if (!@array_key_exists($sp,$instancesBySpecial))
        $instancesBySpecial[$sp]=new Page($sp,3);
      return $instancesBySpecial[$sp];
    }
  }
```

This may look complex at first glance, but it's not all that bad.

There are three methods, `getInstance`, `getInstanceByName`, and `getInstanceBySpecial`, each of which finds the requested page using its own method:

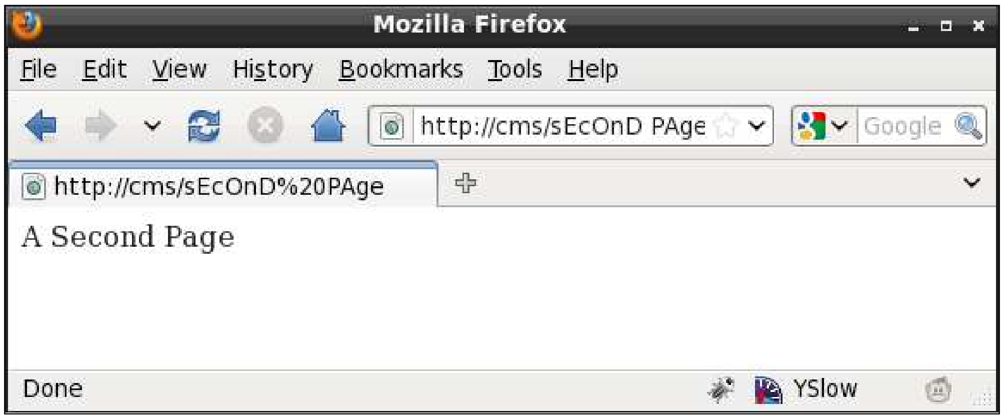
- `getInstance` is used if you know the ID of the page.
- `getInstanceByName` is used if you know the name of the page. We'll expand this later to include hierarchical names such as "`/sub/page/one`".
- `getInstanceBySpecial` is used if there's no particular page requested, but it's a special case. For example, the front page has the value 1. This is recorded as a bit mask, so for example, if a page is both the front page and a sitemap (shown later), then it would be recorded as 3, which is 1 plus 2 (values of Home Page and Sitemap respectively).

With this code in place, you can now load up pages. Here's an example using the page name "Home", as seen in the next screenshot:



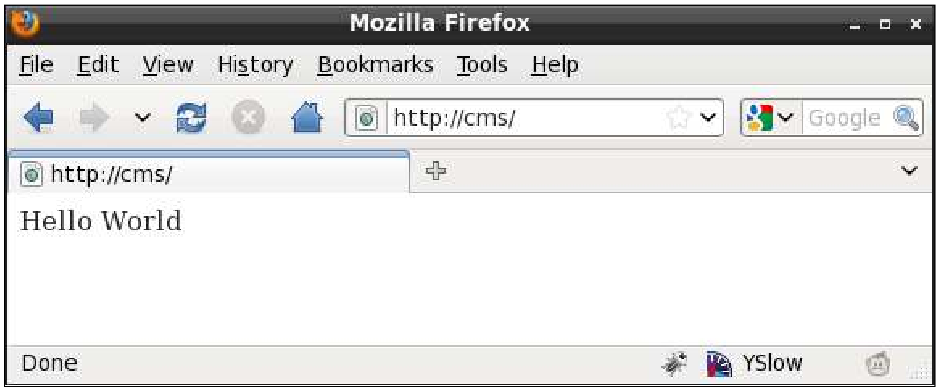
Hello World

Notice that the request uses the lower-case **home** instead of the upper-case "Home". Because MySQL is case-insensitive by default, and humans tend to not care whether something is upper-case or lower-case, it makes sense to allow any case to be used at all in the page name, as seen in the next screenshot:



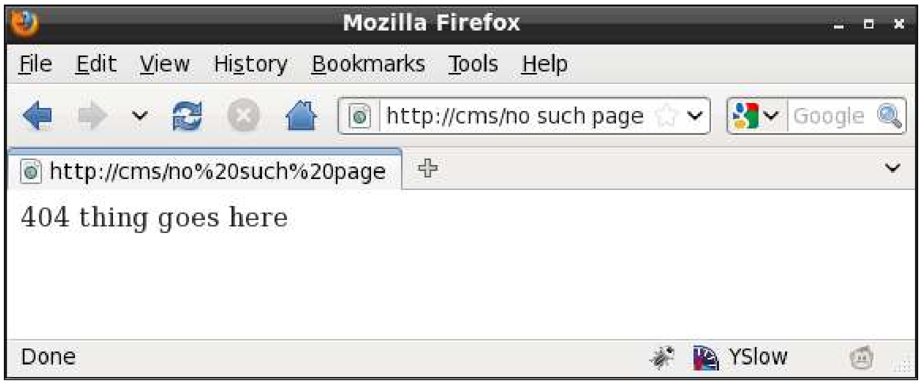
A Second Page

And in the case that no page name is given at all, the `index.php` file will load up using the special "home page" case:



Hello World

And finally, in the case that a page simply doesn't exist at all, we are able to trap that, as seen in the next screenshot:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://cms/no such page
Google
http://cms/no%20such%20page
Done
YSlow

404 thing goes here

Because we can trap this 404, we can do some interesting things such as show a list of possible matches that the reader can then choose from. This won't be handled in this book, but feel free to either redirect to the root or a search page, or any other solution you want.

Summary

In this chapter, we looked at how a CMS works, and built enough of the basics that we could then view a "Hello World" page, in a few different ways, with 404s trapped as well.

In the next chapter, we will discuss how users and groups work, to allow granular permissions, and we will build a login script, including forgotten password functionality and captchas.