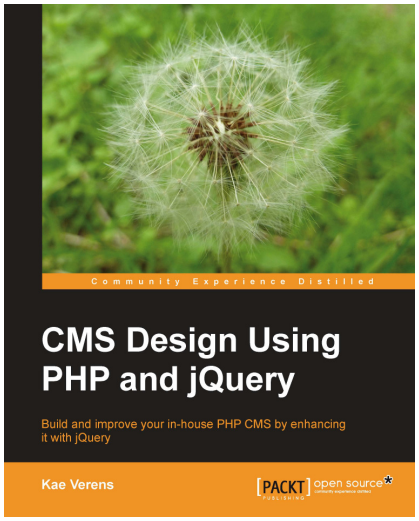
[PACKT] open source
community experience distilled
PUBLISHING

CMS Design Using PHP and jQuery

Kae Verens



Chapter No. 7
"Plugins"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "Plugins"

A synopsis of the book's content

Information on where to buy this book

About the Author

Kae Verens lives in Monaghan, Ireland with his wife Bronwyn and their two kids Jareth and Boann. He has been programming professionally for more than half his life.

Kae started writing in JavaScript in the nineties and started working on server-side languages a few years later. After writing CGI in C and Perl, Kae switched to PHP in 2000, and has worked with it since.

Kae worked for almost ten years with Irish web development company Webworks before branching out to form his own company KV Sites (`http://kvsites.ie/`) a small company which provides CMS and custom software solutions, as well as design, e-mail, and customer support.

Kae wrote the Packt book *jQuery 1.3 with PHP*, which has since become a part of his company's in-house training. Outside of programming, Kae is currently planning a book on budget clavichord design and building, and is the author of the online instructional book *Kae's Guide to Contact Juggling*, available here: `http://tinyurl.com/kae-cj-book`.

Kae is currently the secretary of the Irish PHP Users' Group, `http://php.ie/`, is the owner of the Irish web development company kvsites.ie, `http://kvsites.ie/`, and is the author of popular web-based file manager KFM, `http://kfm.verens.com/`

This is Kae's second book for Packt, having written *jQuery 1.3 with PHP in 2009*.

In his spare time, Kae plays the guitar and piano, likes to occasionally dust the skateboard off and mess around on it, and is studying Genbukan Ninjutsu.

CMS Design Using PHP and jQuery

PHP and jQuery are two of the most famous open source frameworks used for web development. This book will explain how to leverage their power by building a core CMS which can be used for most projects without needing to be written, and how to add custom plugins that can then be tailored to the individual project.

This book walks you through the creation of a CMS core, including basic page creation and user management, followed by a plugin architecture, and example plugins. Using the methods described in this book, you will find that you can create distinctly different websites and web projects using one codebase, web design templates, and custom-written plugins for any site-specific differences. Example code and explanation is provided for the entire project.

This book describes how to use PHP, MySQL, and jQuery to build an entire CMS from the ground up, complete with plugin architecture, user management, template-driven site design, and an installer. Each chapter walks you through the problems and solutions to various aspects of CMS design, with example code and explanation provided for the chosen solutions. A plugin architecture is explained and built, which allows you to enhance your own CMS by adding site-specific code that doesn't involve "hacking" the core CMS.

By the end of this book, you will have developed a full CMS which can be used to create a large variety of different site designs and capabilities.

What This Book Covers

Chapter 1, CMS Core Design, discusses how a content management system works, and the various ways to administrate it, followed by code which allows a page to be retrieved from a database based on the URL requested.

Chapter 2, User Management, expands on the CMS to build an administration area, with user authentication, and finish with a user management system, including forgotten password management, and captchas.

Chapter 3, Page Management – Part One, discusses how pages are managed in a CMS, and will build the first half of a page management system in the administration area.

Chapter 4, Page Management – Part Two, finishes off the page management system in this chapter, with code for rich-text editing, and file management.

Chapter 5, Design Templates – Part One, focuses on the front-end of the site by discussing how Smarty works. We will start building a templates engine for providing cdesign to the front-end, and a simple navigation menu.

Chapter 6, Design Templates – Part Two, improves on the navigation menu we started in the previous chapter by adding some jQuery to it, and will finish up the templating engine.

Chapter 7, Plugins, discusses how plugins work, and we will demonstrate this by building a plugin to handle page comments.

Chapter 8, Forms Plugin, improves on the plugin architecture by building a forms plugin. The improvements allow entirely new page types to be created using plugins.

Chapter 9, Image Gallery Plugin, an image gallery plugin is created, showing how to manage the uploading and management of images.

Chapter 10, Panels and Widgets – Part One, describes how panels and widgets work. These allow for extremely flexible templates to be created, where non-technical administrators can "design" their own page layouts.

Chapter 11, Panels and Widgets – Part Two, finishes up the panels system by creating a Content Snippet widget, allowing HTML sections to be placed almost anywhere on a page, and even select what pages they appear on.

Chapter 12, Building an Installer, shows how an installer can be created, using virtual machines to help test the installer.

7
Plugins

In this chapter, we will enhance the CMS engine so it can use plugins or external code modules, which can be "plugged" into the engine to add new abilities to it.

This chapter will include the following topics:

- What are plugins and triggers and why must a CMS handle them
- The creation of the plugin architecture
- Enabling plugins
- Handling of plugin database tables and upgrades
- Creating an example plugin, Page Comments

After completing this chapter, the CMS could be considered "complete", in that almost every other requested feature can be supplied by writing a plugin for it.

However, it should be noted that a CMS never is actually complete, because each new website may bring a new request that is not yet catered for.

Having said that, using plugins lets you at least complete a "core" engine and concentrate on providing hooks that allow further development to be done, outside that core.

What are plugins?

A **plugin** is a module of code that can be dropped into a directory and enabled, to give a CMS extra capabilities.

Plugins need to be able to change the output and do other tasks, so it is necessary to add various "hooks" throughout the code where the plugins can apply their code.

A very important reason for adding a plugin architecture to a CMS is that it lets you stabilize the core code. The **core** is basically the code that will be available in every instance of the CMS, as opposed to plugin code, which may or may not be present in any particular instance of the CMS.

With a core piece of code that is deemed "complete", it becomes easier to manage bugs. Because you are not always adding to the core code, you are not actively adding to the potential number of bugs.

In a CMS which does not have a stable core, any change to the central code can affect just about anything else.

You really need to get your CMS to a stage where you are no longer developing the central engine. Instead, you are working mostly on external plugins and maybe occasional bug fixes to the core, as they are found.

In my case, for example, I worked for years on building up a CMS before getting around to building in plugins. Every change that was requested was built into the core code. Usually, only the fully-tested code at that time would be the new code, so very often we would miss a problem that the new code would have caused somewhere else in the CMS. Often, this problem would not show up for weeks, so it would not be obvious what the problem was related to!

When all the development of a CMS is shifted to plugins, it becomes less likely that the core is at fault when a problem occurs. Because plugins, by their nature, tend to be isolated pieces of code, if a bug does appear, it is very likely the bug is within the plugin's code and not anywhere else.

Also, because plugins allow a person to develop without touching the core engine, it is possible for the external teams or individuals to create their own plugins that they can use with the engine, without needing to understand all the parts of the core engine.

One more advantage is that if the plugin architecture is solid, it is possible for development to continue on the core completely separately from the plugins, knowing that plugins from one version of the CMS will most likely work with a core from another version.

Events in the CMS

One example of a hook is event triggers.

In JavaScript (and therefore jQuery), there is the concept of **events**, where you can set a block of code to run when a certain trigger happens.

For example, when you move your mouse over an element, there are a number of potential trigger points—`onmouseover`, `onmouseenter`, `onmousemove` (and possibly others, depending on the context).

Obviously, PHP does not have those events, as it's a server-side language. But it is possible to conceive of triggers for your CMS that you could potentially hook onto.

For example, let's say you've just finished figuring out the page content. At this point, you may want to trigger a `page-content-created` event. This could (and will, in this chapter) be used by a **Page Comments** plugin to tack on the comments thread, and any required forms, to the end of that page content.

Another example: Let's say you want to create a custom log for your own purposes. You would then be interested in a **start trigger** that can be used to initialize certain values, such as a timer. After the output has been sent, a **finish trigger** that can be used to tally up a number of figures (compilation time, memory used, size of rendered output, and so on) and record them in a file or database before the script finishes.

Page types

In some cases, you will want the page content to be totally converted. Instead of showing a page body as normal, you may want to show an image gallery or a store checkout.

In this case, you would need to create a "page type" block of code, which the front-end will use instead of the usual page data `render()` call.

In the admin area, this might also require using a customized form instead of the usual rich text editor.

Admin sections

The admin area may need to have new sections added by a plugin. In the Events section, we described a logging plugin. A perfect complement to that is a graphing log viewer, which would be shown as a completely new admin section and have its own entry in the admin menu.

Page admin form additions

You may also want to add extra forms to all the Page forms in the admin, regardless of what page type it is. For example, if you create a security plugin and want to protect various pages depending on who is viewing it, you will need to be able to choose which users or groups have access and what to display if the current user does not have full access. This requires an additional form in the Page admin.

It is very difficult to describe all the possible plugin uses, and the number of triggers that may be required.

The easiest way to proceed is to just adjust the engine as required. If it turns out you forgot to add an event trigger at some point, it should be a small matter to just add it in at that point without affecting the core code beyond that addition.

Example plugin configuration

Create a directory called `/ww.plugins`.

Each plugin you create will be placed in a directory—one directory per plugin.

For our first example, we're going to build a Page Comments plugin, which will allow visitors to your site to leave comments on your pages.

On the admin side, we will need to provide methods to maintain the submitted comments per page and for the whole site.

Anyway, create a directory to hold the plugin called `/ww.plugins/page-comments`.

The CMS will expect the plugin configuration for each plugin to be in a file named `plugin.php`. So the configuration for the Page Comments plugin `/ww.plugins/page-comments/plugin.php` is as follows:

```
<?php
$plugin=array(
  'name' => 'Page Comments',
  'description' => 'Allow your visitors to comment on pages.',
  'version' => '0',
  'admin' => array(
    'menu' => array(
      'Communication>Page Comments' => 'comments'
    ) ,
    'page_tab' => array(
      'name' => 'Comments',
      'function' => 'page_comments_admin_page_tab'
    )
  ),
  'triggers' => array(
    'page-content-created' => 'page_comments_show'
  )
);
```

The `plugin.php` files at least contain an array named `$plugin`, which describes the plugin.

We will expand on the possible configurations of this array throughout the book. For now, let's look at what the current example says. All of these options, except the first two, are optional.

First, we define a name, "Page Comments". This is only ever used in the admin area, when you are choosing your plugins. The same is true of the description field.

The version field is used by the CMS to tell whether a plugin is up-to-date or if some automatic maintenance is needed. This will be explained in more detail later in this chapter.

Next, we have the admin array, which holds details of the admin-only functions.

The menu array is used to edit the admin menu, in case you need to add an admin section for the plugin. In this case, we will add an admin section for Page Comments, which will let you set site-wide settings and view comments site-wide.

If a new tab is to be added to the page admin section, this tab is described in the `page_tab` array. `name` is what appears in the tab header, and `function` is the name of a PHP function that will be called to generate the tab content.

Finally, the triggers array holds details of the various triggers that the plugin should react to. Each trigger calls a function.

Obviously, this is not a complete list, and it is not possible to ever have a complete list, as each new circumstance you are requested to write for may bring up a need for a trigger or plugin config setting that you had not thought of.

You will see as we go through the book that we add on new settings as we go. However, you should also note that as we get closer to the end of the book, there are less and less additions, as the plugin architecture becomes more complete.

From the plugin configuration, you can see that there are some functions named, which we have not defined.

You should define those functions in the same file:

```
function page_comments_admin_page_tab($PAGEDATA){
  require_once SCRIPTBASE.'ww.plugins/page-comments/'
      .'admin/page-tab.php';
  return $html;
}
function page_comments_show($PAGEDATA){
  if(isset($PARENTDATA->vars->comments_disabled) &&
      $PARENTDATA->vars->comments_disabled=='yes')
    return;
  require_once SCRIPTBASE.'ww.plugins/page-comments/'
      .'frontend/show.php';
}
```

The functions are prefixed with an identifier to make sure that they don't clash with the functions from other plugins. In this case, because the plugin is named Page Comments, the prefix is `page_comments_`.

The functions here are essentially stubs. Plugins will be loaded every time any request is made to the server. Because of this, and the obvious fact that not all the functions would be needed in every request, it makes sense to keep as little code in it as possible in the `plugin.php` files.

In most cases, triggers will be called with just the `$PAGEDATA` object as a parameter. Obviously, in cases in the admin area where you're not editing any particular page this would not make sense, but for most plugins, to keep the function calls consistent, the only parameter is `$PAGEDATA`.

Enabling plugins

We have defined a plugin. We could make it such that when you place a plugin in the `/ww.plugins` directory, it is automatically enabled. However, if you are creating a CMS that you intend to reuse for a lot of other clients, it is a lot easier to simply copy the entire CMS source and reconfigure, than to copy the CMS source and then clear out the existing plugins and repopulate carefully with new ones that you would download from a repository that you keep somewhere else.

So, what we do is we give the admin a maintenance page where they choose the plugins they want to load. The CMS then only loads those and does not even look at the other directories.

Edit the `/ww.admin/header.php` file and add a new link (highlighted) to the plugin admin section:

```
<li><a href="/ww.admin/themes.php">Themes</a></li>
<li><a href="/ww.admin/plugins.php">Plugins</a></li>
<li><a href="/ww.incs/logout.php?redirect=/ww.admin/"
    >Log Out</a></li>
```

We will be changing the admin menu later in this chapter to make it customizable more easily, but for now, add in that link manually.

Now create the `/ww.admin/plugins.php` file:

```
<?php
require 'header.php';
echo '<h1>Plugin Management</h1>';
echo '<div class="left-menu">';
echo '<a href="/ww.admin/users.php">Users</a>';
```

```
echo '<a href="/ww.admin/themes.php">Themes</a>';
echo '<a href="/ww.admin/plugins.php">Plugins</a>';
echo '</div>';

echo '<div class="has-left-menu">';
echo '<h2>Plugin Management</h2>';
require 'plugins/list.php';
echo '</div>';

require 'footer.php';
```

You'll have noticed that this is similar to the `/ww.admin/themes.php` and `/ww.admin/users.php` files. They're all related to site-wide settings, so I've placed links to them all in the left-menu. Edit those files and add in the new Plugins link to their menus.

Before we create the page for listing the enabled plugins, we must first set up the array of enabled plugins in `/ww.incs/basics.php`, by adding this to the end of the file:

```
// { plugins
$PLUGINS=array();
if (isset($DBVARS['plugins'])&&$DBVARS['plugins']) {
  $DBVARS['plugins']=explode(',',$DBVARS['plugins']);
  foreach($DBVARS['plugins'] as $pname){
    if (strpos('/',$pname)!==false) continue;
    require SCRIPTBASE . 'ww.plugins/'.$pname.'/plugin.php';
    $PLUGINS[$pname]=$plugin;
  }
}
else $DBVARS['plugins']=array();
// }
```

As you can see, we are again referencing the `$DBVARS` array in the `/.private/config.php`.

Because we already have a function for editing that (`config_rewrite()`, created in the previous chapter), all we need to do to change the list of enabled or disabled plugins, and create and maintain the `$DBVARS['plugins']` array, making sure to resave the config file after each change.

What the code block does is that it reads in the `plugin.php` file for each enabled plugin, and saves the `$plugin` array from each file into a global `$PLUGINS` array.

The `$DBVARS['plugins']` variable is an array, but we'll store it as a comma-delimited string in the config file. Edit `config_rewrite()` in the same file and add this highlighted line:

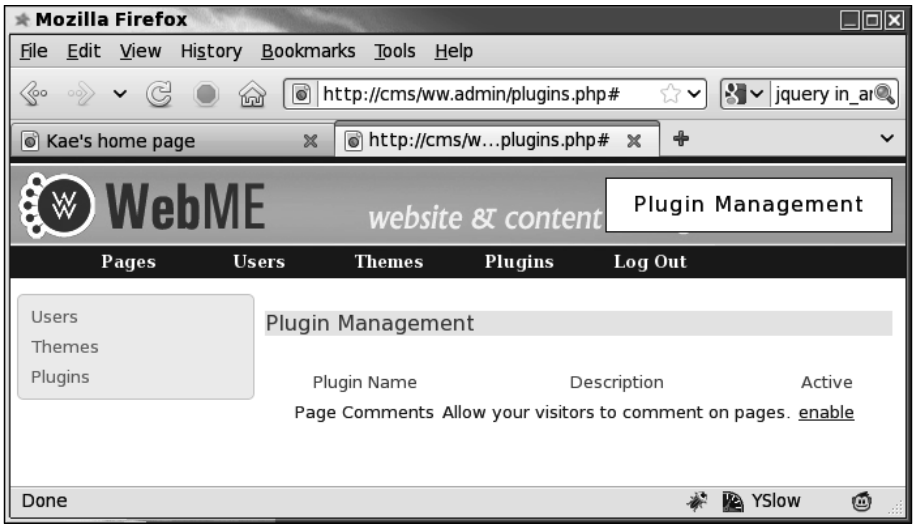
```
$tmparr=$DBVARS;
$tmparr['plugins']=join(',',$DBVARS['plugins']);
$tmparr2=array();
```

We'll enhance the plugin loader in a short while. In the meantime, let's finish the admin plugin maintenance page.

Create the directory `/ww.admin/plugins`, and in it, add `/ww.admin/plugins/list.php`:

```
<?php
echo '<table id="plugins-table">';
echo '<thead><tr><th>Plugin Name</th><th>Description</th>
    <th> </th></tr></thead><tbody>';
// { list enabled plugins first
foreach($PLUGINS as $name=>$plugin){
  echo '<tr><th>',htmlspecialchars(@$plugin['name']),'</th>',
    '<td>',htmlspecialchars(@$plugin['description']),'</td>',
    '<td><a href="/ww.admin/plugins/disable.php?n=',
         htmlspecialchars($name),'">disable</a></td>',
    '</tr>';
}
// }
// { then list disabled plugins
$dir=new DirectoryIterator(SCRIPTBASE . 'ww.plugins');
foreach($dir as $plugin){
  if($plugin->isDot())continue;
  $name=$plugin->getFilename();
  if(isset($PLUGINS[$name]))continue;
  require_once(SCRIPTBASE.'ww.plugins/'.$name.'/plugin.php');
  echo '<tr id="ww-plugin-',htmlspecialchars($name),
       '" class="disabled">',
    '<th>',htmlspecialchars($plugin['name']),'</th>',
    '<td>',htmlspecialchars($plugin['description']),'</td>',
    '<td><a href="/ww.admin/plugins/enable.php?n=',
        htmlspecialchars($name),'">enable</a></td>',
    '</tr>';
}
// }
echo '</tbody></table>';
```

When viewed in a browser, it displays like this:



The script displays a list of already-enabled plugins (we have none so far), and then reads the `/ww.plugins` directory for any other plugins and adds them along with an "enable" link.

Now we need to write some code to do the actual selection/enabling of the plugins.

While it would be great to write some jQuery to do it in an Ajaxy way (so you click on the **enable** link and the plugin is enabled in the background, without reloading the page), there are too many things that might cause problems. For instance, if the plugin caused new items to appear in the menu, we'd have to handle that. If the plugin changed the theme, or did anything else that caused a layout change, we'd have to handle that as well.

So instead, we'll do it the old-fashioned PHP way—you click on **enable** or **disable**, which does the job on the server, and then reloads the plugin page so you can see the change.

Create the `/ww.admin/plugins/enable.php` file:

```
<?php
require '../admin_libs.php';
if(!in_array($_REQUEST['n'],$DBVARS['plugins'])){
  $DBVARS['plugins'][]=$_REQUEST['n'];
  config_rewrite();
}
header('Location: /ww.admin/plugins.php');
```

It simply adds the requested plugin to the `$DBVARS['plugins']` array, then rewrites the config and redirects the browser back to the plugins page.

When clicked, the page apparently just reloads, and the plugin's link changes to **disable**.

The opposite script is just as simple. Write this code block in the file `/ww.admin/plugins/disable.php`:

```
<?php
require '../admin_libs.php';
if(in_array($_REQUEST['n'],$DBVARS['plugins'])){
  unset($DBVARS['plugins'][
    array_search($_REQUEST['n'],$DBVARS['plugins'])
  ]);
  config_rewrite();
}
header('Location: /ww.admin/plugins.php');
```

In this case, all we needed to do was to remove the plugin name from `$DBVARS['plugins']` by unsetting its position in the array.

Plugins are now very simply set up. Here's a screenshot of that page with a number of plugins enabled and disabled. I copied some plugins from a more mature copy of the CMS that I have. We won't cover all of them in this book, but will be looking at a few of them, and building one or two others:

The enabled plugins are moved to the top of the list to make them more visible and the rest are shown below them.

Handling upgrades and database tables

Plugins frequently require database tables to be created or amended.

Because we are not doing a traditional installation when we install a plugin and simply clicking on **enable**, the CMS needs to know if anything needs to be done to the database (or other things, as we'll see).

For this, the CMS needs to keep a record of what version of the plugin is installed.

The way I handle upgrades in the CMS is that there are two copies of the plugin version numbers. One is kept in the `$DBVARS` array and another is kept hardcoded in the plugin's `$plugin` array.

If there is a discrepancy between the two, for example, if you've simply never used the plugin before or if you downloaded a later version of the plugin that has a different version number, you know an upgrade needs to be done.

I'll explain as we create the upgrade architecture. First, edit `/ww.incs/basics.php` and add the following highlighted lines to the plugins section:

```
require SCRIPTBASE . 'ww.plugins/'.$pname.'/plugin.php';
if(isset($plugin['version']) && $plugin['version'] && (
    !isset($DBVARS[$pname.'|version'])
    || $DBVARS[$pname.'|version']!=$plugin['version']
)){
  $version=isset($DBVARS[$pname.'|version'])
    ?(int)$DBVARS[$pname.'|version']
    :0;
  require SCRIPTBASE.'ww.plugins/'.$pname.'/upgrade.php';
  $DBVARS[$pname.'|version']=$version;
  config_rewrite();
  header('Location: '.$_SERVER['REQUEST_URI']);
  exit;
}
$PLUGINS[$pname]=$plugin;
```

How it works is that if the `$plugin` version is greater than 0 and either the `$DBVARS`-recorded version doesn't exist or is not equal to the `$plugin` version, we run an `upgrade.php` script in the plugin's directory and then reload the page.

Note the `$DBVARS[$pname.'|version']` variable name. In the Page Content plugin's case, that will be `$DBVARS['page-content|version']`.

Even if a plugin is eventually disabled, we don't clear that value. Because the upgrade may have made database changes, there's no point removing the value and potentially ruining the database, if you eventually re-enable the plugin.

Let's create the Page Comments `upgrade.php`, `/ww.plugins/page-comments/upgrade.php`:

```
<?php
if($version==0){ // create tables
  dbQuery('create table `page_comments_comment` (
    `id` int(11) NOT NULL auto_increment,
    `comment` text,
    `author_name` text,
    `author_email` text,
    `author_website` text,
    `page_id` int,
    `status` int,
    `cdate` datetime,
    PRIMARY KEY  (`id`)
    ) ENGINE=MyISAM DEFAULT CHARSET=utf8');
  $version++;
}
```

And add a version number to the end of the array in `/ww.incs/page-comments/plugin.php`:

```
  ) ,
  'version' => 1
);
```

Now, let's say you've just enabled the Page Comments plugin. What will happen is:

- 'page-comments' is added to `$DBVARS` and the plugins admin page is reloaded.
- As part of the reload, the `/ww.incs/basics.php` plugins section notices that the plugin has a version number, but the `$DBVARS['page-content|version']` value does not exist. `$version` is set to 0, and the plugin's `upgrade.php` script is run.
- The upgrade script creates the `page_comments_comment` table and increments `$version` to 1.
- The new `$version` is then recorded in `$DBVARS['page-content|version']` and the page is reloaded (again).

So in this case, clicking on **enable** triggered two reloads, one of which also ran an upgrade script.

Now, let's say that you later decided that you needed to also record a moderation e-mail address and whether or not moderation was turned on.

It doesn't make sense to create a whole new database table just to record single values.

Luckily, we already have some code in place that can record single values efficiently. Edit the `upgrade.php` script again and add the following code at the end:

```
if($version==1){ // add moderation details
  $DBVARS[$pname.'|moderation_email']='';
  $DBVARS[$pname.'|moderation_enabled']='no';
  $version++;
}
```

Change the version number in the `plugin.php` file to 2.

Remember that the first run of the script set `$DBVARS['page-content|version']` equal to 1. In this case, when a page is loaded, the upgrade script will skip the first `if` statement and will run the second.

If the plugin was being enabled for the first time, both `if` statements would be run.

The script we just wrote adds the moderation values directly to the `/.private/config.php` file. Notice that we prefixed the values with `page-comments|` so that they would not clash with other plugins.

In my case, that means `/.private/config.php` now looks like this:

```
<?php
$DBVARS=array(
    'username'=>'cmsuser',
    'password'=>'cmspass',
    'hostname'=>'localhost',
    'db_name'=>'cmsdb',
    'theme'=>'basic',
    'plugins'=>'page-comments',
    'page-comments|moderation_email'=>'',
    'page-comments|moderation_enabled'=>'no',
    'page-comments|version'=>'2'
);
```

Also, notice that the second change was not a database one. You can do file updates, send a notification ping to a remote server, send an e-mail, or anything else you want, in those updates.

Custom admin area menu

If you remember, we had the following code lines in `plugin.php`:

```
'menu' => array(
  'Communication>Page Comments' => 'comments'
),
```

Those indicate that we want to add a Page Comments link under a **Communication** top-level menu. When clicked, it should load up an admin maintenance script kept in `/ww.plugins/page-comments/admin/comments.php`.

To make this work, we will need to rewrite the admin menu.

Luckily, we've already installed the Filament Group menu, so we can use that. All we need to do is build a customized `<ul>` menu in the admin header instead of the hardcoded one we already have.

In `/ww.admin/header.php`, remove the entire `#menu-top` element and its contents. We will replace that code with the custom form. Here is the start of it:

```
<?php
  $menus=array(
    'Pages'=>array(
      '_link'=>'/ww.admin/pages.php'
    ),
    'Site Options'=>array(
      'Users'  => array('_link'=>'/ww.admin/users.php'),
      'Themes' => array('_link'=>'/ww.admin/themes.php'),
      'Plugins'=> array('_link'=>'/ww.admin/plugins.php')
    )
  );
  // }
```

First, we create the basic menu array. Any of the plugins that have menu items will add theirs to this.

```
// { add custom items (from plugins)
foreach($PLUGINS as $pname=>$p){
  if(!isset($p['admin'])
      || !isset($p['admin']['menu']))continue;
  foreach($p['admin']['menu'] as $name=>$page){
    if(preg_match('/[^a-zA-Z0-9 >]/',$name))continue;
    $json='{"'.str_replace('>','":{"',$name)
        .'":{"_link":"plugin.php?_plugin='
        .$pname.'&amp;_page='.$page.'"}}'
```

```
            .str_repeat('}',substr_count($name,'>'));
        $menus=array_merge_recursive($menus,
            json_decode($json,true));
      }
    }
    // }
```

Our Page Comments plugin has a menu address **Communication > Page Comments**. This code block takes that string and creates a recursive JSON object from it (**Page Comments** contained in **Communication**), which it then converts to a PHP array, and merges it with `$menus`.

I know it looks difficult to understand—it was a pain to write it as well! I couldn't think of a simpler way to do it which was as concise. If you do, please e-mail me. I prefer my code to be readable by other people.

```
$menus['Log Out']=array('_link'=>
    '/ww.incs/logout.php?redirect=/ww.admin/');
```

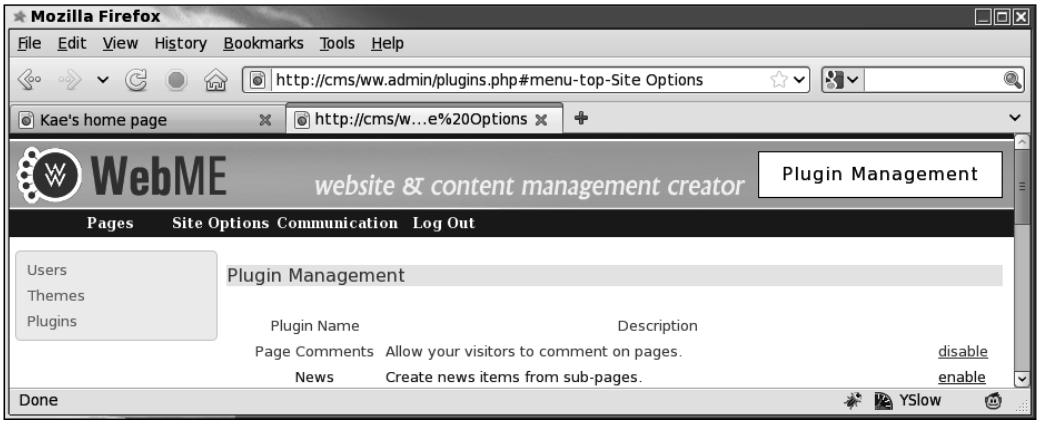
Finally, we add the **Log Out** button at the end of the `$menus` array.

And now, let's output the data in a nested `<ul>` list.

```
// { display menu as UL list
function admin_menu_show($items,$name=false,
                         $prefix,$depth=0){
  if(isset($items['_link']))
      echo '<a href="'.$items['_link'].'">'.$name.'</a>';
  else if($name!='top')
      echo '<a href="#'.$prefix.'-'.$name.'">'.$name.'</a>';
  if(count($items)==1 && isset($items['_link']))return;
  if($depth<2)echo '<div id="'.$prefix.'-'.$name.'">';
  echo '<ul>';
  foreach($items as $iname=>$subitems){
    if($iname=='_link')continue;
    echo '<li>';
    admin_menu_show($subitems,$iname,
        $prefix.'-'.$name,$depth+1);
    echo '</li>';
  }
  echo '</ul>';
  if($depth<2)echo '</div>';
}
admin_menu_show($menus,'top','menu');
// }
?>
```

If an item does not explicitly have a `_link` associated with it, the name is shown and it is not clickable (or at least doesn't do anything when clicked).

With that in place, we have the following menu:



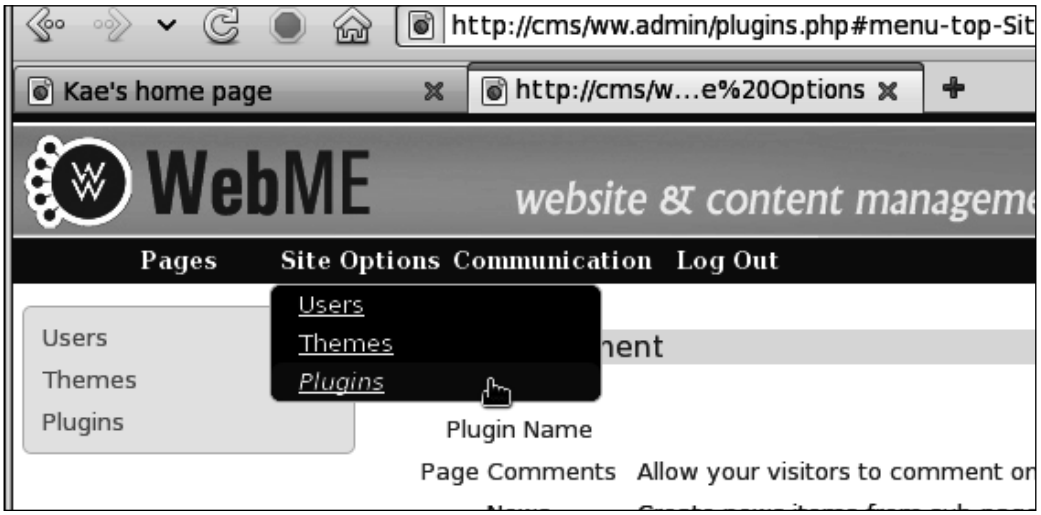
The sub-menus do not yet appear because we haven't enabled the fg-menu.

Edit `/ww.admin/j/admin.js` and add the following highlighted lines to the final section:

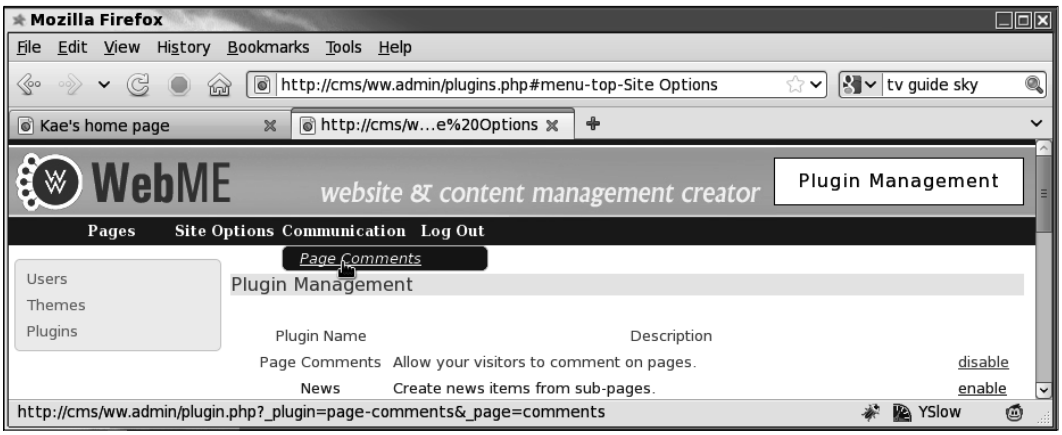
```
    $('input.date-human').each(convert_date_to_human_readable);
    $('#menu-top>ul>li>a').each(function(){
      if(!(/#/.test(this.href.toString())))return;
      $(this).menu({
        content: $(this).next().html(),
        flyOut:true,
        showSpeed: 400,
        callerOnState: '',
        loadingState: '',
        linkHover: '',
        linkHoverSecondary: '',
        flyOutOnState: ''
      });
    });
});
```

That piece of code runs fg-menu on all the items in the menu that do not link to `#`.

After this, we can see that the **Site Options** menu now makes sense:



And we have our **Page Comments** menu item:



Notice the URL in the status bar.

```
http://cms/ww.admin/plugin.php
    ?_plugin=page-comments&_page=comments
```

All the menu items created from plugins are directed to `/ww.admin/plugin.php` (not `/ww.admin/plugins.php`; that has a different purpose), telling the script what plugin is being used (`page-comments`) and what admin form (`comments`) should be used from the plugin's `/admin` directory.

Create the file `/ww.admin/plugin.php`:

```
<?php
require 'header.php';
$pname=$_REQUEST['_plugin'];
$pagename=$_REQUEST['_page'];
if(preg_match('/[^\-a-zA-Z0-9]/',$pagename) || $pagename=='')
    die('illegal character in page name');
if(!isset($PLUGINS[$pname]))die('no plugin of that name ('
    .htmlspecialchars($pname).') exists');
$plugin=$PLUGINS[$pname];
$_url='/ww.admin/plugin.php?_plugin='.urlencode($pname)
    .'&amp;_page='.$pagename;
echo '<h1>'.htmlspecialchars($pname).'</h1>';
if(!file_exists(SCRIPTBASE.'/ww.plugins/'.$pname.'/admin/'
    .$pagename.'.php')){
  echo '<em>The <strong>'.htmlspecialchars($pname).'</strong>
      plugin does not have an admin page named <strong>'
      .$pagename.'</strong>.</em>';
}
else{
  if(file_exists(SCRIPTBASE.'/ww.plugins/'.$pname
      .'/admin/menu.php')){
    include SCRIPTBASE.'/ww.plugins/'
        .$pname.'/admin/menu.php';
    echo '<div class="has-left-menu">';
    include SCRIPTBASE.'/ww.plugins/'.$pname.'/admin/'
        .$pagename.'.php';
    echo '</div>';
  }
  else include SCRIPTBASE.'/ww.plugins/'.$pname.'/admin/'
      .$pagename.'.php';
}
require 'footer.php';
```

When called, this displays the standard admin area header, including the menu, and then checks the requested plugin data.

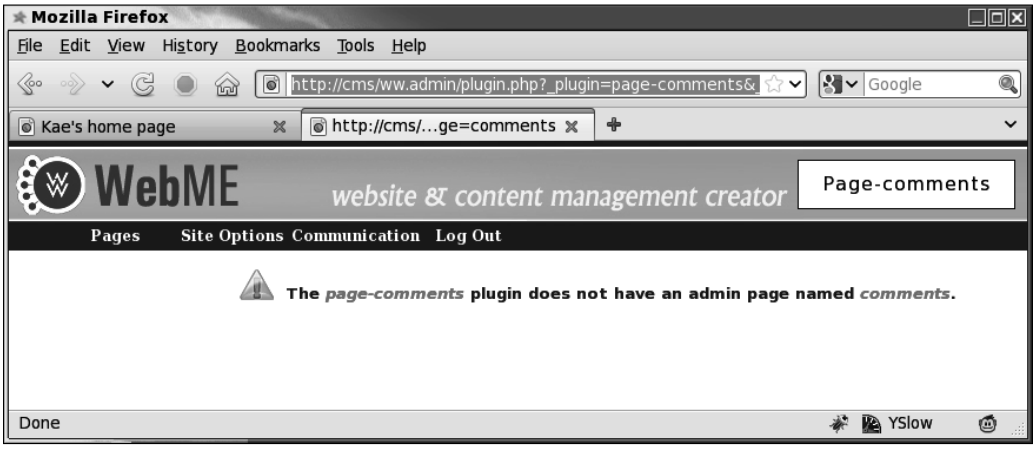
If the plugin doesn't exist, the requested page doesn't exist in the plugin's `/admin` directory, or if the other tests fail, an explanation is shown to the admin and the script is exited.

If all is well, we display the plugin's admin page.

If a `menu.php` file exists in the plugin's `/admin` directory, the menu is shown in a column on the left-hand side and the rest of the page is on the right-hand side.

Otherwise, the page takes over the entire space available.

We haven't created the admin page for comments yet, so here's what the error message looks like:



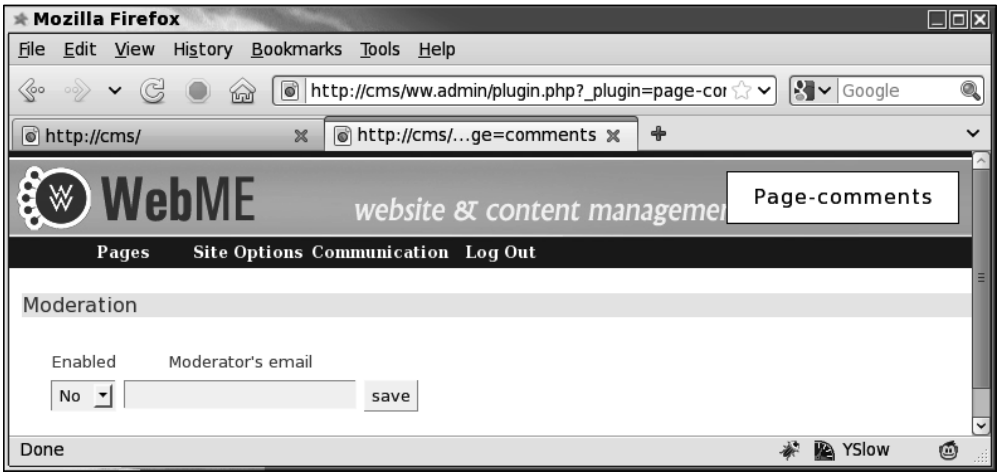
Ideally, the admin should never see that page at all, but if they go playing around with the contents of the URL bar, we need to take care of any eventualities.

Now, we'll write a simple script for that. Create the `/ww.plugins/page-comments/admin/` directory, and create a file in it called `comments.php`, with the following code:

```
<?php
$htmlurl=htmlspecialchars('/ww.admin/plugin.php?_plugin='
  .'page-comments&_page=comments');
// { moderation settings
echo '<form action="',$htmlurl,'" method="post">'
  ,'<h2>Moderation</h2><table><tr><th>Enabled</th>'
  ,'<th>Moderator\'s email</th></tr>';
// { moderation enabled
echo '<tr><td><select name="moderation_enabled">'
  ,'<option value="no">No</option><option value="yes"';
if($DBVARS['page-comments|moderation_enabled']=='yes')
  echo ' selected="selected"';
echo '>yes</option></select></td>';
// }
// { moderation email
echo '<td><input name="moderation_email" value="'
```

```
    ,htmlspecialchars(
      $DBVARS['page-comments|moderation_email'])
    ,'" /></td>';
  // }
  echo '<td><input type="submit" name="action" value="save" '
    ,'/></td></tr></table></form>';
```

This code, when viewed in the browser, shows the following:



The first thing we do is to set `$htmlurl`. This is the HTML-encoded URL of the current plugin admin page. You will need to use this in all the actions so that the CMS knows what plugin you're working with.

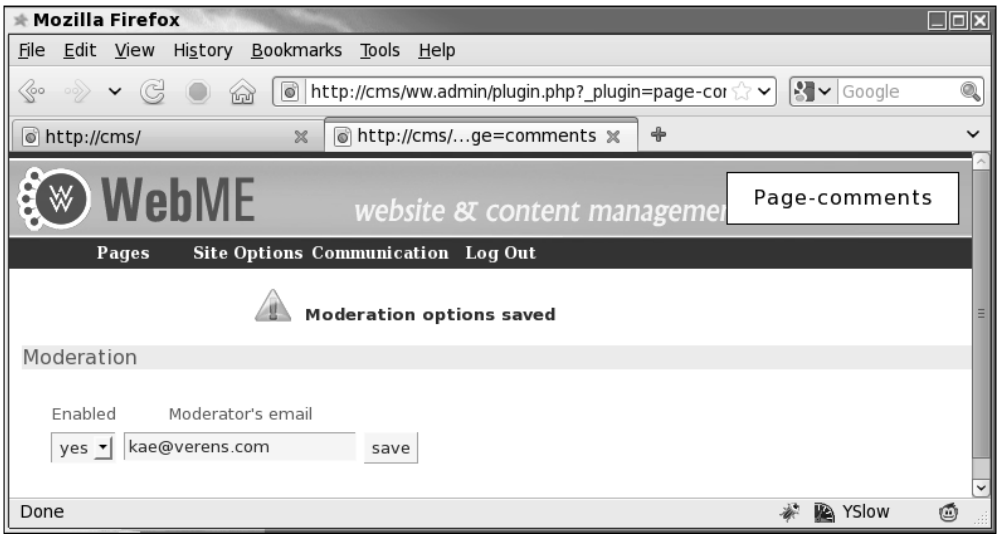
We use it, for example, in the `<form>` that we set to use the POST method (otherwise, when the form is submitted, it may override the `?_plugin=page-comments&_page=comments` part of the URL).

Let's add the code for saving that now. Add it after the opening the `<?php` line in the file.

```
if(isset($_REQUEST['action']) && $_REQUEST['action']=='save'){
  $mod=($_REQUEST['moderation_enabled']=='yes')?'yes':'no';
  $email=$_REQUEST['moderation_email'];
  if(($mod=='yes' && $email=='') ||
      ($mod=='yes' && !
      filter_var($email,FILTER_VALIDATE_EMAIL))){
    echo '<em>error: email is not valid. please retry</em>';
  }
  else{
    $DBVARS['page-comments|moderation_email']=$email;
```

```
        $DBVARS['page-comments|moderation_enabled']=$mod;
        config_rewrite();
        echo '<em>Moderation options saved</em>';
      }
    }
```

This just does a bit of validation on the submitted form, then saves it using the `config_rewrite()` function to write it directly to the config file.



Okay, that's enough from the admin area for now. Let's work on the front-end.

Adding an event to the CMS

We want it so that after the content of a page is figured out, we can trigger a plugin to run some code on it. The obvious place for this trigger to run is immediately at the end of the "set up pagecontent" block in `/ww.index.php` (highlighted):

```
    // other cases will be handled here later
  }
  plugin_trigger('page-content-created',$PAGEDATA);
  // }
  // { set up metadata
```

We will create that function in `/ww.incs/basics.php`:

```
  function plugin_trigger($trigger_name){
    global $PLUGIN_TRIGGERS,$PAGEDATA;
```

```
    if(!isset($PLUGIN_TRIGGERS[$trigger_name]))return;
    foreach($PLUGIN_TRIGGERS[$trigger_name] as $fn)
      $fn($PAGEDATA);
}
```

This checks to see if a plugin trigger of that name (`page-content-created`) exists in the global `$PLUGIN_TRIGGERS` array, which we'll create in a moment, and if so, it runs all functions associated with the name, sending `$PAGEDATA` as a parameter.

In the same file as we are creating the `$PLUGINS` array, we should also be creating the `$PLUGINS_TRIGGERS` array.

Change the start of the plugins block to this:

```
// { plugins
$PLUGINS=array();
$PLUGINS_TRIGGERS=array();
if(isset($DBVARS['plugins'])&&$DBVARS['plugins']){
```

And near the end of the block:

```
      $PLUGINS[$pname]=$plugin;
      if(isset($plugin['triggers'])){
        foreach($plugin['triggers'] as $name=>$fn){
          if(!isset($PLUGIN_TRIGGERS[$name]))
              $PLUGIN_TRIGGERS[$name]=array();
          $PLUGIN_TRIGGERS[$name][]=$fn;
        }
      }
    }
}
else $DBVARS['plugins']=array();
```

And it's as simple as that. We can now create triggers anywhere in the CMS core, and they will execute any that are in the plugins.

If you remember, the Page Comments plugin triggers the function `page_comments_show()` when `page-content-created` is triggered.

We've already written a stub function for this, which then loads up the file `/ww.plugins/page-comments/frontend/show.php`.

Create that file now (create the directory `ww.plugins/page-comments/frontend` first):

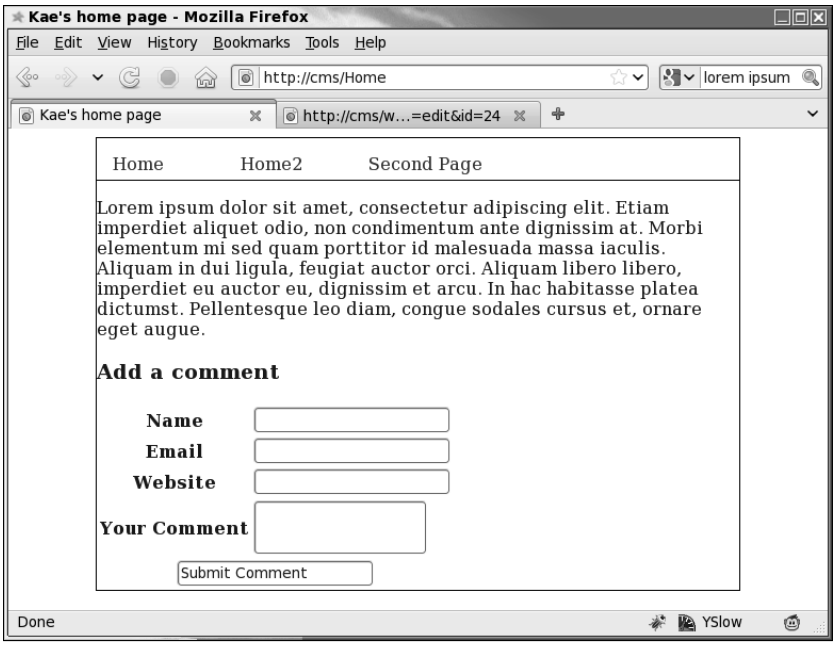
```
<?php
global $pagecontent,$DBVARS;

$c='';
$message='';
// { add submitted comments to database
// }
// { show existing comments
// }
// { show comment entry form
$c.='<a name="page-comments-submit"></a>'
    .'<h3>Add a comment</h3>';
if($message)$c.=$message;
$c.='<form action="'.$PAGEDATA->getRelativeURL()
    .'#page-comments-submit" method="post"><table>';
$c.='<tr><th>Name</th><td><input name="page-comments-name" />'
    .'</td></tr>';
$c.='<tr><th>Email</th><td><input type="email" '
    .'name="page-comments-email" /></td></tr>';
$c.='<tr><th>Website</th><td><input '
    .'name="page-comments-website" /></td></tr>';
$c.='<tr><th>Your Comment</th><td><textarea '
    .'name="page-comments-comment"></textarea></td></tr>';
$c.='<tr><th colspan="2"><input name="action" '
    .'value="Submit Comment" /></th></tr>';
$c.='</table></form>';
// }
$pagecontent.='<div id="page-comments-wrapper">'.$
    c.'</div>';
```

Simple enough; this adds a comment box onto the global `$pagecontent` variable.

Note that the `#page-comments-submit` anchor appended to the page URL. If someone submits a comment, they will be brought back to the comment form.

I've adjusted the basic theme we've been using, to make it a little neater and added some Lorem Ipsum text to the body of the home page, so we can see what it looks like with some text in it.

Here's what the home page looks like with the Page Comments plugin enabled:



Now we need to take care of what happens when the comment is submitted.

Edit the `/ww.plugins/page-comments/frontend/show.php` file, changing the "add submitted comments to database" section to this:

```
// { add submitted comments to database
if(isset($_REQUEST['action']) &&
    $_REQUEST['action']=='Submit Comment'){
  if(!isset($_REQUEST['page-comments-name']) ||
      $_REQUEST['page-comments-name']=='')
      $message.='<li>Please enter your name.</li>';
  if(!isset($_REQUEST['page-comments-email']) ||
      !filter_var($_REQUEST['page-comments-email'],
      FILTER_VALIDATE_EMAIL))
      $message.='<li>Please enter your email address.</li>';
  if(!isset($_REQUEST['page-comments-comment']) ||
      !$_REQUEST['page-comments-comment'])
      $message.='<li>Please enter a comment.</li>';
  if($message)$message='<ul
      class="error page-comments-error">'.$message.'</ul>';
  else{
    $website=isset($_REQUEST['page-comments-website'])
```

```
            ?$_REQUEST['page-comments-website']:'';
        if($DBVARS['page-comments|moderation_enabled']=='yes'){
          $status=0;
          mail($DBVARS['page-comments|moderation_email'],
            '['.$_SERVER['HTTP_HOST'].'] comment submitted',
            'A new comment has been submitted to the page "'
            .$PAGEDATA->getRelativeUrl().'". Please log into '
            'the admin area of the site and moderate it using '
            'that page\'s admin.',
            'From: noreply@'.$_SERVER['HTTP_HOST']
            ."\nReply-to: noreply@".$_SERVER['HTTP_HOST']);
          $message='<p>Comments are moderated. It may be a '
              .'few minutes before your comment appears.</p>';
        }
        else $status=1;
        dbQuery('insert into page_comments_comment set comment="'
          .addslashes($_REQUEST['page-comments-comment'])
          .'",author_name="'
          .addslashes($_REQUEST['page-comments-name'])
          .'",author_email="'
          .$_REQUEST['page-comments-email']
          .'",author_website="'.addslashes($website)
          .'",cdate=now(),page_id='.$PAGEDATA->id.',status='
          .$status);
      }
    }
    // }
```

This will record the comment in the database. Notice the `status` field. This says whether a comment is visible or not.

This can be enhanced in many ways. You can change the e-mail that's sent to the moderator to add links back to the right places, you can add an `is_spam` field to the database and check the comment using the Akisment service (`http://akismet.com/`), or you can have client-side jQuery form validation.

I haven't added these, as I am currently simply explaining how plugins work.

Finally in this section, we need to display any comments that have been successfully entered and moderated. To simulate this, I temporarily turned off moderation in my own copy before doing the following (so comments go through with status set to 1).

Before we write the code for showing comments, we will add a new function to `/ww.incs/basics.php`:

```
function date_m2h($d, $type = 'date') {
  $date = preg_replace('/[- :]/', ' ', $d);
  $date = explode(' ', $date);
  if ($type == 'date') {
    return date('l jS F, Y', mktime(0, 0, 0,
        $date[1], $date[2], $date[0]));
  }
  return date(DATE_RFC822, mktime($date[5],
      $date[4], $date[3], $date[1], $date[2], $date[0]));
}
```

This function `m2h` (stands for "mysql to human") takes a MySQL date and converts it to a format that can be read by humans.

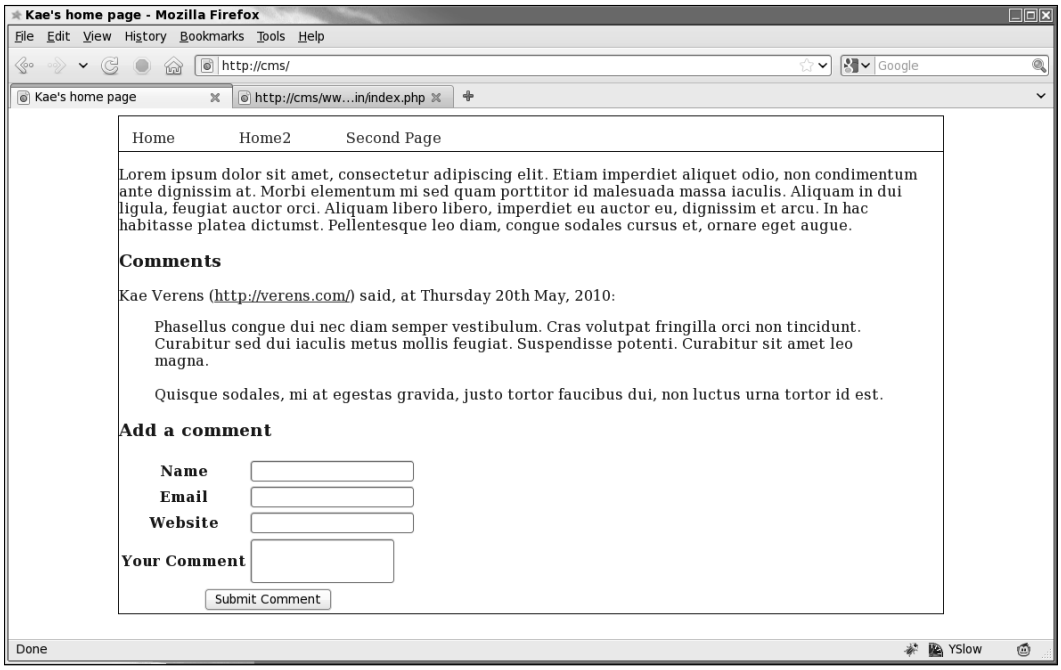
It's easy to write two or three lines and get the same result (or one complex line), but why bother, when it just takes a single function call?

Now, edit `/ww.plugins/page-comments/frontend/show.php` again and this time change the "show existing comments" section to this:

```
// { show existing comments
$c.='<h3>Comments</h3>';
$comments=dbAll('select * from page_comments_comment where
    status=1 and page_id='.$PAGEDATA->id.' order by cdate');
if(!count($comments)){
  $c.='<p>No comments yet.</p>';
}
else foreach($comments as $comment){
  $c.=htmlspecialchars($comment['author_name']);
  if($comment['author_website'])$c.=' (<a href="'
      .htmlspecialchars($comment['author_website']).'">'
      .htmlspecialchars($comment['author_website']).'</a>)';
  $c.=' said, at '.date_m2h($comment['cdate'])
    .':<br /><blockquote>'
    .nl2br(htmlspecialchars($comment['comment']))
    .'</blockquote>';
}
// }
```

This code takes comments from the `page_comments_comment` table and shows them in the page, as long as they have already been accepted/moderated (their status is 1), and they belong to that page.

Here's a screenshot of our home page now, with a submitted comment:



And now we get to the final major plugin method for the CMS in this chapter (there is still more to come): adding a tab to the page admin.

Adding tabs to the page admin

When a comment is submitted (by the way, turn moderation back on), an e-mail is sent to the moderator, who is asked to log into the admin area to check messages that may have been sent.

We had this in the `plugin.php` file:

```
'page_tab' => array(
  'name' => 'Comments',
  'function' => 'page_comments_admin_page_tab'
)
```

We will use this information to add a new tab to the page admin named Comments, which will be populated by calling a function `page_comments_admin_page_tab()`, which we've already built as a stub. It loads and runs the file `/ww.plugins/page-comments/admin/page-tab.php`, which generates a string named `$html` (it contains HTML) and returns that to the page admin to be displayed.

Note that it is also very useful to have a central area where comments from all pages can be moderated. This chapter does not discuss that, but if you wish to see how that would be implemented, please see the comments plugin here: `http://code.google.com/p/webworks-webme/source/browse/#svn/ww.plugins/comments`.

So first, let's adapt the page admin so it will run the plugin's function. Edit `/ww.admin/pages/forms.php` and before the line which opens the `<form>`, add the highlighted code given as follows:

```
// }
// { generate list of custom tabs
$custom_tabs=array();
foreach($PLUGINS as $n=>$p){
  if(isset($p['admin']['page_panel'])){
    $custom_tabs[$p['admin']['page_panel']['name']]
        =$p['admin']['page_panel']['function'];
  }
}
// }
echo '<form id="pages_form" method="post">';
```

This code builds up a `$custom_tabs` array from the global `$PLUGINS` array.

Next, we display the tab names . Replace the "// add plugin tabs here" line with this highlighted line:

```
    ,'<li><a href="#tabs-advanced-options">Advanced
        Options</a></li>';
foreach($custom_tabs as $name=>$function)echo '<li><a
    href="#tabs-custom-'
    ,preg_replace('/[^a-z0-9A-Z]/','',$name)
    ,'">',htmlspecialchars($name),'</a></li>';
echo '</ul>';
```

Finally, we add in the actual tab content (highlighted code) after the "Advanced Options" section:

```
// }
// { tabs added by plugins
foreach($custom_tabs as $n=>$p){
  echo '<div id="tabs-custom-'
      ,preg_replace('/[^a-z0-9A-Z]/','',$n),'">'
      ,$p($page,$page_vars),'</div>';
}
```

```
// }
echo '</div><input type="submit" name="action" value="',
    ($edit?'Update Page Details':'Insert Page Details')
    ,'" /></form>';
```

This creates the tab bodies by calling the plugin functions with two parameters; the main `$page` table data, and the custom variables of the page, `$page_vars`.

The result is then echoed to the screen.

So, let's create the `/ww.plugins/page-comments/admin/page-tab.php` file:

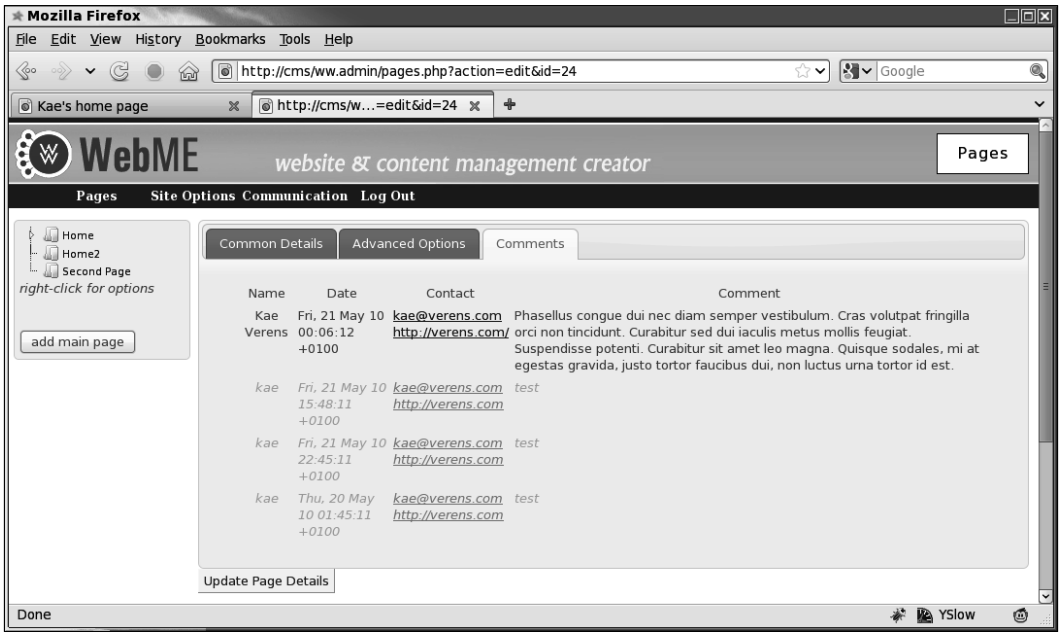
```
<?php
$html='';
$comments=dbAll('select * from page_comments_comment where
    page_id='.$PAGEDATA['id'].' order by cdate desc');
if(!count($comments)){
  $html='<em>No comments yet.</em>';
  return;
}
$html.='<table id="page-comments-table"><tr><th>Name</th>'
    ,'<th>Date</th><th>Contact</th>'
    ,'<th>Comment</th><th> </th></tr>';
foreach($comments as $comment){
  $html.='<tr class="';
  if($comment['status'])$html.='active';
  else $html.='inactive';
  $html.='" id="page-comments-tr-'.$comment['id'].'">';
  $html.='<th>'.htmlspecialchars($comment['author_name'])
    .'</th>';
  $html.='<td>'.date_m2h($comment['cdate'],'datetime')
    .'</td>';
  $html.='<td>';
  $html.='<a href="mailto:'
    .htmlspecialchars($comment['author_email']).'">'
    .htmlspecialchars($comment['author_email']).'</a><br />';
  if($comment['author_website'])$html.='<a href="'
    .htmlspecialchars($comment['author_website']).'">'
    .htmlspecialchars($comment['author_website']).'</a>';
  $html.='</td>';
  $html.='<td>'.htmlspecialchars($comment['comment']).'</td>';
  $html.='<td></td></tr>';
}
$html.='</table><script src="/ww.plugins/page-comments'
  .'/admin/page-tab.js"></script>';
```

In this code block, we build up a string variable named `$html` that holds details on all the comments for a specified page.

We've left a blank `<td>` at the end of each row, which will be filled by jQuery with some actionable links.

Enhancements that you could build in here might be to limit the number of characters available in each table cell or also to add pagination for pages with vast numbers of comments.

We can already see that the tab is working:



Now we need to add in the actions.

Create the file `/ww.plugins/page-comments/admin/page-tab.js`:

```
function page_comments_build_links(){
  var stat=this.className;
  if(!stat)return;
  var id=this.id.replace('page-comments-tr-','');
  var html='<a href="javascript:page_comments_'+(
    (stat=='active')
      ?'deactivate('+id+');">deactivate'
      :'activate('+id+');">activate'
    )+'</a> | <a href="javascript:'
```

```
      +'page_comments_delete('+id+');">delete</a>';
    $(this).find('td:last-child').html(html);
  }
  $(function(){
    $('#page-comments-table tr')
      .each(page_comments_build_links);
  });
```

This script takes all the `<tr>` rows in the table, checks their classes, and builds up links based on whether the link is currently active or inactive.

The reason we do this through JavaScript and not straight in the PHP is that we're going to moderate the links through AJAX, so it would be a waste of resources to do it in both PHP and jQuery.

The page now looks like this:

Okay, now we just need to make those links work. Add these functions to the same file:

```
function page_comments_activate(id){
  $.get('/ww.plugins/page-comments/admin/activate.php'
      +'?id='+id,function(){
    var el=document.getElementById('page-comments-tr-'+id);
    el.className='active';
```

```
      $(el).each(page_comments_build_links);
    });
  }
  function page_comments_deactivate(id){
    $.get('/ww.plugins/page-comments/admin/deactivate.php'
        +'?id='+id,function(){
      var el=document.getElementById('page-comments-tr-'+id);
      el.className='inactive';
      $(el).each(page_comments_build_links);
    });
  }
  function page_comments_delete(id){
    $.get('/ww.plugins/page-comments/admin/delete.php'
        +'?id='+id,function(){
      $('#page-comments-tr-'+id).fadeOut(500,function(){
        $(this).remove();
      });
    });
  }
```

These handle the deletion, activation, and deactivation of comments from the client-side. I haven't included tests to see if they were successful (this is a demo).

The deletion event is handled by fading out the table row and then removing it. The others simply change classes on the row and links in the right cell.

First, let's build the activation PHP script `/ww.plugins/page-comments/admin/activate.php`:

```
<?php
require $_SERVER['DOCUMENT_ROOT'].'/ww.admin/admin_libs.php';

$id=(int)$_REQUEST['id'];
dbQuery('update page_comments_comment set status=1
    where id='.$id);
echo '1';
```

The next one is to deactivate the comment, `/ww.plugins/page-comments/admin/deactivate.php`:

```
<?php
require $_SERVER['DOCUMENT_ROOT'].'/ww.admin/admin_libs.php';

$id=(int)$_REQUEST['id'];
dbQuery('update page_comments_comment set status=0
    where id='.$id);
echo '0';
```

Yes, they're the same except for two numbers. I have them as two files because it might be interesting in the future to add in separate actions that happen after one or the other, such as sending an e-mail when a comment is activated and other such actions.

The final script is the deletion one, `/ww.plugins/page-comments/admin/delete.php`:

```
<?php
require $_SERVER['DOCUMENT_ROOT'].'/ww.admin/admin_libs.php';

$id=(int)$_REQUEST['id'];
dbQuery('delete from page_comments_comment where id='.$id);
echo '1';
```

Very simple and it completes our Page Comments example!

As pointed out, there are a load of ways in which this could be improved. For example, we didn't take into account people that are already logged in (they shouldn't have to fill in their details and should be recorded in the database), there is no spam control, we did not use client-side validation, you could add "gravatar" images for submitters, and so on.

All of these things can be added on at any point. We're onto the next plugin now!

Summary

In this chapter, we built the framework for enabling plugins and looked at a number of ways that the plugin can integrate with the CMS.

We also changed the admin menu so it could incorporate custom items from the plugins.

We built an example plugin, Page Comments, which used a few different plugin hook types, page admin tabs, a standalone admin page, and a `page-content-created` trigger.

In the next chapter, we will create a plugin that allows the admin to create a form page for submission as an e-mail or for saving in the database.

Where to buy this book

You can buy CMS Design Using PHP and jQuery from the Packt Publishing website: `https://www.packtpub.com/cms-design-using-php-and-jquery/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/cms-design-using-php-and-jquery/book