3
Page Management – Part One

In this chapter, we will create the forms for page management, and will build a system for moving the pages around using drag-and-drop.

We will discuss the following topics:

- How pages are requested and generated
- Listing the pages in the admin area
- Administration of pages

Page management will be concluded in the next chapter, where we will discuss saving the pages, and integrate a rich-text editor and a file manager.

How pages work in a CMS

As we discussed in *Chapter 1*, *CMS Core Design*, a "page" is simply the main content which should be shown when a certain URL is requested.

In a non-CMS website, this is easy to see, as a single URL returns a distinct HTML file. In a CMS though, the page is generated dynamically, and may include features such as plugins, different views depending on whether the reader was searching for something, whether pagination is used, and other little complexities.

In most websites, a page is easily identified as the large content area in the middle (this is an over-simplification). In others, it's harder to tell, as the onscreen page may be composed of content snippets from other parts of the site.

We handle these differences by using page "types", each of which can be rendered differently on the front-end. Examples of types include gallery pages, forms, news contents, search results, and so on.

In this chapter, we will create the simplest type, which we will call "normal". This consists of a content-entry textarea in the admin area, and direct output of that content on the front-end. You could call this "default" if you want, but since a CMS is not always used by people from a technical background, it makes sense to use a word that they are more likely to recognize. I have been asked before by clients what "default" means, but I've never been asked what "normal" means.

If you remember from the first chapter, we discussed what should go in the core, and what should be a plugin.

At the very least, a CMS needs some way to create the simplest of web pages. This is why the "normal" type is not a plugin, but is built into the core.

Listing pages in the admin area

To begin, we will add `Pages` to the admin menu. Edit `/ww.admin/header.php` and add the following highlighted line:

```
<ul>
  <li><a href="/ww.admin/pages.php">Pages</a></li>
  <li><a href="/ww.admin/users.php">Users</a></li>
```

And one more thing—when we log into the administration part of the CMS, it makes sense to have the "front page" of the admin area be the `Pages` section. After all, most of the work in a CMS is done in the `Pages` section.

So, we change `/ww.admin/index.php` so it is a synonym for `/ww.admin/pages.php`. Replace the `/ww.admin/index.php` file with this:

```
<?php
require 'pages.php';
```

Next, let's get started on the `Pages` section.

First, we will create `/ww.admin/pages.php`:

```
<?php
require 'header.php';
echo '<h1>Pages</h1>';
// { load menu
echo '<div class="left-menu">';
require 'pages/menu.php';
echo '</div>';
// }
// { load main page
echo '<div class="has-left-menu">';
require 'pages/forms.php';
```

```
echo '</div>';
// }
echo '<style type="text/css">
  @import "pages/css.css";</style>';
require 'footer.php';
```

Notice how I've commented blocks of code, using `// {` to open the comment at the beginning of the block, and `// }` at the end of the block.

This is done because a number of text editors have a feature called "**folding**", which allows blocks enclosed within delimiters such as { and } to be hidden from view, with just the first line showing.

For instance, the previous code example looks like this in my Vim editor:

What the `page.php` does is to load the headers, load the menu and page form, and then load the footers. There will be more added later in this chapter.

For now, create the directory `/ww.admin/pages` and create a file in it called `/ww.admin/pages/forms.php`:

```
<h2>FORM GOES HERE</h2>
```

And now we can create the page menu. Use the following code to create the file `/ww.admin/pages/menu.php`:

```
<?php
echo '<div id="pages-wrapper">';
$rs=dbAll('select id,type,name,parent from pages order by ord,name');
$pages=array();
foreach($rs as $r){
  if(!isset($pages[$r['parent']]))$pages[$r['parent']]=array();
  $pages[$r['parent']][]=$r;
}
```

```
function show_pages($id,$pages){
  if(!isset($pages[$id]))return;
  echo '<ul>';
  foreach($pages[$id] as $page){
    echo '<li id="page_'.$page['id'].'">'
      .'<a href="pages.php?id='.$page['id'].'"'>'
      .'<ins> </ins>'.htmlspecialchars($page['name'])
      .'</a>';
    show_pages($page['id'],$pages);
    echo '</li>';
  }
  echo '</ul>';
}
show_pages(0,$pages);
echo '</div>';
```

That will build up a `<ul>` tree of pages.

Note the use of the "parent" field in there. Most websites follow a hierarchical "parent-child" method of arranging pages, with all pages being a child of either another page, or the "root" of the site. The parent field is filled with the ID of the page within which it is situated.

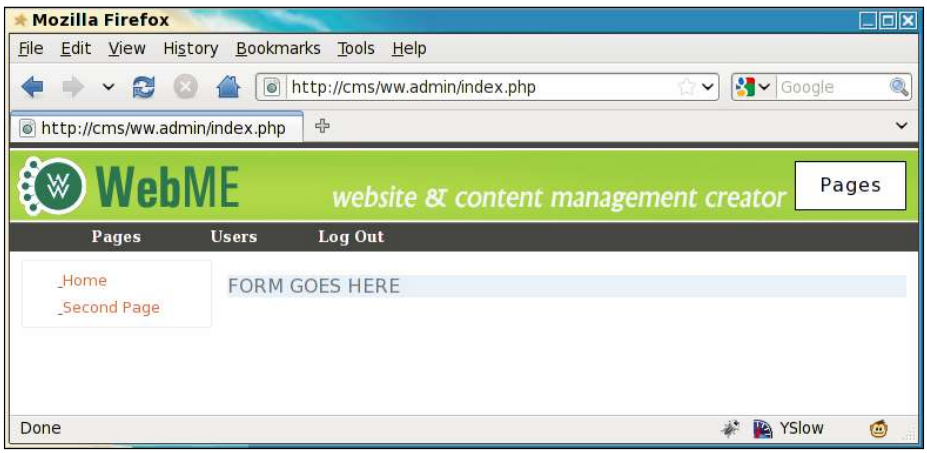
There are two main ways to indicate which page is the "front" page (that is, what page is shown when someone loads up `http://cms/` with no page name indicated).

1. You can have one single page in the database which has a parent of 0, meaning that it has no parent—this page is what is looked for when `http://cms/` is called. In this scheme, pages such as `http://cms/pagename` have their parent field set to the ID of the one page which has a parent of 0.
2. You can have many pages which have 0 as their parent, and each of these is said to be a "top-level" page. One page in the database has a flag set in the `special` field which indicates that this is the front page. In this scheme, pages named like `http://cms/pagename` all have a parent of 0, and the page corresponding to `http://cms/` can be located anywhere at all in the database.

Case 1 has a disadvantage, in that if you want to change what page is the front page, you need to move the current page under another one (or delete it), then move all the current page's child-pages so they have the new front page's ID as a parent, and this can get messy if the new front-page already had some sub-pages—especially if there are any with the same names.

Case 2 is a much better choice because you can change the front page whenever you want, and it doesn't cause any problems at all.

When you view the site in your browser now, it looks like this (based on the pages we created manually back in *Chapter 1, CMS Core Design*):



Hierarchical viewing of pages

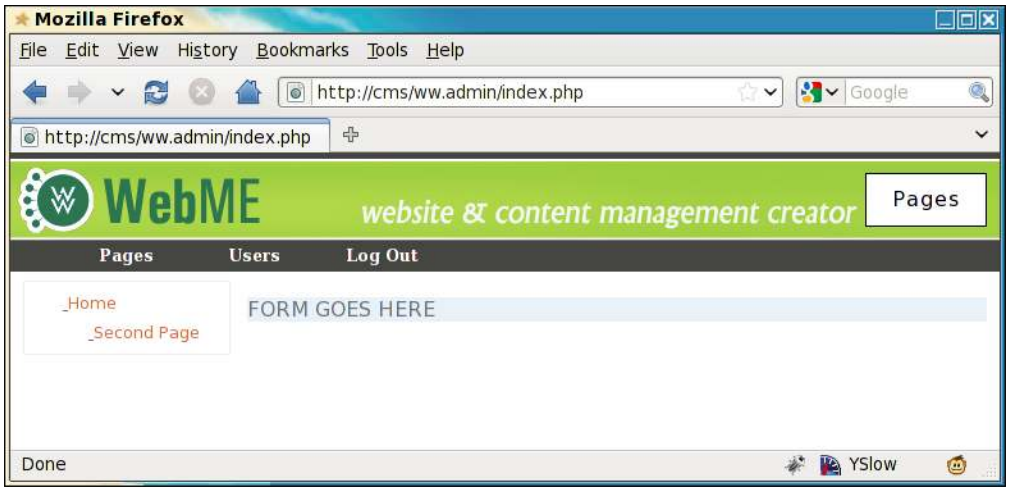
Let's update the database slightly so that we can see the hierarchy of the site pages visually.

Go to your MySQL console, and change the `Second Page` so that its parent field is the ID of the `Home` page:

```
mysql> select id,name,parent from pages;
+----+-------------+--------+
| id | name        | parent |
+----+-------------+--------+
| 24 | Home        |      0 |
| 25 | Second Page |      0 |
+----+-------------+--------+
2 rows in set (0.00 sec)

mysql> update pages set parent=24 where id=25;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

After the update, we refresh the site in the browser:



You can see the **Second Page** has indented slightly because it is now a child page of **Home**, and is contained in a sub-`<ul>` in the HTML.

We can improve on this vastly, though.

There is a jQuery plugin called `jstree` which re-draws `<ul>` trees in a way that is more familiar to users of visual file managers.

It also has the added features that you can drag the tree nodes around, and attach events to clicks on the nodes.

We will use these features later in the chapter to allow creation and deletion of pages, and changing of page parents through drag-and-drop.

Create the directory `/j/` in the root of the website.

Remember that we indicated in the first chapter that the CMS directories would all include dots in them, unless they were less than three characters long.

One of the reasons we name this directory `/j/` instead of `/ww.javascript/`, is that it is short, thus saving a few bytes of bandwidth for the end-user, who may be using something bandwidth-light such as a smartphone.

This may not be a big deal, but if we got into the habit of making small shortcuts like this whenever possible, then the small shortcuts would eventually add up to a second or two of extra speed.

Every unnoticeable optimization can help to make a noticeable one when combined with many more.

Anyway—create the `/j/` directory, and download the `jstree` script from `http://jstree.com/` such that when extracted, the `jquery.tree.js` file is located at `/j/jquery.jstree/jquery.tree.js`.

I have used version `0.9.9a` in the CMS described in this book.

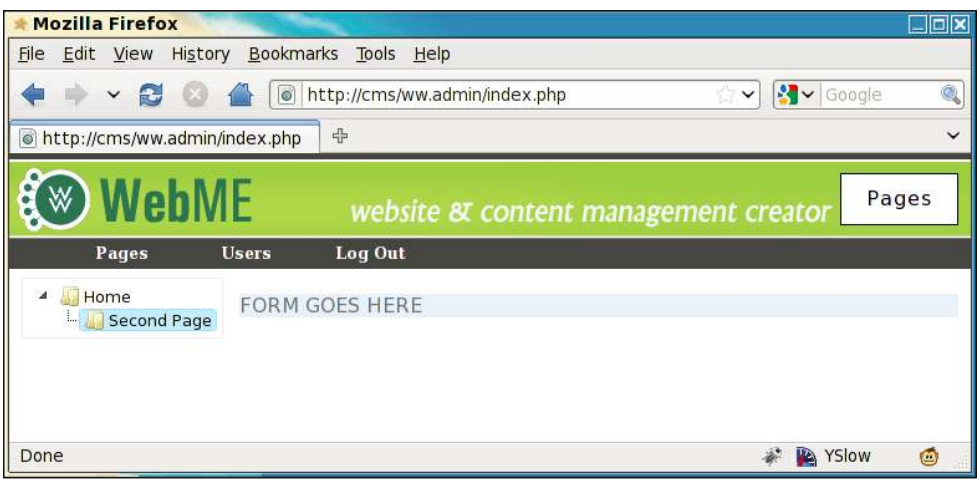
Now edit `/ww.admin/pages/menu.php` and add the following highlighted lines before the first line:

```
<script src="/j/jquery.jstree/jquery.tree.js"></script>
<script src="/ww.admin/pages/menu.js"></script>
<?php
```

And create the file `/ww.admin/pages/menu.js`:

```
$(function(){
  $('#pages-wrapper').tree();
});
```

And immediately, we have a beautified hierarchical tree, as seen in the following screenshot:



If you try, you'll see that you can drag those page names around, with little icons and signs indicating where a page can be dropped, as shown in the next two screenshots:

Before we get into the actual editing of pages, let's improve on this menu one last time. We will add a button to indicate we want to create a new top-level page, and we will also record drag-and-drop events so that they actually do move the pages around.

Change the `/ww.admin/pages/menu.js` file to this:

```
$(function(){
 $('#pages-wrapper').tree({
  callback:{
   onchange:function(node,tree){
    document.location='pages.php?action=edit&id='
      +node.id.replace(/.*_/,'');
   },
   onmove:function(node){
    var p=$.tree.focused().parent(node);
    var new_order=[],nodes=node.parentNode.childNodes;
    for(var i=0;i<nodes.length;++i)
      new_order.push(nodes[i].id.replace(/.*_/,''));
    $.getJSON('/ww.admin/pages/move_page.php?id='
      +node.id.replace(/.*_/,'')+'&parent_id='
      +(p==-1?0:p[0].id.replace(/.*_/,''))
      +'&order='+new_order);
   }
  }
 });
 var div=$(
   '<div><i>right-click for options</i><br /><br /></div>');
 $('<button>add main page</button>')
  .click(pages_add_main_page)
  .appendTo(div);
 div.appendTo('#pages-wrapper');
});
function pages_add_main_page(){}
```

We've added a few pieces of functionality to the tree here.

First, we have the `onchange` callback.

When a tree node (a page name) is clicked, the browser is redirected to `pages.php?edit=` with the page's ID at the end-note that when creating the `<ul>` tree, we added an ID to every `<li>`, such that a page with the ID 24 would have an `<li>` with the ID `page_24`.

So, all we need to do when a node (the `<li>`) is clicked, is to remove the `page_` part, and use that to open up `page.php` for editing that page.

Second, we added an `onmove` callback. This is called after a drag-and-drop event has completed.

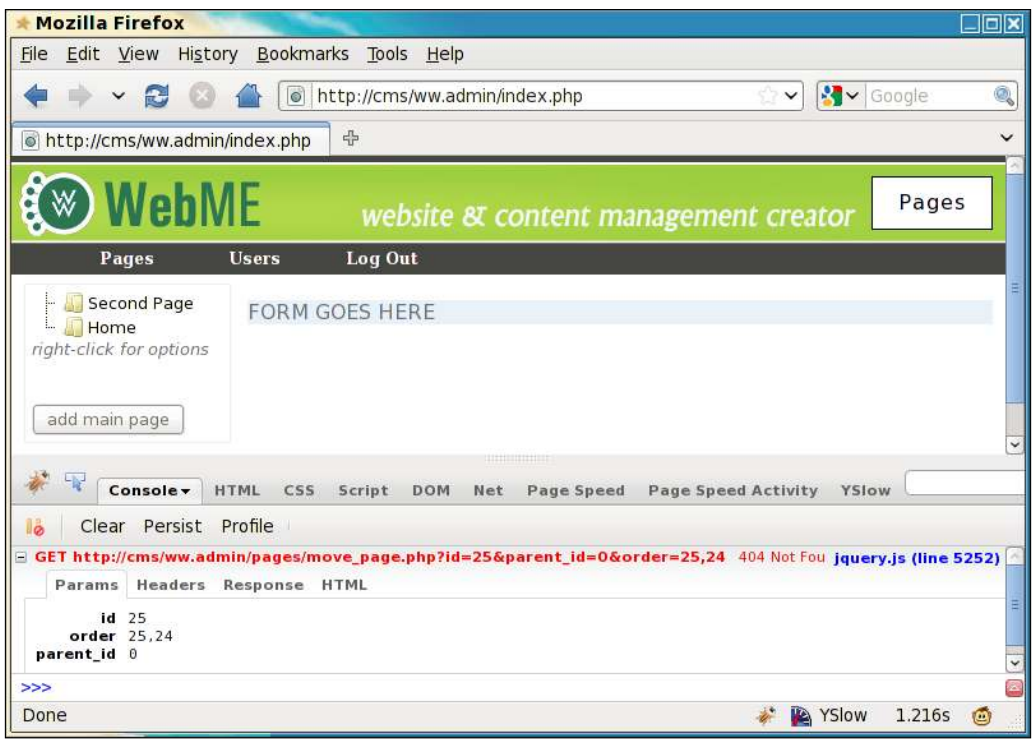
What we do in this is slightly more complex—we get the new parent's ID, and we make an array which records the IDs of all its direct descendant child pages. We then send all that data to `/ww.admin/pages/move_page.php`, which we'll create in just a moment.

Finally, we've added a message to right-click on the tree for further functionality, which we'll detail later in the chapter, and a button to create a new top-level page, which we'll also detail later in the chapter. A dummy function needs to be added so this code will run without error. We'll replace it with a real one later.

Moving and rearranging pages

Now when you drag a page name to a different place on the tree, an Ajax call is made to `/ww.admin/pages/move_page.php`, with some details included in the call.

Here's a screenshot showing (using Firebug) what is sent in a sample drag:



We are sending the page ID (`25`), the new parent ID (`0`), and the new page order of pages which have the parent ID `0` (`25, 24`).

So, let's create `/ww.admin/pages/move_page.php`:

```
<?php
require '../admin_libs.php';
$id=(int)$_REQUEST['id'];
$to=(int)$_REQUEST['parent_id'];
$order=explode(',',$_REQUEST['order']);
dbQuery('update pages set parent='.$to.' where id='.$id  );
for($i=0;$i<count($order);++$i){
  $pid=(int)$order[$i];
  dbQuery("update pages set ord=$i where id=$pid");
  echo "update pages set ord=$i where id=$pid\n";
}
```

Simple! It records exactly what it was sent.

Administration of pages

Okay—we now have a list of the existing pages. Let's add some functionality to edit them.

The form for creating a page is a bit long, so what we'll do is to build it up a bit at a time, explaining as we go. Replace the file `/ww.admin/pages/forms.php` with the following:

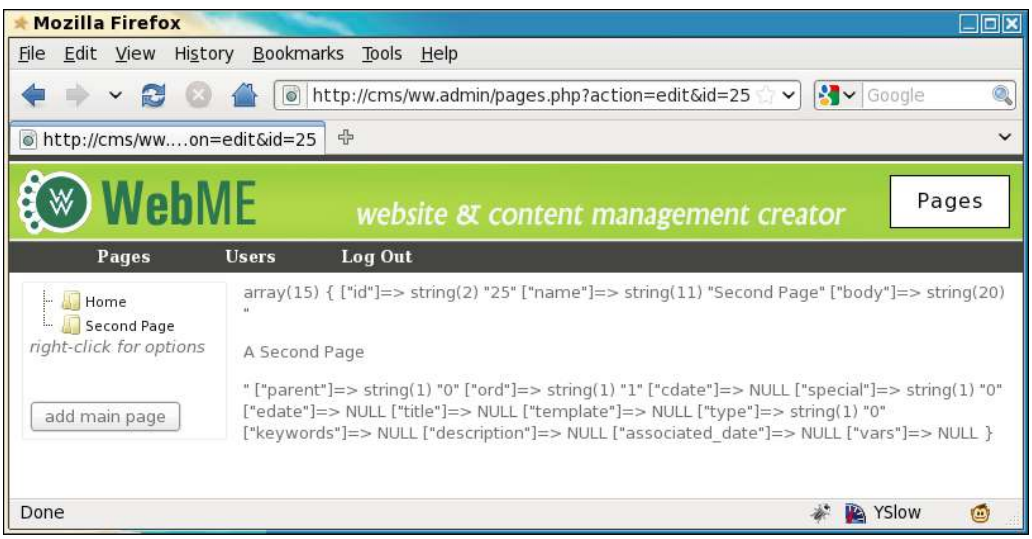
```
<?php
if(isset($_REQUEST['id']))$id=(int)$_REQUEST['id'];
else $id=0;
if($id){ // check that page id exists
  $page=dbRow("SELECT * FROM pages WHERE id=$id");
  if($page!==false){
    $page_vars=json_decode($page['vars'],true);
    $edit=true;
  }
}
if(!isset($edit)){
  $parent=isset($_REQUEST['parent'])?
    (int)$_REQUEST['parent']:0;
  $special=0;
  if(isset($_REQUEST['hidden']))$special+=2;
  $page=array('parent'=>$parent,'type'=>'0','body'=>'',
```

```
        'name'=>'','title'=>'','ord'=>0,'description'=>'',
        'id'=>0,'keywords'=>'','special'=>$special,
        'template'=>'');
    $page_vars=array();
    $id=0;
    $edit=false;
}
```

What the given code does is to initialize an array named `$page` for the main page details, and another named `page_vars` for any custom details that are not part of the main page table—for example, data recorded as part of a plugin.

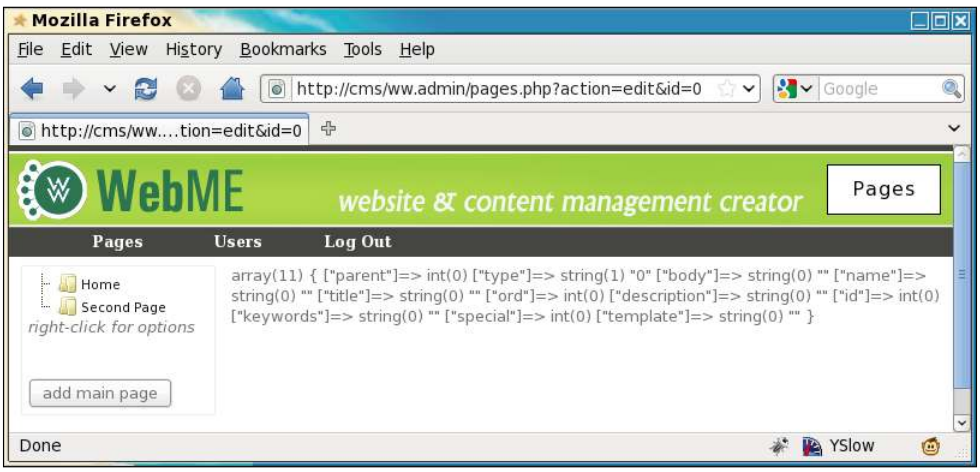
If an ID is passed as part of the URL, then that page's data is loaded.

As an example, if I add the line `var_dump($page);` and then load up `/ww.admin/pages.php?action=edit&id=25` in my browser (a page which exists in my database), this is what's shown:



This shows all the data about that page available in the database table.

If the ID passed in the URL is 0, or any other ID which does not correspond to an existing page ID, then we still initialize `$page` but with empty values:



Because pages can get quite complex, especially when we add in different page types using plugins later in the book, we break the page form into different tabs.

For the "normal" page type, we will have two tabs—common details, and advanced options.

The common details tab will contain options that are changed very often, such as page name, page content, and so on.

The advanced options tab will contain more rarely-changed options such as meta tags, templates, and so on. We call it "advanced", but that's only because "rarely changed options" doesn't sound as concise, and also because most website administrators will not know what to do with some of these options.

So, let's add the tab menu to `/ww.admin/pages/forms.php`:

```
// { if page is hidden from navigation, show a message saying that
if($page['special']&2)
  echo '<em>NOTE: this page is currently hidden from the
        front-end navigation. Use the "Advanced Options" to
        un-hide it.</em>';
// }
echo '<form id="pages_form" method="post">';
echo '<input type="hidden" name="id" value="',$id,'" />'
    ,'<div class="tabs"><ul>'
    ,'<li><a href="#tabs-common-details">Common Details</a></li>'
    ,'<li><a href="#tabs-advanced-options">Advanced Options</a></li>'
```

```
    ;
// add plugin tabs here
echo '</ul>';
```

Above the page form, we display a small message if the page we're viewing is currently not visible in the navigation menu on the front-end of the site—if the `special` field has its 2 bit flagged, then that means that the page is not shown in the navigation menu.

Bitmasks are useful for when you have "yes/no" values and don't want to take up a whole database field for each value.

After this, we open the form.

Note that an action parameter is not provided in my code. Although the W3C HTML 4.01 specification says that the action is required, no browsers actually enforce this. If a browser comes across a form which has no action, then it defaults to the same page.

This is also true of `<style>`, where type defaults to `text/css`, and `<script>`, where type defaults to `javascript`.

Next we display the tab menu, which is the list of tabs to be shown.

Note the second-last line, which is a comment about plugin tabs. When we get to plugins in a later chapter, some of them may have enough extra options that they need a new tab on the page form. We'll handle that when we get to it.

Next, let's add the common details tab to the same file:

```
// { Common Details
echo '<div id="tabs-common-details"><table
    style="clear:right;width:100%;"><%;">< tr>';
// { name
echo '<th width="5%">name</th><td width="23%">
  <input
    id="name" name="name"
    value="',htmlspecialchars($page['name']),'" /></td>';
// }
// { title
echo '<th width="10%">title</th><td width="23%">
  <input
    name="title"
    value="',htmlspecialchars($page['title']),'" /></td>';
// }
// { url
echo '<th colspan="2">';
```

```
if($edit){
  $u='/'.str_replace(' ','-',$page['name']);
  echo '<a style="font-weight:bold;color:red" href="',$u
      ,'" target="_blank">VIEW PAGE</a>';
}
else echo ' ';
echo '</th>';
// }
echo '</tr><tr>';
// { type
echo '<th>type</th><td><select name="type"><option
    value="0">normal</option>';
// insert plugin page types here
echo '</select></td>';
// }
// { parent
echo '<th>parent</th><td><select name="parent">';
if($page['parent']){
  $parent=Page::getInstance($page['parent']);
  echo '<option value="',$parent->id,'">'
    ,htmlspecialchars($parent->name),'</option>';
}
else echo '<option value="0"> -- ','none',' -- </option>';
echo '</select>',"\n\n",'</td>';
// }
if(!isset($page['associated_date']) || !preg_match(
  '/^[0-9]{4}-[0-9]{2}-[0-9]{2}$/',$page['associated_date']
  ) || $page['associated_date']=='0000-00-00'
    $page['associated_date']=date('Y-m-d');
echo '<th>Associated Date</th><td><input
    name="associated_date" class="date-human" value="',
    $page['associated_date'],'" /></td>';
echo '</tr>';
// }
// { page-type-specific data
echo '<tr><th>body</th><td colspan="5">';
echo '<textarea name="body">',
    htmlspecialchars($page['body']),'</textarea>';
echo '</td></tr>';
// }
echo '</table></div>';
// }
```

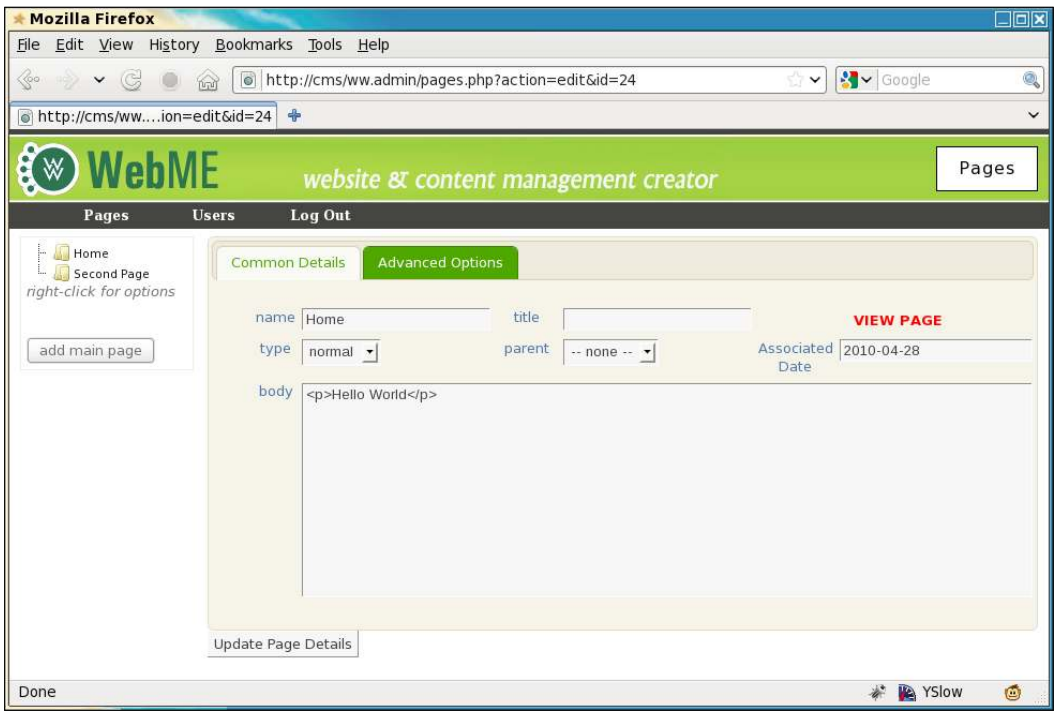
The given code shows the commonly changed details of the page database table. They include:

- `name`
- `title`
- `type`
- `parent`
- `associated_date`
- `body`

We'll enhance a few of those after we've finished the form. For now, a few things should be noted about the form and its options.

- **URL**: When you are editing a page, it's good to view it in another window or tab. To do this, we provide a link to the front-end page. Clicking on the link opens a new tab or window.
- **Type**: The page type by default is "normal", and in the select-box we built previously, that is the only option. We will enhance that when we get to plugins in a later chapter.
- **Parent**: This is the page which the currently edited page is contained within. In the earlier form, we display only the current parent, and don't provide any other options. There's a reason for that which we'll explain after we finish the main form HTML.
- **Associated date**: There are a number of dates associated with a page. We record the created and last-edited date internally (useful for plugins or logging), but sometimes the admin wants to record a date specific to the page. For example, if the page is part of a news system, we will enhance this date input box after the form is completed.
- **Body**: This is the content which will be shown on the front-end. It's plain HTML. Of course, writing HTML for content is not a task you should push on the average administrator, so we will enhance that.

Here's a screenshot of the first tab (I've temporarily completed the jQuery tabs to get this shot—we'll do it in the chapter later on):



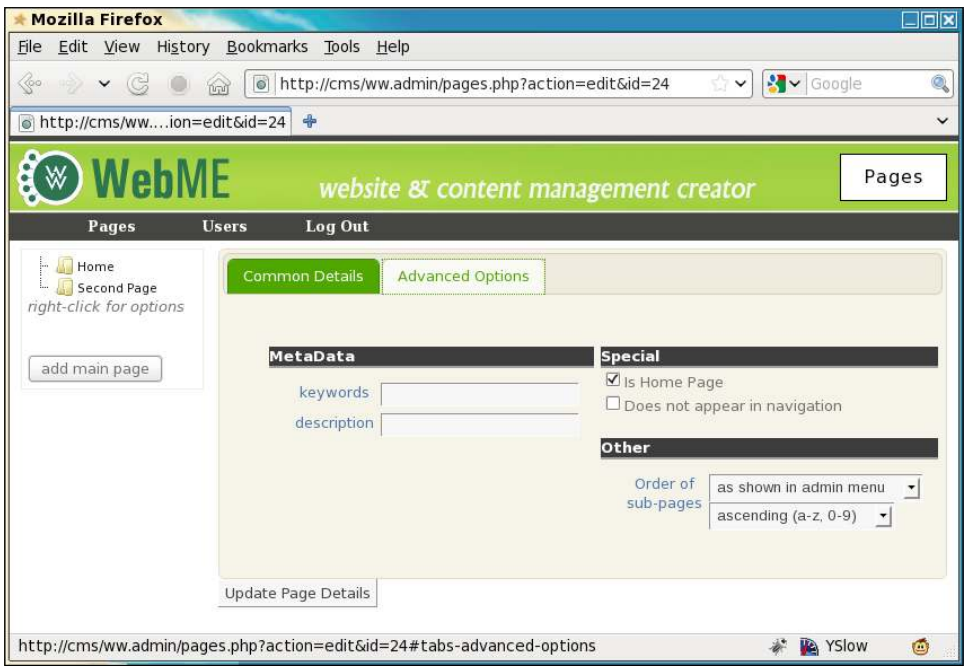
You can see that the date input box is quite large. There's a reason for that, which we'll see in the next chapter.

The second tab will be a bit shorter. Let's add that now. Add the following code to the `/ww.admin/pages/forms.php` file:

```
// { Advanced Options
echo '<div id="tabs-advanced-options">';
echo '<table><tr><td>';
// { metadata
echo '<h4>MetaData</h4><table>';
echo '<tr><th>keywords</th><td>
  <input name="keywords"
    value="',htmlspecialchars($page['keywords']),'"
  /></td></tr>';
echo '<tr><th>description</th><td>
  <input name="description"
    value="',htmlspecialchars($page['description']),'"
  /></td></tr>';
```

```
// { template
// we'll add this in the next chapter
// }
echo '</table>';
// }
echo '</td><td>';
// { special
echo '<h4>Special</h4>';
$specials=array('Is Home Page',
    'Does not appear in navigation');
for($i=0;$i<count($specials);++$i){
  if($specials[$i]!=''){
    echo '<input type="checkbox" name="special[',$i,']"';
    if($page['special']&pow(2,$i))echo ' checked="checked"';
    echo ' />',$specials[$i],'<br />';
  }
}
// }
// { other
echo '<h4>Other</h4>';
echo '<table>';
// { order of sub-pages
echo '<tr><th>Order of sub-pages</th><td><select name="page_
vars[order_of_sub_pages]">';
$arr=array('as shown in admin menu','alphabetically',
    'by associated date');
foreach($arr as $k=>$v){
  echo '<option value="',$k,'"';
  if(isset($page_vars['order_of_sub_pages']) &&
      $page_vars['order_of_sub_pages']==$k)
      echo ' selected="selected"';
  echo '>',$v,'</option>';
}
echo '</select>';
echo '<select name="page_vars[order_of_sub_pages_dir]">
    <option value="0">ascending (a-z, 0-9)</option>';
echo '<option value="1"';
if(isset($page_vars['order_of_sub_pages_dir']) &&
    $page_vars['order_of_sub_pages_dir']=='1')
    echo ' selected="selected"';
echo '>descending (z-a, 9-0)</option></select></td></tr>';
// }
echo '</table>';
// }
echo '</td></tr></table></div>';
// }
```

There's not a lot to explain here. There are some extra "advanced" options which I've not added here, which are useful for the system when it's been more completed (plugins added, themes or templates completed, and so on).



First, we add inputs for **keywords** and **description** meta-data. Most people appear to leave these alone, which is why it's not on the front tab.

We will add templates and themes in the next chapter. For now, I've added a commented placeholder.

After this, we show a list of "specials". I've included just two here—a marker to say whether the current page is the home page, and another marker to indicate that the page should not appear in front-end navigation.

Finally (for now), we show two drop-down boxes, to let the administrator decide what order the current page's sub-pages should be shown in the front-end navigation. For example, you might want a list of authors to be alphabetical or new items to appear by date descending, but in most cases you will want the pages to appear in the same order as they appear in the admin area (which you can change by dragging page names in the navigation menu on the left-hand side).

Okay—now let's complete the form and add in the tabs code.

There is one more section which we could add—some plugins might want to add tabs to this form. We'll get to that later in the book.

Add this code to the file `/ww.admin/pages/forms.php`:

```
echo '</div><input type="submit" name="action" value="',
    ($edit?'Update Page Details':'Insert Page Details')
    ,'" /></form>';
echo '<script>window.currentpageid='.$id.';</script>';
echo '<script src="/ww.admin/pages/pages.js"></script>';
```

And let's create the file `/ww.admin/pages/pages.js`:

```
$(function(){
  $('.tabs').tabs();
});
```

The `window.currentpageid` variable will be used in the next section.

That completes the basics of the form.

Next, let's look at those inputs we highlighted earlier as needing some enhancements.

Filling the parent selectbox asynchronously

In very large websites, it can sometimes be very slow to load up the `Page` form, because of the "parents" drop-down. This select-box tells the server what page the current page is located under.

If you fill that at the time of loading the form, then the size of the downloaded HTML can be quite large.

A solution for this problem was developed for my previous book (*jQuery 1.3 with PHP*), and as part of that book, the solution was packaged into a jQuery plugin which solves the problem here.

Download the `remoteselectoptions` plugin from `http://plugins.jquery.com/project/remoteselectoptions` and unzip it in your `/j/` directory.

What this plugin does, is that in the initial load of your page's HTML, you enter just one option in the select-box, and it will get the rest of the options only when it becomes necessary (that is, when the select-box is clicked).

To get this to work with the **parents** select-box, change the `/ww.admin/pages/pages.js` file to this:

```
$(function(){
  $('.tabs').tabs();
  $('#pages_form select[name=parent]').remoteselectoptions({
    url:'/ww.admin/pages/get_parents.php',
    other_GET_params:currentpageid
  });
});
```

And because this plugin is useful for quite a few places in the admin, let's add it to `/ww.admin/header.php` (the highlighted line):

```
<script src="http://ajax.googleapis.com/ajax/libs
    /jqueryui/1.8.0/jquery-ui.min.js"></script>
<script src="/j/jquery.remoteselectoptions
    /jquery.remoteselectoptions.js"></script>
<link rel="stylesheet" href="http://ajax.googleapis.com/ajax
    /libs/jqueryui/1.8.0/themes/south-street/jquery-ui.css"
    type="text/css" />
```

And you can see from the `pages.js` file that another file is required to build up the actual list of page names. Create this as `/ww.admin/pages/get_parents.php`:

```
<?php
require '../admin_libs.php';

function page_show_pagenames($i=0,$n=1,$s=0,$id=0){
  $q=dbAll('select name,id from pages where parent="'
      .$i.'" and id!="'.$id.'" order by ord,name');
  if(count($q)<1)return;
  foreach($q as $r){
    if($r['id']!=''){
      echo '<option value="'.$r['id'].'" title="'
          .htmlspecialchars($r['name']).'"';
      echo($s==$r['id'])?' selected="selected">':'>';
      for($j=0;$j<$n;$j++)echo ' ';
      $name=$r['name'];
      if(strlen($name)>20)$name=substr($name,0,17).'...';
      echo htmlspecialchars($name).'</option>';
      page_show_pagenames($r['id'],$n+1,$s,$id);
    }
  }
}

$selected=isset($_REQUEST['selected'])
```

```
    ?$_REQUEST['selected']:0;
$id=isset($_REQUEST['other_GET_params'])
    ?(int)$_REQUEST['other_GET_params']:-1;
echo '<option value="0"> --  none  -- </option>';
page_show_pagenames(0,0,$selected,$id);
```

The `remoteselectoptions` plugin sends a query to this page, with two parameters—the currently selected parent's ID, and the current page ID.

The previous code builds up an option list, taking care to not allow the admin to choose to place a page within itself, or within any page which is contained hierarchically under itself. That would make the page disappear from all navigation, including the admin navigation.

For the current example, that means that the only options available are either **none** (that is, the page is a top-level one), or **Second Page**, as in our example, there are currently only two pages, and obviously you can't place **Home** under **Home**.

Okay—we've done enough now that you can take a break before we start on the next chapter, where we'll finish off page creation.

Summary

In this chapter, we built the basics of page management, including creation of the form for page management, and a few jQuery tools for making page location management easy and improving the selection of large select-boxes.

In the next chapter, we will complete the page management section, and build a simple menu system for the front-end so we can navigate between pages.