4
Page Management – Part Two

In this chapter, we will complete the page-management section, and will build a simple navigation menu for the front-end.

We will discuss the following topics:

- How to make human-readable dates
- Rich-text editing
- File management for images and files

At the end of this chapter, we will have a completed page management system.

Dates

Dates are annoying. The scheme I prefer is to enter dates the same way MySQL accepts them—`yyyy-mm-dd hh:mm:ss`. From left to right, each subsequent element is smaller than the previous. It's logical, and can be sorted sensibly using a simple numeric sorter.

Unfortunately, most people don't read or write dates in that format. They'd prefer something like `08/07/06`.

Dates in that format do not make sense. Is it the 8th day of the 7th month of 2006, or the 7th day of the 8th month of 2006, or even the 6th day of the 7th month of 2008? Date formats are different all around the world.

Therefore, you cannot trust human administrators to enter the dates manually.

A very quick solution is to use the jQuery UI's `datepicker` plugin.

Temporarily (we'll remove it in a minute) add the highlighted lines to `/ww.admin/pages/pages.js`:

```
        other_GET_params:currentpageid
      });
      $('.date-human').datepicker({
        'dateFormat':'yy-mm-dd'
      });
    });
```

When the date field is clicked, this appears:

It's a great calendar, but there's still a flaw: Before you click on the date field, and even after you select the date, the field is still in `yyyy-mm-dd` format.

While MySQL will thank you for entering the date in a sane format, you will have people asking you why the date is not shown in a humanly readable way.

We can't simply change the date format to accept something more reasonable such as "May 23rd, 2010", because we would then need to ensure that we can understand this on the server-side, which might take more work than we really want to do.

So we need to do something else.

The `datepicker` plugin has an option which lets you update two fields at the same time. This is the solution—we will display a dummy field which is humanly readable, and when that's clicked, the calendar will appear and you will be able to choose a date, which will then be set in the human-readable field and in the real form field.

Don't forget to remove that temporary code from `/ww.admin/pages/pages.js`.

Because this is a very useful feature, which we will use throughout the admin area whenever a date is needed, we will add a global JavaScript file which will run on all pages.

Edit `/ww.admin/header.php` and add the following highlighted line:

```
<script src="/j/jquery.remoteselectoptions
    /jquery.remoteselectoptions.js"></script>
<script src="/ww.admin/j/admin.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 then we'll create the `/ww.admin/j/` directory and a file named `/ww.admin/j/admin.js`:

```
function convert_date_to_human_readable(){
  var $this=$(this);
  var id='date-input-'+Math.random().toString()
    .replace(/\./,'');
  var dparts=$this.val().split(/-/);
  $this
    .datepicker({
      dateFormat:'yy-mm-dd',
      modal:true,
      altField:'#'+id,
      altFormat:'DD, d MM, yy',
      onSelect:function(dateText,inst){
        this.value=dateText;
      }
    });
  var $wrapper=$this.wrap(
      '<div style="position:relative" />');
  var $input=$('<input id="'+id+'" class="date-human-readable"
      value="'+date_m2h($this.val())+'" />');
  $input.insertAfter($this);
  $this.css({
    'position':'absolute',
    'opacity':0
  });
  $this
    .datepicker(
      'setDate', new Date(dparts[0],dparts[1]-1,dparts[2])
    );
}
$(function(){
  $('input.date-human').each(convert_date_to_human_readable);
});
```

This takes the computer-readable date input and creates a copy of it, but in a human-readable format.

The original date input box is then made invisible and laid across the new one. When it is clicked, the date is updated on both of them, but only the human-readable one is shown.

Much better. Easy for a human to read, and also usable by the server.

Saving the page

We created the form, and except for making the body textarea more user-friendly, it's just about finished. Let's do that now.

When you click on the **Insert Page Details** button (or **Update Page Details**, if an ID was provided), the form data is posted to the server.

We need to perform these actions before the page menu is displayed, so it is up-to-date.

Edit `/ww.admin/pages.php`, and add the following highlighted lines before the `load menu` section:

```
echo '<h1>Pages</h1>';

// { perform any actions
if(isset($_REQUEST['action'])){
  if($_REQUEST['action']=='Update Page Details'
    || $_REQUEST['action']=='Insert Page Details'){
    require 'pages/action.edit.php';
  }
  else if($_REQUEST['action']=='delete'){
    'pages/action.delete.php';
  }
}
// }
// { load menu
```

If an `action` parameter is sent to the server, then the server will use this block to decide whether you want to edit or delete the page. We'll handle deletes later in the chapter.

Notice that we are handling inserts and updates with the same file, `action.edit.php`—in the database, there is almost no difference between the two when using MySQL.

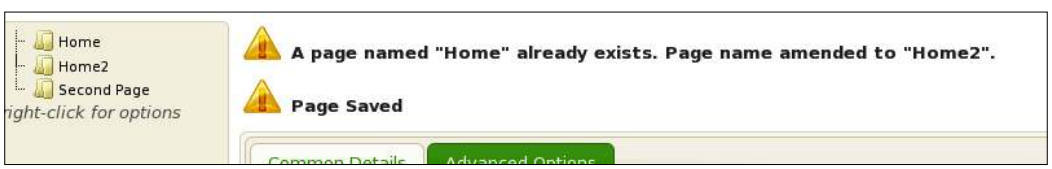
So, let's create that file now. We'll do it a bit at a time, like how we did the form, as it's a bit long.

Create `/ww.admin/pages/action.edit.php` with this code:

```
<?php
function pages_setup_name($id,$pid){
  $name=trim($_REQUEST['name']);
  if(dbOne('select id from pages where
      name="'.addslashes($name).'" and parent='.$pid.'
      and id!='.$id,'id')){
    $i=2;
    while(dbOne('select id from pages where
        name="'.addslashes($name.$i).'" and parent='.$pid.'
        and id!='.$id,'id'))$i++;
    echo '<em>A page named "'.htmlspecialchars($name).'"
        already exists. Page name amended to "'
        .htmlspecialchars($name.$i).'".</em>';
    $name=$name.$i;
  }
  return $name;
}
```

The first piece is a function which tests the submitted page name. If that name is the same as another page which has the same parent, then a number is added to the end and a message is shown explaining this.

Here's an example, creating a page named "Home" in the top level (we already have a page named "Home"):



Next we'll create a function for testing the inputted `special` variable. Add this to the same file:

```
function pages_setup_specials($id=0){
  $special=0;
  $specials=isset($_REQUEST['special'])
      ?$_REQUEST['special']:array();
  foreach($specials as $a=>$b)
      $special+=pow(2,$a);
  $homes=dbOne("SELECT COUNT(id) AS ids FROM pages
      WHERE (special&1) AND id!=$id",'ids');
  if($special&1){ // there can be only one homepage
    if($homes!=0){
      dbQuery("UPDATE pages SET special=special-1
          WHERE special&1");
    }
  }
  else{
    if($homes==0){
      $special+=1;
      echo '<em>This page has been marked as the site\'s
        Home Page, because there must always be one.</em>';
    }
  }
  return $special;
}
```

In this function, we build up the `special` variable, which is a bit field.

A **bit** field is a number which uses binary math to combine a few "yes/no" answers into one single value. It's good for saving space and fields in the database.

Each value has a value assigned to it which is a power of two. The interesting thing to note about powers of two is that in binary, they're always represented as a 1 with some 0s after it. For example, 1 is represented as 00000001, 2 is 00000010, 4 is 00000100, and so on.

When you have a bit field such as 00000011 (each number here is a bit), it's easy to see that this is composed of the values 1 and 2 combined, which are 2^0 and 2^1 respectively.

The `&` operator lets us check quickly if a certain bit is turned on (is 1) or not. For example, `00010011 & 16` is true, and `00010011 & 32` is false, because the 16 bit is on and the 32 bit is off.

In the database, we set a bit for the homepage, which we say has a value of 1. In the previous function, we need to make sure that after inserting or updating a page, there is always exactly one homepage.

The only other one we've set so far is "does not appear in navigation menu", which we've given the value 2. If we added a third bitflag ("is a 404 handler", for example), it would have the value 4, then 8, and so on.

Okay—now we will set up our variables. Add this to the same file:

```
// { set up common variables
$id              =(int)$_REQUEST['id'];
$pid             =(int)$_REQUEST['parent'];
$keywords        =$_REQUEST['keywords'];
$description     =$_REQUEST['description'];
$associated_date =$_REQUEST['associated_date'];
$title           =$_REQUEST['title'];
$name            =pages_setup_name($id,$pid);
$body            =$_REQUEST['body'];
$special         =pages_setup_specials($id);
if(isset($_REQUEST['page_vars']))
    $vars=json_encode($_REQUEST['page_vars']);
else $vars='[]';
// }
```

Then we will add the main body of the page update SQL to the same file:

```
// { create SQL
$q='edate=now(),type="'.addslashes($_REQUEST['type']).'",
    associated_date="'.addslashes($associated_date).'",
    keywords="'.addslashes($keywords).'",
    description="'.addslashes($description).'",
    name="'.addslashes($name).'",
    title="'.addslashes($title).'",
    body="'.addslashes($body).'",parent='.$pid.',
    special='.$special.',vars="'.addslashes($vars).'"';
// }
```

This is SQL which is common to both creating and updating a page.

Finally we run the actual query and perform the action. Add this to the same file:

```
// { run the query
if($_REQUEST['action']=='Update Page Details'){
  $q="update pages set $q where id=$id";
  dbQuery($q);
}
```

```
else{
  $q="insert into pages set cdate=now(),$q";
  dbQuery($q);
  $_REQUEST['id']=dbLastInsertId();
}
// }
echo '<em>Page Saved</em>';
```

In the first case, we simply run an update.

In the second, we run an `insert`, adding the creation date to the query, and then setting `$_REQUEST['id']` to the ID of the entry that we just created.

Creating new top-level pages

If you've been trying all this, you'll have noticed that you can create a top-level page simply by clicking on the admin area's **Pages** link in the top menu, and then you're shown an empty **Insert Page Details** form.

It makes sense, though, to also have it available from the pagelist on the left-hand side.

So, let's make that **add main page** button useful.

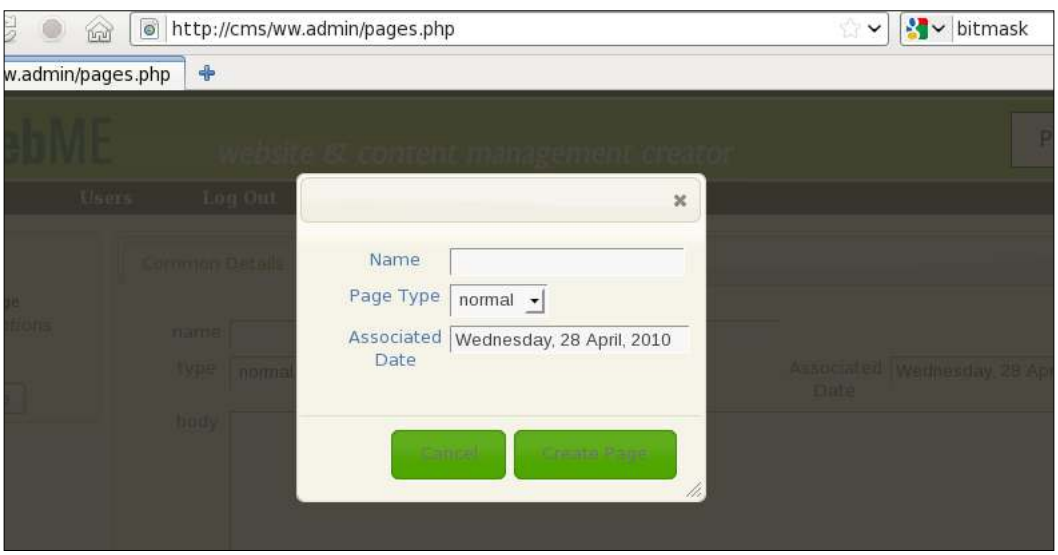
If you remember, we created a `pages_add_main_page` function in the `menu.js` file, just as a placeholder until we got everything else done.

Open up that file now, `/ww.admin/pages/menu.js`, and replace that function with the following two new functions:

```
function pages_add_main_page(){
  pages_new(0);
}
function pages_new(p){
  $('<form id="newpage_dialog" action="/ww.admin/pages.php"
       method="post">
      <input type="hidden" name="action"
          value="Insert Page Details" />
      <input type="hidden" name="special[1]"
          value="1" />
      <input type="hidden" name="parent" value="'+p+'" />
      <table>
       <tr><th>Name</th><td><input name="name" /></td></tr>
       <tr><th>Page Type</th><td><select name="type">
           <option value="0">normal</option>
           </select></td></tr>
```

```
          <tr><th>Associated Date</th><td>
              <input name="associated_date" class="date-human"
              id="newpage_date" /></td></tr>
          </table>
         </form>')
       .dialog({
        modal:true,
        buttons:{
          'Create Page': function() {
            $('#newpage_dialog').submit();
          },
          'Cancel': function() {
            $(this).dialog('destroy');
            $(this).remove();
          }
        }
      });
      $('#newpage_date').each(convert_date_to_human_readable);
      return false;
    }
```

When the **add main page** button is clicked, a dialog box is created asking some basic information about the page to create:



We include a few hidden inputs.

- `action`: To tell the server this is an **Insert Page Details** action.
- `special`: When **Create Page** is clicked, the page will be saved in the database, but we should hide it initially so that front-end readers don't see a half-finished page. So, the `special[1]` flag is set ($2^1 == 2$, which is the value for hiding a page).
- `parent`: Note that this is a variable. We can use the same dialog to create sub-pages.

When the dialog has been created, the date input box is converted to human-readable, the same as we did earlier.

Creating new sub-pages

We will add sub-pages by using context menus on the page list. Note that we have a message saying **right-click for options** under the list.

First, add this function to the `/ww.admin/pages/menu.js` file:

```
function pages_add_subpage(node,tree){
  var p=node[0].id.replace(/.*_/,'');
  pages_new(p);
}
```

We will now need to activate the context menu. This is done by adding a `contextmenu` plugin to the `jstree` plugin. Luckily, it comes with the download, so you've already installed it. Add it to the page by editing `/ww.admin/pages/menu.php` and add this highlighted line:

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

And now, we edit the `.tree()` call in `/ww.admin/menu.js` to tell it what to do:

```
  $('#pages-wrapper').tree({
    callback:{
// SKIPPED FOR BREVITY - DO NOT DELETE THESE LINES
    },
    plugins:{
      'contextmenu':{
        'items':{
```

```
            'create' : {
              'label' : "Create Page",
              'icon'  : "create",
              'visible' : function (NODE, TREE_OBJ) {
                if(NODE.length != 1) return 0;
                return TREE_OBJ.check("creatable", NODE);
              },
              'action':pages_add_subpage,
              'separator_after' : true
            },
            'rename':false,
            'remove':false
          }
        }
      }
    });
```

By default, the contextmenu has three links: `create`, `rename`, and `remove`. You need to turn off any you're not currently using by setting them to false.

Now if you right-click on any page name in the pagelist, you will have a choice to create a sub-page under it.

Deleting pages

We will add deletions in the same way, using the context menu.

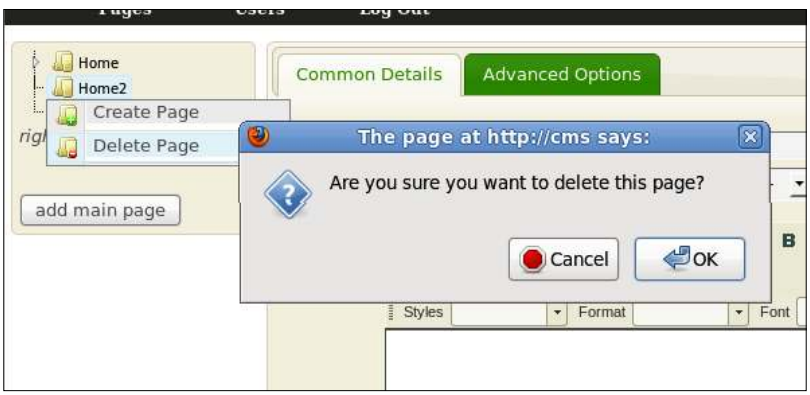
Edit the same file, and this time in the contextmenu code, replace the `remove: false` line with these:

```
            'remove' : {
              'label' : "Delete Page",
              'icon'  : "remove",
              'visible' : function (NODE, TREE_OBJ) {
                if(NODE.length != 1) return 0;
                return TREE_OBJ.check("deletable", NODE);
              },
              'action':pages_delete,
              'separator_after' : true
            }
```

And add the `pages_delete` function to the same file:

```
function pages_delete(node,tree){
  if(!confirm(
      "Are you sure you want to delete this page?"))return;
  $.getJSON('/ww.admin/pages/delete.php?id='
      +node[0].id.replace(/.*_/,''),function(){
    document.location=document.location.toString();
  });
}
```

One thing to always keep in mind is whenever creating any code that deletes something in your CMS, you must ask the administrator if he/she is sure, just to make sure it wasn't an accidental click. If the administrator confirms that the click was intentional, then it's not your fault if something important was deleted.



So, the `pages_delete` function first checks for this, and then calls the server to remove the file. The page is then refreshed because this may significantly change the page list tree, as we'll see now.

Create the `/ww.admin/pages/delete.php` file:

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

$id=(int)$_REQUEST['id'];
if(!$id)exit;

$r=dbRow("SELECT COUNT(id) AS pagecount FROM pages");
if($r['pagecount']<2){
  die('cannot delete - there must always be one page');
}
else{
  $pid=dbOne("select parent from pages
```

```
        where id=$id",'parent');
    dbQuery("delete from pages where id=$id");
    dbQuery("update pages set parent=$pid where parent=$id");
}
echo 1;
```

First, we ensure that there is always at least one page in the database. If deleting this page would empty the table, then we refuse to do it. There is no need to add an alert explaining this, as it should be clear to anyone that deleting the last remaining page in a website leaves the website with no content at all. Simply refusing the deletion should be enough.

Next, we delete the page.

Finally, any pages which were contained within that page (pages which had this one as their parent), are moved up in the tree so they are contained in the deleted page's old parent.

For example, if you had a page, `page1>page2>page3` (where `>` indicates the hierarchy), and you removed `page2`, then `page3` would then be in the position `page1>page3`.

This can cause a large difference in the treestructure if there were quite a few pages contained under the deleted one, so the page needs to be refreshed.

Rich-text editing using CKeditor

Anything entered into the body textarea will be displayed directly on the front-end. We've used a plain textarea for now, but this is not ideal.

It's not a good idea to assume that the administrator knows HTML. Most of them will not.

For a long time, the only reasonable solution for this was to use a text markup language such as Textism or BBCode, which allow you to enter text such as `this _is_ a *word*`, which will be converted to `this <em>is</em> a <strong>word</strong>`.

While that's a good compromise, in the last few years it has become possible to use "what you see is what you get"-style editors, where you can type into the textarea and use buttons or key combinations to style the text and see it right there.

These editors are known as **Rich-text Editors** (**RTEs**). When describing them to clients, though, I find it's easier to describe them as small Word-like editors. In fact, they're usually designed very similar to the wordprocessor packages that people use in their normal office work.

The first one I used was HTMLarea, but that project eventually was discontinued, and I moved onto FCKeditor. Recently, that project has been rewritten and is now available as CKeditor from `http://ckeditor.com/`.

While it is interesting to offer a choice of RTE or plaintext editing to the administrator (WordPress offers the choice, for example), I've never had a client which asked for plain text. After all, the point of a CMS is to ease the editing of websites and their pages and content, so why ruin this by then writing HTML instead of using an RTE?

Apart from CKeditor, the only other very popular RTE is TinyMCE. There are many other editors, but when you read about them, they are usually compared against CKeditor or TinyMCE.

So, let's start by downloading CKeditor from `http://ckeditor.com/download`—I'm using version 3.2.1. Download it and extract to `/j/`. It will create a directory called `/j/ckeditor/`.

CKeditor is useful enough that we will use it a lot in the admin area. So, we will add it as a plugin that's loaded on all pages. Edit `/ww.admin/header.php` and add the highlighted line:

```
<script src="/ww.admin/j/admin.js"></script>
<script src="/j/ckeditor/ckeditor.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" />
```

To display CKeditor, the method I prefer to use is to create a textarea, and convert it to an editor afterwards using some JavaScript.

The code to do this is repetitive enough that it makes sense to create a small function for it. Add this to `/ww.admin/admin_libs.php`:

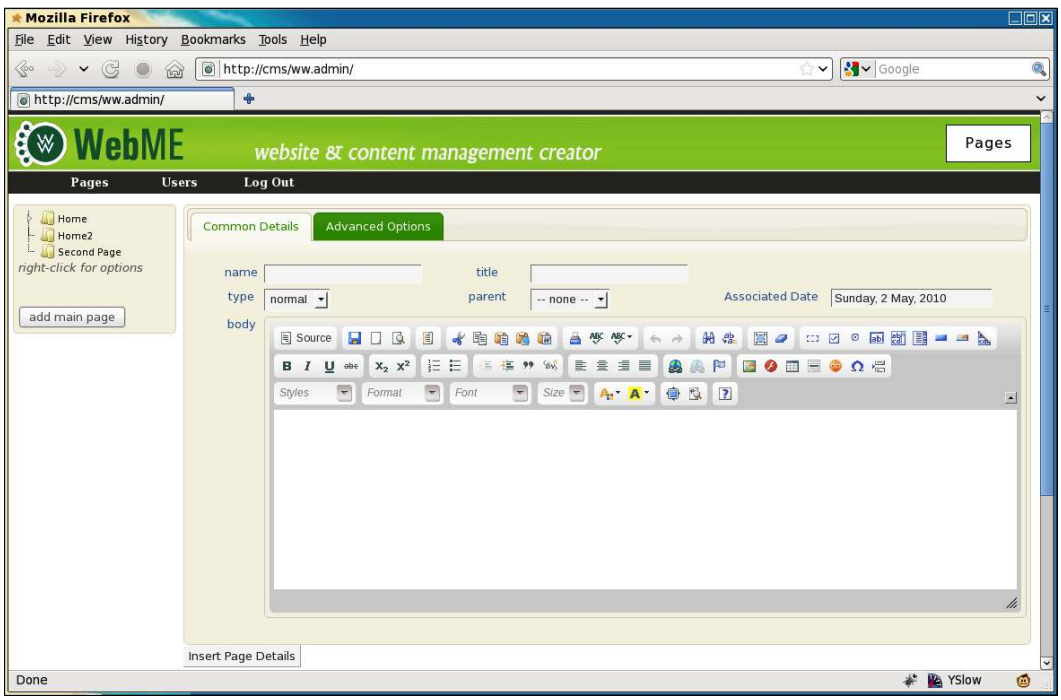
```
function ckeditor($name,$value='',$height=250){
  return '<textarea style="width:100%;height:'.$height.'px"
      name="'.addslashes($name).'">'.htmlspecialchars($value)
      .'</textarea><script>$(function(){
          CKEDITOR.replace("'.addslashes($name).'",{
          });
      });</script>';
}
```

The second parameter for the `CKEDITOR.replace` function call is to supply options to CKeditor. We'll get to that in a minute.

Now, let's use it. In `/ww.admin/pages/forms.php`, change the `page-type-specific data` block to the following:

```
// { page-type-specific data
echo '<tr><th>body</th><td colspan="5">';
echo ckeditor('body',$page['body']);
echo '</td></tr>';
// }
```

Now when you load up the page admin, you'll see we have the RTE embedded:



That toolbar is much too full though. The first time an administrator sees that, they would be terrified! Most things an admin will want to do in a web page are really simple—make something bold, insert an image or table, and so on.

I prefer to give the admin a much smaller toolbar.

You can do this by editing the `/j/ckeditor/config.js` file. Here's what I have:

```
CKEDITOR.editorConfig = function( config )
{
  config.skin="v2";
  config.toolbar="WebME";
```

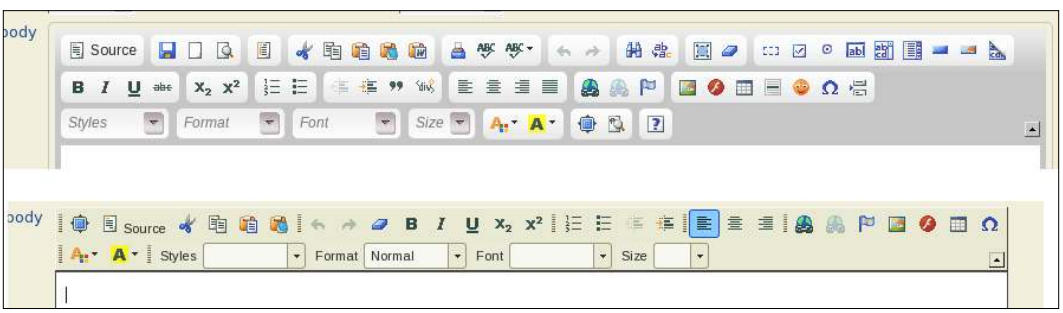
```
    config.toolbar_WebME=[
      ['Maximize','Source','Cut','Copy','Paste','PasteText'],
      ['Undo','Redo','RemoveFormat','Bold','Italic',
          'Underline','Subscript','Superscript'],
      ['NumberedList','BulletedList','Outdent','Indent'],
      ['JustifyLeft','JustifyCenter','JustifyRight'],
      ['Link','Unlink','Anchor','Image','Flash','Table',
          'SpecialChar'],
      ['TextColor','BGColor'],
      ['Styles','Format','Font','FontSize']
    ];
};
```

First, I've set the CKeditor skin to `"v2"`, which looks a bit more like what people are used to (Open Office Writer or MS Word). There are others in `/j/ckeditor/skins/` if you prefer to use a different one.

Next we set the editor to use the `"WebME"` set of buttons, and finally, we define those buttons.

Each sub-array is a group of buttons which are similar in purpose. If you resize the editor (notice that the bottom right-hand side of the editor is CMS Design With PHP and jQuery) then each of those groups of buttons will be kept together.

You can compare the default toolbar and the custom toolbar in the following screenshot, the new one is at the bottom:



You can see the custom toolbar is more compact, allowing more room in the textarea to see what is written, and is more like what you would find in the default toolbar of a word processor.

An administrator can use this editor as easily as they would use any word processor—you can copy or paste from sources such as websites or word processor documents and the formats will be mostly retained.

In fact, you can even copy from websites and any images in the copied text will also be copied! The `src` parameter of the image will be the original source, so you should not do this on websites which are not yours. Embedding an image on your site and yet linking to the original site can be seen as rude (it's called **inline linking**, but is also known as **leeching** and **hot-linking**) as you are using someone else's bandwidth to supply the image.

File management using KFM

So, let's say you want to upload your own images or files and link to them through the RTE?

When FCKeditor was the main RTE on the Internet, it had a file manager built in. This file manager allowed you to do some very basic things such as create directories or upload files.

That was about the limit of its capabilities though—if you wanted to rename a file, move it, delete it, and so on, there was simply no way to do it.

The file manager was limited for a number of reasons.

It was designed to work with several separate server-side languages, such as PHP, ASP, Java, and Perl, and adding any one feature meant writing the server-side implementation for it several times over. Concentrating on one single language was not acceptable to the development team, as they wanted it to appeal to all users of the RTE.

There is also that the team was developing a commercial file manager add-on for CKeditor called CKfinder, so enhancing the free file manager was a conflict of interest.

Luckily, the CKeditor plugin system is not hard to work with, so in 2006 I started building my own file manager specifically for use in a CMS with FCKeditor/CKeditor, which I called **KFM** (**Kae's File Manager**).

I started developing KFM with MooTools, but it soon became obvious that jQuery was much better for it, so it was converted to use jQuery. Using jQuery meant less code for me (and my co-developer Benjamin) to write, and also, the more jQuery I used the less I had to maintain myself, because there is a large community out there catering to jQuery.

Download a copy of KFM from `http://kfm.verens.com/`. You can use either the latest stable copy (1.4.5 at the time of writing this book), or the Nightly version, which is compiled each night from subversions. I will use the Nightly version, which you can get by clicking on **nightly** on the front page.

Extract it in `/j/`. It will create a directory called `trunk`. Rename that to `kfm`.

Now delete the `/j/kfm/admin/` directory. We will handle administration through the configuration files, and will not allow configuration on-the-fly by administrators.

How KFM works is that you tell it what directory it is to maintain (`/f/` in our case) and what database to use to manage the files. There are many other configuration settings, but those are the most important.

For the database, you can use PostGres, MySQL, or SQLite. I prefer to use SQLite, because it is stored as a single file, which you can actually save within the `/f/` directory itself (KFM creates a hidden directory called `/f/.files` and saves its data in there).

This makes it easy for you to back up the entire file storage system and move it to another server if you want, including the database as well. With MySQL or PostGres, you'd have to export the data to a file, move the files, then import the data on the far end.

KFM does a lot of image manipulation, in order to provide thumbnails, and so on. To do this, your PHP server needs to either have GD compiled (this is usually true), or to have ImageMagick installed.

I prefer to use ImageMagick, because it is quicker than GD, and less resource-hungry.

If you are building your CMS on a Windows server, you may have to use GD.

Configuration is managed by creating a file `/j/kfm/configuration.php`:

```
<?php
$kfm_userfiles_address=$_SERVER['DOCUMENT_ROOT'].'/f/';
$kfm_userfiles_output='/f/';
$kfm_dont_send_metrics=1;
```

You can see the full list of configuration settings in `/j/kfm/configuration.dist.php`. KFM loads the `dist` file first, and then your own file, so you only need to enter the options that you want to change.

In the previous code snippet, we set `$kfm_userfiles_address` to the full local address of the website's `/f/` directory (as if you were on the machine from a console and wanted to navigate to it).

We also set the `$kfm_userfiles_output` setting to `'/f/'`, to tell KFM where in the website's online-accessible directories the files are kept.

Finally, we set `$kfm_dont_send_metrics` to `1`. By default, when KFM is loaded up for the first time every day, it "phones home" to tell the KFM server how many people are using it, and what version is being used—this is so the developers have a reasonable idea what versions of KFM they should support and what can be safely decided to be obsolete. Setting this option to `1` tells KFM not to bother doing this (as the main developer of KFM, I am usually using the most up-to-date version).

Now, create the directory `/f/`, and make sure it is writable by the web server.

In Linux, you can accomplish this in a number of different ways. The simplest is to change the permissions on the directory to "0777" (which means read/write/execute for everyone). This is not advisable on servers where other people may have user accounts, such as virtual host accounts.

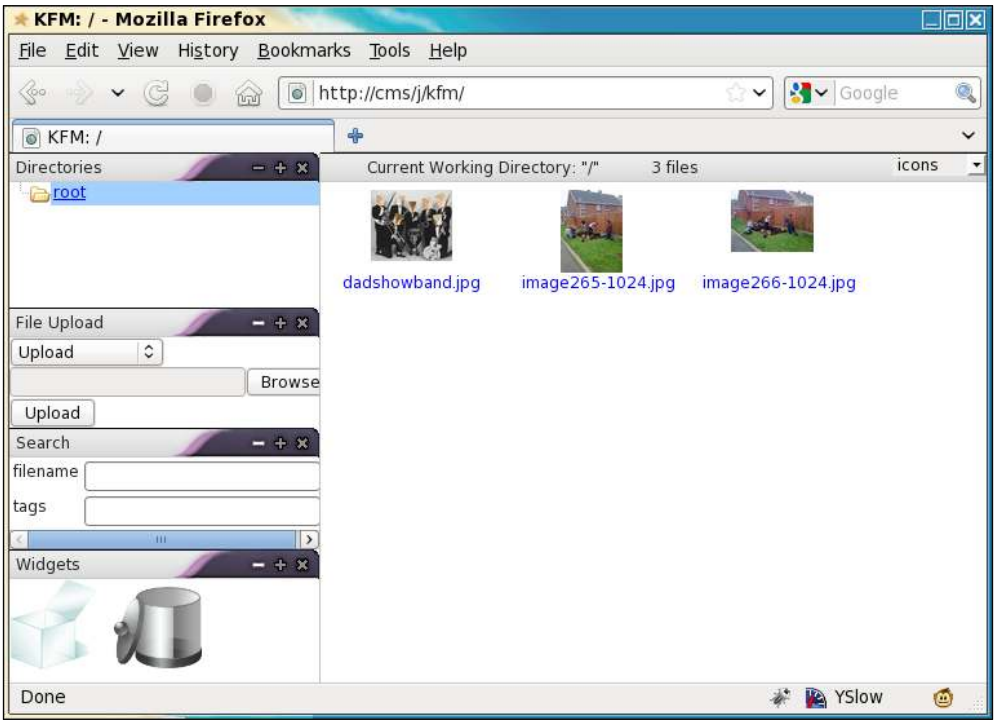
A better method is to use suPHP (`http://www.suphp.org/`), which runs PHP scripts using your own username instead of the webserver's. This means that you can secure the files so that they are only accessible or editable by you, and yet the webserver can still work with them.

Setup of suPHP is outside the scope of this book. Feel free to use whichever you want.

For the sake of simplicity, I will assume you are on your own server and no other person has an account on it (this is becoming much more popular lately, thanks to server virtualization), and so you can use the "0777" method.

You should now have an installed copy of KFM.

Test it by going to `/j/kfm/` in your browser. You should see something similar to the following screenshot (I've uploaded a few images through the **File Upload** section to test it):



If your installation fails, ask for help on the `kfm.verens.com` forum.

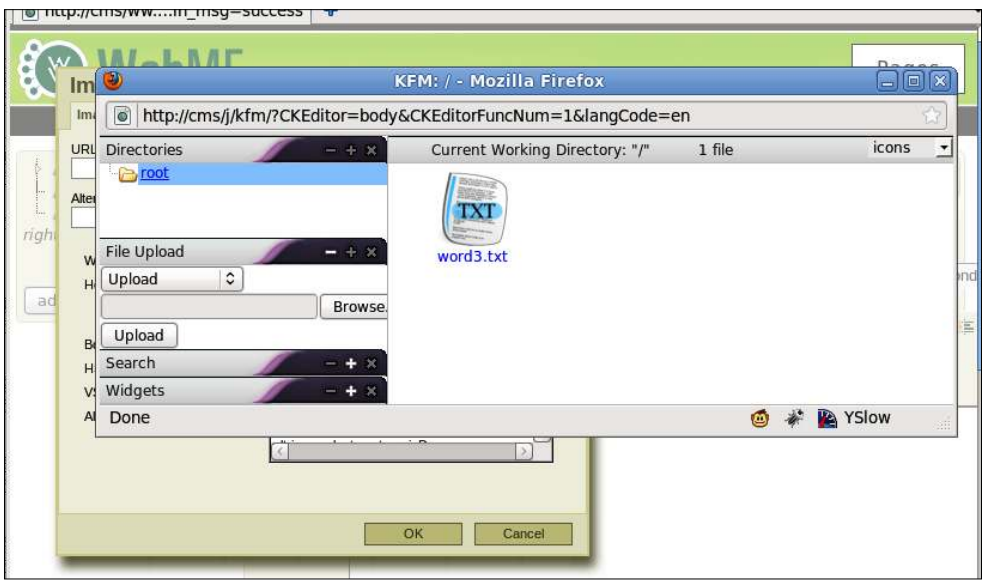
Okay—you should now have KFM installed. Now, let's connect it to CKeditor.

Hooking the two together is easy—in `/ww.admin/admin_libs.php`, add the following highlighted line to the `CKEDITOR` function:

```
CKEDITOR.replace("'.addslashes($name).'",{
    filebrowserBrowseUrl:"/j/kfm/"
});
```

Now when you click on the image icon or link icon in CKeditor, the pop-up window will have a **Browse Server** button:-

Clicking on that will pop up a new window with KFM in it to let you select the file you want:



We are almost finished.

We now need to make sure that only authorized users can log into KFM.

After KFM has loaded its configuration, it then loads up another file if it exists, `/j/kfm/api/config.php`, which is there for developers, in case they want to integrate their CMSes with KFM.

By default this file does not exist in the usual distribution.

Create it and add this content:

```
<?php
if($_SERVER['PHP_SELF']=='/j/kfm/get.php' ||
  (isset($kfm_api_auth_override) && $kfm_api_auth_override))
      $inc='/ww.incs/basics.php';
else $inc='/ww.admin/admin_libs.php';
include_once $_SERVER['DOCUMENT_ROOT'].$inc;

$kfm_userfiles_address=$_SERVER['DOCUMENT_ROOT'].'/f/';
if(!session_id()){
  if(isset($_GET['cms_session']))
      session_id($_GET['cms_session']);
  session_start();
}
if($_SERVER['PHP_SELF']=='/j/kfm/get.php'){
  $kfm_do_not_save_session=true;
}
$kfm_api_auth_override=true;
$kfm->defaultSetting('file_handler','return');
$kfm->defaultSetting('file_url','filename');
$kfm->defaultSetting('return_file_id_to_cms',false);
```

The first few lines are the most important.

When a file is retrieved through KFM, it always comes through `/j/kfm/get.php`. Within the KFM interface, for example, the thumbnails are retrieved through that script, and if a file is downloaded, it is downloaded through that script.

So, if that file is loaded in a browser, it needs to be allowed.

Later in the book, we will see plugins which will use KFM's functions to accomplish some things, such as the gallery plugin. To allow those plugins to use KFM, we need to set a variable in the plugin to true (`$kfm_api_auth_override`). If KFM loads and that variable is set, then permission is granted.

Otherwise, the browser must be logged in as an administrator.

The rest of the lines are additional configuration options which are generally done through the KFM admin area (which we've deleted), and some optimization settings as well.

One final thing needs to be done.

KFM uses an `__autoload` function to load its classes. WebME also uses an `__autoload` function. You can only have one of these.

So, we'll rewrite the `__autoload` function in `/ww.incs/basics.php` to only get set if no other function of that name exists. Add the following highlighted lines:

```
if(!function_exists('__autoload')){
  function __autoload($name) {
    require $name . '.php';
  }
}
```

And that's it! Now if you log out of the admin area, and try go to `/j/kfm/`, you will see that you are asked to log in again.

Summary

In this chapter, we finished the page management system.

This included the display and management of page data in the admin area, embedding a rich-text editor, and adding a filemanagement package as well.

In the next chapter, we will look at theme management, and displaying pages and page navigation menus on the front-end.