

8

Forms Plugin

In this chapter, we will create a plugin for providing a form generator which can be used for sending the submitted data as an e-mail, or saving in the database.

By building this plugin, we will extend the core engine so that it can display custom content forms in the page admin and custom page types. We will also adjust the output on the front-end so it can display from the plugin instead of straight from the page's body field.

How it will work

There are a number of ways to create a form. Probably the simplest way is to allow the administrator to "draw" the form using a rich text editor, and treat the submitted `$_POST` values as being correct.

There are a number of reasons why POST should be used for forms instead of GET:

A form may contain file-upload inputs, requiring multi-part encoded POST data.

A form may contain textareas, which could have arbitrarily long tracts of text pasted in them (GET has a rather short limit on the number of characters allowed).

When a form is submitted through POST, and the user reloads the page, the browser pops up a warning asking if the user is sure that he/she wants to post the form data again. This is better than accidentally sending forms over and over again.

This method has the disadvantage that the server doesn't know what type of data was supposed to be inputted, so it can't validate it.

A more robust method is to define each field that will be in the form, then create a template which will be shown to the user.

This allows us to autogenerate server-side and client-side validation. For example, if the input is to be an e-mail address, then we can ensure that only e-mails are entered into it. Similar tests can be done for numbers, select-box values, and so on.

We could make this comprehensive and cover all forms of validation. In this chapter, though, we will build a simple forms plugin that will cover most cases that you will meet when creating websites.

We will also need to define what is to be done with the form after it's submitted – e-mail it or store in a database.

Because we're providing a method of saving to database, we will also need a way of exporting the saved values so they can be read in an office application. CSV is probably the simplest format to use, so we'll write a CSV exporter.

And finally, because we don't want robots submitting rubbish to your form or trying to misuse it, we will have the option of using a captcha.

The plugin config

Plugins are usually started by creating the definition file. So, create the directory / `ww.plugins/forms` and add this file to it:

```
<?php
$plugin=array(
    'name' => 'Form',
    'admin' => array(
        'page_type' => array(
            'form' => 'form_admin_page_form'
        )
    ),
    'description' =>
        'Generate forms for sending as email or saving in the database',
    'frontend' => array(
        'page_type' => array(
            'form' => 'form_frontend'
        )
    ),
    'version' => 3
);
function form_admin_page_form($page,$page_vars){
    $id=$page['id'];
    $c='';
    require dirname(__FILE__).'/admin/form.php';
```

```

    return $c;
}
function form_frontend($PAGEDATA) {
    require dirname(__FILE__) . '/frontend/show.php';
    return $PAGEDATA->render().form_controller($PAGEDATA);
}

```

In the admin section of the `$plugin` array, we have a new value, `page_type`. We are going to handle forms as if they were full pages. For example, you may have a contact page where the page is predominantly taken over by the form itself.

The `page_type` value tells the server what function to call in order to generate custom forms for the page admin.

It's an array, in case one plugin handles a number of different page types.

Because we've provided an admin-end `page_type`, it also makes sense to provide the front-end equivalent, so we also add a `page_type` to the frontend array.

I've set the version to 3 here, because while developing it, I made three adjustments to the database.

Here's the `upgrade.php` file, which should go in the same directory:

```

<?php
if($version==0){ // forms_fields
    dbQuery('CREATE TABLE IF NOT EXISTS `forms_fields` (
        `id` int(11) NOT NULL auto_increment,
        `name` text,
        `type` text,
        `isrequired` smallint(6) default 0,
        `formsId` int(11) default NULL,
        `extra` text,
        PRIMARY KEY (`id`)
    ) ENGINE=MyISAM DEFAULT CHARSET=utf8');
    $version=1;
}
if($version==1){ // forms_saved
    dbQuery('CREATE TABLE IF NOT EXISTS `forms_saved` (
        `forms_id` int(11) default 0,
        `date_created` datetime default NULL,
        `id` int(11) NOT NULL auto_increment,
        PRIMARY KEY (`id`)
    ) ENGINE=MyISAM DEFAULT CHARSET=utf8');
    $version=2;
}

```

```
if($version==2){ // forms_saved_values
  dbQuery('CREATE TABLE IF NOT EXISTS `forms_saved_values` (
    `forms_saved_id` int(11) default 0,
    `name` text,
    `value` text,
    `id` int(11) NOT NULL auto_increment,
    PRIMARY KEY (`id`)
  ) ENGINE=MyISAM DEFAULT CHARSET=utf8');
  $version=3;
}
```

When the plugin is enabled (enable it in the plugins section of the admin area), that script will be run, and the tables added to the database.

- The `forms_fields` table holds information about the fields that will be shown in the form.
- The `formsId` value links to the page ID, and the extra value is used to hold values in cases where the input type needs extra data – such as select-boxes, where you need to tell the form what the select-box options are.
- The `forms_saved` table holds data on forms that have been saved in the database, including the date and time that the form was saved.
- The `forms_saved_values` holds the saved data and is related via `forms_saved_id` to the `forms_saved` table.

Okay – we have the config and the database tables installed. Now let's get down to the administration.

Page types in the admin

When you click on **add main page** in the page admin section, the pop up appears as seen here:



The **Page Type** in the form only holds one value at the moment, **normal**. We need to change this so that it can add types created by plugins.

We first assume that most pages will be "normal", so we can leave that as the single value in the select-box; we then use the `RemoteSelectOptions` plugin described earlier in the book to add any others if the select-box is used.

Edit `/ww.admin/pages/menu.js` and add the following highlighted line to the `pages_new()` function:

```
$('#newpage_date').each(convert_date_to_human_readable);  
$('#newpage_dialog select[name=type]'  
  .remoteselectoptions({  
    url: '/ww.admin/pages/get_types.php'  
  });  
return false;
```

And here's the `/ww.admin/pages/get_types.php` file:

```
<?php  
require '../admin_libs.php';  
echo '<option value="0">normal</option>';  
foreach($PLUGINS as $n=>$plugin){  
  if(!isset($plugin['admin']['page_type']))continue;  
  foreach($plugin['admin']['page_type'] as $n=>$p){  
    echo '<option value="'.htmlspecialchars($n).'">' .  
      htmlspecialchars($n).'</option>';  
  }  
}
```

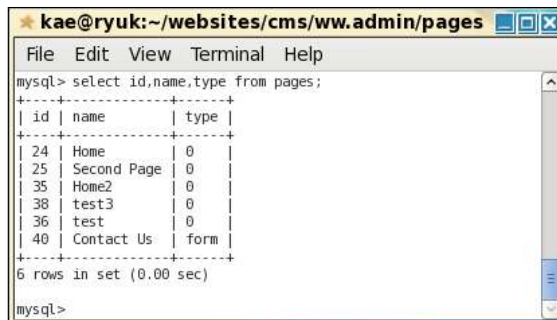
All it does is to echo out the `normal` type as an option, then goes through all installed plugins and displays their `page_type` values as well.

The **add main page** pop up now has the new `form` page type added, as seen in the next screenshot:



When you submit this form, a new page is created. The new page's form says its type is `normal`, but that's because we haven't added the code yet to the main form.

You can see that it's been done correctly by checking the database, as seen in the next screenshot:



We add the page types to the main page form in the same way as we did with the **add main page** form.

Edit the `/ww.admin/pages/pages.js` file and add these highlighted lines to the `$(function...)` section:

```
    other_GET_params:currentpageid
  });
  $('#pages_form select[name=type]').remoteselectoptions({
    url: '/ww.admin/pages/get_types.php'
  });
});
```

This does exactly the same as the previous one. However, on doing it you'll see that the page form still says `normal`:

A screenshot of a web form. The form has three fields: 'name' with the value 'Contact Us', 'type' with a dropdown menu showing 'normal', and 'body' with a rich text editor toolbar. The 'type' field is highlighted with a yellow background.

The reason for this is that the HTML of the page is generated without knowing about the other page types. We need to add code to the form itself.

Because the name of the type is stored in the page table (except if it's `normal`, in which case it's stored as 0), all we need to do is to output that name on its own.

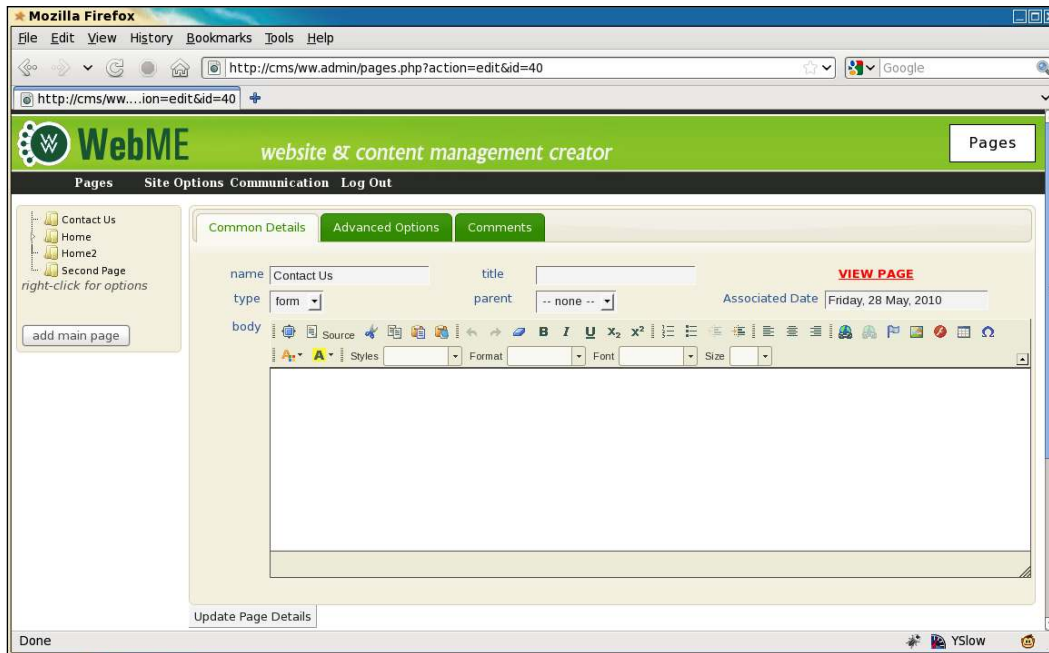
Edit `/ww.admin/pages/forms.php`, in the **Common Details** section, change the `type` section to this:

```
// { type
echo '<th>type</th><td><select name="type"><option';
if (!$page['type'])echo ' value='0'>normal';
else echo '>' . htmlspecialchars($page['type']);
echo '</option></select></td>';
// }
```

Now, let's get back to the plugin and adding its admin forms to the page.

Adding custom content forms to the page admin

The page admin form appears as seen in the next screenshot:



We saw in the previous chapter how to add another tab along the top.

When creating the custom form for the plugin, we could use the same method and add another tab.

However, it makes more sense to convert the body section so that it can be tabbed.

In the end, it all gets added to the database in the same way, but visually, tabs along the top of the page appear to be "meta" data (data about the page), whereas tabs in the body section appear to be "content" data.

The difference is subtle, but in my experience, admins tend to find it easier to use forms that are arranged in this way.

So, we will add the forms to the body section.

Open `/ww.admin/pages/forms.php` again, and change the generate list of custom tabs section to this (changed lines are highlighted):


```
// { gather plugin data
$custom_tabs=array();
$custom_type_func='';
foreach($PLUGINS as $n=>$p){
    if(isset($p['admin']['page_tab'])){
        $custom_tabs[$p['admin']['page_tab']['name']]
            =$p['admin']['page_tab']['function'];
    }
    if(isset($p['admin']['page_type'])){
        foreach($p['admin']['page_type'] as $n=>$f){
            if($n==$page['type'])$custom_type_func=$f;
        }
    }
}
// }
```

We rename it to gather plugin data because it's no longer specifically about tabs.

This loops through all installed plugins, getting any tabs that are defined, and setting `$custom_type_func` to the plugin's `page_type` function if it exists.

And later in the same file, change the page-type-specific data section to this:

```
// { page-type-specific data
if($custom_type_func && function_exists($custom_type_func)){
    echo '<tr><td colspan="6">'
        . $custom_type_func($page, $page_vars) . '</td></tr>';
}
else{
    echo '<tr><th>body</th><td colspan="5">';
    echo ckeditor('body', $page['body']);
    echo '</td></tr>';
}
// }
```

This outputs the result of the `page_type` function if it was set and the function exists.

The function requires a file that we haven't yet created, so loading the admin page will display only half the form before crashing.

Create the directory `/ww.plugins/forms/admin` and create the file `form.php` in it:

```
<?php
$c.='<div class="tabs">';
// { table of contents
$c.='<ul><li><a href="#forms-header">Header</a></li>'
```

```
. '<li><a href="#forms-main-details">Main Details</a></li>'
. '<li><a href="#forms-fields">Fields</a></li>'
. '<li><a href="#forms-success-message">Success Message</a></li>'
. '<li><a href="#forms-template">Template</a></li></ul>';
// }
// { header
$c.='<div id="forms-header"><p>Text to be shown
    above the form</p>';
$c.=ckeditor('body', $page['body']);
$c.='</div>';
// }
// { main details
// }
// { fields
// }
// { success message
// }
// { template
// }
$c.='</div>';
```

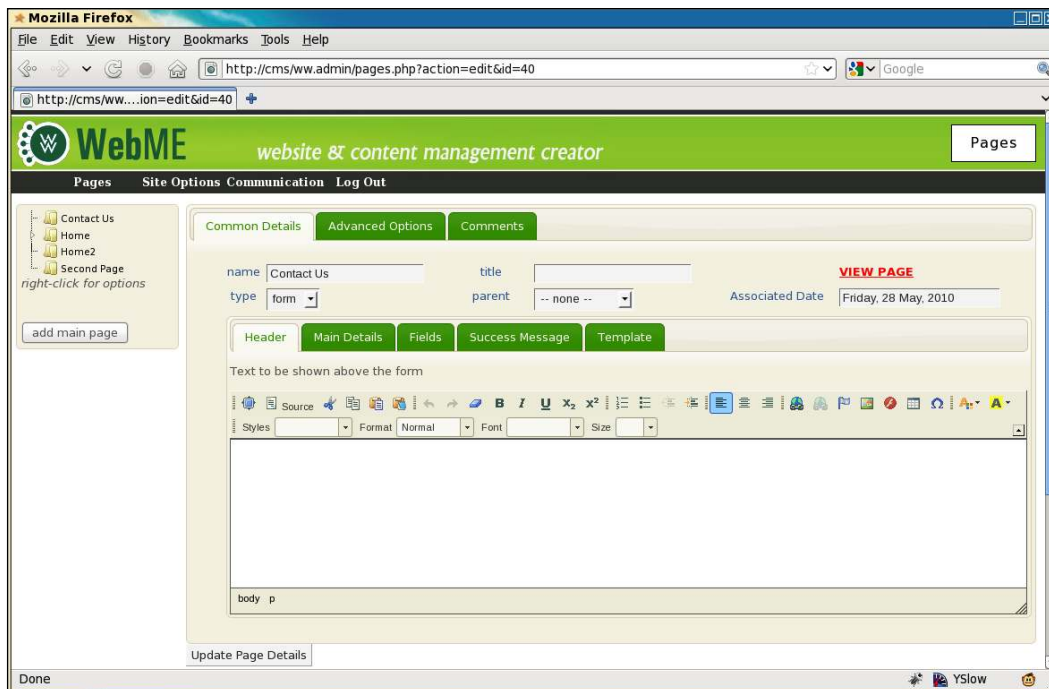
I've left the `main details`, `fields` (and other) sections empty on purpose. We'll fill them in a moment.

This code creates a tab structure. You can see the table of contents matches the commented sections.

Creating a "skeleton" of a config form can be useful because it lets you view your progress in the browser, and you can leave reminders to yourself in the form of commented sections that have not yet been filled in.

Doing this also helps you to develop the habit of commenting your code, so that others can understand what is happening at various points of the file.

The previous code snippet can now be viewed in the browser, and renders as seen in the following screenshot:



So we have two rows of tabs. You can now see what I meant – the bottom collection of tabs is obviously about the page content, while the others are more about the page itself.

Before we work on the `fields` tab, let's do the other three.

First, replace the `template` section with this:

```
// { template
$c.= '<div id="forms-template">';
$c.= '<p>Leave blank to have an auto-generated
    template displayed.</p>';
$c.= ckeditor('page_vars[forms_template]',
    $page_vars['forms_template']);
$c.= '</div>';
// }
```

The template defines how you want the form to appear on the front-end. We start off with the assumption that the admin does not know (or want to know) how to fill this in, so we leave a message saying that if the template is left blank, it will be auto-generated on the front-end.

When we get to displaying the form on the front-end, we'll discuss this one more.

Notice that we use `page_vars[forms_template]` as the name for the template's input box. With this, we will not need to write server-side code to save the data, as it will be handled by the page admin's own saving mechanism.

Next, replace the success message section with this:

```
// { success message
$c.= '<div id="forms-success-message">';
$c.= '<p>What should be displayed on-screen after the
      message is sent.</p>';
if(!isset($page_vars['forms_successmsg']))
    $page_vars['forms_successmsg']=
        '<h2>Thank You</h2>
        <p>We will be in contact as soon as we can.</p>';
$c.= ckeditor('page_vars[forms_successmsg]',
    $page_vars['forms_successmsg']);
$c.= '</div>';
// }
```

This defines the message which is shown to the form submitter after they've submitted the form. We initialize this with a simple non-specific message (**We will be in contact as soon as we can**), as we cannot be certain what the form will be used for.

The final straightforward tab is the `main details` section. Replace it with the following code. It may be a little long, but it's just a few fields. A screenshot after the code will explain what it does:

```
// { main details
$c.= '<div id="forms-main-details"><table>';
// { send as email
if(!isset($page_vars['forms_send_as_email']))
    $page_vars['forms_send_as_email']=1;
$c.= '<tr><th>Send as Email</th><td><select
      name="page_vars[forms_send_as_email]"><option
      value="1">Yes</option><option value="0">
if(!isset($page_vars['forms_send_as_email']))
    $c.= ' selected="selected"';
$c.= '>No</option></select></td>';
// }

// { recipient
if(!isset($page_vars['forms_recipient']))
    $page_vars['forms_recipient']=
        $_SESSION['userdata']['email'];
```

```

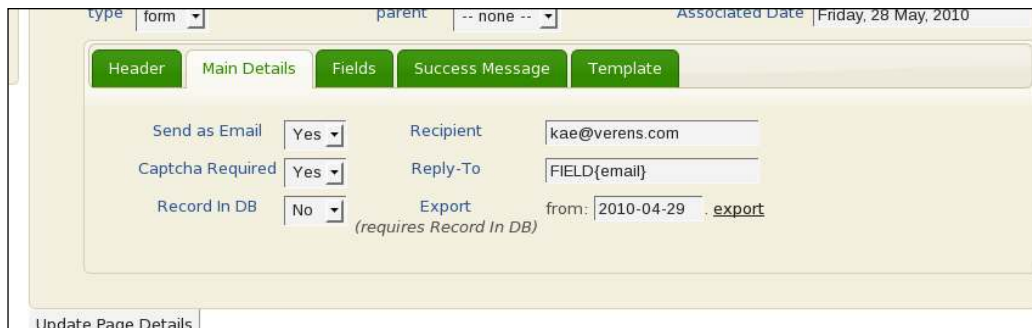
$c.= '<th>Recipient</th><td><input
      name="page_vars[forms_recipient]"
      value="'.htmlspecialchars($page_vars['forms_recipient'])
      .'

```

```
}
else{   $c.='<td colspan="2">&nbsp;</td></tr>';
}
// }

$c.= '</table></div>';
// }
```

This code builds up the **Main Details** tab, which looks like this in the browser:



The screenshot shows a web form configuration interface. At the top, there are tabs: Header, Main Details (selected), Fields, Success Message, and Template. Below the tabs, there are several settings:

- Send as Email: Yes (dropdown)
- Captcha Required: Yes (dropdown)
- Record in DB: No (dropdown)
- Recipient: kae@verens.com (text input)
- Reply-To: FIELD{email} (text input)
- Export: from: 2010-04-29 .export (text input)

At the bottom left, there is a button labeled "Update Page Details".

The **Send as Email** and **Captcha Required** are defaulted to **Yes**, while the **Record in DB** is defaulted to **No**.

Recipient is the person that the form is e-mailed to, which is set initially to the e-mail address of the administrator that created the form.

If the form is e-mailed to the recipient, and the recipient replies to the e-mail, then we need to know who we're replying to. We default this to `FIELD{email}`, which is a code indicating that the reply-to should be equal to whatever was filled in in the form in its `email` field (assuming it has one). We'll talk more on this later on. You could just as well enter an actual e-mail address such as `no-reply@no-such.domain`.

The **Export** field lets you export saved form details to a CSV file. We'll work on this later in the chapter as well.

For now, let's define the form fields.

Defining the form fields

Before we look at the code, we should define what it is that we are trying to achieve with the `fields` tab.

We want to be able to define each field that will be entered in the form by the user.

While some validation will be available, we should always be aware that the forms plugin will be used by an administrator who may be daunted by complex controls, so we will try to keep the user interface as simple as possible.

Validation will be kept to a minimum of whether the field must be entered in the form, and whether the entered value matches the field type. For example, if the field is defined as an e-mail, then the entered value must be an e-mail address. If the field is a select-box, then the entered value must match one of the entries we've defined as belonging to that select-box, and so on.

We will not use complex validation, such as if one entry is entered, then another must not. That kind of validation is rarely required by a simple website, and it would make the user interface much more cluttered and difficult to use.

Now, the `fields` tab is made of two parts—first, any fields that are already associated with the form will be printed out by the PHP, and we will then add some JavaScript to allow more fields to be added "on-the-fly".

Here is the PHP of it (replace the `fields` section in the file with this):

```
// { fields
$c.= '<div id="forms-fields">';
$c.= '<table id="formfieldsTable" width="100%"><tr><th
width="30%">Name</th><th width="30%">Type</th><th
width="10%">Required</th><th id="extrasColumn"><a
href="javascript:formfieldsAddRow()">add
field</a></th></tr></table><ul id="form_fields"
style="list-style:none">';
$q2=dbAll('select * from forms_fields where formsId="'. $id.'"
order by id');
$i=0;
$arr=array('input box','email','textarea','date',
'checkbox','selectbox','hidden');
foreach($q2 as $r2){
$c.= '<li><table width="100%"><tr>';
// { name
$c.= '<td width="30%"><input name="formfieldElementsName['
.$i.']" value="'.htmlspecialchars($r2['name']).'" />
.</td>';
// }
// { type
$c.= '<td width="30%"><select name="formfieldElementsType['
.$i.']">';
foreach($arr as $v){
```

```
    $c.='<option value="' . $v . '"';
    if ($v==$r2['type']) $c.=' selected="selected"';
    $c.='>' . $v . '</option>';
  }
  $c.='</select></td>';
  // }

  // { is required
  $c.='<td><input type="checkbox"
    name="formfieldElementsIsRequired['.($i).']"' .
  if ($r2['isrequired']) $c.=' checked="checked"';
  $c.=' /></td>';
  // }

  // { extras
  $c.='<td>';
  switch ($r2['type']) {
    case 'selectbox': case 'hidden': {
      $c.='<textarea class="small"
        name="formfieldElementsExtra['.($i++)
          .']">' . htmlspecialchars ($r2['extra']) . '</textarea>';
      break;
    }
    default: {
      $c.='<input type="hidden"
        name="formfieldElementsExtra['.($i++) .']"
        value="' . htmlspecialchars ($r2['extra']) . '" />';
    }
  }
  $c.=' </td>';
  // }

  $c.='</tr></table></li>';
}
$c.=' </ul></div>';
// }
```

If you've read through that, you'll see that it simply outputs a number of rows of field data. We'll have a look at a screenshot shortly. First, let's add the JavaScript.

At the end of the file, add these lines:

```
$c.='<script>var formfieldElements=' . $i . ';</script>';
$c.='<script src="/ww.plugins/forms/admin/forms.js">
</script>';
```

The variable `$i` here was set in the previous code-block, and represents the number of field rows that are already printed on the screen.

Now create the file `/ww.plugins/forms/admin/forms.js`:

```

window.form_input_types=['input box','email','textarea',
    'date','checkbox','selectbox','hidden'];
function formfieldsAddRow(){
    formfieldElements++;
    $('<li><table width="100%"><tr><td width="30%"><input '
        +'name="formfieldElementsName['+formfieldElements+']" '
        +'/></td><td width="30%"><select class="form-type" name=" '
        +'formfieldElementsType['+formfieldElements+']"><option>'
        +'form_input_types.join('</option><option>')
        +'/></select></td><td width="10%"><input '
        +'type="checkbox" name=" '
        +'formfieldElementsIsRequired['+formfieldElements+']" '
        +'/></td><td><textarea name=" '
        +'formfieldElementsExtra['+formfieldElements+']" '
        +'style="display:none" class="small"></textarea></td>'
        +'/tr></table></li>'
    ).appendTo($('#form_fields'));
    $('#form_fields').sortable();
    $('#form_fields input,#form_fields select,#form_fields
        textarea').bind('click.sortable mousedown.sortable',
        function(ev){
            ev.target.focus();
        });
    }
    $('select.form-type').live('change',function(){
        var val=$(this).val();
        var display=(val=='selectbox' || val=='hidden')
            ?'inline':'none';
        $(this).closest('tr').find('textarea')
            .css('display',display);
    });
    if(!formfieldElements)var formfieldElements=0;
    $(function(){
        formfieldsAddRow();
    });
}

```

First, we define the list of available field types.

The `formfieldsAddRow()` function adds a new field row to the `fields` tab. The row is a simple line of HTML, replicating what we did in the PHP earlier.

Notice that we add a hidden textarea. This is to hold data on select-box values or hidden values if we choose to set the field type to either of those.

Next, we make the rows sortable using the jQuery UI's `.sortable()` plugin. This is so that the admin can reorder the field values if they want to.

Note that the `.sortable()` plugin makes it tricky to click on the input, select, and textarea boxes in the field row, as it hijacks the click and mousedown events, so the next line overrides the `.sortable()` event grab if you click on one of those elements. If you want to sort the rows, you should drag from a point outside those elements.

Next, we add a `live` event, which says that whenever a select-box with the class `form-type` is changed, we should change the visibility of the `extras` textarea in that row based on what you changed it to.

And finally, we initialize everything by calling `formfieldsAddRow()` so that the form has at least one row in it.

Note that we could have replaced the last three lines with this:

```
$( formfieldsAddRow );
```

However, when we get around to exporting saved data, we will want to add some more to the initialization routine, so we do it the long way.

The screenshot shows a web-based form configuration interface. At the top, there are five tabs: 'Header', 'Main Details', 'Fields', 'Success Message', and 'Template'. The 'Fields' tab is active. Below the tabs is a table with the following columns: 'Name', 'Type', 'Required', and 'add_field'. The table contains five rows of field configurations:

Name	Type	Required	add_field
Name	input box	<input checked="" type="checkbox"/>	
Email	email	<input checked="" type="checkbox"/>	
Address	textarea	<input type="checkbox"/>	
Request	selectbox	<input type="checkbox"/>	<div style="border: 1px solid black; padding: 5px;">Send me your brochure Add me to your mailing list I wish to complain I have a comment on your product</div>
Comment	textarea	<input type="checkbox"/>	

At the bottom left of the interface, there is a button labeled 'Update Page Details'.

In the screenshot, you can see the result of all this. I took this snapshot as I was dragging the **Request** field above the **Comment** one. Notice that the **Request** field has its `extras` textarea visible and filled in, one option per line.

Next we need to save the inputs.

Edit the file `/ww.plugins/forms/plugin.php`, and change the function `form_admin_page_form()` to the following (changes are highlighted):

```

function form_admin_page_form($page,$page_vars){
    $id=$page['id'];
    $c='';
    if(isset($_REQUEST['action'])
        && $_REQUEST['action']=='Update Page Details')
        require dirname(__FILE__).'/admin/save.php';
    require dirname(__FILE__).'/admin/form.php';
    return $c;
}

```

And then all that's required is to do the actual save. Create the file `/ww.plugins/forms/admin/save.php`:

```

<?php
dbQuery('delete from forms_fields where formsId="'. $id. '"');
if(isset($_POST['formfieldElementsName'])
    &&is_array($_POST['formfieldElementsName'])) {
    foreach($_POST['formfieldElementsName'] as $key=>$name) {
        $name=addslashes(trim($name));
        if($name!='') {
            $type=addslashes($_POST['formfieldElementsType'][$key]);
            $isrequired=
                (isset($_POST['formfieldElementsIsRequired'][$key])
                 ?1:0);
            $extra=
                addslashes($_POST['formfieldElementsExtra'][$key]);
            $query='insert into forms_fields set name="'. $name. '"
                ,type="'. $type. '", isrequired="'. $isrequired. '"
                ,formsId="'. $id. '",extra="'. $extra. '"';
            dbQuery($query);
        }
    }
}

```

First, the old existing fields are deleted if they exist, and then a fresh set are added to the database.

This happens each time you edit the form, because updating existing entries is much more complex than simply starting from scratch each time. This is especially true if you are moving them around, adding new ones, and so on.

Note that we check to see if the field's name was entered. If not, that row is not added to the database. So, to delete a field in your form, simply delete the name and update the page.

Now, let's show the form on the front-end.

Showing the form on the front-end

Showing the form is a matter of taking the information from the database and rendering it in the page HTML.

First, we need to tell the controller (`/index.php`) how to handle pages which are of a type other than `normal`.

Edit the `/index.php` file, and in the switch in set up `pagecontent`, replace the other cases will be handled here later line with the following default case:

```
default: // { plugins
    $not_found=true;
    foreach($PLUGINS as $p){
        if(isset($p['frontend']['page_type'][$PAGEDATA->type])){
            $pagecontent=$p['frontend']['page_type']
                [$PAGEDATA->type]($PAGEDATA);
            $not_found=false;
        }
    }
    if($not_found)$pagecontent='<em>No plugin found to handle
        page type <strong>'.htmlspecialchars($PAGEDATA->type)
        .'/</strong>. Is the plugin installed and
        enabled?</em>';
    // }
```

If the page type is not `normal` (type 0 in the switch that we've edited), then we check to see if it's a plugin.

This code runs through the array of plugins that are loaded, and checks to see if any of them have a `frontend page_type` that matches the current page. If so, then the associated function is run, with the `$PAGEDATA` object as a parameter.

We've already created the function as part of the `plugin.php` file. Now let's work on rendering the form.

Create the file `/ww.plugins/forms/frontend/show.php` (create the directory first):

```
<?php
require_once SCRIPTBASE.'ww.incs/recaptcha.php';
function form_controller($page){
    $fields=dbAll('select * from forms_fields where
        formsId="'.$page->id.'" order by id');
    if(isset($_POST['_form_action']))
        return form_submit($page,$fields);
    return form_display($page,$fields);
}
```

The first thing we do is to load up the field data from the database, as this is used when submitting and when rendering.

When the page is first loaded, there is no `_form_action` value in the `$_POST` array, so the function `form_display()` is then run and returned.

Add that function to the file now:

```
function form_display($page,$fields){
    if(isset($page->vars->forms_template)){
        $template=$page->vars->forms_template;
        if($template=='&nbsp;')$template=false;
    }
    else $template=false;
    if(!$template)
        $template=form_template_generate($page,$fields);
    return form_template_render($template,$fields);
}
```

We first check the form's template to see that it is created and is not blank.

Next, if the template was blank, we build one using the field data as a guide.

And finally, we render the template and return it to the page controller.

Okay – the first thing we're missing is the `form_template_generate()` function. Add that to the file as follows:

```
function form_template_generate($page,$fields){
    $t='<table>';
    foreach($fields as $f){
        if($f['type']=='hidden')continue;
        $name=preg_replace('/[^a-zA-Z0-9_]/','',$f['name']);
        $t.='<tr><th>'.htmlspecialchars($f['name'])
            .'</th><td>{{$f.$name.}}</td></tr>';
    }
    if($page->vars->forms_captcha_required){
        $t.='<tr><td>&nbsp;</td><td>{{CAPTCHA}}</td></tr>';
    }
    return $t.'</table>';
}
```

Simple enough – we iterate through each row, and generate some Smarty-like code. Here's an example output of the function (formatted for easier reading):

```
<table class="forms-table">
  <tr><th>Name</th><td>{{$Name}}</td></tr>
```

```
<tr><th>Email</th><td>{{ $Email }}</td></tr>
<tr><th>Address</th><td>{{ $Address }}</td></tr>
<tr><th>Request</th><td>{{ $Request }}</td></tr>
<tr><th>Comment</th><td>{{ $Comment }}</td></tr>
<tr><td>&nbsp;</td><td>{{ CAPTCHA }}</td></tr>
</table>
```

We're not going to actually use Smarty on this one, as it would be too much — we just want to do a little bit of code replacement, so adding the full power of Smarty would be a waste of resources.

We use the Smarty-like code so that the admin doesn't have to remember different types of code. We could have also used BBCode, or simply placed % on either end of the field names, and so on.

Note that we don't output a line for hidden fields. Those fields are only ever seen by the administrator when the form is submitted.

Finally, we get to the rendering.

This function is kind of long, so we'll do it in bits.

```
function form_template_render($template,$fields){
    if(strpos($template,'{{CAPTCHA}}')!==false){
        $template=str_replace('{{CAPTCHA}}',
            recaptcha_get_html(RECAPTCHA_PUBLIC),$template);
    }
    foreach($fields as $f){
        $name=preg_replace('/[^a-zA-Z0-9_]/','',$f['name']);
        if($f['isrequired'])$class=' required';
        else $class='';
        if(isset($_POST[$name])){
            $val=$_POST[$name];
        }
        else $val='';
    }
}
```

We first initialize the function and render the captcha if it's turned on. We're using the same captcha code that we used for the admin authentication.

Next, we start looping through each field value.

If the form has already been submitted, and we're showing it again, then we set `$val` to the value that was submitted. This is so we can show it in the form again.

Next, we figure out what should go into the template for the field:

```
switch($f['type']){
  case 'checkbox': // {
    $d='<input type="checkbox" id="forms-plugin-'. $name
      .' name="'. $name.'";
    if($val)$d.=' checked="'. $_REQUEST[$name].'";
    $d.=' class="'. $class.'" />';
    break;
  // }
  case 'date': // {
    if(!$val)$val=date('Y-m-d');
    $d='<input id="forms-plugin-'. $name.'" name="'. $name
      .' value="'. htmlspecialchars($val).' class="date'
      .' $class.'" />';
    break;
  // }
  case 'email': // {
    $d='<input type="email" id="forms-plugin-'. $name.'"
      name="'. $name.'" value="'. htmlspecialchars($val).'
      class="email' . $class.'" />';
    break;
  // }
  case 'selectbox': // {
    $d='<select id="forms-plugin-'. $name.'" name="'. $name
      .' class="'. $class.'">';
    $arr=explode("\n",htmlspecialchars($f['extra']));
    foreach($arr as $li){
      if($li=='')continue;
      $li=trim($li);
      if($val==$li)$d.='<option selected="selected">'. $li
        .'</option>';
      else $d.='<option>'. $li.'</option>';
    }
    $d.='</select>';
    break;
  // }
  case 'textarea': // {
    $d='<textarea id="forms-plugin-'. $name.'" name="'.
      .' $name.'" class="'. $class.'">'
      .' htmlspecialchars($val).'</textarea>';
    break;
  // }
  default: // {
    $d='<input id="forms-plugin-'. $name.'" name="'. $name
      .' value="'. htmlspecialchars($val).' class="text'
      .' $class.'" />';
  // }
}
```

This switch block checks what type of field it is, and generates an appropriate HTML string to represent it.

Note that we've added classes to the inputs. These classes can be used for client-side validation, or for CSS.

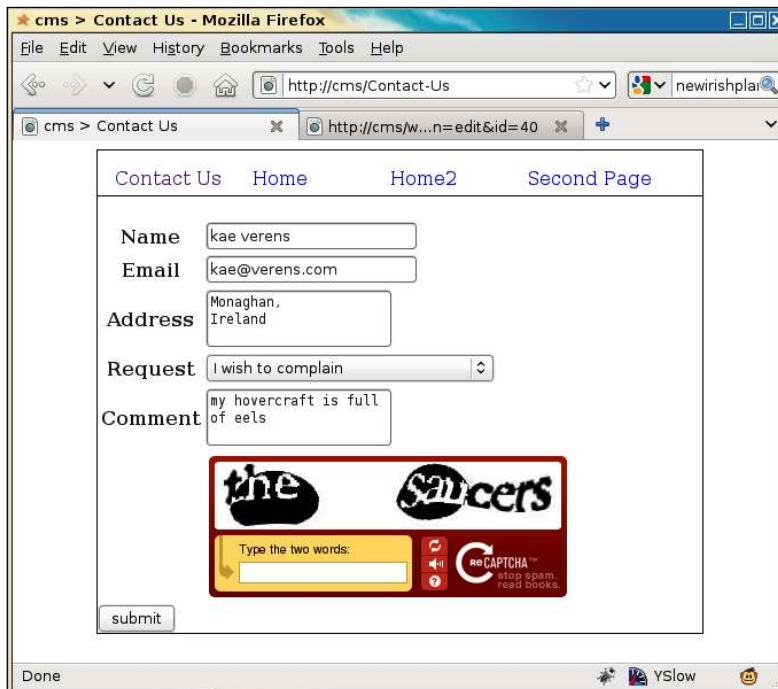
Finally:

```
$template=str_replace('{{$.name.}}', $d, $template);
}
return '<form method="post" id="forms-plugin">'.$template
.'<input type="submit" name="_form_action"
value="submit" /></form>
<script src="/ww.plugins/forms/frontend/forms.js">
</script>';
}
```

We replace the Smarty-like code with the HTML string for each field, and finally return the generated form with the `submit` button attached.

We also load up a JavaScript file, to handle validation on the client-side. We'll get to that shortly.

Having finished all of this, here's a screenshot of an example filled-in form:



The form can be easily marked up in CSS to make it look better. But we're not here to talk style – let's get on with the submission of the form.

Handling the submission of the form

When submit is clicked, the form data is sent to the same page (we didn't put an `action` parameter in the `<form>` element, so it goes back to the same page by default).

The submit button itself has the name `_form_action` which, when the form controller is loaded, triggers `form_submit()` to be called.

Add that function to the same file:

```
function form_submit($page,$fields) {
    $errors=form_validate($page,$fields);
    if(count($errors)) {
        return '<ul id="forms-plugin-errors"><li>'
            .join('</li><li>',$errors)
            . '</ul>'
            .form_display($page,$fields);
    }
    if($page->vars->forms_send_as_email)
        form_send_as_email($page,$fields);
    if($page->vars->forms_record_in_db)
        form_record_in_db($page,$fields);
    return $page->vars->forms_successmsg;
}
```

The first thing we do is validate any submitted values.



Always write your validation for the server-side first.

If you do your validation on the client-side first, then you may forget to do it on the server-side. You'd also have to disable your client-side validation in order to test the server-side work.

After validation, we send the form off in an e-mail and save the form in the database if that's how it was set up in the admin area.

Finally, we return the success message to the page controller.

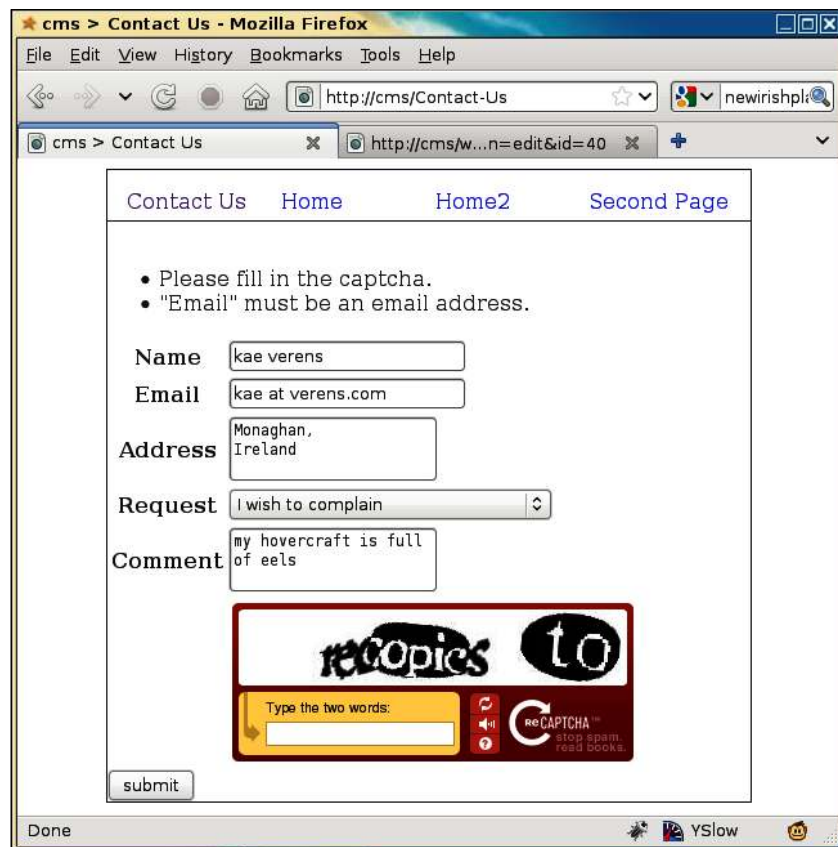
There are three functions to add.

The first is the validation function:

```
function form_validate($page,$fields){
    $errors=array();
    if($page->vars->forms_captcha_required){
        $resp=recaptcha_check_answer(
            RECAPTCHA_PRIVATE,
            $_SERVER["REMOTE_ADDR"],
            $_POST["recaptcha_challenge_field"],
            $_POST["recaptcha_response_field"]
        );
        if(!$resp->is_valid)$errors[]='Please fill in
            the captcha.';
    }
    foreach($fields as $f){
        $name=preg_replace('/[^a-zA-Z0-9_]/','',$f['name']);
        if(isset($_POST[$name])){
            $val=$_POST[$name];
        }
        else $val='';
        if($f['isrequired'] && !$val){
            $errors[]='The "'.$htmlspecialchars($f['name']).'" field
                is required.';
            continue;
        }
        if(!$val)continue;
        switch($f['type']){
            case 'date': // {
                if(preg_replace('/[0-9]{4}-[0-9]{2}-[0-9]{2}/','',$val)=='')continue;
                $errors[]=' "'.$htmlspecialchars($f['name']).'" must be
                    in yyyy-mm-dd format.';
                break;
            // }
            case 'email': // {
                if(filter_var($val,FILTER_VALIDATE_EMAIL))continue;
                $errors[]=' "'.$htmlspecialchars($f['name']).'" must be
                    an email address.';
                break;
            // }
            case 'selectbox': // {
                $arr=explode("\n",$htmlspecialchars($f['extra']));
                $found=0;
                foreach($arr as $li){
```

```
        if($li=='')continue;
        if($val==trim($li))$found=1;
    }
    if($found)continue;
    $errors[]='You must choose one of the options in
        "'.htmlspecialchars($f['name']).'".';
    break;
    // }
}
}
return $errors;
}
```

If you create dummy functions for `form_send_as_email()` then you can test the given code, and its output should appear as seen in the next screenshot:



Okay, so validation works.

Sending by e-mail

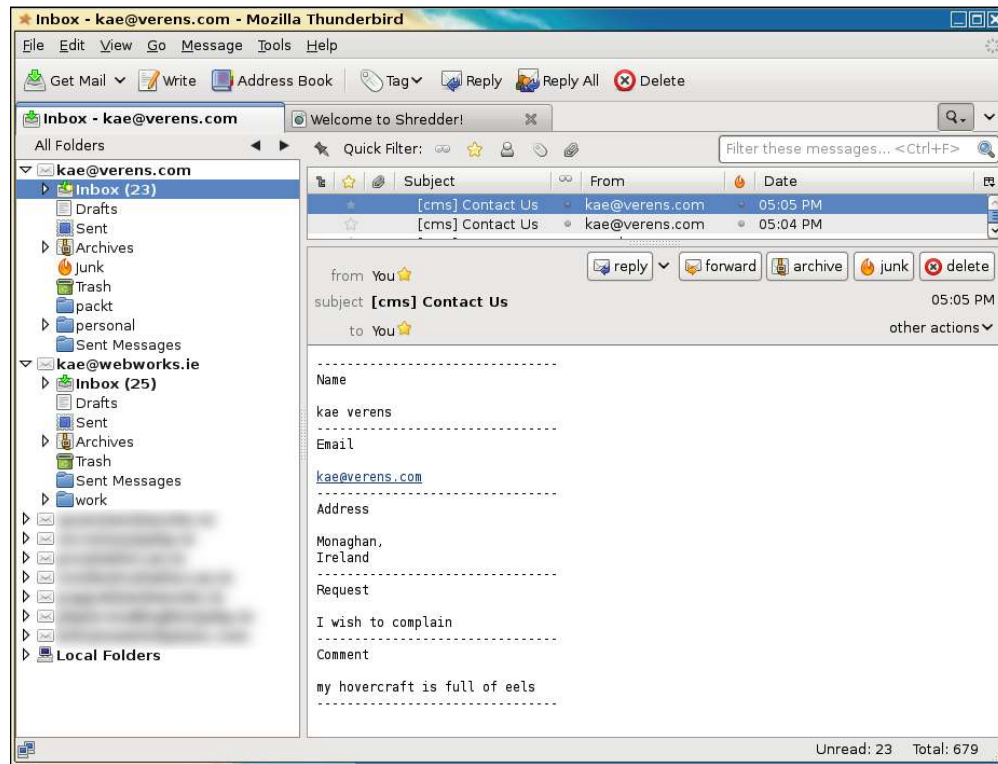
Next, we will add the e-mail sender function.

The e-mail that we create does not need to be fancy – we're submitting a simple list of questions and responses, and it's not to a client, so it doesn't need a template to be created.

With that in mind, it's reasonable to create the following simple function:

```
function form_send_as_email($page,$fields){
    $m="-----\n";
    foreach($fields as $f){
        $name=preg_replace('/^[a-zA-Z0-9_]/','',$f['name']);
        if(!isset($_POST[$name]))continue;
        $m.=$f['name']."\n\n";
        $m.=$_POST[$name];
        $m.="-----\n";
    }
    $from=preg_replace('/^FIELD{|\}$/', '',
        $page->vars->forms_replyto);
    $to=preg_replace('/^FIELD{|\}$/', '',
        $page->vars->forms_recipient);
    if($page->vars->forms_replyto!=$from)
        $from=$_POST[preg_replace('/^[a-zA-Z0-9_]/','',$from)];
    if($page->vars->forms_recipient!=$to)
        $to=$_POST[preg_replace('/^[a-zA-Z0-9_]/','',$to)];
    mail($to,['.$_SERVER['HTTP_HOST'].'] '
        .addslashes($page->name),$m,
        "From: $from\nReply-to: $from");
}
```

With this in place, the system will send the form contents as an e-mail:



Perfectly readable and simple.

Saving in the database

Next, we tackle saving the form into the database:

```

function form_record_in_db($page,$fields) {
    $formId=$page->id;
    dbQuery("insert into forms_saved (forms_id,date_created)
values ($formId,now())");
    $id=dbOne('select last_insert_id() as id','id');
    foreach($fields as $r){
        $name=preg_replace('/[^a-zA-Z0-9_]/','',$r['name']);
        if (isset($_POST[$name])) $val=addslashes($_POST[$name]);
        else $val='';
        $key=addslashes($r['name']);
        dbQuery("insert into forms_saved_values (forms_saved_
id,name,value) values($id,'$key','$val)");
    }
}

```

This records the values of the form in the database:

```
mysql> select * from forms_saved;
+-----+-----+-----+
| forms_id | date_created      | id |
+-----+-----+-----+
|         40 | 2010-06-01 05:58:16 | 1 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from forms_saved_values \G
***** 1. row *****
forms_saved_id: 1
      name: Name
      value: Kae Verens
      id: 6
***** 2. row *****
forms_saved_id: 1
      name: Email
      value: kae@verens.com
      id: 7
***** 3. row *****
forms_saved_id: 1
      name: Address
      value: Monaghan,
Ireland
      id: 8
***** 4. row *****
forms_saved_id: 1
      name: Request
      value: I wish to complain
      id: 9
***** 5. row *****
forms_saved_id: 1
      name: Comment
      value: my hovercraft is full of eels
      id: 10
5 rows in set (0.00 sec)
```

We cannot expect the admin to use the MySQL, so we need to write the export function now.

Exporting saved data

Back in the admin area, we had the following part of the **Forms** config:

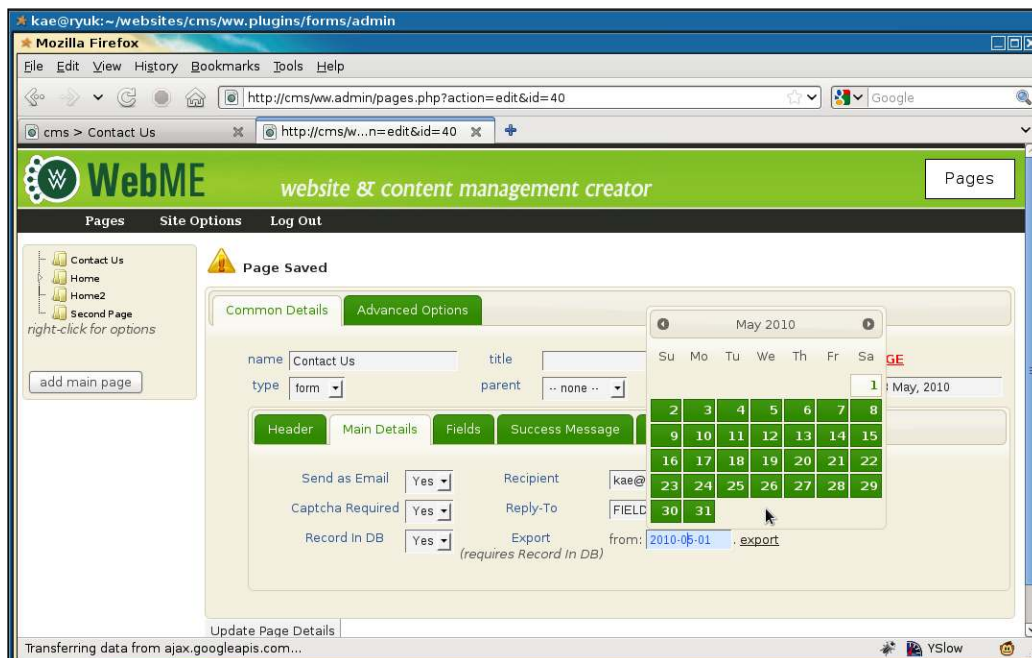
A screenshot of a form configuration field. It features a dropdown menu with 'Yes' selected, followed by the text 'Export (requires Record In DB)'. To the right is a date input field containing '2010-05-01' and a text input field containing '.export'.

First, let's make that date area more interesting.

Edit the file `/ww.plugins/forms/admin/forms.js` and add the following highlighted line to the `$(function)` part:

```
$(function() {
    formfieldsAddRow();
    $('#export_from').datepicker({dateFormat: 'yy-m-d'});
});
```

This simple line then adds calendar functionality to that input, as seen in the next screenshot:



When the input is clicked, the calendar pops up, so the admin doesn't have to write the date and get the format right.

Now let's add the function for handling the export (to that same file):

```
function form_export(id){
    if(!id)return alert('cannot export from an empty
        form database');
    if(!(+$('select[name="page_vars\[forms_record_in_db\]"')
        .val()))
        return alert('this form doesn\'t record to database');
    var d=$('#export_from').val();
    document.location='/ww.plugins/forms/admin/export.php?date='
        +d+'&id='+id;
}
```

This function checks first to see if the form is marked to save in the database. If so, then it does a redirect to `/ww.plugins/forms/admin/export.php`. Create that file now:

```
<?php
require $_SERVER['DOCUMENT_ROOT'].'/ww.admin/admin_libs.php';
if(isset($_REQUEST['id']))$id=(int)$_REQUEST['id'];
else exit;
if(!$id)exit;
$date=$_REQUEST['date'];
if(!preg_match('/^20[0-9][0-9]-[0-9][0-9]-[0-9][0-9]$/',$date))die('invalid date format');
header('Content-type: application/octet-stream');
header('Content-Disposition: attachment; filename="form'
    .$id.'-export.csv"');
// { ids
$ids=array();
$rs=dbAll("select id,date_created from forms_saved where
    forms_id=$id and date_created>'$date'");
foreach($rs as $r){
    $ids[$r['id']]=$r['date_created'];
}
// }
// { columns
$cols=array();
$rs=dbAll('select name from forms_fields where formsId="'
    .$id.'" order by id');
foreach($rs as $r){
    $cols[]=$r['name'];
}
// }
// { do the export
```



```

echo '"Date Submitted",';
echo join('"',",",$cols).'"'. "\n";
foreach($ids as $id=>$date){
    echo '"'.$date.'",';
    for($i=0;$i<count($cols);++$i){
        $r=dbRow('select value from forms_saved_values where
            forms_saved_id='.$id.' and name="'.addslashes($cols[$i])
            .'");
        echo '"'.str_replace('\\"','"',addslashes($r['value']))
            .'";
        if($i<count($cols)-1)echo ',';
        else echo "\n";
    }
}
// }

```

This exports the data as a CSV file.

Because the Content-type is application/octet-stream and browsers would not normally know how to handle that, the file is forced to download, instead of displaying in the browser. You can then open that exported file up in a spreadsheet:

	A	B	C	D	E	F
1	Date Submitted	Name	Email	Address	Request	Comment
2	2010-06-01 05:58:16	Kae Verens	kae@verens.com	Monaghan, Ireland	I wish to complain	my hovercraft is full of eels
3						
4						

With the export finished, we've completed a functional forms plugin.

Summary

In this chapter, we added the ability for a plugin to create a full page type, instead of just a trigger.

We also added content tabs to the page admin.

With the Forms plugin created, the admin can now create contact pages, questionnaires, and other types of page that request data in the form of user input.

In the next chapter, we will create an Image Gallery plugin.