

# ROCKABLE



Rohan Mehta



# ROCKABLE✱

Rockablepress.com  
Envato.com

© Rockable Press 2011

All rights reserved. No part of this publication may be reproduced or redistributed in any form without the prior written permission of the publishers.

## Contents

<b>Introduction</b>	<b>6</b>
<i>Files Included With The Book</i>	7
<i>A Code Convention</i>	8
<b>Setup &amp; Managing Files</b>	<b>10</b>
<i>Managing Files</i>	10
<i>The functions.php File</i>	11
<i>Basic Setup</i>	13
<i>The wp_nav_menu() Function</i>	18
<i>JavaScript Files</i>	20
<i>Image Functions</i>	22
<b>Theme Options Pages</b>	<b>28</b>
<i>The Options Page Design</i>	28
<i>Options Files</i>	29
<i>Generating the Pages</i>	33
<i>Styling and Scripts</i>	50
<i>Including the Files</i>	56
<i>Using this Theme Options Framework in Another Theme</i>	57
<i>The Home Page Slider Options</i>	59
<i>The JavaScript Code</i>	60
<i>Coding the Slider</i>	60
<b>Custom Post Types and Taxonomies</b>	<b>64</b>
<i>template_portfolio.php</i>	75
<i>single-portfolio.php</i>	78
<i>taxonomy-portfolio_cat.php</i>	79

<b>Meta Boxes</b>	<b>82</b>
<i>The Meta Box Options Files</i>	84
<i>Coding the Meta Box Class</i>	86
<i>Including the Files</i>	93
<i>Using the Class in Other Themes</i>	93
 <b>Custom Widgets</b>	 <b>96</b>
<i>Making the Theme Widget-Ready</i>	96
<i>Custom Widgets</i>	99
<i>Latest Portfolio Item Widget</i>	109
<i>Contact Form Widget</i>	112
<i>Tabbed Widget</i>	119
 <b>Miscellaneous Items</b>	 <b>122</b>
<i>Breadcrumbs</i>	122
<i>Shortcodes</i>	127
<i>Threaded Comments</i>	130
<i>Post Thumbnails</i>	134
 <b>Internationalization and Translation</b>	 <b>137</b>
<i>Text Domains</i>	137
<i>I18n Functions</i>	138
<i>Different i18n Scenarios</i>	139
<i>Generating the .po File</i>	141
<i>Using the PO File</i>	145
 <b>Afterword</b>	 <b>148</b>
 <b>About The Author</b>	 <b>149</b>
 <b>Your Download Link</b>	 <b>150</b>

# INTRODUCTION

# Introduction

WordPress today is no longer regarded as just a blogging tool, though quite possibly that was the reason it was created. Its flexibility and ease of customization has led WordPress to be one of the most popular Content Management Systems (CMS) used in the web design and development industry.

As a web designer or developer, WordPress is one of the most potent tools available at your disposal. WordPress 3.0 brings with it a multitude of new features — the long-awaited Menu Manager and Custom Post Types which expand WordPress' CMS capabilities. In this book, I am going to teach you to take advantage of WordPress' more advanced features, from shortcodes to custom post types, from creating options pages to custom widgets. Once you're done with this book, you will have a thorough grasp of exactly what is needed to make a WordPress theme usable, flexible and useful.

In this book, I'm not going to be following the conventional methods of writing that most books on WordPress development follow. The approach being taken here assumes that you already know intermediate WordPress coding (for example, you've read the [Rockstar WordPress Designer](#) book) and have an understanding of PHP. I've already converted the theme from HTML to WordPress. You can find the complete theme in the folder *WordPress*, entitled *Rockable*. What you are going to be working with is a stripped down version of this theme. You will find a folder in the *WordPress* directory called *BasicFramework*. Take a quick glance at it. You will notice that there are no files in the functions, language, and widgets folders. This is the main difference between the framework and the complete theme. As we go along, you will be filling up these folders with the necessary code — if you have any problems, you can just take a look at the completed theme to figure things out. Note that the *BasicFramework* is **not** a theme. It lacks many critical files to work as a theme, so WordPress will just show you

errors if you try to install it as a theme. You have to start with this *BasicFramework* and make it into a complete theme.

So, jump in and get ready to rumble!

## Files Included With The Book

Packaged with this book, you will find several folders with files relating to this theme. They are:

- **Photoshop Files:** The PSD files for the design used in this book.
- **HTML Theme:** This is the HTML version of the PSD theme. It consists of the various pages used in the theme — home page, static page, blog page, single blog post, etc.
- **HTML Admin:** This is the HTML design of the administration panel used in the theme. I find it useful first to create the admin theme in HTML, and then logically break it up into smaller parts which are generated dynamically by PHP. Sounds like a mouthful, but it's quite interesting — I will demonstrate how to do this in the book.
- **BasicFramework:** This is the folder with the basic framework for the theme. Rather than repeat everything you probably already know about WordPress, I decided to convert and create the basic structure for the theme. This consists of the regular WordPress theme files, including the `index.php` file, `style.css` stylesheet and various other regular theme files such as `page.php`, `single.php`, etc. You can read more about WordPress template files [here](#) and [here](#).
- **Completed Theme:** This is the final completed theme which is ready for use. It is basically the *BasicFramework* plus all the advanced features you will learn about in this book.

These files and templates can be used by you for both personal and commercial purposes, but they may not be resold or redistributed.

In addition, I recommend you use the *BasicFramework* as a base to work upon while proceeding through the book. Note a few things about this though:

- The *BasicFramework* is NOT a working theme
- It regularly uses the `get_option()` call. This is a WordPress function that basically fetches an option from the database. Example usage: `get_option('theme_style')`. You can view the WordPress Codex article on this function [here](#). We are going to be creating a WordPress theme options page which adds options to the database, so this function is immensely helpful.

## A Code Convention

Sometimes the code will be longer than can fit on the width of this book's page. If so, there will be a continuation marker (► or ▷) at the end of the line. This means the text on the following line is intended to be typed together with the previous line as one. For example, the following is all typed as one line, with no break:

```
define('ROCKABLE_THEME_DIR',  
    get_bloginfo('template_directory'));
```

A hollow arrow indicates that a space is permissible between the character at the end of the starting line and the first character of the second; a solid arrow indicates that there should be no space. The marker itself is not typed in.





# Setup & Managing Files

## Managing Files

While coding a WordPress theme, you should keep something in mind: keep things as easy to edit as possible. This means that you should reuse code that occurs multiple times, separate files that are used many times, and write functions rather than put, in large pieces of code all over the place.

Open up your *BasicFramework* and take a look at the **search.php**, **archive.php**, and **taxonomy-portfolio\_cat.php** files. These files are respectively used for search results, archives (category, tag, date archives), and archives for the custom taxonomy we define later. You will notice that in all three files an **include** statement is used:

```
<?php include (ROCKABLE_INCLUDES . 'post-excerpt.php'); ?>
```

This file, **post-excerpt.php**, displays the excerpt for a post along with some metadata and a thumbnail. Rather than repeating this code in three different files, I decided instead to include one file that is reusable in many places. That way, if I wanted to make a change, I could make it in one place and the change would be reflected in every page template that uses this file. In addition to this file, there are five other files in the **includes** folder:

- **homepage-slider.php**: This file is for the homepage slider. I have separated this into a file of its own for modularity and so that editing it is easier. It makes sense to separate large code blocks into their own files.
- **pagination.php**: This file includes the plugin required for pagination and displays pagination links.

- **post-meta.php:** This displays the metadata (date of publishing, comments, author, categories, etc.) for the post.
- **post-thumbnail.php:** This displays (optionally) a post-thumbnail for the post.
- **widget-columns.php:** This displays the three widget columns in the footer.

## The functions.php File

The `functions.php` file is a vital part of your theme. From [the Codex](#):

*A theme can optionally use a functions file, which resides in the theme subdirectory and is named `functions.php`. This file basically acts like a plugin, and if it is present in the theme you are using, it is automatically loaded during WordPress initialization (both for admin pages and external pages).*

We're going to use the functions file a lot during this book. Open it up, and add in this code:

```
<?php
$curr_theme = get_theme_data(TEMPLATEPATH . '/style.css');
$theme_version = trim($curr_theme['Version']);
if (!$theme_version) $theme_version = "1.0";

//Define constants:
define('ROCKABLE_FUNCTIONS', TEMPLATEPATH . '/functions/');
define('ROCKABLE_WIDGETS', TEMPLATEPATH . '/widgets/');
define('ROCKABLE_INCLUDES', TEMPLATEPATH . '/includes/');
define('ROCKABLE_THEME', 'WordThemer');
define('ROCKABLE_THEME_DIR',
    get_bloginfo('template_directory'));
```

```
define('ROCKABLE_THEME_DOCS',  
    ROCKABLE_THEME_DIR.'/functions/docs/docs.pdf');  
define('ROCKABLE_THEME_LOGO',  
    ROCKABLE_THEME_DIR.'/functions/img/logo.png');  
define('ROCKABLE_MAINMENU_NAME', 'general-options');  
define('ROCKABLE_THEME_VERSION', $theme_version);
```

In this section of code, we are defining some constants which will be used in our theme. These constants can be used in any file in the theme, and they help in making code more readable and reducing its length. For example, if I wanted to include the file **Rockable/includes/post-excerpt.php**, I would have to use this code:

```
include(TEMPLATEPATH . 'includes/post-excerpt.php');
```

Using my constants, it becomes shorter, and you immediately understand what's happening when you read it:

```
include(ROCKABLE_INCLUDES . 'post-excerpt.php');
```

In addition, there is another advantage: suppose I decided to change my includes folder to modules — if using constants, all I would have to do is change the definition for **ROCKABLE\_INCLUDES** — otherwise, I would have to search through every single file, replacing “/includes” with “/modules” — a rather annoying task.

The first three lines use built-in WordPress functions to get the current theme version from **style.css**.

## Basic Setup

Next step: put the **BasicFramework** folder into your **wp-content** folder, and activate the theme. Now, go to your WordPress site. You should see an error:

```
Fatal error: Call to undefined function rockable_titles()
```

This means that we have called a function **rockable\_titles()** in **header.php**, but haven't defined this function. Let's do that now.

First, create an empty file named **custom-functions.php** in the **functions** folder. This will hold all of our custom functions used in various places in the theme. The function in question dynamically outputs the title for the current page. Here's the code I used:

```
//TITLES
function rockable_titles(){
    $separator=stripslashes(get_option('rockable_separator'));

    if (!$separator)
        $separator="|";
    if (is_front_page())
        bloginfo('name');

    else if (is_single() or is_page() or is_home()){
        bloginfo('name');
        wp_title($separator,true,'');
    }
    else if (is_404()){
        bloginfo('name');
        echo " $separator ";
        _e('404 error - page not found', 'rockable');
    }
    else{
        bloginfo('name');
    }
}
```

```
wp_title($separator,true,'');  
}  
}
```

Quite straightforward. First, get a separator. If none is defined, default to “|”. Then, output a title:

- The blog name on the home page.
- The blog name, separator, and page title on pages and posts — e.g.: **Sitename | Page name**.
- The blog name and “404 error” on 404 error pages.
- A default fallback on other pages.

**Note:**

You will notice, instead of just using

```
echo '404 error - page not found';
```

I use

```
_e('404 error - page not found', 'rockable');
```

The reason for this is theme localization (easy translation). We’ll come to that later, but for now know this: wherever you have ‘hard coded’ text, use `_e` instead of `echo`, and use `__` (double underscore) instead of a direct assignment. Examples:

- `_e('some text', 'rockable')` instead of `echo 'some text'`
- `$str = __('some text', 'rockable')` instead of `$str = 'some text'`

Now that our function is defined, we need to include the file so that this function is available to WordPress. Add this to **functions.php**:

```
//Load all-purpose functions:
require_once (ROCKABLE_FUNCTIONS .
'custom-functions.php');
```

Next, we'll get some basic stuff set up — let's start with menus. WordPress 3.0 created a new type of menu management — you can add any links you want, and are no longer restricted to page-only or category-only menus as before. You can see the new menu interface by going to *Appearance* ➤ *Menus* in your WordPress Dashboard. This will only show up if you have support for menus in your theme. To add support, include this line of code in **functions.php**:

```
add_theme_support( 'menus' );
```

Now you will be able to see the menu interface:

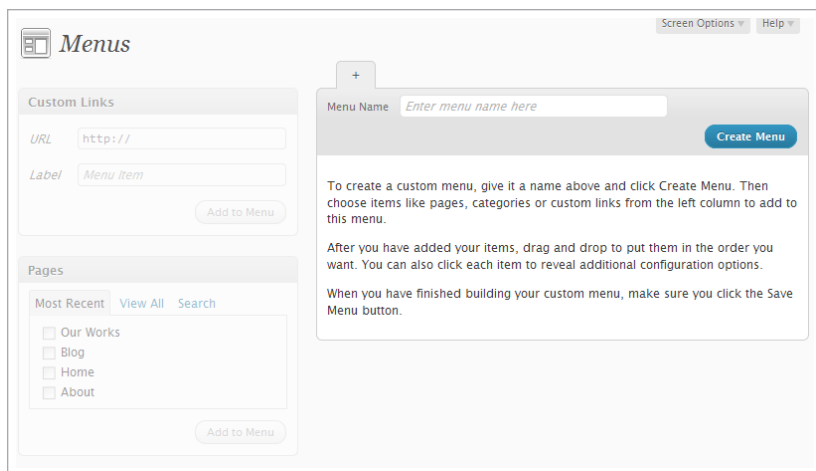


Fig. 1-1. Menus interface.

Next, you have to “register” a menu area. This means that when a user creates a menu, they will see a place in the theme to assign the menu. For example, I could assign a location for a header menu — when a user creates a menu and adds it to the header menu location, the menu will show up there. Create a new file named **register-wp3.php** in the **functions** folder and add this code to it:

```
if (function_exists('register_nav_menu'))  
    register_nav_menu('main_menu', __( 'Main Menu',  
                                     'rockable' ));  
endif;
```

This registers a new nav menu area with the ID “main\_menu” and the (translatable) name “Main Menu”. Include the new file from **functions.php** with this code:

```
require_once (ROCKABLE_FUNCTIONS . 'register-wp3.php');
```

Now, create a new menu. You will see this box when it’s created:

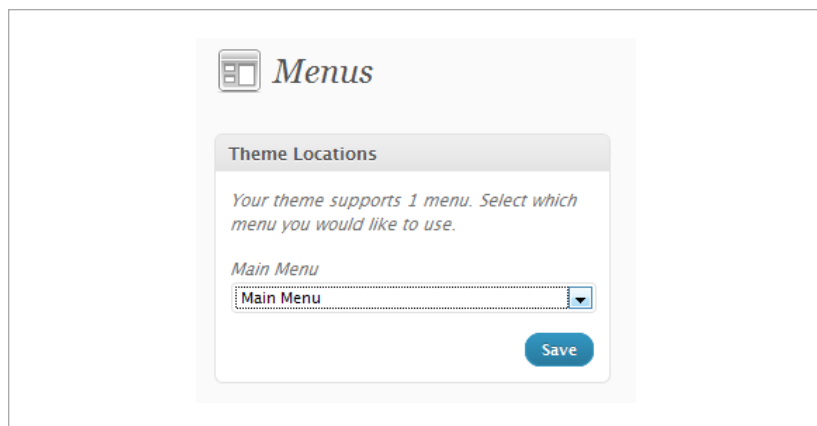


Fig. 1-2. Registered menu locations.



If you look at the HTML version of the theme, you will notice that there is a small description under the menu title. To do this, we need some custom code, as it is not immediately supported in WordPress. In your **custom-functions.php** file, create a new function **rockable\_menu()** and add in this code:

```
function rockable_menu(){  
    if (function_exists('wp_nav_menu') &&  
        has_nav_menu('main_menu')):  
        wp_nav_menu(  
            array(  
                'theme_location' => 'main_menu',  
                'container' => '',  
                'menu_class' => 'sf-menu',  
                'depth' => 2,  
                'walker' => new rockable_menu_walker()  
            );  
  
        else  
            echo '<ul class="sf-menu">';  
            wp_list_pages('depth=1&title_li=');  
            echo '</ul>';  
  
        endif;  
}
```

We're using a few new functions here:

- First, we check if the function **wp\_nav\_menu()** exists. If it does not, the version is less than WP 3.0, so we can't use custom nav menus.
- Then, we use **has\_nav\_menu()** — This function checks if a menu has been assigned to a theme location.
- If either of these are false, then we can't display a nav menu. So, we default to the **wp\_list\_pages()** function.

## The `wp_nav_menu()` Function

The `wp_nav_menu()` function displays a custom nav menu. Its parameters are:

- **menu:** The ID, slug or name of a created menu
- **container:** The wrapping element, e.g. `<div>`
- **container\_class:** The class applied to the container
- **container\_id:** The ID applied to the container
- **menu\_class:** The class assigned to the menu `<ul>`
- **menu\_id:** The ID applied to the menu `<ul>`
- **echo:** Whether to actually display the menu, or just return it; defaults to `true`
- **fallback\_cb:** The fallback function if the menu cannot be displayed, e.g. if no menu is found
- **before, after:** The text to display before and after the link text
- **link\_before, link\_after:** The text to display before and after the link
- **depth:** How many levels of sub-menus to display. Default is `0` (unlimited levels)
- **walker:** a “walker” object to use for the menu (more on this below)
- **theme\_location:** The ID of a registered nav menu (`main_menu`) in our case

For our menu, I have added arguments as per the HTML — no container element, a class of `sf-menu` assigned to the `<ul>` and

a depth of 2 (parents and one sublevel). The interesting part here is the argument for a custom Walker. In 99% of cases, this isn't required — we need it only to display the description. On its own, WordPress uses a *Walker* class to display the navigation. We can simply *extend* this Walker class to output things our way. Read more about classes here: <http://php.net/manual/en/language.oop5.php>.

Rather than extend our Walker without any base, I decided to just modify the regular Walker code. If you see here, in the core code of WordPress: <http://core.trac.wordpress.org/browser/trunk/wp-includes/nav-menu-template.php>, you can see the Walker code at the top of the file. We only need to deal with the `start_el()` function. Add this code to `custom-functions.php`:

```
class rockable_menu_walker extends Walker_Nav_Menu
{
    function start_el(&$output, $item, $depth, $args){
```

The keyword `extends` tells PHP we are writing a class which just builds upon the `Walker_Nav_Menu` class. Basically, we are just overwriting the `start_el()` function.

Now, just copy the entire code of the `start_el()` function from the source code in that link and paste it into your `start_el()` function. Add this line below the `$attributes` assignment:

```
$description = ! empty( $item->description ) ? '<span>'.  
esc_attr( $item->description ).'</span>' : '';
```

That fetches the description of the menu item and assigns it to `$description`.

Now, we need to display this description somewhere. Look for this line in your code:

```
$item_output .= $args->link_before . apply_filters(
    'the_title', $item->title, $item->ID ) .
    $args->link_after;
```

Change that to:

```
$item_output .= $args->link_before . apply_filters(
    'the_title', $item->title, $item->ID );
$item_output .= $description . $args->link_after;
```

The difference: we've added in our **\$description** variable to show in the output. That's it. With those two edits we're done with the backend code. You need to do one more thing to add descriptions though. In the nav menu interface, click on *Screen Options* and check the *Description* checkbox. This will allow you to enter descriptions.

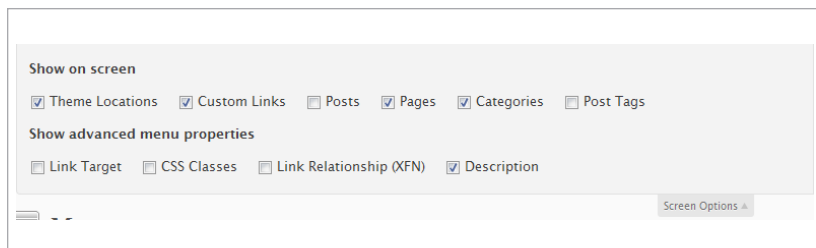


Fig. 1-3. Menus interface — screen options.

And that's it for the nav menus. Before we go further, we should quickly write up a few generic functions which will be used throughout the theme and that can be used in other themes as well.

## JavaScript Files

You might notice, there are no JavaScript files included in **header.php**. Instead, we are going to use a WordPress function called `wp_enqueue_script()`. This function **safely** adds JS files

to the WordPress page. It prevents conflicts: for example, if you added jQuery to the header and a plugin also wanted to add jQuery, there would be two jQuery files included in the page. If you used `wp_enqueue_script()` then it would only be added once — WordPress would take care of that.

So, we are going to enqueue all the JS files used in the theme. Add this to **functions.php**:

```
if ( !is_admin() ) {  
    wp_enqueue_script('jquery');  
    wp_enqueue_script('superfish', ROCKABLE_THEME_DIR.  
        '/assets/js/superfish.js');  
    wp_enqueue_script('jcarousel', ROCKABLE_THEME_DIR.  
        '/assets/js/jquery.jcarousel.min.js');  
    wp_enqueue_script('jqueryui_custom', ROCKABLE_THEME_DIR.  
        '/assets/js/jquery-ui-1.8.1.custom.min.js');  
    wp_enqueue_script('cufon', ROCKABLE_THEME_DIR.  
        '/assets/js/cufon.js');  
    wp_enqueue_script('font', ROCKABLE_THEME_DIR.  
        '/assets/js/myriad_pro.font.js');  
    wp_enqueue_script('rockable', ROCKABLE_THEME_DIR.  
        '/assets/js/rockable.js');  
    wp_enqueue_script('rockable_contact',   
        ROCKABLE_THEME_DIR.'/assets/js/rockable_contact.js');  
    wp_enqueue_script( 'comment-reply' );  
}
```

The usage of the function is:

```
wp_enqueue_script($handle, $src, $deps, $ver, $in_footer);
```

**handle** is the unique name you give to the script, **src** is the source URL, **deps** is the dependency (e.g. if it is reliant on jQuery), **ver** is the version, used to prevent caching when versions are changed

and `in_footer` specifies whether to add the footer to the `<head>` or to the footer.

## Image Functions

Images are a pretty important part of premium WordPress themes. They are used in most themes as post thumbnails, in galleries, etc. So rather than perform the same image-fetching techniques many times, we should automate the process. Here are two functions (to be added to `custom-functions.php`) which will automate this for us.

```
function rockable_post_image(){
    global $post;
    $image = '';

    //Get the image from the post meta box
    $image = get_post_meta($post->ID, 'rockable_post_image', >
        true);
    if ($image) return $image;

    //If the above doesn't exist, get the post thumbnail
    $image_id = get_post_thumbnail_id($post->ID);
    $image = wp_get_attachment_image_src($image_id, >
        'rockable_thumb');
    $image = $image[0];
    if ($image) return $image;

    //If there is still no image, get the first image from >
    the post
    return rockable_get_first_image();
}
function rockable_get_first_image(){
    global $post, $posts;
    $first_img = '';
```

```
ob_start();
ob_end_clean();

$output = preg_match_all('/<img.+src=[\'"]([^\\'"]+)[\\'"].*>/i', $post->post_content, $matches);

$first_img="";

if (isset($matches[1][0]))
    $first_img = $matches[1][0];

return $first_img;
}
```

Explanations: `rockable_post_image()` fetches the image for the current post. It follows an order: first, it looks for a custom field with the key “`rockable_post_image`”. If not found, it looks for the post thumbnail (a built in WordPress feature). If there is still no image found, it scans the content of the post to search for the first image, using regular expressions (the function `rockable_get_first_image()`). Therefore, if you wanted to display the post image, you could use code like this:

```

```

Simple, don't you agree?

**Note:** *You might notice that I am prefixing every function with `rockable_` — e.g. `rockable_post_image()`. This is so that there are no conflicts with other plugins or scripts. If I used a generic function name like `get_image()` there could be conflicts with the core WordPress code or with plugins. Always try to use a unique prefix in your code.*

If you take a look at the `php` folder of the theme, you'll notice a file `timthumb.php` — this is an automatic resizing script — read more

about it here: <http://www.darrenhoyt.com/2008/04/02/timthumb-php-script-released/>

TimThumb is very useful in WordPress themes. Since it dynamically resizes images, by uploading just one image you can have different image sizes throughout the theme without uploading many differently cropped images. It's used like this:

```

```

That will give you a resized image of **image.jpg**, with a width (**w**) and height (**h**) of 100 pixels.

We're going to be using TimThumb in this theme. Now, rather than writing the same code over and over, I've written a function in **custom-functions.php** to generate the TimThumb code for us:

```
function rockable_build_image($img='', $w=false, $h=false, $zc=1 ){
    if ($h)
        $h = "&h=$h";
    else
        $h = '';

    if ($w)
        $w = "&w=$w";
    else
        $w = '';

    $image_url = ROCKABLE_THEME_DIR .
        "/php/timthumb.php?src=" . $img . $h . $w;
    return $image_url;
}
```

The parameters for this function are the image source URL, width, height, and the zoom-crop. One thing to note: if you specify only a



height or width, TimThumb will maintain the aspect ratio. Therefore, you are able to specify only height or width here, if wanted. Sample usage:

- `rockable_build_image('image.jpg', 100)` will give you the URL to an image with a width of 100, and a corresponding height. So if your original image was  $200 \times 70$  pixels, the new one will be  $100 \times 35$ .
- Similarly, `rockable_build_image('image.jpg', false, 100)` will give you the URL to an image with a height of 100, and a corresponding width. So if your original image was  $120 \times 200$  pixels, the new one will be  $60 \times 100$ .

Finally, there is one more generic function to be written. WordPress has a built in function called `the_excerpt()` which displays a 55 word excerpt of the page. But what if we want a different number of words in the excerpt? Here's the code:

```
function rockable_excerpt($len=20, $trim="&hellip;"){
    $limit = $len+1;
    $excerpt = explode(' ', get_the_excerpt(), $limit);
    $num_words = count($excerpt);
    if ($num_words >= $len){
        $last_item = array_pop($excerpt);
    }
    else{
        $trim = "";
    }
    $excerpt = implode(" ", $excerpt) . "$trim";
    echo $excerpt;
}
```

Whoa, you say, what's that? Let me break it down for you. The parameters are the excerpt length (default 20) and the trim character displayed after the excerpt (defaults to the HTML entity `&hellip;` which is an ellipsis: ...). What it does is this:

- First, it gets the excerpt using `get_the_excerpt()`.
- Then, it `explodes` (breaks up) the excerpt into words by separating them by blanks. For example, “Rockable is awesome” would be broken into `array('Rockable', 'is', 'awesome')`. The exploded array is limited to the length of the new excerpt, given by `$len`.
- Then, if the number of words in `$excerpt` is more than the length, it removes the rest of the words.
- Finally, it joins up the array `$excerpt` using the `implode()` function (opposite of `explode()`) and appends the `$trim` to the end. It prints this new excerpt.

Sample usage:

```
rockable_excerpt(5); //prints a 5 word excerpt
```

With this, our basic setup is over and Chapter 1 comes to an end. In the next chapter, you will learn how to create and style theme options pages.

2

# Theme Options Pages

Theme Options pages are administration pages added to the WordPress backend to make customizing a theme easier for users. When creating WordPress themes now, it's almost a necessity to have useful and usable theme options. I'm going to teach you how to create a WordPress theme options panel of any design you want.

## The Options Page Design

The first step to creating a theme options page is the design: what do you want it to look like? Take a look at your WordPress Dashboard. The header and the navigation on the left are part of all WordPress backend pages. However, the rest of the page can be customized to your design.

To start, I've already made an options page design. In the files included with the book, take a look at the directory with the admin panel HTML. You will notice I have styled all the basic input styles: textareas, radios, checkboxes, etc. Along with this, there is a head with the theme logo and some text in the header. Most importantly, each option group has a consistent structure. The HTML looks like this:

```
<div class="rockable_section">
  <div class="rockable_title clearfix">
    <!-- logo, etc. goes here -->
  </div><!-- //rockable_title -->
  <div class="rockable_options">
    <div class="rockable_input rockable_textarea clearfix">
      <div class="label">Input name</div>
      <div class="option_wrap">
        <div class="option_control">
          <input element>
        </div>
      </div>
    </div>
  </div>
</div>
```

```
<div class="description"> [...] </div>
</div><!-- /option_wrap -->
</div><!-- //rockable_textarea -->
</div><!-- //rockable_options -->
</div><!-- //rockable_section -->
```

The part in bold is the input element block — it repeats again and again for new elements. That’s the key to consistent structure. You don’t need to stick with this design — you could design and style something of your own too; it’s simple to integrate.

To begin the process of creating the options page, in your **functions** directory, you can see there are three folders entitled **css**, **img**, and **scripts**. These hold the stylesheets, images, and JavaScript files necessary for the theme options page.

## Options Files

Now, you might have noticed how I’ve been stressing that you should separate code into files and reuse code. Organization is a very important part of web development because it helps you to work more efficiently. In fact, I have my own starter WordPress with a framework for building theme options, meta boxes, etc. So, it’s going to be no different this time — we are going to have our options in separate files. Create a new directory **options** in your **functions** folder. The structure of our options page is going to be as follows.

First, we have some basic information which is relevant to the page:

```
<?php
$info = array(
    'name' => 'general',
    'pagename' => 'general-options',
```

```
'title' => 'General Settings',  
'sublevel' => 'true'  
);
```

**name** is the unique name for this options page. **pagename** is the term going to be used in the URL of the page, i.e. `/wp-admin/admin.php?page=general-options` in this case. **title** is the page title displayed in the options page. **sublevel** is only put in if the page is a submenu and NOT a top level menu page.

Then, we have a PHP array of the options for this page:

```
$options = array(  
    "type" => "",  
    "name" => "",  
    "id" => "",  
    "options" => array(""),  
    "desc" => "",  
    "default" => ""  
);
```

An explanation of the array elements:

- **type:** The input type — text, textarea, select, radio, checkbox, checkbox-nav or image
- **name:** The part displayed in the `div.label`
- **id:** The ID with which the option is stored. For checkboxes, this is an array of IDs:

```
"id" => array("id_1", "id_2", "id_3")
```

- **options:** Doesn't apply to text, textarea or image inputs. An array of options, the actual text displayed for the input.
- **desc:** The description for the element.

- **default:** The default value for the element. For checkboxes, it is an array, each element corresponding to the ID. Example:

```
"id" => array("id_1", "id_2", "id_3"),  
"default" => array("checked", "not", "checked")
```

This would mean a checkbox element with three checkboxes, the first and third being checked by default.

Finally, we have the object creation:

```
$optionspage = new rockable_options_page($info, $options);
```

This generates a theme options page by creating a new *object* of the **rockable\_options\_page** class.

In this way, we build up an array of options for the particular page. Now, from the completed theme, copy over the entire options directory into the **BasicFramework** directory. You can take a look at the different options. The file **slider-settings.php** is a little different; we'll get to that soon.

Most of the options are pretty regular, but you can take a look at **blog-options.php**. Notice the code near the top of the file.

```
$all_categories=get_categories('hide_empty=0&orderby=name');  
$cat_list = array();  
$cat_options = array();  
$checked_cats = array();  
foreach ($all_categories as $category_list){  
    $cat_list[] = "rockable_cat_" . $category_list->cat_ID;
```

```

$cat_options[] = $category_list->cat_name;
$checked_cats[] = "checked";
}

```

First, every category is fetched using `get_categories()`. Then, we loop through the categories. `$cat_list` holds the IDs for the category, prefixed with 'rockable\_cat'. `$cat_options` holds the category names. `$checked_cats[]` holds the original checked/unchecked status for the category. Then, we create a checkbox-nav option for this. The only difference between `checkbox` and `checkbox-nav` is the styling. We use a script to add some custom styling to these checkbox-nav elements.

**Note:** *To actually implement this, we need a function that generates a list of categories to exclude. Add this to `custom-functions.php`:*

```

function rockable_build_cat_exclude(){
    $categories = get_categories('hide_empty=0&orderby=id');
    $exclude="";

    foreach($categories as $cat):
        $cat_field = 'rockable_cat_' . $cat->cat_ID;
        if (get_option($cat_field) && get_option($cat_field) >
            == 'false')
            $exclude .= "-" . $cat->cat_ID . ",";
    endforeach;

    if ($exclude)
        $exclude = substr($exclude, 0, -1); //Remove the last
        comma

    return $exclude;
}

```



The logic involved is simple.

1. Fetch all the categories.
2. Create names similar to the IDs with which the category excludes are stored in the database.
3. Check if the stored option is equal to the string `'false'`. If so, that means that the checkbox for that category was unchecked (implying excluded). Append the category ID to a comma-separated string. Note that a minus sign ( `-` ) is prepended to the ID so that it is excluded, not included.
4. Loop through all the categories like this. You should end up with a string like `-3, -15, -33,`
5. Now, we need to remove the last comma if there is one. If the `$exclude` has been set, then we know there is a trailing comma and we remove the last character.

The function is used like this:

```
$exclude = rockable_build_cat_exclude();  
query_posts('cat=' . $exclude);
```

Now, we need to write some code that actually generates HTML from this array of options.

## Generating the Pages

Options pages rely on a few basic functions to be created. First, we add actions. Adding actions tells WordPress we're going to be doing some stuff here. We need to add actions for generating the menu, as well as for any scripts or styles we want to add. Here's the code:

```
add_action('admin_init', 'initialize_function');  
add_action('admin_menu', 'admin_generating_function');
```

Then, we need to write the initialize and admin generating functions. So, if we wanted to add five options pages, we would have to add 10 actions, and then write 15 functions for generating the pages — init, the menu function, and an HTML outputting function. That's **way** too much work. If you want just one page, it's useful. See an example here: <http://net.tutsplus.com/tutorials/wordpress/how-to-create-a-better-wordpress-options-panel/>

To simplify things for multiple options pages, we're going to be using *classes*. A class is basically a blueprint — like a function, only much more usable. Within classes, we have functions. Classes are used by creating *objects* of them. Think of it like this: a class is like a car factory. We have machines which do the work for us — those represent the functions of the class. Once we build the factory, we can use it to make many, many cars. Similarly, once we build an option generating class, we can make many options pages! Objects are created with the **new** keyword. Example:

```
$object = new classname(); //Creates a new object of the  
class classname
```

Here's what a basic class looks like:

```
class classname{  
    function classname(){  
        //...  
    }  
    //more functions...  
}//end class
```

You might notice that I have a function with the same name as the class — this is called a *constructor*. A constructor is a function that

doesn't need to be called — whenever you create a new object, the constructor is automatically run through.

Don't worry: it's not completely necessary to understand the actual concept of classes. You can understand everything if you know how to use functions.

Coming to our options pages, we need to do the following:

- Add an action for the current menu page — the `admin_menu` action.
- Use the function defined in `admin_menu` to add a menu page.
- Use another function to output the HTML for the page.
- Save the options when the form is submitted.

So, this is an outline of our class. Paste it into a new file in the `functions` folder named `generate-options.php`:

```
class rockable_options_page{
    //constructor:
    function rockable_options_page(){
    }
    //Add menu item
    function rockable_add_menu(){
    }
    //Generate functions page
    function rockable_generate_page(){
    }
    //Save options
    function save_options(){
    }
}
```

Now, we need to fill in those functions one by one. First, let's deal with the constructor. We know from our options pages that we have two variables: **\$info**, the page information, and **\$options**, the options. Add this code above the constructor:

```
//data members:  
var $options;  
var $file_name;  
var $page_title;
```

Those will hold the options, the file name and the page title. Then, use this code for the constructor:

```
//constructor:  
function rockable_options_page($info, $options){  
    $this->info = $info;  
    $this->file_name = $info['pagename']; //filename for the  
    options page  
    $this->page_title = $info['title']; //title, displayed  
    near top of page  
    $this->options = $options;  
  
    add_action('admin_menu', array(&$this,  
        'rockable_add_menu'));  
  
}
```

So, we have two parameters passed to the class: **\$info** and **\$options**. You can see this in the options pages. One thing about classes: You can't just use variables directly. You use the **\$this** keyword to refer to the current object. So, we assign the variables to the object **\$this**. Now, we call the next function, **rockable\_add\_menu**. Again, we can't directly call the function. We use **array(&\$this, 'rockable\_add\_menu')** to refer to the function. The awesome part about this is, whenever we create a new object (with **one** line of code), we don't need to worry about

writing the same function again and again, we just need to call the function for the **current** object!

Next, the **rockable\_add\_menu** function. The code:

```
//Add menu item
function rockable_add_menu(){
    if (!isset($this->info['sublevel'])){
        add_menu_page(ROCKABLE_THEME, ROCKABLE_THEME,
            'administrator', $this->file_name,
            array(&$this, 'rockable_generate_page'),
            get_bloginfo('template_directory').
                '/functions/img/icon.png');
    }
    else{
        add_submenu_page(ROCKABLE_MAINMENU_NAME,
            $this->page_title, $this->page_title,
            'administrator', $this->file_name,
            array(&$this, 'rockable_generate_page'));
    }
}
```

Simple enough. If the *sublevel* index discussed earlier is present, it's a sublevel page. Otherwise, it's a top-level page. We then use **add\_menu\_page** or **add\_submenu\_page** to add the page. The functions are used like this:

```
add_menu_page($page_title, $menu_title, $capability,
    $menu_slug, $function, $icon_url);
add_submenu_page($parent_slug, $page_title, $menu_title,
    $capability, $menu_slug, $function);
```

- **\$page\_title** is the text in the <title> tags.
- **\$menu\_title** is the title on the screen.

- **\$capability** is the user role allowed to access the menu — we only allow administrators to view it. Other roles and capabilities can be seen [here](#).
- **\$menu\_slug** is the slug for the menu page.
- **\$function** is the function used to generate the HTML for the page.
- **\$icon\_url** is the URL to an icon for the menu.
- **\$parent\_slug**, for submenu pages, is the slug of the parent menu under which this is a submenu. We use **ROCKABLE\_MAINMENU\_NAME**, a constant defined in **functions.php**.

Next, we need to write **rockable\_generate\_page**, which generates the HTML from the **\$options** array. Note one thing though: when we generate a page, we are creating a **<form>** which **POSTs** the data back to the **same** page. So, we should save the options on page load. The first part of the code:

```
//Generate functions page
function rockable_generate_page(){
    $this->save_options();
?>

<form method="post" id="rockable_options_form"
class="rockable_options_form">
<input type="hidden" name="action" id="action"
value="rockable_save_options" />
<div class="rockable_section">
    <div class="rockable_title clearfix">
        <div class="rockable_title_image clearfix">
            " />
        <div class="rockable_meta">
            <div>Version: <?php echo ROCKABLE_THEME_VERSION;>
```



```

        case "checkbox-nav":
            $this->display_checkbox_nav($value); break;
        case "radio": $this->display_radio($value); break;
        case "select": $this->display_select($value); break;
    }
}
?>

```

This is a **crucial** part of the function. What we're doing is looping through the **\$options** array using a **foreach** loop. Depending on the option *type*, we display the option using an appropriate function. Of course, we need to write these functions as part of the class.

The final piece of code:

```

<div class="rockable_input rockable_save clearfix">
    <div class="label">Save Options</div>
    <div class="option_wrap">
        <div class="option_control">
            <input type="submit" class="button"
                id="final_submit" name="final_submit"
                value="Save changes" />
        </div>
        <div class="description">Save the options</div>
    </div><!-- //option_wrap -->
</div><!-- //rockable_upload -->
</div><!-- //rockable_options -->
</div><!-- //rockable_section -->
</form><!--//save form-->
<?php
}

```

That displays the static footer for the page. We display one last element with a save button, and then close all the required **divs** and **forms**. Take the entire code and paste it into the file. Now,



imagine you were using a different design with different HTML. Your header and footer parts would naturally be updated to match the HTML of your design. In addition, the functions to generate the options will naturally be updated. Now, we'll write the functions for the options:

```
function display_text($value){
    ?>
        <div class="rockable_input rockable_text clearfix">
            <div class="label"><?php echo $value['name']; ?></div>
            <div class="option_wrap">
                <div class="option_control">
                    <input name="<?php echo $value['id']; ?>"
                        id="<?php echo $value['id']; ?>" type="text"
                        value="<?php if (get_option($value['id']))
                        echo esc_html(stripslashes(get_option($value
                        ['id']))); else echo $value['default']; ?>" />
                    </div>
                    <div class="description"><?php echo $value['desc'];>
                        ?></div>
                </div><!-- //option_wrap -->
            </div><!-- //rockable_text -->
        <?php
    }
    function display_textarea($value){
        ?>
            <div class="rockable_input rockable_textarea clearfix">
                <div class="label"><?php echo $value['name']; ?></div>
                <div class="option_wrap">
                    <div class="option_control">
                        <textarea rows="5" cols="25"
                            id="<?php echo $value['id']; ?>"
                            name="<?php echo $value['id']; ?>"
                        <?php
                            if (get_option($value['id']))
                                echo stripslashes(get_option($value['id'])) ;
```

```

        else
            echo $value['default'];
        ?></textarea>
    </div>
    <div class="description"><?php echo $value['desc'];>
        ?></div>
    </div><!-- /option_wrap -->
</div><!-- //rockable_textarea -->
<?php
}

```

These are the functions for displaying the `<input type='text' />` and `<textarea />` elements. We display the HTML around the element (hence the need for a consistent structure), and then the actual input element. The value of the element is determined using simple logic:

- If the option is stored in the database, we fetch the value and display it.
- If the option is not stored in the database, we display the default value, if any.

The next function, for images, is slightly different:

```

function display_image($value){
    ?>
    <div class="rockable_input rockable_image clearfix">
        <div class="label"><?php echo $value['name']; ?></div>
        <div class="option_wrap">
            <div class="option_control">
                <input type="text" value="<?php
                if (get_option($value['id']))
                    echo stripslashes(get_option($value['id']));
                else
                    echo $value['default'];

```

```

?>" name="<?php echo $value['id']; ?>"/>
<span id="<?php echo $value['id']; ?>"
    class="button upload rockable_upload">Upload >
    Image</span>
<?php if (get_option($value['id'])) :?>
    <span class="button rockable_remove" >
        id="remove_<?php echo $value['id']; ?>" >
        Remove Image</span>
<?php endif; ?>

<div class="rockable_image_preview">
    <?php if (get_option($value['id'])):?>
    
    <?php elseif ($value['default'] != ""):?>
    
    <?php endif; ?>
</div>
</div>
<div class="description"><?php echo $value['desc'];>
?></div>
</div><!-- //option_wrap -->
</div><!-- //rockable_upload -->
<?php
}

```

One thing to note here: we are not going to be using `input type='file'` elements for the image upload. Rather, we're going to be using AJAX image uploads using the script here: <http://github.com/valums/ajax-upload>. So, we have a text input (in case users want to paste in a URL) and a couple of buttons for uploading/removing images. The ID is used in the `<span>` for uploading. We also display a preview for the image. The logic:

- If the option is stored in the database, the URL is drawn from there.

- Otherwise, if the default value isn't a null string, we display the URL from there.

The next two functions deal with the **checkbox** and **checkbox-nav** elements.

```
function display_checkbox($value){
    ?>

    <div class="rockable_input rockable_checkbox clearfix">
        <div class="label"><?php echo $value['name']; ?></div>

        <div class="option_wrap">
            <div class="option_control">
                <?php
                $ctr=-1;
                foreach($value['options'] as $cb_option):
                    $ctr++;
                    $checked='';
                    if (get_option($value['id'][$ctr])){
                        if (get_option($value['id'][$ctr]) == 'true')
                            $checked=' checked="checked"';
                        else $checked='';
                    }
                    else{
                        if ($value['default'][$ctr] == "checked")
                            $checked=' checked="checked"';
                    }
                ?>
                <input type="checkbox"
                    id="<?php echo $value['id'][$ctr]; ?>"
                    <?php echo $checked; ?>
                    name="<?php echo $value['id'][$ctr]; ?>"
                    <label for="<?php echo $value['id'][$ctr]; ?>">
                    <?php echo $value['options'][$ctr]; ?></label>
                <br />
            </div>
        </div>
    </div>
}
```

```
<?php
    endforeach;
?>

</div>

<div class="description"><?php echo $value['desc'];>
?></div>

</div><!-- //option_wrap -->
</div><!-- //rockable_checkbox -->
<?php
}

function display_checkbox_nav($value){
?>

<div class="rockable_input rockable_checkbox_nav clearfix">

<div class="label"><?php echo $value['name']; ?></div>

<div class="option_wrap">
    <div class="option_control">
        <?php
            $ctr=-1;
            foreach($value['options'] as $cb_option):
                $ctr++;
                $checked='';
                if (get_option($value['id'][$ctr])){
                    if (get_option($value['id'][$ctr]) == 'true')
                        $checked=' checked="checked"';
                    else $checked='';
                }
                else{
                    if ($value['default'][$ctr] == "checked")
                        $checked=' checked="checked"';
                }
                $clearfix='';
                if ($ctr%3==0 and $ctr!=0)
                    $clearfix= ' style="clear:both"';
```

```

        $last='';
        if (($ctr+1)%3==0 )
            $last=' last';
    ?>
    <div class="checkbox-nav">?php echo $last; ?>
        <?php echo $clearfix; ?>
        <input class="rockable_super_check"
            type="checkbox"
            id="<?php echo $value['id'][$ctr]; ?>"
            <?php echo $checked; ?>
            name="<?php echo $value['id'][$ctr]; ?>"
            <label for="<?php echo $value['id'][$ctr]; ?>">
                <?php echo $value['options'][$ctr]; ?></label>
        </div>
    <?php
        endforeach;
    ?>
</div>
    <div class="description"><?php echo $value['desc'];>
        ?></div>
    </div><!-- //option_wrap -->
</div><!-- //rockable_checkbox -->
<?php
}

```

These two functions are larger than normal because of the check for whether or not they are checked by default. Here's how that is determined:

- If the option exists in the database and its value is equal to **'true'**, then it is checked.
- If the option exists in the database but its value is not **'true'**, then it is not checked.
- If the option doesn't exist in the database and its corresponding **\$defaults** value is **'checked'**, it is checked.

Another thing to note: Since the checkboxes have `$id` as an array, we use a variable `$ctr` to loop through it and get the corresponding value for `$options` and `$default`.

Finally, we have the functions for the select and radio elements:

```
function display_radio($value){
    ?>
    <div class="rockable_input rockable_radio clearfix">
        <div class="label"><?php echo $value['name']; ?></div>

        <div class="option_wrap">
            <div class="option_control">
                <?php
                $ctr=-1;
                if (get_option($value['id']))
                    $default=get_option($value['id']);
                else $default = $value['default'];
                foreach($value['options'] as $rd_option):
                    $ctr++;
                    $checked='';
                    if ($value['values'][$ctr] == $default)
                        $checked=' checked="checked"';

                ?>
                <input type="radio" id="<?php echo $value['id']; ?>">
                    name="<?php echo $value['id']; ?>"
                    value="<?php echo $value['values'][$ctr]; ?>"
                    <?php echo $checked; ?>>
                <label for="<?php echo $value['id']; ?>">
                    <?php echo $value['options'][$ctr]; ?></label>
                <br />
            <?php
            endforeach;
        ?>
    </div>
}
```

```

        </div>
        <div class="description"><?php echo $value['desc'];>
        ?></div>
    </div><!-- //option_wrap -->
</div><!-- //rockable_radio -->
<?php
}

function display_select($value){
?>
    <div class="rockable_input rockable_select clearfix">
        <div class="label"><?php echo $value['name']; ?></div>

        <div class="option_wrap">
            <div class="option_control">
                <select id="<?php echo $value['id']; ?>"
                    name="<?php echo $value['id']; ?>"
                <?php
                    if (get_option($value['id']))
                        $default=get_option($value['id']);
                    else $default = $value['default'];

                    foreach($value['options'] as $sel_opt):
                        $selected='';
                        if ($sel_opt == $default) $selected=' selected=>
                            "selected"';
                ?>
                <option <?php echo $selected;?>><?php echo
                    $sel_opt; ?></option>
                <?php
                endforeach;
                ?>
            </select>
        </div>
        <div class="description"><?php echo $value['desc'];>
        ?></div>

```



```

        </div><!-- //option_wrap -->
    </div><!-- //rockable_select -->
<?php
    }

```

These are pretty simple once you've read the functions before. They follow the same logic. One thing to note: In the radio, the same name is used throughout.

That concludes the option generating functions. Whew! You can, of course, change the HTML if you are using a different HTML structure. Moving on, we need to write the last function for saving options. Here's the code:

```

function save_options(){
    if (isset($_POST['action']) && $_POST['action'] == 'rockable_save_options' ) {
        foreach ($this->options as $value) {
            $the_type=$value['type'];

            if ($the_type=="checkbox" or $the_type=="checkbox-nav"){
                $ctr=0;
                foreach( $value['options'] as $cbopt):
                    $curr_id=$value['id'][$ctr];
                    if (isset($_POST[$curr_id]))
                        update_option($curr_id, 'true');
                    else
                        update_option($curr_id, 'false');
                    $ctr++;
                endforeach;
            }

            if ($the_type!="checkbox" and $the_type!="checkbox-nav"){

```

```
        update_option($value['id'],
        $_POST[ $value['id'] ]);
    }
}
echo '<div id="message" class="updated fade"><p>
    <strong>'.ROCKABLE_THEME.' settings saved.</strong>
    </p></div>';
}
}
```

Quick explanation:

- First, we check if the save button has been clicked to load this page. This is done by checking if the action variable has been posted. If so, we know that the options have been posted.
- Then, we loop through the options.
- If the type is **checkbox** or **checkbox-nav**, we loop through their IDs as well. We check if each ID has been **POSTed**. If so, then the checkbox was checked and we use **update\_option** to update the value to the string **'true'**. Otherwise, we update it to the string **'false'**.
- If the type is **not** either of those, we just update the value to the **POSTed** value.
- Then, we echo an *Options Saved* message.

That's all. With that function, we're done with generating options!

## Styling and Scripts

Create a new file named **admin-helper.php** in the **functions** folder. Paste in this code snippet:

```

<?php
function rockable_admin_head(){
?>

<link rel='stylesheet' id='rockable-admin-css' href=
'<?php echo ROCKABLE_THEME_DIR; ?>/functions/css/
admin.css' type='text/css' media='all' />

<link rel='stylesheet' id='rockable-admin-css' href=
'<?php echo ROCKABLE_THEME_DIR; ?>/functions/css/
checkbox.css' type='text/css' media='all' />

<link rel='stylesheet' id='rockable-prettyphoto-css'
href='<?php echo ROCKABLE_THEME_DIR; ?>/functions/
scripts/prettyPhoto/css/prettyPhoto.css' type='text/css'
media='all' />

<script type="text/javascript" src="<?php echo
ROCKABLE_THEME_DIR; ?>/functions/scripts/
jquery.checkbox.min.js"></script>

<script type="text/javascript" src="<?php echo
ROCKABLE_THEME_DIR; ?>/functions/scripts/ajaxupload.js">
</script>

<script type="text/javascript" src="<?php echo
ROCKABLE_THEME_DIR; ?>/functions/scripts/rotator.js">
</script>

```

Those will be the static JS and CSS files for the page. Then, we need to write the JavaScript for the AJAX uploads. We use the code for the AJAX upload script mentioned above.

```

<script type="text/javascript">
jQuery(document).ready(function() {
jQuery('.rockable_super_check').checkbox({empty:
'<?php echo ROCKABLE_THEME_DIR; ?>/functions/css/
empty.png'}});

```

```
//Check if element exists
jQuery.fn.exists = function(){
    return jQuery(this).length;}

//AJAX upload
jQuery('.rockable_upload').each(function(){
    var the_button = jQuery(this);
    var image_input = jQuery(this).prev();
    var image_id = jQuery(this).attr('id');
    new AjaxUpload(image_id, {
        action: ajaxurl,
        name: image_id,
        // Additional data
        data: {
            action: 'rockable_ajax_upload',
            data: image_id
        },
        autoSubmit: true,
        responseType: false,
        onChange: function(file, extension){},
        onSubmit: function(file, extension) {
            the_button.html("Uploading...");
        },
        onComplete: function(file, response) {
            the_button.html("Upload Image");

            if (response.search("Error") > -1){
                alert("There was an error uploading:
                \n"+response);
            }

            else{
                image_input.val(response);
                var image_preview='
                class="rockable_image_preview" />';
```

```

        var remove_button_id = 'remove_'+image_id;
        var rem_id = "#"+remove_button_id;
        if (!jQuery(rem_id).exists())){
            the_button.after('<span
                                class="button rockable_remove"
                                id="'+remove_button_id+'">Remove Image
                                </span>');
        }

        the_button.next().next().html(image_preview);
    }
}

});

});

//AJAX image remove
jQuery('.rockable_remove').live('click', function(){
    var remove_button = jQuery(this);
    var image_remove_id = jQuery(this).prev().attr('id');
    remove_button.html('Removing...');

    var data = {
        action: 'rockable_ajax_remove',
        data: image_remove_id
    };

    jQuery.post.ajaxurl, data, function(response) {
        remove_button.prev().prev().val('');
        remove_button.next().html('');
        remove_button.remove();
    });

});

});

</script>
<?php

```

```
}
?>
```

The code, despite its seeming complexity, is actually very simple. We attach an AJAX upload to the buttons with the class **rockable\_upload**. When a file is selected (**onSubmit**) we change the button HTML to '**Uploading...**'. When the file is uploaded (**onComplete**) we change it back to '**UploadImage**'. If there is no error, we update the image preview to the new image, and add a remove button if not already present.

We also attach an event to the remove button. Both AJAX results are posted to the URL **ajaxurl**. This is a URL automatically generated by WordPress in the head where AJAX requests are meant to be posted. The AJAX requests are handled there.

We need to write PHP code for handling the AJAX image upload. Create a new file named **ajax-image.php** in the **functions** folder. Add in this code:

```
<?php
//Save image via AJAX
add_action('wp_ajax_rockable_ajax_upload',
    'rockable_ajax_image_upload'); // Add support for AJAX save
function rockable_ajax_image_upload(){
    global $wpdb; //Now WP database can be accessed

    $image_id = $_POST['data'];
    $image_filename = $_FILES[$image_id];
    $override['test_form'] = false; //see
        http://wordpress.org/support/topic/269518?replies=6
    $override['action'] = 'wp_handle_upload';

    $uploaded_image = wp_handle_upload($image_filename,
        $override);
```

```

        if (!empty($uploaded_image['error'])) {
            echo 'Error: ' . $uploaded_image['error'];
        }
        else {
            update_option($image_id, $uploaded_image['url']);
            echo $uploaded_image['url'];
        }

        die();
    }
    //Remove image via AJAX
    add_action('wp_ajax_rockable_ajax_remove',
        'rockable_ajax_image_remove'); // Add support for AJAX save
    function rockable_ajax_image_remove() {
        global $wpdb; //Now WP database can be accessed

        $image_id = $_POST['data'];

        $query = "DELETE FROM $wpdb->options WHERE option_name
            LIKE '$image_id'";
        $wpdb->query($query);

        die();
    }
    ?>

```

Explanation: first, we need to add **actions** to tell WordPress we are handling the AJAX upload with the named function. Here's what the code looks like:

```
add_action('wp_ajax_AJAX_ACTION', 'function name');
```

You may notice that in the AJAX upload script, we add a data variable action: **'rockable\_ajax\_upload'**. This is necessary. ALL AJAX requests to the **ajaxurl** MUST have an action variable posted. This action variable is written after **wp\_ajax\_** in the

`add_action` statement. Then, in the function, we get the uploaded file and tell WordPress to handle it with `wp_handle_upload`. The code in `$override` is necessary to prevent errors. Once it's handled, we either tell the AJAX requester that there is an error or success and update the related option.

Similarly in the `rockable_ajax_image_remove()` function, we just remove the option with the POSTed ID.

## Including the Files

Now, we include the required files. First, create a file named `include-options.php` in the functions directory. Paste in this code:

```
<?php
include('options/general-options.php');
include('options/homepage-options.php');
include('options/integrate-options.php');
include('options/blog-options.php');
include('options/footer-options.php');
?>
```

This file is responsible for including every options page we need to make.

Open up `functions.php` and add in this code:

```
if (is_admin()) :
    require_once (ROCKABLE_FUNCTIONS . 'admin-helper.php');
    require_once (ROCKABLE_FUNCTIONS . 'ajax-image.php');
    require_once (ROCKABLE_FUNCTIONS . 'generate-options.php');
    require_once (ROCKABLE_FUNCTIONS . 'include-options.php');
endif;
```



We include all the files we created here. Note that we only include the files if it is an admin page ( `is_admin()` ). Then we add an action for the scripts and styles. Paste in this line:

```
add_action('admin_head', 'rockable_admin_head');
```

The hook `admin_head` refers to the creation of the `<head>` of the page. Now go to your WordPress dashboard and refresh — you should see the menu in the navigation!

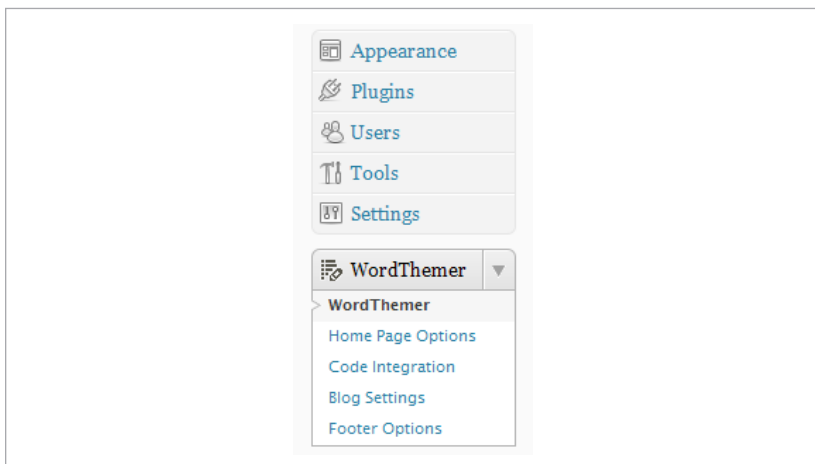


Fig. 2-1. Theme options.

## Using this Theme Options Framework in Another Theme

Now that we have a class, it's super simple to use it in other themes as well. Here are the steps:

1. First, copy over the files `generate-options.php`, `admin-helper.php`, `ajax-image.php` and the folders `css`, `img` and `scripts` with no change.

2. Add an **options** directory with as many options files as need be. The structure of the options file should be like this:

```
$info=array(  
    'name' => '', //The options page identifier  
    'pagename' => '', //The page slug  
    'title' => '', //The Page title  
    'sublevel' => 'yes' //Only included if a submenu page  
);  
$options = array(  
    //options  
);  
$optionspage = new rockable_options_page($info,   
    $options);
```

3. Include all these files from **include-options.php**.
4. From **functions.php**, include all the PHP files mentioned above. Make sure the following constants are defined:  
**ROCKABLE\_THEME\_DOCS, ROCKABLE\_THEME\_LOGO,  
ROCKABLE\_MAINMENU\_NAME, ROCKABLE\_THEME\_VERSION.**
5. Also add this code to **functions.php** for adding the CSS and JS files:

```
add_action('admin_head', 'rockable_admin_head');
```

That's it! By just changing the files in the options directory, we can have entirely new options pages! Awesome, isn't it?

Next up: creating a submenu admin page for managing the slider, and coding the actual slider.

## The Home Page Slider Options

To create the slider manager, we will write another class. Rather than go through the entire class once again, just copy over the file `functions/generate-slider.php` from the complete theme into the folder you're working with. The slider uses default WordPress CSS classes and IDs for styling.

I'll explain some of the code there. As usual, the constructor assigns variables and sets up the `admin_menu` hook. Then, `rockable_add_menu()` adds the submenu page. Following this, `rockable_generate_page()` generates the page HTML. Simple logic is followed:

- First, all the fields (image URL, description, title, etc.) are fetched. Each field is stored as an array in the database.
- Then, the number of elements in the `$image_set`, which holds the images, is counted. If it is equal to 0, then no changes have been saved. In this case, one row of fields is displayed.
- If the `$image_set` has one or more images stored, we loop through the arrays and display the already saved data. Note the structure of the input elements:

```
<input type="text" name="<?php echo $this->slider_name; ▶  
?>[image][]" />
```

You can see that the name of the input element is like a PHP array. This allows PHP to handle it better — when the data is `POST`ed, it can be processed by our PHP code directly as an array. That way, we don't need to keep track of the number of images by appending digits or any such complex code.

- We also display a set of controls at the side, for adding or removing more slider rows. This will be done by the JavaScript, which I explain below.
- Finally, the `save_options()` function handles saving. We just get the `POST`ed arrays and update the options. The naming of elements as arrays helps a lot.

## The JavaScript Code

Open up the file `functions/scripts/rotator.js` and take a look at the code there.

- First, we paste in the HTML of a single row from the slider page and assign it to a variable.
- We attach click events to the 'Add Above' And 'Add Below' buttons. Every time one of them is clicked, the contents of the JavaScript variable above are appended or prepended to the current row.
- We attach another event to the delete button, which removes the current row with a little fancy animation.
- Finally, whenever the 'Add New' button at the top or bottom is clicked, a new row is inserted at the bottom.

## Coding the Slider

Take a look at the code in the file `includes/homepage-slider.php`. In our theme options, we offered the user two options: the slider could be filled with either recent posts from the Portfolio custom posts type, or from the slider manager. The process followed is:

- First, use `get_option()` to get the user's selected option. Note that I used the option call like this:

```
get_option('rockable_slider_source', 'Latest Posts')
```

The function has two parameters — the name of the option and the default value if the option doesn't exist. That way, even if the user hasn't selected any options, we still have a value to work with.

- If the user has chosen **Latest Posts**, we grab the latest portfolio posts using the `WP_Query` class. We specify two arguments — the post type (`portfolio`) and the number of posts as specified by the user. We then loop through the posts and display the HTML required. The usual template tags are used, like `the_title()` and `the_permalink()`. We use our custom `rockable_excerpt()` function with a length of 25 words.
- If the user has chosen to use the slider manager instead, we use `get_option()` and get the images, links, descriptions, and titles. We loop through these arrays and display the required HTML.

Now, in `functions.php`, also include this file in the `if(is_admin())` check above `include-options.php`:

```
require_once (ROCKABLE_FUNCTIONS . 'generate-slider.php');
```

Then, add this line of code to `include-options.php`:

```
include('options/slider-settings.php');
```

Now go and refresh your page. A new link should show up under the *WordThemer* menu, called *WordThemer Rotator*. The page looks as shown in Figure 2-2.

And with that, our theme options part comes to an end. You can add options by simply adding a new array element in the options pages, and add new options pages by creating a file similar to

**WordThemer Rotator**

[Add new item](#)

Settings	Description	Controls
Image URL (*): <input type="text"/> Link: <input type="text"/> Title: <input type="text"/>		<a href="#">Add above</a> <a href="#">Add below</a> <a href="#">Delete item</a>

[Save changes](#)

Fig. 2-2. the WordThemer Rotator output.

the files in the folder **functions/options/** and including it via **include-options.php**. To fetch options, use the function like this:

```
$variable = get_option($option_id, $default_value);
```

The **\$default\_value**, if not specified, defaults to the Boolean **false**. Examples:

```
$var = get_option('rockable_logo'); //Gets the rockable
logo option from the database. If not present, returns
false
$var = get_option('rockable_logo', 'logo.jpg'); //Gets the
rockable logo option from the database. if not present,
returns logo.jpg
```

Note: WordPress encodes slashes, quotes, etc. So if you enter the string “**It's raining**” for an option, it will be stored as **It\'s raining** — a slash is added. To remove these slashes, be sure to use **stripslashes**:

```
$var = stripslashes(get_option('rockable_logo'));
```

That will get rid of any errant slashes. Next chapter: Creating and working with custom posts types and custom taxonomies.

3

# Custom Post Types and Taxonomies

In WordPress, there are five types of posts built in:

1. Posts
2. Pages
3. Revisions (drafts and modifications of posts and pages)
4. Attachments (images, videos, etc.)
5. Nav Menus

Sometimes, these post types don't suffice. Imagine you were running an e-commerce site, and you wanted to add a product. What would you prefer — creating a new post, adding it to the portfolio category, adding a couple of custom fields, choose a post template at the side **or** just creating a new *Products* post? The second one not only involves fewer steps, it's more logical and allows us to customize the page display for the post type.

*Custom Post Types* are added using the function `register_post_type()`. The function is used like this:

```
<?php register_post_type( $post_type, $args ) ?>
```

`$post_type` is the name of the post type, e.g. *portfolio*. It has a maximum length of 20 characters.

`$args` is an array of arguments. These are explained below:

- **label:** The name of the post type in plural. Defaults to `$post_type`.



- **labels:** An array of labels for the post type:
  - **name:** Same as the label attribute above.
  - **singular\_name:** Name for a single post of this post type.
  - **add\_new:** The text replacing *Add New*.
  - **add\_new\_item:** The text replacing *Add New Item*.
  - **edit\_item:** The text replacing *Edit Post/Edit Page*.
  - **new\_item:** The text replacing *New Post/New Page*.
  - **view\_item:** The text replacing *View Post/View Page*.
  - **search\_items:** The text replacing *Search Posts/Search Pages*.
  - **not\_found:** The text replacing *No posts found/No pages found*.
  - **not\_found\_in\_trash:** The text replacing *No posts found in Trash/No pages found in Trash*.
  - **parent\_item\_colon:** The text replacing *Parent Page* in hierarchical post types.
- **description:** A description of the post type.
- **public:** Whether the post type has a shown user interface, is publicly queryable, included in searches and shows up in navigation menus, or not.
- **publicly\_queryable:** Whether the **post\_type** queries can be performed from the front end.
- **exclude\_from\_search:** Whether the post is excluded from search or not.
- **show\_ui:** Whether to generate a default UI for this.
- **capability\_type:** The post type to use for checking read, edit, and delete capabilities. Defaults to **post**.

- **capabilities:** An array of the capabilities for this post type.
- **hierarchical:** Whether or not post parents can be set (e.g. *Pages*) or not (e.g. *Posts*).
- **supports:** The features supported for the post type, e.g. *title*, *editor*, *excerpt*, *comments*, etc.
- **register\_meta\_box\_cb:** A function to add meta boxes if required.
- **taxonomies:** A list of registered taxonomies to be used with this post, e.g. *Categories*, *Tags*, etc.
- **menu\_position:** Position in the menu for this post type. Defaults to below *Comments*. Otherwise:
  - 5: below *Posts*.
  - 10: below *Media*.
  - 20: below *Pages*.
  - **menu\_icon:** The icon to be used for the post type.
- **rewrite:** Rewrite permalinks with this format.
- **query\_var:** Name of the query variable to use for this post type.
- **can\_export:** Whether or not this post type can be exported via the WordPress Exporter.
- **show\_in\_nav\_menus:** Whether the post type is available to add to the nav menu.
- **\_builtin, \_edit\_link:** If the post type is built-in, and the edit link for posts. **These should always be false for custom post types created by us.**

The process for creating a custom post type is:

1. Hook into `init` with a function.
2. Use the above mentioned function to create the post type.

Add this code to `functions/register-wp3.php`:

```
add_action('init', 'rockable_create_post_types');
function rockable_create_post_types(){
    $labels = array(
        'name' => __( 'Portfolio' ),
        'singular_name' => __( 'Portfolio' )
    );
    $args = array(
        'labels' => $labels,
        'label' => __( 'Portfolio' ),
        'singular_label' => __( 'Portfolio' ),
        'public' => true,
        'show_ui' => true,
        '_builtin' => false,
        'capability_type' => 'post',
        'hierarchical' => false,
        'rewrite' => false,
        'supports' => array('title','editor','excerpt',
        'revisions','thumbnail'),
        'taxonomies' => array('portfolio_cat', 'post_tag'),
        'menu_icon' => get_bloginfo('template_directory').
        '/functions/img/icon.png'
    );
    if (function_exists('register_post_type'))
        register_post_type('portfolio', $args);
    endif;
}
```

As you can see, I've named our *portfolio* custom post type with the singular name *Portfolio*. It is a public post type which is not

hierarchical, is not rewritten (more on this later), and supports the title, editor, excerpt, revisions, and thumbnail editing features. In addition, it supports two taxonomies: *portfolio\_cat* (which is a custom taxonomy we will create soon) and *post\_tag* (tags).

Now go over to your WordPress Dashboard. A new post type will have appeared:

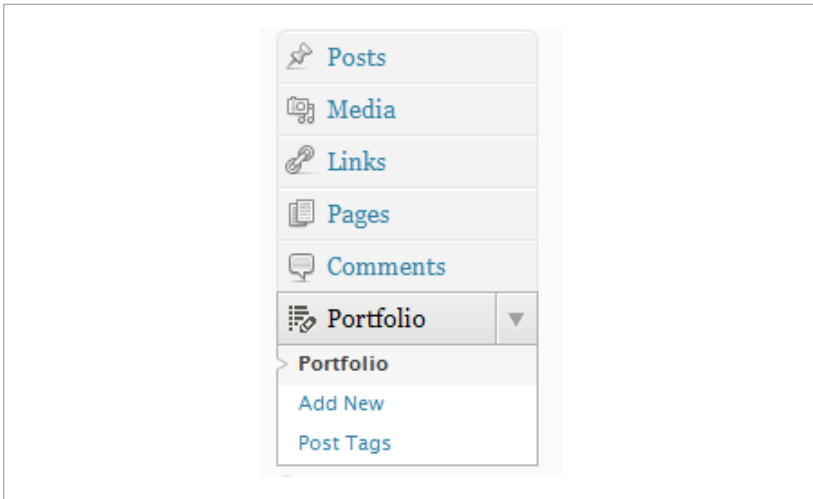


Fig. 3-1. The new Portfolio custom post type.

Next, we should create our custom taxonomy. A *taxonomy* is a way of classifying things. By default, WordPress has two default taxonomies built in — *Categories* and *Tags*. Sometimes, we need to build our own custom taxonomies which are more intuitive and logical, similar to post types.

Custom taxonomies are added using the function `register_taxonomy()`. The function is used like this:

```
<?php register_taxonomy($taxonomy, $object_type, $args); ?>
```

**\$taxonomy** is the name of the taxonomy. **\$object\_type** is an array of the object types for the custom taxonomy. **\$args** is a list of arguments.

Here are the arguments in **\$args**:

- **label**: The name of the post type in plural.
- **labels**: An array of labels for the post type:
  - **name**: Same as the label attribute above.
  - **singular\_name**: Name for a single post of this post type.
  - **search\_items**: The search items text.
  - **popular\_items**: the popular items text. Default is `__(Popular Tags')` or null.
  - **all\_items**: The all items text.
  - **parent\_item**: The parent item text. This string is not used on non-hierarchical taxonomies such as post tags.
  - **parent\_item\_colon**: The same as **parent\_item**, but with a colon at the end.
  - **edit\_item**: The edit item text.
  - **update\_item**: The update item text.
  - **add\_new\_item**: The add new item text.
  - **new\_item\_name**: The new item name text.
  - **separate\_items\_with\_commas**: The separate item with commas text used in the taxonomy meta box. This string isn't used on hierarchical taxonomies.
  - **add\_or\_remove\_items**: The add or remove items text and used in the meta box when JavaScript is disabled. This string isn't used on hierarchical taxonomies.

- **choose\_from\_most\_used:** The choose from most used text used in the taxonomy meta box. This string isn't used on hierarchical taxonomies.
- **public:** Whether the taxonomy has a shown user interface.
- **show\_ui:** Whether to generate a default UI for managing this taxonomy.
- **show\_tagcloud:** Whether to show a tag cloud of this taxonomy in the admin.
- **hierarchical:** Whether or not the post type can have descendants, e.g. *Categories* are hierarchical, *Tags* are not.
- **rewrite:** Used for customizing the **query\_var** and front base.
- **query\_var:** To change the **query\_var**.
- **capabilities:** The capabilities for this post type.
- **\_builtin:** If the taxonomy is built-in. This should always be false for custom taxonomies created by us.

The process for creating custom taxonomies is the same as for custom post types — hook into **init** and use a function to create it. Put this code into **register-wp3.php**:

```
add_action('init', 'rockable_taxonomies', 0);

function rockable_taxonomies(){
    $labels = array(
        'name' => _x('Portfolio Categories', 'taxonomy general name', 'rockable'),
        'singular_name' => _x('Portfolio Category', 'taxonomy singular name', 'rockable'),
        'search_items' => __('Search Portfolio', 'rockable'),
```

```

        'all_items' => __('All Portfolio Categories',
        'rockable'),
        'parent_item' => __('Parent Portfolio Category',
        'rockable'),
        'parent_item_colon' => __('Parent Portfolio Category:',
        'rockable'),
        'edit_item' => __('Edit Portfolio Category',
        'rockable'),
        'update_item' => __('Update Portfolio Category',
        'rockable'),
        'add_new_item' => __('Add New Portfolio Category',
        'rockable'),
        'new_item_name' => __('New Portfolio Category Name',
        'rockable')
    );

    register_taxonomy('portfolio_cat', array('portfolio'),
        array(
            'hierarchical' => true,
            'labels' => $labels,
            'show_ui' => true,
            'query_var' => true,
            'rewrite' => array('slug' => 'portfolio_categories')
        ));
}

```

We create a taxonomy with the name **portfolio\_cat**, usable in the portfolio custom post types, which is hierarchical and has a slug **portfolio\_categories**.

Notice the `__x()` function we've used. This is similar to `__()` except that it also adds a context argument as the second parameter. This specifies where the translatable string is being used.

Now, if you take a look at the Dashboard again, you will see a new menu item under *Portfolio: Portfolio Categories*.

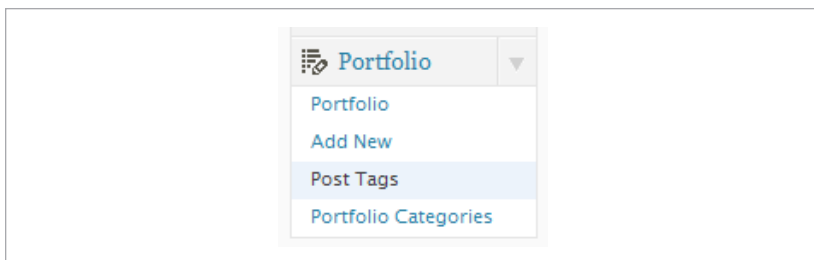


Fig. 3-2. The Portfolio Categories custom taxonomy shows up.

Now, if you create a portfolio post and take a look at the UI for it, you will notice something:

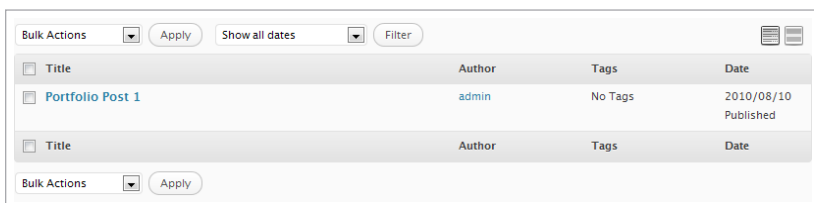


Fig. 3-3. The Portfolio post type UI.

Those columns, *Title*, *Author*, *Tags* and *Date* aren't really that useful on their own. Shouldn't we be able to see the *Portfolio Categories* too? Well, managing the columns there is pretty simple! We simply add one filter and one function. Add this code to **register-wp3.php**:

```
add_filter("manage_edit-portfolio_columns",
    "rockable_portfolio_columns");
add_action("manage_posts_custom_column",
    "rockable_portfolio_custom_columns");
```

Now you need to write the two functions mentioned there: **rockable\_portfolio\_columns()** and **rockable\_portfolio\_custom\_columns()**.

Here's the code for the functions:



```
function rockable_portfolio_columns($columns){
    $columns = array(
        "cb" => "<input type=\"checkbox\" />",
        "title" => _x("Portfolio Title", "portfolio title column", 'rockable'),
        "author" => _x("Author", "portfolio author column", 'rockable'),
        "portfolio_cats" => _x("Portfolio Categories", "portfolio categories column", 'rockable'),
        "date" => _x("Date", "portfolio date column", 'rockable')
    );
    return $columns;
}
```

The columns are in an array. They are:

- **cb:** The checkboxes at the side. Don't ever omit this.
- **title:** The post title.
- **author:** The post author.
- **portfolio\_cats:** The Portfolio Categories.
- **date:** the date of publishing.

Next, the function which actually **displays** these columns:

```
function rockable_portfolio_custom_columns($column){
    global $post;
    switch ($column)
    {
        case "author":
            the_author();
            break;
```

```

        case "portfolio_cats":
            echo get_the_term_list( $post->ID, 'portfolio_cat', '
                ', ' ', ' ', ' ' );
            break;
        }
    }
}

```

We do a **switch-case** through the columns. **cb**, **title** and **date** are automatically handled by WordPress. In the case of **author**, we use the template tag **the\_author()**. For **portfolio\_cats**, we display a list of the terms using **get\_the\_term\_list()**. The function is used like this:

```

<?php get_the_term_list( $id, $taxonomy, $before, $sep,
    $after ) ?>

```

The parameters are:

- **id**: the ID of the post for which taxonomies are displayed.
- **taxonomy**: The registered name of the taxonomy (**portfolio\_cats** in our case).
- **before**: The leading text before each taxonomy name.
- **sep**: The separator between taxonomy names.
- **after**: The trailing text after each taxonomy name.

Using our code, we separate the categories by a comma. Take a look at the Dashboard now.

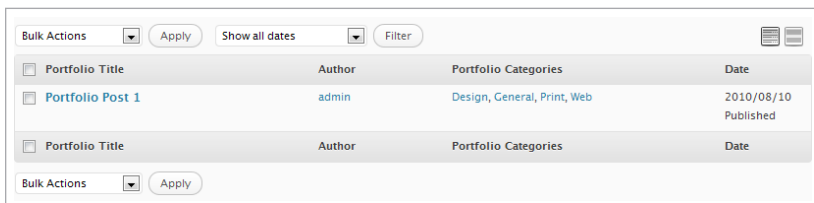


Fig. 3-4. The new columns for the Portfolio post type.

Our columns have been customized as we want! To add new columns, you can just add them to the array in `rockable_portfolio_columns()` and then a display case in `rockable_portfolio_custom_columns()`.

The next step is to make templates for the display of these custom post types and taxonomies. The new templates needed are:

1. A template to display all *Portfolio* posts, with pagination — a custom page template, `template_portfolio.php`.
2. A template for a single *Portfolio* post — `single-portfolio.php`.
3. An archive template for the *Portfolio Categories* template — `taxonomy-portfolio_cat.php`.

The custom page template (#1) can be named anything, but #2 and #3 follow a prefixed naming convention.

- The single post template for any post type must be named `single-{post-type}.php`. So we name it `single-portfolio.php`.
- The archive template for a custom taxonomy must be named `taxonomy-{taxonomy-name}.php`. So we call it `taxonomy-portfolio_cat.php`.

For more on naming conventions, see here: [http://codex.wordpress.org/Template\\_Hierarchy](http://codex.wordpress.org/Template_Hierarchy).

## template\_portfolio.php

To make the portfolio template, we need to make a Page Template similar to the `index.php` file. Make a copy of `index.php` and rename it to `template_portfolio.php`. Now, we need to identify

this as a page template. Add this code in between the opening `<?php` tag and the `get_header()` call:

```
/*
Template Name: Portfolio
*/
```

Next, we need to change the query. We now need to get only the posts in the portfolio post type. Find this piece of code:

```
<?php
global $query_string; //We now have access to the original WordPress Query
$exclude = rockable_build_cat_exclude(); //Build the list of categories to exclude
if ($exclude)
    $exclude = '&cat=' . $exclude;

$posts = query_posts($query_string . $exclude); //Tell WordPress to exclude some categories, if required.
if (have_posts()) : while(have_posts()) : the_post(); ?>
    <?php include (ROCKABLE_INCLUDES . 'post-excerpt.php'); ?>

<?php endwhile; ?>
```

Remove it and replace it with this:

```
<?php
global $paged;
$paged = (get_query_var('paged')) ? get_query_var('paged') : 1; //For pagination
$posts = query_posts('post_type=portfolio&paged='.$paged); //Make sure we let WordPress know we need posts ONLY from the portfolio post type
if (have_posts()) : while(have_posts()) : the_post();
    $image_url = rockable_post_image(); //Use the function
```

```

        to fetch the portfolio image
    if ($image_url)
        $image_url = rockable_build_image($image_url, 160, 100);
    ?>

    <div class="entry">
        <div class="title">
            <h2><a href="<?php the_permalink(); ?>">
                <?php the_title(); ?></a></h2>
            <?php include(ROCKABLE_INCLUDES . 'post-meta.
                php'); ?>
        </div>
        <?php if ($image_url): ?>
            <a href="<?php the_permalink(); ?>">
                </a>
            <?php endif; ?>
            <?php the_excerpt(); ?>
        </div>
    <?php endwhile; ?>

```

Explanation:

- First, we get the **paged** query variable. This refers to the page number of the current page. For example, if we have eight portfolio posts and choose to display three posts per page, then we will have three pages of posts. Specifying the **paged** value will show the correct posts. If **paged** is not set in the query, we default it to one.
- Then, we set up a new query getting only the posts from the *portfolio* post type.
- The rest is pretty straightforward: we loop through the posts and display the HTML. A 160 × 100 pixel thumbnail is also displayed.

**Note:** Remember when we created the custom post type, I didn't enable rewrites? This line:

```
'rewrite' => false,
```

What happens when we set rewrite to **false** is, the URL changes. If you have the permalinks to your posts set up as `/%postname%/`, then your portfolio posts would display like this:

```
http://domain.com/portfolio/lorem-ipsum
```

However, now that we have disabled rewrites, it will be like this:

```
http://domain.com/?portfolio=lorem-ipsum
```

This may be a little SEO unfriendly, but it's the only way to allow pagination in the portfolio posts templates that you create. If you **did** enable rewrites, you would have to display **every single portfolio post on one page**. The decision is up to you, which path you wish to take. I have a feeling this is just temporary though. Future versions of WordPress are likely to have a way to display custom post types in a page like the `index.php` template without such hacks.

Anyway, this is the final code for the file. To create a new portfolio page, just create a new WordPress Page and assign the *Portfolio* page template to it.

## single-portfolio.php

In our case, the single post templates for blog posts and portfolio posts are more or less identical. So, make a copy of `single.php` and rename it to `single-portfolio.php`. Remove the line that reads:

```
<?php comments_template(); ?>
```

That will disable comments on our single portfolio pages. Of course, if you wish to retain comments, you may leave it in. The other thing that is different between the two is the post meta data. Blog posts show the categories they are posted in, and portfolio posts the Portfolio categories. Take a look at `includes/post-meta.php`. There is a piece of code that reads:

```
<?php if (get_post_type( $post ) == 'portfolio'): ?>
| <?php printf( __( 'Posted by %1$s in %2$s', 'rockable' ), >
get_the_author(), get_the_term_list( $post->ID,
| 'portfolio_cat', '', ' ', ' ' ) ); ?>
<?php else: ?>
| <a href="<?php comments_link(); ?>"><?php comments_ >
number(__( 'No Comments', 'rockable' ), __( '1 Comment', >
| 'rockable' ), __( '% Comments', 'rockable' ) ); ?></a>
| <?php printf( __( 'Posted by %1$s in %2$s', 'rockable' ), >
get_the_author(), get_the_category_list( ' ', ' ' ) ); ?>
<?php endif; ?>
```

The function `get_post_type($post)` gets the post type of the current post. `$post` refers to the current post object. This function returns `post` for blog posts, `page` for pages, and importantly for us, `portfolio` for portfolio posts. Also, the number of comments isn't displayed for portfolio posts since we haven't enabled comments. You can show that if you wish, though.

## taxonomy-portfolio\_cat.php

This is basically a file for the archives of the `portfolio_cat` taxonomy (*Portfolio Categories*). In this theme, there is going to be no difference between the `archive.php` and `taxonomy-portfolio_cat.php` files. Just duplicate `archive.php` and rename it. The only reason I showed you what to do is for understanding. If you want to make layout changes then you need to have a different file. This is similar to the default WordPress functionality — to display a

category archive, it first searches for `category.php`. If not found, it defaults to `archive.php`. Similarly, for *Portfolio Categories*, first `taxonomy-portfolio_cat.php` is searched for. If not found, `archive.php` is used.

That brings this chapter to an end. Next up: creating and using meta boxes.



4

# Meta Boxes

Meta boxes are an easy way to manage custom fields. So what are custom fields anyway? From [the Codex](#):

*WordPress has the ability to allow post authors to assign custom fields to a post. This arbitrary extra information is known as meta-data. Meta-data is handled with key/value pairs. The key is the name of the meta-data element. The value is the information that will appear in the meta-data list on each individual post that the information is associated with.*

This is what the custom field management looks like in WordPress:

The screenshot shows the 'Custom Fields' management interface in WordPress. It features a table with two columns: 'Name' and 'Value'. The first row contains the text 'custom\_field\_key' in the 'Name' column and 'custom\_field\_value' in the 'Value' column. Below the 'Name' column, there are 'Delete' and 'Update' buttons. Below the table, there is a section titled 'Add New Custom Field:' which includes a dropdown menu with the text '— Select —', a link 'Enter new', and an 'Add Custom Field' button. At the bottom, a note states: 'Custom fields can be used to add extra metadata to a post that you can [use in your theme](#).'

Fig. 4-1. Custom fields in WordPress.

Custom fields can be pretty useful for theme developers. For example, if you wanted some extra information about the post like a post thumbnail, or if you wanted to include the location from where the post was written, custom fields can be handy. To retrieve a custom field, you use the function `get_post_meta()`:

```
get_post_meta($post_id, $key, $single);
```

The parameters are:

- **post\_id:** The ID of the post from which you want to retrieve the meta data.
- **key:** The name of the custom field you want to retrieve.
- **single:** Whether or not to return as a single element (use **true**) or an array (use **false**).

Despite being extremely useful, custom fields do have their drawbacks — they are difficult to use, for example. Initially, when creating new custom fields, the user has to remember the name of the custom field. In addition, it isn't always clear what a custom field does. For example, if I told you to add a custom field with the name `rockable_p_i`, you wouldn't know what I meant. Developers use names that suit them, not the user.

To make things easier, we create a meta box, which allows users to directly enter the custom fields with a more intuitive interface. Figure 4-2 shows what a meta box might look like.

The meta box automatically updates or deletes the custom fields when the post is saved. This is much more user-friendly and easy to use. To add a meta box, we need to follow these steps:

1. First, hook into the `admin_init` with a function to add a meta box.

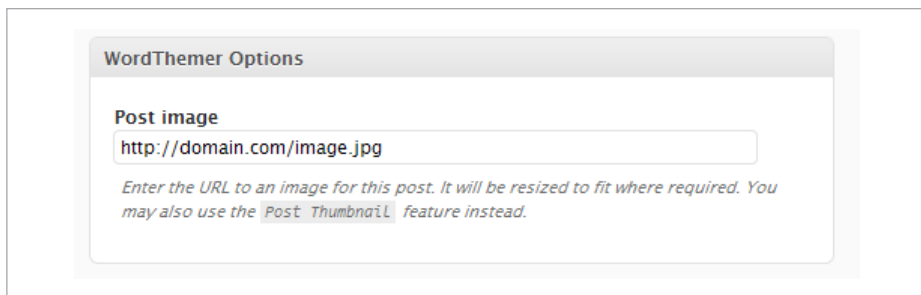


Fig. 4-2. A sample meta box.

2. Use the `add_meta_box()` function to add a meta box.
3. Display the meta box with a third function.
4. Also hook into the `save_post` to save the custom field values.

We will be using a class for this too, so that we can add multiple meta boxes.

## The Meta Box Options Files

Before we begin coding the class, it's a good idea to take a look at the meta box files. Copy over the `functions/meta-boxes/` folder from the completed theme into your `functions` folder. Take a look at the `meta-box.php` file. The structure is:

```
<?php
$info = array(
    'box_name' => 'rockable-meta-box',
    'title' => 'WordThemer Options',
    'location' => array('post', 'portfolio')
);
```

Here, the variables in `$info` are:

- **box\_name:** a unique identifier for the meta box.
- **title:** The title of the meta box.
- **location:** An array of the locations for the meta box: *post*, *page*, or a custom post type, e.g. *portfolio*.

Then we have the options:

```
$options = array(
    array(
        "type" => "text",
        "title" => "Post image",
        "name" => "rockable_post_image",
        "desc" => "Enter the URL to an image for this post.
        It will be resized to fit where required. You may
        also use the <code>Post Thumbnail</code> feature
        instead.",
        "default" => "" ),
    );
```

In this particular file there is just one option, but you can have as many as you like, similar to the theme options. The components of the options are also the same as the theme options. One thing to note though: only three input types have been coded: *text*, *textarea*, and *select*. These are the three used most frequently. It's easy to add more input types, however.

Finally, we have the object:

```
$metabox_post = new rockable_meta_box($info, $options);
```

This creates a new object of the **rockable\_meta\_box** class, generating a new meta box.

## Coding the Meta Box Class

Here's the basic structure of the class:

```
class rockable_meta_box{
    function rockable_meta_box($info, $options){
    }
    function rockable_add_metabox(){
    }
    function rockable_display_metabox(){
    }
    function rockable_save_metabox(){
    }
}
```

Create a new file named **generate-meta-box.php** in the **functions** folder, and paste in that class framework. To start, paste this above the constructor (the first function):

```
var $options; //Options to display for the meta box
var $box_title; //Title, displayed at the top of the meta box.
var $location; //Where to display the meta box - post, page, custom post type, etc.
var $box_name; //Unique name for the meta box
```

Those are the basic components of the meta box — the options in the meta box, the title at the top of the box, where to display the box, and a unique identifier for the box.

Next, here's the code for **rockable\_meta\_box()**:

```
function rockable_meta_box($info, $options){
    $this->info=$info;
    $this->box_title=$info['title'];
    $this->location=$info['location'];
```

```
$this->box_name = $info['box_name'];
$this->options = $options;
add_action('admin_init', array(&$this, 'rockable_add_
    metabox'));
add_action('save_post', array(&$this, 'rockable_save_
    metabox'));
}
```

Very straightforward: we assign the parameters to the variables and add two actions: one for adding the meta box and one for saving it. The structure is similar to that of the theme options class. Next, the `rockable_add_metabox()` function.

```
function rockable_add_metabox(){
    if ( function_exists('add_meta_box') && is_array(
        $this->location)):
        foreach($this->location as $loc):

            add_meta_box(
                $this->box_name, //ID
                $this->box_title, //Title
                array(&$this, 'rockable_display_metabox'), //Callback>
                    function to print HTML
                $loc, //Place to display
                'high' //Context
            );

        endforeach;
    endif;
}
```

In this function, we loop through each of the locations (*post*, *page*, etc.) and add a new meta box using the `add_meta_box()` function. The function works like this:

```
add_meta_box( $id, $title, $callback, $page, $context,
    $priority, $callback_args );
```

The arguments are:

- **id:** A unique identifier for this meta box.
- **title:** The title of the meta box.
- **callback:** The function that displays the HTML for the meta box.
- **context:** An array of locations to display the meta box — *post*, *page*, *link* or a custom post type.
- **priority:** The priority within the page where the boxes should show — *high* or *low*. A higher priority will show the meta box higher in the page.
- **callback\_args:** Any arguments to be passed to the callback function. The **\$post** object is passed by default.

Next, we need to fill in the `rockable_display_metabox()` function. Here's the code:

```
function rockable_display_metabox(){
    global $post;
    ?>
    <div class="form-wrap">
    <?php
        wp_nonce_field( plugin_basename( __FILE__ ), $this->
            box_name . '_wpnonce', false, true ); //Security field
            for verification

        foreach ($this->options as $value):
            $data = get_post_meta($post->ID, $value['name'], true);
            if (!$data) $data=$value['default'];
        ?>
```



```

<div class="form-field form-required">
  <label for="<?php echo $value['name']; ?>"><strong>
    <?php echo $value['title']; ?></strong></label>
  <?php
    switch($value['type']) {
      case "text": $this->display_meta_text($value, $data);
        break;
      case "textarea": $this->display_meta_textarea($value, $data); break;
      case "select": $this->display_meta_select($value, $data); break;
    }
  ?>
</div>
<?php
  endforeach;
?>
</div><!-- //form_wrap -->

<?php
}

```

Explanation of this function:

- First, we generate a nonce field for this meta box using **wp\_nonce\_field()**. A *nonce* is a cryptographic feature that allows verification of pages — basically, it makes sure that WordPress security is maintained.
- Next, we loop through the options and display the HTML for them. Default WordPress HTML is used so as to make the meta box fit in with the page layout. You can, of course, use custom HTML and style it too.

We also have a function to display each of the text, textarea and select inputs. They are described below:

```

function display_meta_text($meta_box, $data){
    ?>
        <input type="text" id="<?php echo $meta_box['name'];
    ?>" name="<?php echo $meta_box['name']; ?>"
        value="<?php echo esc_html($data); ?>" />
    <p>
        <?php echo $meta_box[ 'desc' ]; ?>
    </p>
<?php
}

function display_meta_textarea($meta_box, $data){
    ?>
        <textarea id="<?php echo $meta_box['name']; ?>"
            name="<?php echo $meta_box['name']; ?>"><?php echo
            stripslashes($data); ?></textarea>
        <p>
            <?php echo $meta_box[ 'desc' ]; ?>
        </p>
<?php
}

function display_meta_select($meta_box, $data){
    ?>
        <select name="<?php echo $meta_box['name']; ?>"
            id="<?php echo $meta_box['name']; ?>" >
    <?php
        foreach ($meta_box['options'] as $option):
            echo '<option>';

            if ( $data == $option) {
                echo ' selected="selected"';
            }

            echo '>'.$option.'</option>';

        endforeach;
    ?>

    </select>

```

```

<p>
    <?php echo $meta_box[ 'desc' ]; ?>
</p>
<?php
}

```

The logic involved in these functions is simple and is pretty much the same as in the theme options.

The final function we need to write is the **rockable\_save\_ > metabox()** function. Here's the code:

```

function rockable_save_metabox(){
    global $post;
    if ( !wp_verify_nonce( $_POST[ $this->box_name .
        'wpnonce' ], plugin_basename(__FILE__) ) )
        return $post->ID;
    if ( !current_user_can( 'edit_post', $post->ID ) )
        return $post->ID;

    foreach( $this->options as $meta_box ):

        $data = $_POST[ $meta_box['name'] ];
        if ( $data=="")
            delete_post_meta($post->ID, $meta_box['name'],
                $data);
        else
            update_post_meta($post->ID, $meta_box['name'],
                $data);

    endforeach;
}

```

What we're doing is:

- First we verify the nonce with `wp_verify_nonce()`. The parameter in the function is the same as the string with which we generated the nonce. If WordPress returns *false*, meaning an unverified or insecure nonce, we return the post ID.
- Next, we check if the current user is allowed to edit the post. If not, again we return the post ID.
- Finally, we loop through the options.
- If the option has not been **POSTed** or does not have a value, we use `delete_post_meta()` to delete the custom field value. The function is used like this:

```
<?php delete_post_meta($post_id, $key, $value); ?>
```

The parameters are:

- **post\_id:** The ID of the post for which the meta has to be deleted.
- **key:** The name of the meta field to be deleted.
- **value:** The previous value of the meta field. If there are multiple custom fields with the same name but different values, this helps distinguish between them.
- If the option has been **POSTed** and has a value, we use `update_post_meta()` to update it. The function is used like this:

```
update_post_meta($post_id, $meta_key, $meta_value, $prev_value);
```

The parameters are:

- **post\_id:** The ID of the post for which the meta has to be updated.
- **meta\_key:** The name of the meta field to be updated
- **meta\_value:** The value of the above mentioned meta field.
- **prev\_value:** The previous value of the meta field. If there are multiple custom fields with the same name but different values, this helps distinguish between them.

## Including the Files

To make all this code usable, we need to include the files. In `functions.php`, include these files in the `if(is_admin())` check above `include-options.php`:

```
require_once (ROCKABLE_FUNCTIONS . 'generate-meta-box.php');  
require_once (ROCKABLE_FUNCTIONS . 'include-meta-boxes.php');
```

Make sure that `include-meta-boxes.php` includes all the meta box files you want to include.

## Using the Class in Other Themes

Since we've created a class, it's helpful and easy to use it in other themes. Here are the steps:

1. First, copy over the file `generate-meta-box.php`.
2. Add a **meta-boxes** directory with as many meta box options files as need be. The structure of the options file should be like this:

```
$info = array(  
    'box_name' => '', //e.g. 'rockable-meta-box-1'
```

```
'title' => '', //e.g. 'WordThemer Options'
'location' => array('') //e.g. array('post', 'page', 'portfolio')
);
$options = array(
//options
);
$metabox = new rockable_meta_box($info, $options);
```

3. Include all these files from **include-meta-boxes.php**.
4. From **functions.php**, include all the PHP files mentioned above.

That's all! Just by editing the files in the meta-boxes directory, you can add different meta boxes to posts, pages, custom post types, or a combination of these.

That brings to an end of this chapter. Next up: learning to code a set of advanced custom widgets!

5

# Custom Widgets

Widgets are something you can use to customise your WordPress site without actually delving into coding. WordPress has a number of default Widgets, which are found at *Appearance* > *Widgets*:

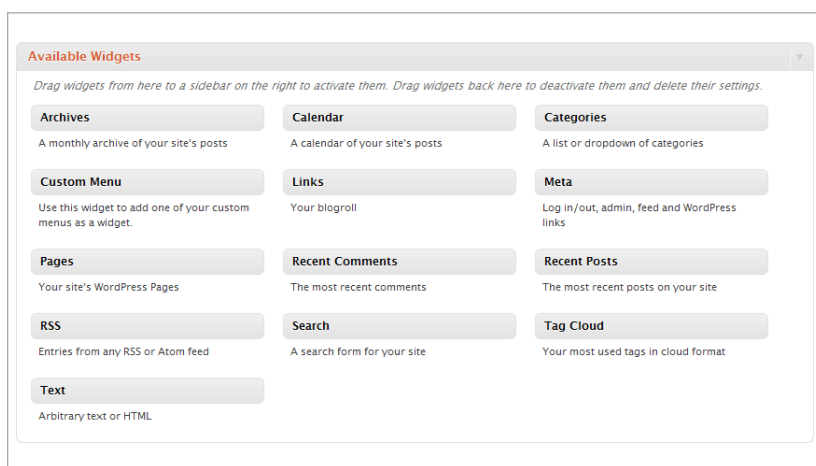


Fig. 5-1. The WordPress widgets panel.

To allow widgets to be used, you have to define *sidebars* in your theme. Sidebars are basically widget-ready locations, and widgets can be dragged and dropped to them.

## Making the Theme Widget-Ready

You have to register at least one widget area to make the theme widget ready. This is done by using the function `register_sidebar()`.

Create a new file named `register-wp3.php` in the `functions` folder. Add this code:



```

if (function_exists('register_sidebar')){
    register_sidebar(array(
        'name'          => 'Footer Column 1',
        'description'   => 'The first column in the footer',
        'before_widget' => '<li>',
        'after_widget'  => '</li>',
        'before_title'  => '<h2>',
        'after_title'   => '</h2>' )
    );
}

```

The parameters are self explanatory. The widget outputs code like this:

```

before_widget
before_title
<widget_title>
after_title

<widget_content>
after_widget

```

We just specify these variables as per the HTML we want generated.

Duplicate the above code twice more, changing the **names** to “Footer Column 2” and “Footer Column 3” respectively, and modifying the descriptions. Then, add this code:

```

if (function_exists('register_sidebar')){
    register_sidebar(array(
        'name'          => 'Sidebar',
        'description'   => 'Sidebar on all pages',
        'before_widget' => '',
        'after_widget'  => '',
        'before_title'  => '<h2>',

```

```
'after_title' => '</h2>' )  
);  
}
```

Now include the `register-widgets.php` file from `functions.php`. Add the code below the `require` in `register-wp3.php`:

```
//Register widget areas:  
require_once (ROCKABLE_FUNCTIONS . 'register-widgets.php');
```

Go to the *Widgets* subpanel and take a look. Your widget areas should show up:

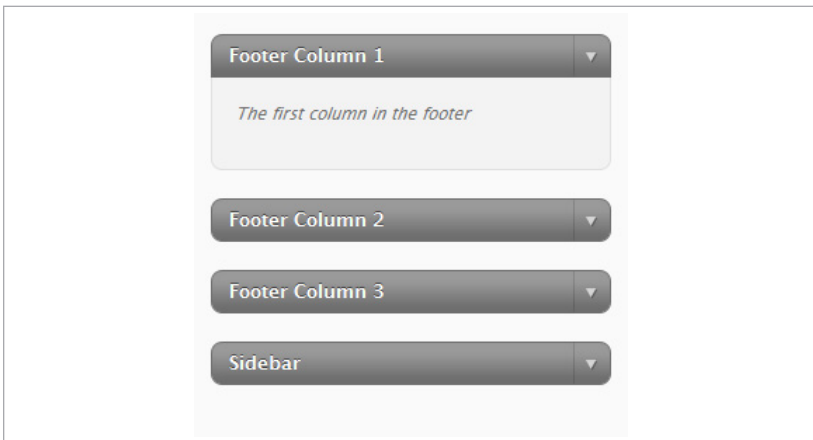


Fig. 5-2. The registered widget areas.

You are, of course, free to register as many more widget areas as you want. For example, you could define widget areas for only posts, only pages, search results pages, etc.

To show the widgets of a particular sidebar, you just have to call the `dynamic_sidebar()` function. For example:

```
<div id="sidebar">
  <?php
    if (function_exists('dynamic_sidebar') &&
        dynamic_sidebar('Sidebar')) :
    endif;
  ?>
</div>
```

The name of the registered sidebar should be in the `dynamic_ sidebar()` function call. The file `sidebar.php` can be included using the `get_sidebar()` function. To include other files that have sidebar code, just use `require` or `include`. In this theme, the files `sidebar.php` and `includes/widget-columns.php` have sidebar code.

## Custom Widgets

Sometimes, the default WordPress widgets just aren't enough. Say you wanted a Twitter widget, for example — WordPress doesn't have this built in, but they have an API to make building widgets extremely easy. To create a new widget, you have to make a class that extends the class `WP_Widget`. The skeleton code for all custom widgets should be this:

```
class Widget_Name extends WP_Widget {
    function Widget_Name() {
    }
    function widget($args, $instance) {
    }
    function form($instance) {
    }
    function update($new_instance, $old_instance) {
    }
}
```

This is the starting point.

- **Widget\_Name** is the name of the class for your widget, e.g. **Rockable\_Latest**.
- You have a function with the same name as the class. Here you give some basic settings for the widget like a description.
- The function **widget()** outputs the widget content.
- The function **form()** displays the form which shows on the *Widgets* management panel.
- The function **update()** updates the widget options when saved.

This theme comes with four custom widgets:

- A *Latest Posts* widget, which displays post excerpts as well.
- A *Latest Portfolio Item* widget, which displays, well, your latest portfolio item.
- A *Contact Form* widget.
- A tabbed widget which shows the most recent and most popular posts and recent comments.

Let's start with the *Latest Posts* widget. Create a new file named **latest-posts.php** in the **widgets** folder. Paste in this skeleton code:

```
class Rockable_Latest_Posts extends WP_Widget {  
    function Rockable_Latest_Posts() {  
    }  
    function form($instance) {  
    }  
}
```

```
function widget($args, $instance) {
}

function update($new_instance, $old_instance) {
}

}
```

Let's start with the constructor:

```
function Rockable_Latest_Posts(){
    $widget_ops = array( 'classname' => 'latest_posts',
        'description' => 'Show recent/popular/random posts
        from your blog, with excerpts.' );
    $this->WP_Widget( 'rockable_latest', 'Rockable Posts',
        $widget_ops );
}
```

The **\$widget\_ops** is an array with the widget options. **classname** is the CSS class applied to the widget. The **description** shows up on the widgets page.

Then, we add a new widget with the unique ID **rockable\_latest** and the title **Rockable Posts**.

Next we write the **form()** function. Before writing it, we need to decide what settings we need to give the user control over. I have decided on the widget title, the categories, sorting order (popular/recent/random) and number of posts. So here's the code:

```
function form($instance) {

    $instance = wp_parse_args( (array) $instance,
        array('number' => 2 ) );

    $title = esc_attr($instance['title']);
    $number = absint( $instance['number'] );
    $sort_by = $instance['sort_by'];
```

```

        $categories = (array) $instance['categories'];

    ?>
    <p>
        <label for="<?php echo $this->get_field_id('title'); ?>"
        ?>">
            Title:
        </label>
        <input class="widefat" id="<?php echo $this->
            get_field_id('title'); ?>" name="<?php echo $this->
            get_field_name('title'); ?>" type="text" value="
            <?php echo $title; ?>" />
    </p>
    <p>
        <label> Select categories: </label>
        <br />
        <?php
            $all_categories = get_categories('hide_empty=0
            &orderby=name');
            foreach ($all_categories as $cat ):
                $cat_id = intval($cat->cat_ID);
                $cat_name = $cat->cat_name;
                $selected = '';
                if (in_array($cat_id, $categories))
                    $selected = ' checked="checked"';

    ?>
        <input value="<?php echo $cat_id; ?>" class="checkbox"
        type="checkbox"<?php echo $selected; ?> id="<?php
        echo $this->get_field_id('categories'); echo $cat_id;
        ?>" name="<?php echo $this->get_field_name(
        'categories'); ?>[]" /> <label for="<?php echo
        $this->get_field_id('categories'); echo $cat_id;
        ?>"><?php echo $cat_name; ?></label> <br />
    <?php
endforeach;

```

```

        ?>
    </p>
    <p>
        <label for="php echo $this-&gt;get_field_id('sort_by'); ?&gt;"&gt; Sort them by: &lt;/label&gt;
        &lt;select name="<?php echo $this-&gt;get_field_name(
            'sort_by'); ?&gt;" id="<?php echo $this-&gt;get_field_id(
            'sort_by'); ?&gt;" class="widefat"&gt;
            &lt;option value="popular"&lt;?php selected(
                $instance['sort_by'], 'popular' ); ?&gt;&gt;Most Popular
            &lt;/option&gt;
            &lt;option value="recent"&lt;?php selected(
                $instance['sort_by'], 'recent' ); ?&gt;&gt;Most Recent
            &lt;/option&gt;
            &lt;option value="random"&lt;?php selected(
                $instance['sort_by'], 'random' ); ?&gt;&gt;Random&lt;/option&gt;
        &lt;/select&gt;
    &lt;/p&gt;
    &lt;p&gt;
        &lt;label for="<?php echo $this-&gt;get_field_id('number'); ?&gt;"&gt;Number of posts to get:&lt;/label&gt;
        &lt;input class="widefat" id="<?php echo $this-&gt;
            get_field_id('number'); ?&gt;" name="<?php echo $this-&gt;
            get_field_name('number'); ?&gt;" type="text" value=
            "&lt;?php echo $number; ?&gt;" /&gt;
    &lt;/p&gt;
&lt;?php
}
</pre

```

We use `wp_parse_args()` to add a default for the number of posts. Then, we display the inputs as in a regular HTML form. The difference is, we don't use the regular name or input field values.

- Instead of `name='title'`, we use `name="php echo $this-&gt;get_field_name('title'); ?&gt;"</code`

The function `get_field_name()` gets the *name* for this particular instance of the widget (since the widget can be used multiple times).

- Instead of `id='title'`, we use `name="<?php echo $this->get_field_id('title'); ?>"` ▶

The function `get_field_ID()` gets the *ID* for this particular instance of the widget (since the widget can be used multiple times).

- To check if a dropdown option is selected by default, we use `<?php selected( $instance['sort_by'], 'popular' ); ?>`. The function is used like this: ▶

```
<?php selected(current value, this option's value); ?>
```

As you can see, `$instance['sort_by']` will give the current value.

To display the categories, we use a loop similar to the one in the theme options. We loop through all the categories, and if it is in the selected categories array, we add a `selected` attribute to the checkbox.

Next, the function `widget()`. First, we get the widget settings and set up a custom query:

```
extract($args);

$title = apply_filters('widget_title', $instance['title']);
if ( empty($title) ) $title = false;
$number = absint( $instance['number'] );
$sort_by = $instance['sort_by'];
$categories = (array) $instance['categories'];
echo $before_widget;
if ($title):
    echo $before_title;
```



```
        echo $title;
        echo $after_title;
    endif;

    $args=array();
    //Number
    $args['posts_per_page'] = $number;
    //Categories
    $args['category__in'] = $categories;

    //Order by
    if ($sort_by == "popular"):
        $args['orderby'] = "comment_count";
    elseif ($sort_by == "random"):
        $args['orderby'] = "rand";
    else:
        $args['orderby'] = "date";
    endif;
```

By default, the `widget()` function is passed an array `$instance` which contains the settings for the current widget instance. So to get a widget option named `title`, just use `$instance['title']`.

Then, look at this code:

```
echo $before_widget;
if ($title):
    echo $before_title;
    echo $title;
    echo $after_title;
endif;
```

This needs to be part of every widget. It helps keep the code proper. Remember that when widget areas are registered, the `before_widget`, `before_title`, `after_title` and `after_widget` variables are specified. We need to display these.

Next, we set up the query arguments. The number of posts is set by `posts_per_page`, the categories by `category__in` and the sorting by `orderby`. For the `orderby` parameter, we use the values `comment_count` for popularity, `rand` for random and `date` for recent posts. Read more about these query parameters here: [http://codex.wordpress.org/Function\\_Reference/query\\_posts](http://codex.wordpress.org/Function_Reference/query_posts).

Next, add this code in:

```
$get_posts_query = new WP_Query($args);

if ($get_posts_query->have_posts()): while($get_posts_query->have_posts()): $get_posts_query->the_post();
global $post;

?>
<div>
    <span><a href="<?php the_permalink(); ?>"><?php
        the_title(); ?> | <?php the_time('F j, Y'); ?> </a>
    </span>
    <p><?php rockable_excerpt(); ?></p>
    <span><?php _e('at', 'rockable');?> <?php the_time(
        'g:i A'); ?> <a href="<?php comments_link(); ?>">
        <?php comments_number__( '0 Comments', 'rockable' ),>
        __( '1 Comment', 'rockable' ), __( '% Comments',
        'rockable' ) ); ?></a></span>
    </div>

<?php
    endwhile; endif;

    echo $after_widget;

}
```

We then create a new `WP_Query` object with the arguments array we made above. We then loop through the posts and display the HTML as it is in the HTML theme files. Finally, we display the `after_widget` variable to maintain consistency.

Now, write the `update()` function and end the class:

```
function update($new_instance, $old_instance) {
    $instance=$old_instance;

    $instance['title'] = strip_tags($new_instance['title']);
    $instance['categories'] = (array)$new_instance[
        'categories'];
    $instance['sort_by'] =$new_instance['sort_by'];
    $instance['number'] = absint( $new_instance['number'] );
    return $instance;
}

} //end class
?>
```

In this function, you get two parameters sent by WordPress — the old instance and the new instance. We create a new array `$instance` duplicating the new instance. Then, set the `$instance` array with each of the elements you defined in the `form()` function. You can make any changes — for example, we strip all tags from the title and make sure that the categories are stored as an array. Finally, return `$instance` and end the class.

You now need to load and register this widget. Open `functions.php` and add in this code:

```
require_once (ROCKABLE_WIDGETS . 'latest-posts.php');

add_action( 'widgets_init', 'rockable_load_widgets' );

function rockable_load_widgets() {
```

```
register_widget( 'Rockable_Latest_Posts' );  
}
```

First, include the widget file. Then, hook into the `widgets_init` and make sure that you register the widgets using `register_widget()`. The parameter inside that function is the name of the class in your widget file (class `Rockable_Latest_Posts` in our case.)

Now head over to the widget page, and you will be able to see the widget:

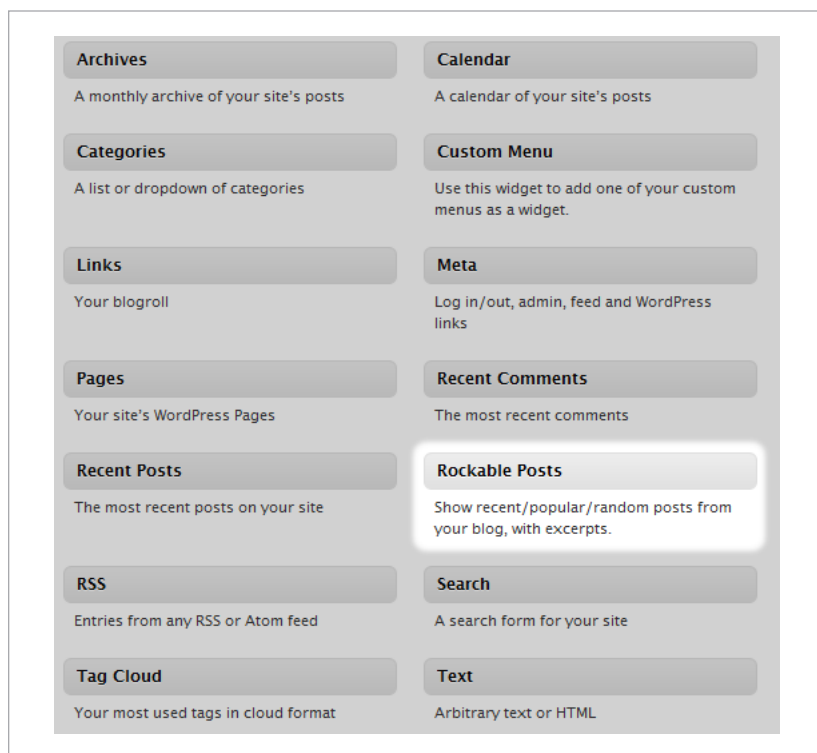


Fig. 5-3. The new Rockable Posts widget.

## Latest Portfolio Item Widget

Next up, the *Latest Portfolio Item* widget. Create a new file named `latest-portfolio.php` in the `widgets` folder. Create the basic widget skeleton and name the class and constructor `Rockable_Latest_Portfolio`. Add this code to the constructor:

```

        widget_ops = array( 'classname' => 'latest_portfolio',
        'description' => 'Show the latest portfolio post you
        have written, with an image and excerpt.' );
        $this->WP_Widget( 'rockable_portfolio', 'Rockable Latest
        Portfolio', $widget_ops );

```

Next, the `form()` function.

```

function form($instance) {
    $title= esc_attr($instance['title']);
    ?>
    <p>
        <label for="<?php echo $this->get_field_id('title');
        ?>">
            Title:
        </label>
        <input class="widefat" id="<?php echo $this->
            get_field_id('title'); ?>" name="<?php echo $this->
            get_field_name('title'); ?>" type="text" value="
            <?php echo $title; ?>" />
        </p>
        <p>
            <code><em>This widget shows your latest portfolio post.</em></code>
        </p>
    <?php
    }

```

The only option is the *title*. I have also added a description of the widget in `<code>` and `<em>` tags.

Next, the `widget()` function.

```
function widget($args, $instance) {
    extract($args);

    $title = apply_filters('widget_title', $instance['title']);
    if ( empty($title) ) $title = false;

    echo $before_widget;
    if ($title):
        echo $before_title;
        echo $title;
        echo $after_title;
    endif;

    $args = array(
        'post_type' => 'portfolio',
        'posts_per_page' => '1'
    );

    $portfolio_posts_query = new WP_Query($args);

    if ($portfolio_posts_query->have_posts()):
        while($portfolio_posts_query->have_posts()):
            $portfolio_posts_query->the_post();
            global $post;

            ?>
            <div class="project_details">
                <div class="avatar">
                    <span><a href="<?php the_permalink(); ?>"><?php
                        the_title(); ?></a></span>
                    " />
    </div>
    <p><?php rockable_excerpt(); ?></p>
</div>

<?php
    endwhile; endif;
    echo $after_widget;
}

```

No big stuff here — we create a new query querying for only one post from the portfolio custom post type, and display the HTML as it is in the HTML theme files. Next, the `update()` function and the class end:

```

function update($new_instance, $old_instance) {
    $instance = $old_instance;
    $instance['title'] = strip_tags($new_instance['title']);
    return $instance;
}

} //end class

```

The same logic as the first widget applies — grab the form elements (only *title* in this case), make any changes you want (e.g. `strip_tags()`) update the `$instance` array and return it. Now go to `functions.php` and include this file under the other widget `require` statement. Next, in the `rockable_load_widgets()` function, add another line:

```

register_widget( 'Rockable_Latest_Portfolio' );

```

If you have any trouble, just refer to the completed theme.

## Contact Form Widget

Now we're going to make a contact form widget similar to the one in the HTML. Before we start the widget, take a look at the `assets/js/rockable_contact.js` file. That AJAXifies the contact form. The logic followed is:

- When the form is submitted, get the three form elements with `document.getElementById()`.
- If their values are equal to the default values, make their values equal to the null string `""`. This is so that we don't end up submitting the form with the default values (the strings *Your Name*, *Email* and *Message* or their translated values).
- Next, we make sure the `div.alert` is empty and disable the submit button.
- We serialize the inputs (make it into a string like `name= John+Doe&email=john@doe.com&message=Lorem+Ipsum` so that PHP can access it).
- Next, we make a `POST` request to the `ajaxurl` (defined by us in `header.php`). Note that we need to actually define `ajaxurl` because it is not added by default to the `<head>`, unlike in the WordPress admin.
- We then set the contents of `div.alert` to the PHP server response and re-enable the form. We also reset the form values to their defaults if necessary.
- Finally, we add a `return false;` statement to make sure the form is not submitted in the regular way, since we have already submitted it by AJAX.

Now, create a new file named `contact-form.php` in the widgets folder and create a new widget skeleton with the class and



constructor named **Rockable\_Contact**. Here's the constructor code:

```
function Rockable_Contact(){
    $widget_ops = array( 'classname' => 'rockable-contact',
        'description' => 'Display a contact form in a widget
        area. The recipient email is set in the theme options.'
    );
    $this->WP_Widget( 'rockable_contact', 'Rockable Contact',
        $widget_ops );
}
```

You should make sure you change the first parameter in the **WP\_Widget()** call; it's quite often missed out. If you have the same unique ID for two different widgets, only one will actually be registered.

Next, the **form()** function:

```
function form($instance) {
    $title = esc_attr($instance['title']);
    ?>
    <p>
        <label for="<?php echo $this->get_field_id('title');
        ?>">
            Title:
        </label>
        <input class="widefat" id="<?php echo $this->
            get_field_id('title'); ?>" name="<?php echo $this->
            get_field_name('title'); ?>" type="text" value="
            <?php echo $title; ?>" />
    </p>

    <p>
        <code><em>This widget is best used in the footer
        columns widget area. It will display an AJAX contact form.
    </code>
    </p>
}
```

```

        </em></code>
    </p>

    <?php
    }

```

Again, there is only one input (*title*) and a short description which is optional. Next, the **widget()** function.

```

function widget($args, $instance){
    extract($args);
    $title = apply_filters('widget_title', $instance['title']);
    if ( empty($title) ) $title = false;

    echo $before_widget;
    if ($title):
        echo $before_title;
        echo $title;
        echo $after_title;
    endif;

    ?>

    <form method="post" action="<?php echo admin_url(
        'admin-ajax.php'); ?>" class="contact">
    <div class="alert"></div>
    <fieldset>
        <input type="hidden" value="rockable_contact_form"
            name="action" />
        <input type="text" name="name" value="<?php _e('Your
            Name', 'rockable'); ?>" onfocus="if(this.value==
            this.defaultValue){this.value=''}" onblur="if(
            this.value==''){this.value=this.defaultValue}"
            id="contact_name" />

```

```

        <input type="text" name="email" value="<?php _e(
            'Email', 'rockable'); ?>" onfocus="if(this.value==
            this.defaultValue){this.value=''}" onblur="if(
            this.value==''){this.value=this.defaultValue}"
            id="contact_email" />
        <textarea rows="8" cols="25" onfocus="if(this.value
            ==this.defaultValue){this.value=''}" onblur="if(
            this.value==''){this.value=this.defaultValue}"
            name="message" id="contact_message"><?php
            _e('Message', 'rockable'); ?></textarea>
        <input type="submit" class="button" value="<?php
            _e('Submit', 'rockable'); ?>" />
    </fieldset>
</form>
<?php

    echo $after_widget;
}

```

We display the `<form>` element here. The code more or less mimics the HTML files, apart from the following changes:

- We add a hidden input named `action` with the value `rockable_contact_form`. This is for the AJAX submission — don't forget, AJAX requests are made to the WordPress backend which **requires** an *action* to be specified.
- There is a `div` with the class `alert`, which displays the response from the server (success/error messages).
- We use `_e()` for the default form values to make them translatable.

Next, the `update()` function and class end:

```
function update($new_instance, $old_instance) {
    $instance = $old_instance;
    $instance['title'] = strip_tags($new_instance['title']);
    return $instance;
}
}
```

This is the same as for the *Latest Portfolio* widget.

That's it for the actual widget code. We also need to do another thing: since the AJAX is posted to WordPress' `ajaxurl`, we need to add *actions* for that. Add this code to the `contact-form.php` file after ending the class:

```
//AJAX Contact form submission
add_action('wp_ajax_rockable_contact_form',
    'rockable_send_contact');
add_action('wp_ajax_nopriv_rockable_contact_form',
    'rockable_send_contact');
```

You might ask why there are **two** actions. The reason is this: by default, a hook for `wp_ajax_{action-name}` will work only if the user is logged into WordPress. However, it will just return `-1` if not. To allow this AJAX action to be performed even if the user is not logged in, we use another hook — `wp_ajax_nopriv_{action- ▶ name}`. This works the exact same way as the other hook, except for the `_nopriv` added before the action name.

Now we need to write the `rockable_send_contact()` function referenced in the hook. Add in this code:

```
function rockable_send_contact(){
    $name    = $_REQUEST['name'];
    $email    = $_REQUEST['email'];
    $message = $_REQUEST['message'];
    if (trim($name) == '') {
```

```

        _e('Please enter your name', 'rockable');
        die();
    } else if (trim($email) == '') {
        _e('Please enter your email', 'rockable');
        die();
    } else if (!is_email($email)) {
        _e('Please enter a valid email', 'rockable');
        die();
    } else if (trim($message) == '') {
        _e('Please enter a message', 'rockable');
        die();
    }
}

```

First, we get the values sent to the function. We then check the form values — if any of them is equal to a null string, we display an error message (using `_e()` and not `echo()`) and end the function with `die()`. The email address is verified using the WordPress function `is_email()`.

Next, add in this code:

```

if (get_magic_quotes_gpc())
    $message = stripslashes($message);

$address = get_option('rockable_email');
if (!$address or $address=="")
    $address=get_option('admin_email');

$placeholders=array("%sitename%", "%name%");
$replacements=array(get_bloginfo('name'), $name);
$subject = stripslashes(get_option('rockable_form_
    subject'));
$subject=str_replace($placeholders, $replacements,
    $subject);

```

```

if (wp_mail($address, $subject, $message, "From:
$email\r\nReply-To: $email\r\nReturn-Path: $email\r\n"))
    _e('Thanks! Your message was sent.', 'rockable');
else
    _e('There was an error sending your message!',
        'rockable');

die();
}

```

Next, we apply `stripslashes()` on the message. The contact form email is fetched from the theme options using `get_option()`. If it hasn't been set, we default back to the administrator's URL, which WordPress stores by default with the option name `admin_email`. In the theme options, we gave users the option to configure the email message subject, with the placeholders `%sitename%` and `%name%` for the site name and the sender's name. We now replace the placeholders using the built in PHP function `str_replace()`. The site name is fetched using `get_bloginfo('name')`, and the sender's name from the variable `$name`. Finally, we send the mail using `wp_mail()`. `wp_mail()` is identical to the PHP `mail()` function in its parameters. If the mail was sent, we display a success message, or else we display an error message. We then end the page with `die()` and end the function. That's all. When the contact form is submitted, this function will be executed and will do the work for us.

Now, add this code to `functions.php` to include the widget file:

```
require_once (ROCKABLE_WIDGETS . 'contact-form.php');
```

Then, add this to the `rockable_load_widgets()`:

```
register_widget( 'Rockable_Contact' );
```

The widget should show up in your admin area. Try submitting it, entering incorrect data and see what happens.

The image displays two side-by-side contact form widgets. Each widget contains three input fields: a text field for the name (pre-filled with 'Rockable Press'), a text field for the email (pre-filled with 'Email' on the left and 'email@domain.com' on the right), and a large text area for the message (pre-filled with 'This is a sample email message.'). Below the input fields is a yellow 'Submit' button. The left widget has a red error message 'Please enter your email' above the name field. The right widget has a red error message 'There was an error sending your message!' above the email field.

*Fig. 5-4. Contact form; and an error message when the `mail()` function isn't enabled or configured properly.*

Note that for the contact form to work, the `mail()` function should work properly on your server. On a localhost server such as XAMPP, it usually won't work without some extra configuration. In this case, the `wp_mail()` function will return `false` and it will display an error even if the data is correctly entered (Fig. 5-4, right).

## Tabbed Widget

The final widget to be coded is the tabbed widget. However, I'm not going to be coding this. I'll give you some help, but it's your task to code it. If you are stuck, just refer to the file `widgets/tabbed-widget.php` in the completed theme. Keep the following in mind:

- I named the widget class `Rockable_Tabbed`.
- The class and constructor function should have the same name.

- Make sure you define a unique name for the first parameter in the `WP_Widget()` call.
- The options I have in the file are:
  - Widget Title
  - Title for the tab *Popular* (the text replacing 'Popular')
  - Title for the tab *Recent* (the text replacing 'Recent')
  - Title for the tab *Comments* (the text replacing 'Comments')
  - Number of posts per tab
- You will need to make three custom queries, one for each tab, in the `widget()` function. Study the HTML in the HTML theme files and use the same HTML structure for the tabs.
- Don't forget to display the `before_widget`, `before_title`, `after_title` and `after_widget` values.
- After finishing the widget, make sure you include the widget file and register the widget in the `rockable_load_widgets()` function.

Have fun coding! Remember, you can always refer to the file `widgets/tabbed-widget.php` in the completed theme if you run into trouble. After you're done, drag your widget to the sidebar and revel in its awesomeness! Next, we're going to be writing some miscellaneous functions relevant to the theme.



6

# Miscellaneous Items

In this chapter, we're going to be doing the following:

- Implement breadcrumbs
- Learn to make shortcodes
- Implement threaded comments
- Understand the post thumbnail feature

## Breadcrumbs

Breadcrumbs are navigation aids which allow users to keep track of where they are in the site. They are also said to have a positive impact on search engine optimization.



*Fig. 6-1. Breadcrumb examples.*

If you take a look at the theme files, we use a function called `rockable_breadcrumbs()` for the breadcrumbs. Open up `custom-functions.php` and start coding!

```

<?
function rockable_breadcrumbs(){
    //No breadcrumbs if disabled
    if (get_option('rockable_show_breadcrumbs', 'true') ==
        'false')
        return;
    //No breadcrumbs on homepage
    if (is_front_page())
        return;

    $breadcrumb_sep = ' / '; // Separator
    global $post;
?>
    <div class="bread_crums">
        <a href="<?php bloginfo('url'); ?>"><?php _e('Home',
            'rockable'); ?></a>
    <?php echo $breadcrumb_sep; ?>

```

Here, we first check if breadcrumbs are enabled. If they are not, we end the function. Similarly, if it is the front page (home page), we end the function. Then, we get the separator for the breadcrumb trail, defaulting to “ / ”. To start, we echo the site name wrapped with the URL. Next:

```

<?php
$blog_page_id = get_option('page_for_posts');
//Single post
if (is_single()){
    //Portfolio posts
    if (get_query_var('post_type') == 'portfolio')
        _e('Portfolio', 'rockable');
    //Blog posts
    else{
        echo '<a href="' . get_permalink($blog_page_id) .
            '>';
        echo get_the_title($blog_page_id);
    }
}

```

```

        echo '</a>';
    }
    echo $breadcrumb_sep;
    the_title();

}

if ( is_home() ) {
    echo get_the_title($blog_page_id);
}

if ( is_page() && $post->post_parent == 0 ) {
    the_title();
}

```

Next, we get the ID for the page which shows the blog posts. On single posts, we check if it's a portfolio post. If so, we echo "Portfolio" (translatable) or else we echo the name of the blog page. We follow this with the post name to get a breadcrumbs structure like *Home / Blog / Post Name* or *Home / Portfolio / Post name*. The next check is for the home page (actually, this refers to the blog posts page). In this case, we display the blog page name. Finally, we check if it is a page **without** a parent (in this case, the `post_parent` will be 0). We display the page name. Next:

```

elseif ( is_page() && $post->post_parent != 0 ) {
    $parent_id = $post->post_parent;
    $breadcrumbs = array();
    while ($parent_id) {
        $page = get_page($parent_id);
        $breadcrumbs[] = '<a href="' . get_permalink(
            $page->ID) . '">' . get_the_title($page->ID) .
            '</a>';
        $parent_id = $page->post_parent;
    }
    $breadcrumbs = array_reverse($breadcrumbs);
    foreach ($breadcrumbs as $crumb)

```

```

        echo $crumb . $breadcrumb_sep;
    the_title();
}
elseif (is_category() ) {
    _e('Archive for category', 'rockable');
    echo ' &#39;';
    single_cat_title();
    echo '&#39;';
}

```

On pages which **do** have parents, we loop through all the parent pages and display them e.g. *Home / Page 1 / Subpage 1.2 / Subpage 1.2.3*. If the page is a category archive, we display translatable text saying it is a category archive for the named category. Next:

```

elseif ( is_tax() ) {
    global $wp_query;
    $term = $wp_query->get_queried_object();
    $taxonomy = get_taxonomy ( get_query_var('taxonomy') );
    $term = $term->name;
    _e('Archive for', 'rockable');
    echo ' ' . strtolower($taxonomy->labels->singular_name)>
        . ' &#39;' . $term . '&#39;';
}
elseif ( is_day() ) {
    echo '<a href="' . get_year_link(get_the_time('Y')) . >
        '">' . get_the_time('Y') . '</a> / ' ;
    echo '<a href="' . get_month_link(get_the_time('Y'), >
        get_the_time('m')) . '">' . get_the_time('F') . >
        '</a> / ' ;
    echo get_the_time('d');
}
elseif ( is_month() ) {
    echo '<a href="' . get_year_link(get_the_time('Y')) . >
        '">' . get_the_time('Y') . '</a> / ' ;

```

```
        echo get_the_time('F');
    }
    elseif ( is_year() ) {
        echo get_the_time('Y');
    }
```

We have cases for custom taxonomies (`is_tax()`) and different date archives — day, month and year. This code is fairly simple. The final piece of code:

```
    elseif ( is_search() ) {
        _e('Search results for', 'rockable');
        echo ' &#39;' . get_search_query() . '&#39;';
    }
    elseif ( is_tag() ) {
        _e('Posts tagged', 'rockable');
        echo ' &#39;';
        single_tag_title();
        echo '&#39;';
    }

    if ( get_query_var('paged') ) {
        printf( __( ' (Page %s) ', 'rockable' ),
            get_query_var('paged') );
    }
?>
</div>
<?php
}
```

Here, we take care of search results by displaying the search term, and tag archives in a manner similar to category archives. Finally, we check if this is a **paged** page. If so, we display the page number.

Once that's done, you can go to a blog post or archives page and take a look at your cool new breadcrumbs!

## Shortcodes

Shortcodes are a way to make writing posts easier for the user. Rather than coding stuff and editing theme files, they help in customising the site without coding. For example, you could implement this shortcode:

```
[column_half]...some content...[/column_half]
[column_half]...some more content...[/column_half]
```

Which would produce this HTML:

```
<div class="column_half">
  ...some content...
</div>
<div class="column_half">
  ...some more content...
</div>
```

Or you could implement a shortcode like this:

```
[archives number="5"]
```

Which would list the five most recent posts.

Shortcodes are made using the shortcode API. Here's how it works:

- You need to have a function to handle the shortcode and return some HTML:

```
function shortcode_handler($atts, $content=null){
    extract(shortcode_atts(array(
```

```
'option 1' => 'something',  
'option 2' => 'something else',  
) , $atts));  
//Make some changes to $content and then:  
return $content;  
}
```

Basically, you write a function that accepts two parameters — **\$atts**, the shortcode attributes (options) and **\$content**, the content between the opening and closing shortcode tags.

- Then, you use the function **add\_shortcode()** to add a new shortcode:

```
add_shortcode(shortcode_name,shortcode_handler_function);
```

```
//Example:
```

```
add_shortcode('my_shortcode', 'my_handler')
```

That would result in a **shortcode** **[my\_shortcode]** ... ▸ **[/my\_shortcode]** which is handled by **my\_handler()**.

The applications for shortcodes are limited only by your imagination — here are some ideas:

- Columns — ½ width, ⅓ width, ¼ width, etc. You write a handler that generates the required HTML
- Fancy button, e.g. **[button url="#" ]Button Text** ▸ **[/button]** that creates the extra HTML required for a button.
- Display ad code in a post. e.g. **[show\_ads]**
- Quotes, e.g. **[quote]Lorem ipsum[/quote]** which display a styled quote.



To explain what I'm talking about, we're going to create a basic shortcode that adds a highlight. Create a new file, **functions/shortcodes.php**, and add in this code: ▶

```
//Highlight
function shortcode_highlight($atts, $content=null){
    extract( shortcode_atts( array(
        'color' => '#000',
        'textcolor' => '#484747'
    ), $atts ) );

    $style = ' style="color:' . $textcolor . ' ;
    background-color:' . $color . ' ; padding: 2px 4px;";' ;
    return '<span' . $style . '>'.do_shortcode($content).
    '</span>';
}
add_shortcode('hilite', 'shortcode_highlight');
```

What we do first is extract the attributes — **color** and **textcolor**. These are respectively the background color and text color of the text within the shortcode. These default to **#000** and **#484747** respectively. We then generate the HTML using a **<span>** element with inline styles. We also add a **do\_shortcode()** function wrapped around the content. This is in case there are nested shortcodes — WordPress performs shortcode handling on the text again. We then add a new shortcode **[hilite]**.

Now, include this file from **functions.php**:

```
require_once (ROCKABLE_FUNCTIONS . 'shortcodes.php');
```

For example, this text:

```
Quis aute iure reprehenderit in [hilite color='#B1E4FA'
textcolor='#666'] voluptate velit esse cillum dolore
```

```
[/hilite] eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat.
```

Produces this output:

## Aliquam Elit

August 7, 2010 | 0 Comments | Posted by admin in Announcements, News

Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat.

Fig. 6-2. The `[hilite]` shortcode in effect.

## Threaded Comments

WordPress 2.7 introduced threaded comments — comments can be replied to and with some simple CSS, be styled to look like a discussion. There have been a few changes since then, making threaded comments both easier to implement as well as easier to use.



Fig. 6-3. Threaded comments in our theme.

To begin with, we need to include a `comments.php` file. The best way to proceed with this is to copy over the `comments.php` file from the default WordPress theme (e.g. TwentyTen) and modify it as required. I have already done that for you. The changes made are:

- Changes have been made to accommodate the HTML we have. For example, the `<ol>` has been changed to a `<ul>`.
- The *callback* in `wp_list_comments()` has been changed to match our own function.
- Rather than using `comment_form()`, the comment form has been manually displayed.

There are two functions to know while writing the `comments.php` file:

- `wp_list_comments()`: This function replaces the old method of looping through the comments. Instead you can just use this to display the comments. To manage the HTML for each comment, just specify a callback function; in our case it is `rockable_comments()`.
- `comment_form()`: This can be used to display a comment form if you don't need any special HTML or styling.

To make sure that threaded comments work on your theme, you need to make sure of these things:

- First, the WordPress `comment-reply` script should be present in your header. This will allow replies to happen without refreshing the page. We have already done this in `functions.php` with this line of code:

```
wp_enqueue_script( 'comment-reply' );
```

- The textarea in the comment form must have the ID “comment”. This is essential to the script working properly. Also, no other element should have an ID of “comment”, naturally.
- Next, you have to make sure that these two lines of code are in your **comments.php**, just before the closing **<form>** tag. They generate some hidden input fields essential to comment threading.

```
<?php comment_id_fields(); ?>
<?php do_action('comment_form', $post->ID); ?>
```

- Lastly, your entire form should be wrapped in a **div** with the ID “respond”. So you should have:

```
<div id="respond">
    <form>
        [...]
    </form>
</div>
```

These have already been done. Now, you just need to write the callback function for the comment list. Create a new file named **comment-list.php** in the **functions** folder. Paste in this code:

```
<?php
function rockable_comments($comment, $args, $depth) {

    $GLOBALS['comment'] = $comment; ?>

    <li <?php comment_class(); ?> id="li-comment-<?php
        comment_ID() ?>">

        <div id="comment-<?php comment_ID(); ?>" class=
            "user_info">

            <h3><?php comment_author_link()?></h3> <small>
```

```

        <?php _e('Says', 'rockable'); ?></small>
        <?php echo get_avatar($comment,$size='70'); ?>
        <p><?php printf(__('%1$s at %2$s', 'rockable'),
            get_comment_date('F j, Y'), get_comment_time(
                'g:i a')) ?></p>
        <p><?php comment_reply_link( array_merge( $args,
            array('reply_text' => __('Reply', 'rockable'),
                'depth' => $depth, 'max_depth' =>
                    $args['max_depth'] ) ) ); ?></p>
    </div>
    <div class="comment">
        <?php if ($comment->comment_approved == '0') : ?>
            <em>Your comment is awaiting moderation</em>
        <?php endif; ?>
        <?php echo get_comment_text(); ?>
    </div>
<?php
}
?>

```

Quick explanation: Since the listing is a list element, we start with an `<li>` tag. The rest of the HTML is similar to the HTML which the HTML theme files have. There is one function of importance:

```

<?php comment_reply_link( array_merge( $args, array(
    'reply_text' => __('Reply', 'rockable'), 'depth' => $depth,
    'max_depth' => $args['max_depth'] ) ) ); ?>

```

That generates a link which allows the user to reply to a comment. If you've done everything correctly, clicking that link will allow you to reply to a comment by moving the form up, and not refreshing the page. Try it!

## Post Thumbnails

WordPress 2.9 introduced a new *Post Thumbnail* feature. It allows users to easily upload an image for a post. You might remember that there was a check for post thumbnail images in the `rockable_post_image()` function — this is the post thumbnail we spoke of. To enable post images, you need to first add this code to `functions.php`:

```
add_theme_support( 'post-thumbnails' );
```

This is similar to the support for menus. By default, WordPress creates a few different post thumbnail sizes. To add one of your own, add a line like this:

```
add_image_size( 'rockable_thumb', 270, 170, true );
```

You can then display the image by writing this code:

```
if ( has_post_thumbnail() )  
    the_post_thumbnail('rockable_thumb');
```

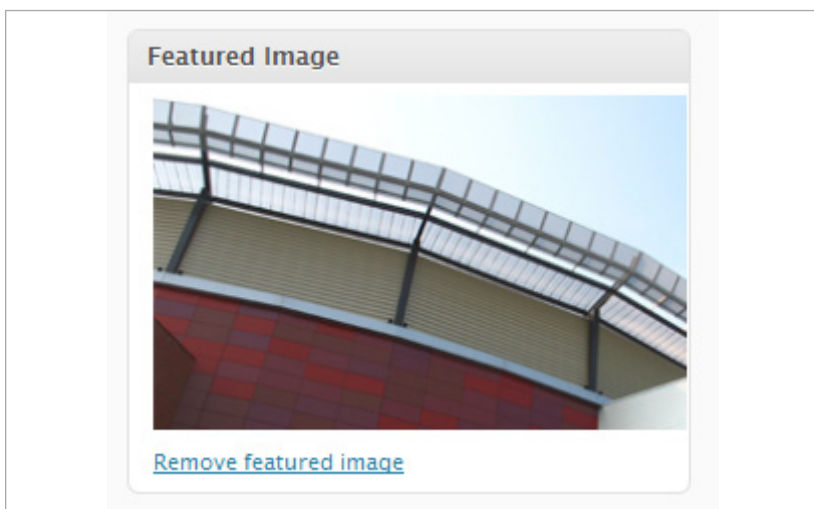
We actually need the URL of the image in `rockable_post_image()`, so we use this code:

```
$image_id = get_post_thumbnail_id($post->ID);  
$image = wp_get_attachment_image_src($image_id,  
    'rockable_thumb');  
$image = $image[0];  
if ($image) return $image;
```

First, we get the ID of the post thumbnail using `get_post_thumbnail_id()`. Then, we get the image's source URL using the image ID and `wp_get_attachment_image_src()` and the `rockable_thumb` image size we specified. This returns an array

in `$image` — we get the first array index ( `[0]` ), which is the URL, and return it.

By enabling support for post thumbnails, this box shows up while editing posts:



*Fig. 6-4. The Post Thumbnail feature.*

This allows the user two ways to set a post thumbnail — either by entering a URL from the meta box we wrote earlier, or by using the *Post Thumbnail* feature.

That pretty much concludes the writing of advanced features for a WordPress theme. If you run into any errors or problems, just refer to the finished theme. In the next and final chapter, you are going to find out how to make your theme translation ready and learn to create `.po` and `.mo` files for translation.

7



# Internationalization and Translation

WordPress is a very widespread publishing platform — people from all over the world use it. Therefore it stands to reason that people would like to make sites in their own language. While developing a WordPress theme, developers often “hard code” English words into the theme. For example:

```
<div class="title">  
    Search results for <?php echo get_search_query(); ?>  
</div>
```

If the theme user runs an English-based site, this is completely acceptable. But what if he wants to run a site in French? He would have to look through tens, possibly hundreds of PHP files, searching for hard coded text and updating it. As if that wasn't tough enough, if he wanted to make a small change later on, he'd have to search through the files all over again! To make things much easier for users, you can make the theme translation ready. This process is known as *internationalization*, often abbreviated as *i18n*. I will teach you the different ways to internationalize text below.

## Text Domains

WordPress uses a library called GNU gettext for translation. Basically, you have to “tag” pieces of text for translation, to make things easy for the translator. Before you begin, you have to define something known as a *text domain* for your theme. The text domain is a unique identifier, which makes sure WordPress can distinguish between all loaded translations. For this theme, I am using **rockable** as the text domain. To define the text domain, add this code to **functions.php**:

```
load_theme_textdomain( 'rockable', TEMPLATEPATH .  
'/language' );
```

This does two things:

1. First, it tells WordPress that this theme uses the text domain **rockable** for i18n, and all texts marked with this text domain should be translated with the **.po** file as mentioned in the following point.
2. Second, it tells WordPress that the **.po** file for translation will be found in the folder **language** within the theme. For example, if you are using WordPress in French (**fr\_FR**), it will look for the file **/language/fr\_FR.po** for translation.

## I18n Functions

Once you have a text domain, you are ready to start i18n. There are three basic WordPress functions to be used for marking these texts for translation:

1. For strings (i.e. those that are not echoed to the browser), you can use the function **\_\_()** (that's *underscore underscore*). Example:

```
$str = 'This is some text'; //Not marked for translation  
$str = __('This is some text', 'rockable'); //Marked  
for translation with the domain rockable
```

2. For echoing strings, you can use the **\_e()** function. An HTML example:

```
<h2>Add a comment</h2> <!-- Not marked for translation  
-->  
<h2><?php _e('Add a comment', 'rockable'); ?></h2> <!--  
Marked for translation with the domain rockable -->
```

An example in PHP:

```
<?php
echo 'Add a comment'; // Not marked for translation
__e('Add a comment', 'rockable'); //Marked for translation with the domain rockable
?>
```

3. If the same text is used in different contexts — for example, *Post* can be used as in *Add a Post* or *Post Comment* — then how do you mark these texts differently? Answer: `__x()`. You can specify a context for the string. Example:

```
__x('Post', 'noun -- e.g. new post', 'rockable');
__x('Post', 'verb -- e.g. post comment', 'rockable');
```

These are marked separately, translated by context.

## Different i18n Scenarios

While the three functions mentioned above should suffice for many situations, there are some situations which need more than those. For example, suppose you were displaying a dynamic variable (e.g. a number) in a string, that can't be translated, obviously. Example:

```
$users = get_number_of_users();
echo "There are $users users";
```

For this, if you wrote the string as:

```
__e("There are $users users", 'rockable');
```

Then it would not work properly. Instead, you have to use the PHP functions `printf` and `sprintf`. Here's the correct code:

```
printf( __('There are %d users', 'rockable'), $users )
```

If you had multiple dynamic strings, like this:

```
echo "There are $users users and $active active users";
```

Then, the `printf` equivalent would be:

```
printf( __('There are %1$s users and %2$s active users.',  
'rockable'), $users, $active) )
```

This is known as *argument swapping*. It is explained in detail in the PHP manual. The translator could even exchange the `%1$s` and `%2$s` to reverse the order (e.g. “There are 25 active users and 50 users.”).

`sprintf` works much the same as `printf`, except for the fact that `sprintf` returns the text while `printf` echoes the text.

Another scenario is plurals. What if there was only one user? Then the text above would read “There are 1 active users.” — definitely incorrect grammar. In this case, use `_n()`. Example:

```
printf( _n('There is %d active user', 'There are %d active  
users', $users, 'rockable'), $users) )
```

The syntax for the function is:

```
_n(Singular text, Plural text, count, domain);
```

The parameter `count` is the number to compare to — if `count` is 1, `Singular text` will be used. Otherwise, `Plural text` will be used.

Here are some examples of translatable text from our theme:

```
//From 404.php  
_e('404 Error - Page Not Found', 'rockable');
```

```
//From custom-functions.php
printf( __( ' (Page %s) ', 'rockable' ),
    get_query_var('paged') );

//From register-wp3.php
$columns = array(
    "cb" => "<input type=\"checkbox\" />",
    "title" => _x("Portfolio Title", "portfolio title
        column", 'rockable'),
    "author" => _x("Author", "portfolio author column",
        'rockable'),
    "portfolio_cats" => _x("Portfolio Categories",
        "portfolio categories column", 'rockable'),
    "date" => _x("Date", "portfolio date column", 'rockable')
);
```

## Generating the .po File

Now, we need to make a .po file for translators. To do this, download the translation software POEdit from here: <http://www.poedit.net/download.php>. POEdit is one of the most popular translation software out there. What's not so well known is, it's also super easy to make .po files with it. Once you have it downloaded, click *File* ➤ *New Catalog*. A box should open up (Fig. 7-1).

Fill in a Project Name of your choice — I used *Rockable*. Next, go to the *Paths* tab. The *Paths* tab allows you to configure the folders in which files are to be found for translation. Let's leave the base path as '.' which refers to the directory in which the catalog is (language for us). We have files with translation ready text in the root folder of the theme, the **functions** folder, **includes** folder and **widgets** folder. Naturally, if you had more or fewer folders, you would modify the paths. This is what the tab should look like, just add new items (Fig. 7-2).

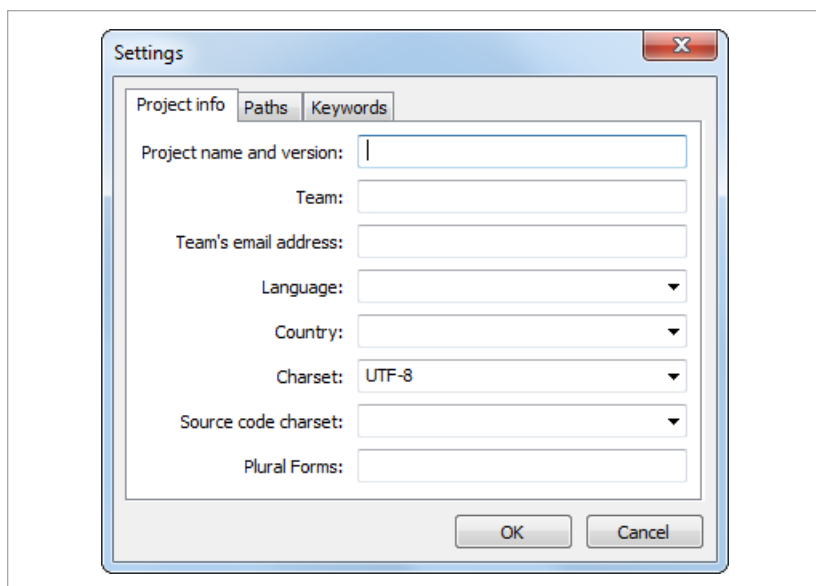


Fig. 7-1. POEdit's New Catalog setup.

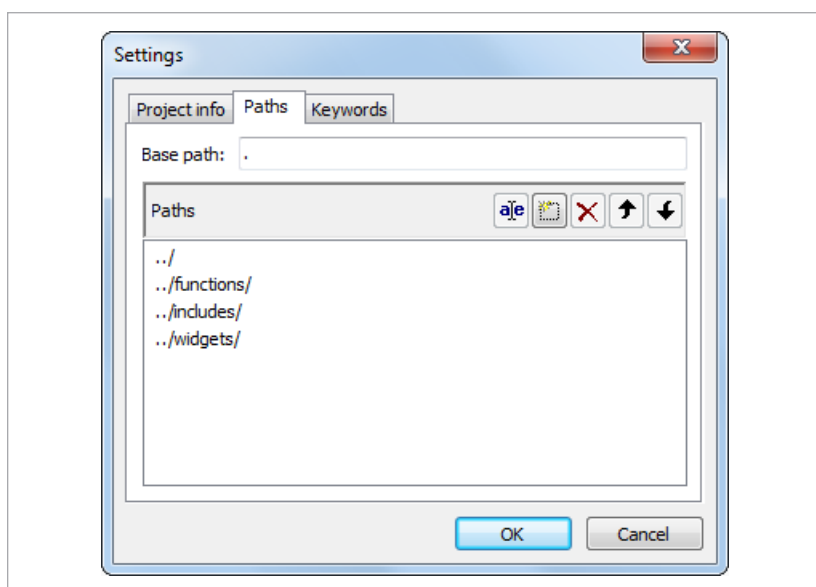


Fig. 7-2. The Paths tab.

Next, go to the *Keywords* tab. Remove all the items there, and add in these keywords:

- `__`
- `_e`
- `_n`
- `_x`

If you used any other functions for translation, add them in too. Don't add the brackets, and don't add PHP functions like `printf`. Your *Keywords* tab should look like this:

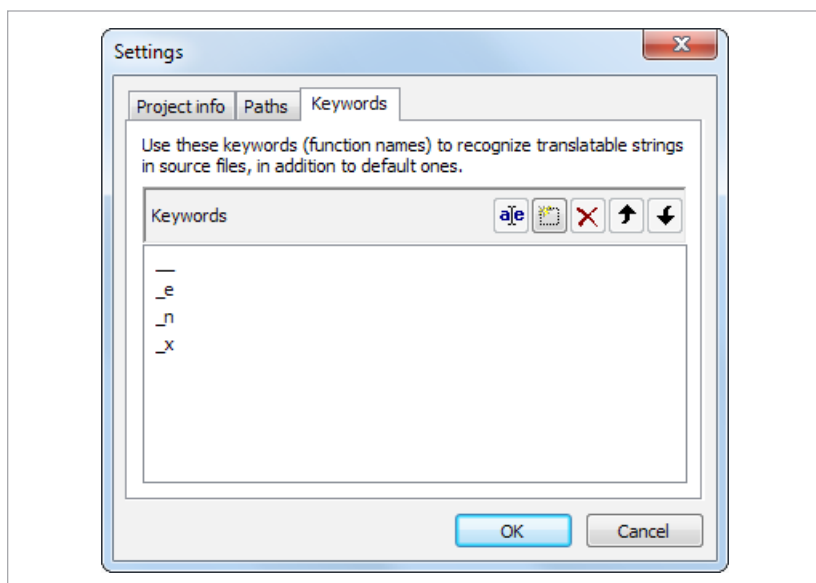


Fig. 7-3. The *Keywords* tab.

Now click *OK*. Save the file in your **language** folder, with a name of your choice. You should get a box looking like Figure 7-4. Now the file is ready for translation. The translation screen appears as in Figure 7-5.

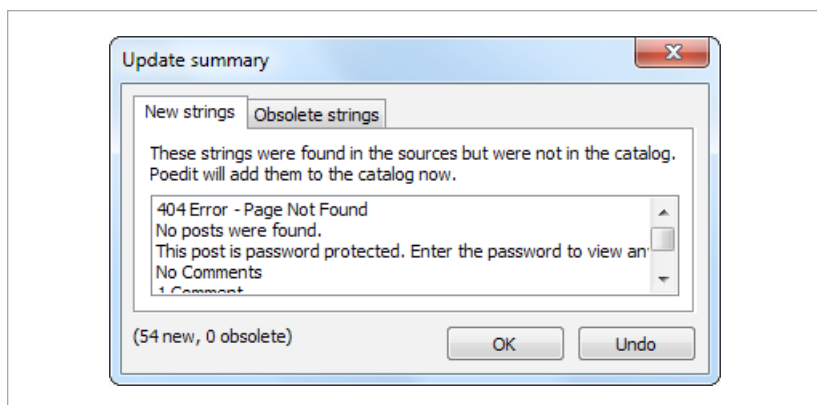


Fig. 7-4. Output after parsing PHP files.

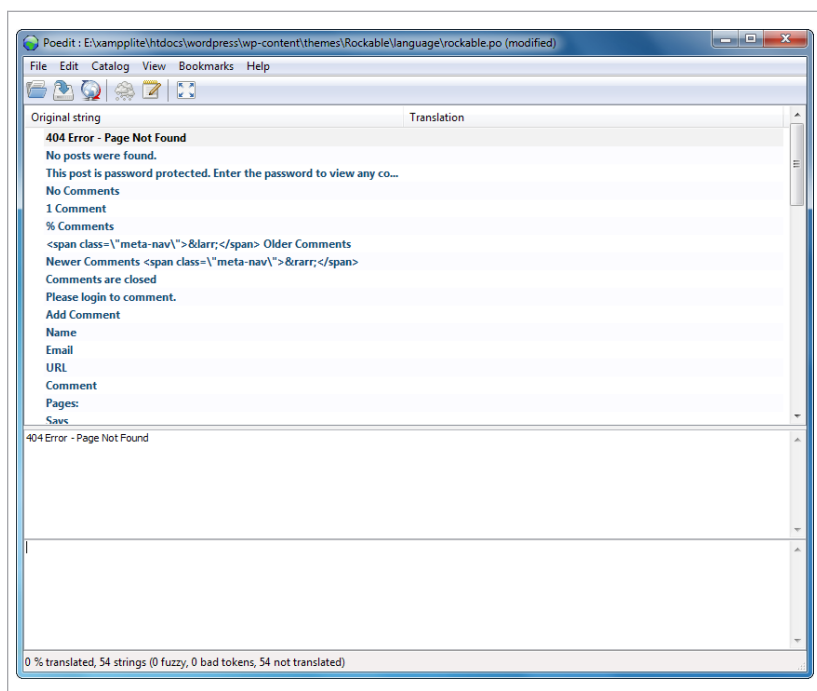


Fig. 7-5. The translate screen.



## Using the PO File

To use the file, you first have to define a locale. To do this, open the `wp-config.php` file in the root WordPress directory. Find this line:

```
define ('WPLANG', '');
```

Change it to match your locale, e.g.:

```
define ('WPLANG', 'fr_FR'); //French
```

A full list of locales can be found here: [http://codex.wordpress.org/WordPress\\_in\\_Your\\_Language](http://codex.wordpress.org/WordPress_in_Your_Language).

Then, open the `rockable.po` file in POEdit. Translate all the strings you want, then click *File* ➤ *Save As*. Save the file name with the same name as the locale. This is vital. In this case, I saved it as `fr_FR.po`, in the `language` folder. Now refresh the site. You should see the texts in your translated text:

### Get in Touch

S'il vous plaît entrer votre nom

*(Please enter your name)*

---



**admin** Dit

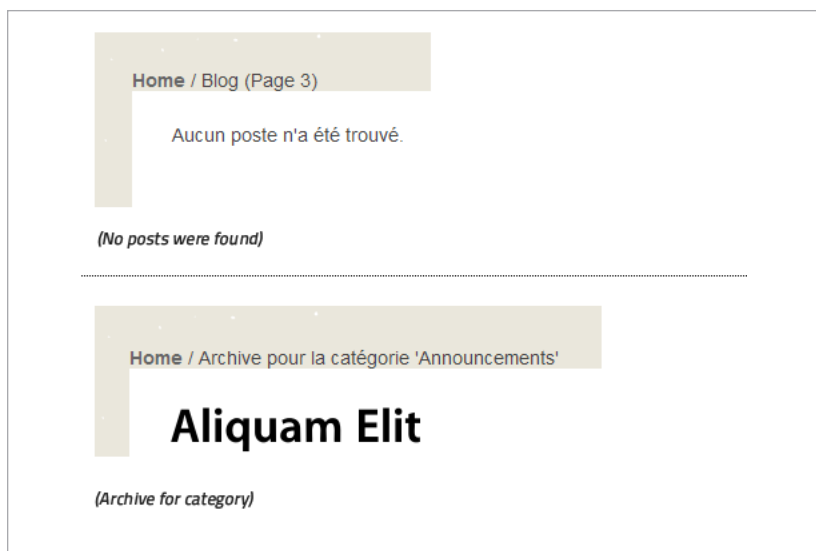
August 16, 2010 at 8:28 am

Réponse

In hac habitasse platea dictumst. Aliquam et turpis ante. Nullam justo orci, blz posuere ut, posuere ac sem. Fusce elit dui, imperdiet id tempus ac, lobortis s lorem.

*(Reply-to text)*

Fig. 7-6. Sample translated texts.



*Fig. 7-6. Sample translated texts, continued.*

That concludes internationalization. This is a feature that is pretty simple to integrate, but adds a lot of value to your themes, especially if you are distributing them to a large and varied audience.

# AFTERWORD

## Afterword

In this book, I've shown you how powerful WordPress can get as a content management platform, and not just as a blogging platform. Entire sites can be run on WordPress, and the ways to customize WordPress are limitless.

You have learnt how to develop code for WordPress that is reusable across projects — the theme options and meta box classes can be adapted across themes extremely easily and without much effort at all. You can try your hand at developing classes and functions that suit your project, and improve your workflow and efficiency by reusing code.

No matter what your objective is while creating WordPress themes, remember to keep code clean and reusable, observe coding conventions and best practices, and above all — enjoy yourself and don't feel scared to innovate and create new things!

—Rohan Mehta

## About The Author



Rohan Mehta is a Wordpress and jQuery artist, with significant experience in Java and web technologies. He is passionate about racquet sports, reading, and of course, code. In his opinion, code is a form of art, and he expresses his creativity in Java, PHP and JavaScript. Rohan is currently in pursuit of a Bachelor's degree in Computer Science, and resides in the United States and India; he can be reached at his email, [rohan@getrohan.com](mailto:rohan@getrohan.com).

## Your Download Link

To download your project files, please use the link below.

[http://rockable.s3.amazonaws.com/wordpress-designer-2/  
wordpress-2-projectfiles.zip](http://rockable.s3.amazonaws.com/wordpress-designer-2/wordpress-2-projectfiles.zip)

As a web designer or developer, WordPress is one of the most potent tools at your disposal. In Rockstar WordPress Designer 2, Rohan Mehta picks up where the original Rockstar WordPress Designer left off and teaches you to take advantage of WordPress 3.0's more advanced features. Included in the book:

- \* Using Metaboxes
- \* Advanced image functions
- \* Custom widgets and option pages
- \* Using shortcodes and custom post types
- \* Internationalization and translation issues
- \* Crafting threaded comments, breadcrumb
- \* navigation, and more!

Rockstar WordPress Designer 2 also includes EVERY project file referenced in the book, helping you get off to a running start. Grab a copy of Rockstar WordPress Designer 2 and improve your WordPress mastery.



**ROCKABLE\***