SMASHING MAGAZINE

WordPress Tutorials

Detailed Techniques

Imprint

Version 2: October 2012

ISBN: 978-3-943075-19-9

Cover Design: Ricardo Gimenes

PR & Press: Stephan Poppe

eBook Strategy: Andrew Rogerson & Talita Telma Stöckle

Technical Editing: Andrew Rogerson

Idea & Concept: Smashing Media GmbH

ABOUT SMASHING MAGAZINE

Smashing Magazine is an online magazine dedicated to Web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy. Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is — and always has been — a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised.

ABOUT SMASHING MEDIA GMBH

Smashing Media GmbH is one of the world's leading online publishing companies in the field of Web design. Founded in 2009 by Sven Lennartz and Vitaly Friedman, the company's headquarters is situated in southern Germany, in the sunny city of Freiburg im Breisgau. Smashing Media's lead publication, Smashing Magazine, has gained worldwide attention since its emergence back in 2006, and is supported by the vast, global Smashing community and readership. Smashing Magazine had proven to be a trustworthy online source containing high quality articles on progressive design and coding techniques as well as recent developments in the Web design industry.

About this eBook

You've probably already gathered a lot of knowledge on how to improve your WordPress skills, but applying this on your own might be a complicated process. **WordPress Tutorials** will facilitate putting all this knowledge into practice. Through the accurate methods which were exclusively selected, you will certainly acquire the expertise you need for your future WordPress publishing projects without any complications.

Table of Contents

How To Create Custom Post Meta Boxes In WordPress

Justin Tadlock

What seems like one of the most complicated bits of functionality in WordPress is adding meta boxes to the post editing screen. This complexity only grows as more and more tutorials are written on the process with weird loops and arrays. Even meta box “frameworks” have been developed. I’ll let you in on a little secret though: it’s not that complicated.

Creating custom meta boxes is extremely simple, at least it is once you've created your first one using the tools baked into WordPress' core code. In this tutorial, I'll walk you through everything you need to know about meta boxes:

- Creating meta boxes.
- Using meta boxes with any post type.
- Handling data validation.
- Saving custom meta data.

- Retrieving custom meta data on the front end.

Note: When I use the term “post” throughout this tutorial, I’m referring to a post of any post type, not just the default blog post type bundled with WordPress.

What is a post meta box?

A post meta box is a draggable box shown on the post editing screen. Its purpose is to allow the user to select or enter information in addition to the main post content. This information should be related to the post in some way.

Generally, two types of data is entered into meta boxes:

- Metadata (i.e. custom fields),
- Taxonomy terms.

Of course, there are other possible uses, but those two are the most common. For the purposes of this tutorial, you’ll be learning how to develop meta boxes that handle custom post metadata.

What is post metadata?

Post metadata is data that’s saved in the `wp_postmeta` table in the database. Each entry is saved as four fields in this table:

- `meta_id`: A unique ID for this specific metadata.
- `post_id`: The post ID this metadata is attached to.
- `meta_key`: A key used to identify the data (you’ll work with this often).

- `meta_value`: The value of the metadata.

In the following screenshot, you can see how this looks in the database.

	meta_id	post_id	meta_key	meta_value
	1395	628	superawesome	false
	1397	628	superawesome	false
	55	21	Thumbnail	http://localhost/wp-content/uploads/2010/07/w
	71	24	Thumbnail	http://localhost/wp-content/uploads/2010/07/or
	1284	297	Thumbnail	http://localhost/wp-content/uploads/2010/09/Li
	1285	584	Thumbnail	http://localhost/wp-content/uploads/2010/10/Je
	1287	540	Thumbnail	http://localhost/wp-content/uploads/2010/10/1c
	1288	469	Thumbnail	http://localhost/wp-content/uploads/2010/10/cc
	1289	448	Thumbnail	http://localhost/wp-content/uploads/2010/10/Cl
	74	24	Title	The beast mode
	1057	469	Title	Image alignment in posts!
	2891	584	Title	This image is extremely large
	2204	9	ttw_tag	"<script>alert('XSS');</script>"
	3545	817	Video	#
	449	59	Views	51
	450	175	Views	6
	451	24	Views	67

When you get right down to it, metadata is just key/value pairs saved for a specific post. This allows you to add all sorts of custom data to your posts. It is especially useful when you're developing custom post types.

The only limit is your imagination.

Note: One thing to keep in mind is that a single meta key can have multiple meta values. This isn't a common use, but it can be extremely powerful.

Working with post metadata

By now, you're probably itching to build some custom meta boxes. However, to understand how custom meta boxes are useful, you must understand how to add, update, delete, and get post metadata.

I could write a book on the various ways to use metadata, but that's not the main purpose of this tutorial. You can use the following links to learn how the post meta functions work in WordPress if you're unfamiliar with them.

- add_post_meta(): Adds post metadata.
- update_post_meta(): Updates post metadata.
- delete_post_meta(): Deletes post metadata.
- get_post_meta(): Retrieves post metadata.

The remainder of this tutorial assumes that you're at least familiar with how these functions work.

The setup

Before building meta boxes, you must have some ideas about what type of metadata you want to use. This tutorial will focus on building a meta box that saves a custom post CSS class, which can be used to style posts.

I'll start you off by teaching you to develop custom code that does a few extremely simple things:

- Adds an input box for you to add a custom post class (the meta box).
- Saves the post class for the `smashing_post_class` meta key.
- Filters the `post_class` hook to add your custom post class.

You can do much more complex things with meta boxes, but you need to learn the basics first.

All of the PHP code in the following sections goes into either your custom plugin file or your theme's functions.php file.

Building a custom post meta box

Now that you know what you're building, it's time to start diving into some code. The first two code snippets in this section of the tutorial are mostly about setting everything up for the meta box functionality.

Since you only want your post meta box to appear on the post editor screen in the admin, you'll use the `load-post.php` and `load-post-new.php` hooks to initialize your meta box code.

```
/* Fire our meta box setup function on the post editor screen.
*/
add_action( 'load-post.php',
'smashing_post_meta_boxes_setup' );
add_action( 'load-post-new.php',
'smashing_post_meta_boxes_setup' );
```

Most WordPress developers should be familiar with how hooks work, so this should not be anything new to you. The above code tells WordPress that you want to fire the `smashing_post_meta_boxes_setup` function on the post editor screen. The next step is to create this function.

The following code snippet will add your meta box creation function to the `add_meta_boxes` hook. WordPress provides this hook to add meta boxes.

```
/* Meta box setup function. */
function smashing_post_meta_boxes_setup() {

  /* Add meta boxes on the 'add_meta_boxes' hook. */
```

```
  add_action( 'add_meta_boxes',
'smashing_add_post_meta_boxes' );
}
```

Now, you can get into the fun stuff.

In the above code snippet, you added the `smashing_add_post_meta_boxes()` function to the `add_meta_boxes` hook. This function's purpose should be to add post meta boxes.

In the next example, you'll create a single meta box using the add_meta_box() WordPress function. However, you can add as many meta boxes as you like at this point when developing your own projects.

Before proceeding, let's look at the `add_meta_box()` function:

```
add_meta_box( $id, $title, $callback, $page, $context =
'advanced', $priority = 'default', $callback_args = null );
```

- `$id`: This is a unique ID assigned to your meta box. It should have a unique prefix and be valid HTML.
- `$title`: The title of the meta box. Remember to internationalize this for translators.
- `$callback`: The callback function that displays the output of your meta box.
- `$page`: The admin page to display the meta box on. In our case, this would be the name of the post type (post, page, or a custom post type).
- `$context`: Where on the page the meta box should be shown. The available options are normal, advanced, and side.
- `$priority`: How high/low the meta box should be prioritized. The available options are default, core, high, and low.

- `$callback_args`: An array of custom arguments you can pass to your $callback function as the second parameter.

The following code will add the post class meta box to the post editor screen.

```
/* Create one or more meta boxes to be displayed on the post
editor screen. */
function smashing_add_post_meta_boxes() {

  add_meta_box(
      'smashing-post-class',           // Unique ID
      esc_html__( 'Post Class', 'example' ),      // Title
      'smashing_post_class_meta_box',     // Callback function
      'post',                // Admin page (or post type)
      'side',                // Context
      'default'                 // Priority
  );
}
```

You still need to display the meta box's HTML though. That's where the `smashing_post_class_meta_box()` function comes in (`$callback` parameter from above).

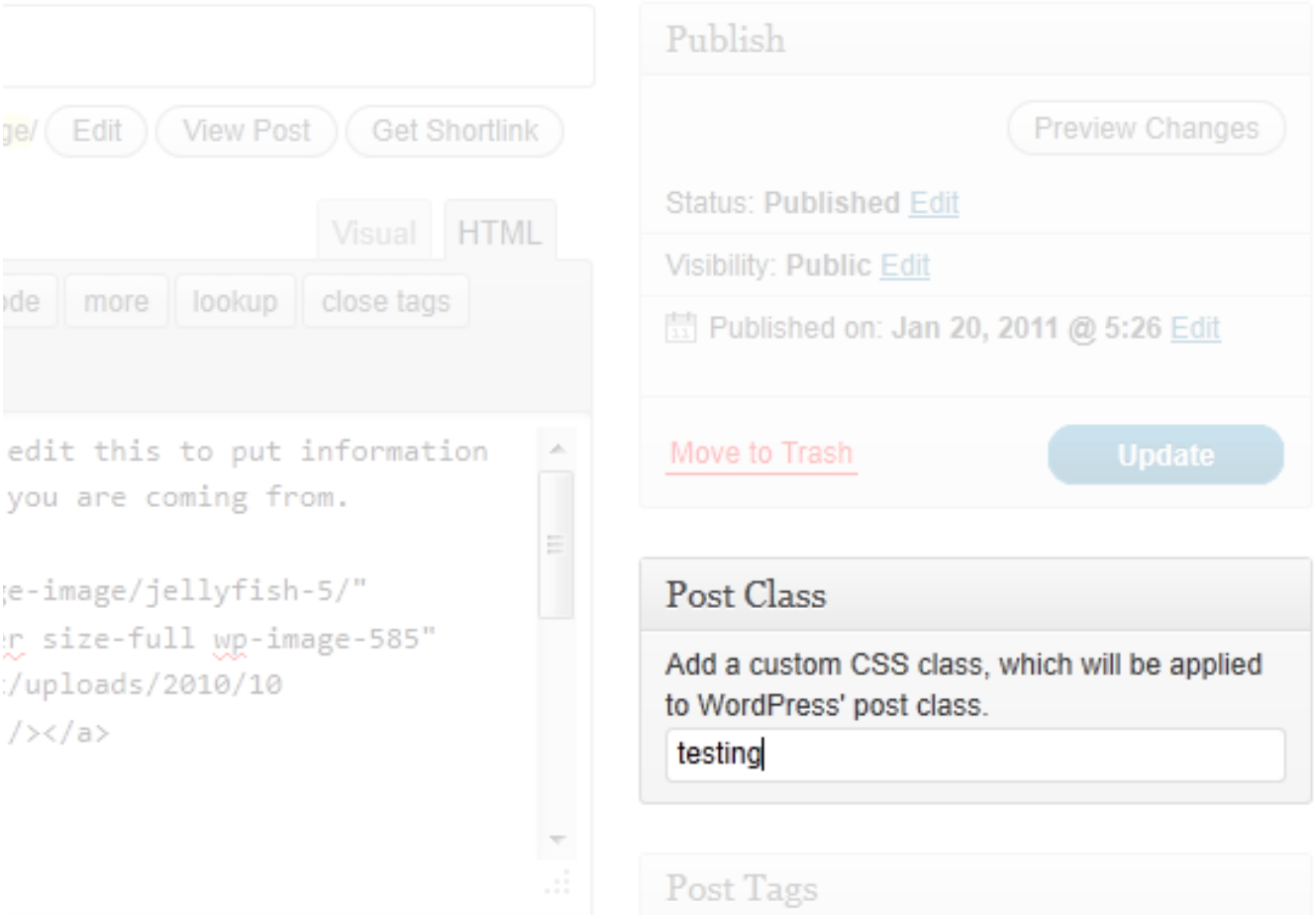
```
/* Display the post meta box. */
function smashing_post_class_meta_box( $object, $box ) { ?>

  <?php wp_nonce_field( basename( __FILE__ ),
'smashing_post_class_nonce' ); ?>

  <p>
      <label for="smashing-post-class"><?php _e( "Add a custom
CSS class, which will be applied to WordPress' post class.",
'example' ); ?></label>
      <br />
      <input class="widefat" type="text" name="smashing-post-
class" id="smashing-post-class" value="<?php echo
esc_attr( get_post_meta( $object->ID, 'smashing_post_class',
true ) ); ?>" size="30" />
  </p>
<?php }
```

What the above function does is display the HTML output for your meta box. It displays a hidden nonce input (you can read more about nonces on the WordPress Codex). It then displays an input element for adding a custom post class as well as output the custom class if one has been input.

At this point, you should have a nice-looking meta box on your post editing screen. It should look like the following screenshot.



The meta box doesn't actually do anything yet though. For example, it won't save your custom post class. That's what the next section of this tutorial is about.

Saving the meta box data

Now that you've learned how to create a meta box, it's time to learn how to save post metadata.

Remember that `smashing_post_meta_boxes_setup()` function you created earlier? You need to modify that a bit. You'll want to add the following code to it.

```
/* Save post meta on the 'save_post' hook. */
add_action( 'save_post', 'smashing_save_post_class_meta', 10,
2 );
```

So, that function will actually look like this:

```
/* Meta box setup function. */
function smashing_post_meta_boxes_setup() {

  /* Add meta boxes on the 'add_meta_boxes' hook. */
  add_action( 'add_meta_boxes',
'smashing_add_post_meta_boxes' );

  /* Save post meta on the 'save_post' hook. */
  add_action( 'save_post', 'smashing_save_post_class_meta',
10, 2 );
}
```

The new code you're adding tells WordPress that you want to run a custom function on the `save_post` hook. This function will save, update, or delete your custom post meta.

When saving post meta, your function needs to run through a number of processes:

- Verify the nonce set in the meta box function.
- Check that the current user has permission to edit the post.

- Grab the posted input value from `$_POST`.
- Decide whether the meta should be added, updated, or deleted based on the posted value and the old value.

I've left the following function somewhat generic so that you'll have a little flexibility when developing your own meta boxes. It is the final snippet of code that you'll need to save the metadata for your custom post class meta box.

```
/* Save the meta box's post metadata. */
function smashing_save_post_class_meta( $post_id, $post ) {

  /* Verify the nonce before proceeding. */
  if ( !isset( $_POST['smashing_post_class_nonce'] ) || !
wp_verify_nonce( $_POST['smashing_post_class_nonce'],
basename( __FILE__ ) ) )
      return $post_id;

  /* Get the post type object. */
  $post_type = get_post_type_object( $post->post_type );

  /* Check if the current user has permission to edit the
post. */
  if ( !current_user_can( $post_type->cap->edit_post,
$post_id ) )
      return $post_id;

  /* Get the posted data and sanitize it for use as an HTML
class. */
  $new_meta_value = ( isset( $_POST['smashing-post-class'] ) ?
sanitize_html_class( $_POST['smashing-post-class'] ) : '' );

  /* Get the meta key. */
  $meta_key = 'smashing_post_class';

  /* Get the meta value of the custom field key. */
  $meta_value = get_post_meta( $post_id, $meta_key, true );

  /* If a new meta value was added and there was no previous
value, add it. */
```

```
  if ( $new_meta_value && '' == $meta_value )
      add_post_meta( $post_id, $meta_key, $new_meta_value,
true );

  /* If the new meta value does not match the old value,
update it. */
  elseif ( $new_meta_value && $new_meta_value != $meta_value )
      update_post_meta( $post_id, $meta_key, $new_meta_value );

  /* If there is no new meta value but an old value exists,
delete it. */
  elseif ( '' == $new_meta_value && $meta_value )
      delete_post_meta( $post_id, $meta_key, $meta_value );
}
```

At this point, you can save, update, or delete the data in the “Post Class” meta box you created from the post editor screen.

Using the metadata from meta boxes

So you have a custom post meta box that works, but you still need to do something with the metadata that it saves. That’s the point of creating meta boxes. What to do with your metadata will change from project to project, so this is not something I can answer for you. However, you will learn how to use the metadata from the meta box you’ve created.

Since you’ve been building a meta box that allows a user to input a custom post class, you’ll need to filter WordPress’ `post_class` hook so that the custom class appears alongside the other post classes.

Remember that `get_post_meta()` function from much earlier in the tutorial? You’ll need that too.

The following code adds the custom post class (if one is given) from your custom meta box.

```
/* Filter the post class hook with our custom post class
function. */
add_filter( 'post_class', 'smashing_post_class' );

function smashing_post_class( $classes ) {

  /* Get the current post ID. */
  $post_id = get_the_ID();

  /* If we have a post ID, proceed. */
  if ( !empty( $post_id ) ) {

      /* Get the custom post class. */
      $post_class = get_post_meta( $post_id,
'smashing_post_class', true );

      /* If a post class was input, sanitize it and add it to
the post class array. */
      if ( !empty( $post_class ) )
          $classes[] = sanitize_html_class( $post_class );
  }

  return $classes;
}
```

If you look at the source code of the page where this post is shown on the front end of the site, you'll see something like the following screenshot.

Pretty cool, right? You can use this custom class to style posts however you want in your theme's stylesheet.

Security

One thing you should keep in mind when saving data is security. Security is a lengthy topic and is outside the scope of this article. However, I thought it best to at least remind you to keep security in mind.

You've already been given a link explaining nonces earlier in this tutorial. The other resource I want to provide you with is the WordPress Codex guide on data validation. This documentation will be your best friend when learning how to save post metadata and will provide you with the tools you'll need for keeping your plugins/themes secure.

Bonus points to anyone who can name all of the security measures used throughout this tutorial.

Create a custom meta box

Once you've copied, pasted, and tested the bits of pieces of code from this tutorial, I encourage you to try out something even more complex. If you really want to see how powerful meta boxes and post metadata can be, try doing something with a single meta key and multiple meta values for that key (it's challenging).

How To Create Tabs On WordPress Settings Pages

Elio Rivero

Using tabs in a user interface can help you better organize content, so it's only natural that WordPress themes that have a lot of options would benefit from tabs on their settings page. In this tutorial, you will learn how to create a tabbed settings page, and you'll get to download a WordPress theme that implements the code.

OVERVIEW

To get a quick grasp of the tabs we'll be creating, go to `Appearance/Themes` in the WordPress admin area. You will find two tabs there: "Manage Themes" and "Install Themes." When you click on one, the content changes and the tab's title is highlighted.

The process is actually fairly simple: we set and send a `tab` variable when a tab is clicked. By querying this tab variable later, in `$_GET['tab']`, we will know which tab was selected so that we can highlight the corresponding title and display the corresponding tab.

In our approach, there are three times when we will need to know which tab the user is currently on:

1. **When we initially display the tabs and the form fields for the settings (in order to display the correct set of fields);**
2. When the user saves their settings (in order to save the correct fields);
3. When redirecting the user after they have saved their settings (in order to redirect the user to the correct tab).

For the sake of brevity, we won't explain all of the code, only the snippets that are relevant to this approach. You can, however, find all of the code in the accompanying theme.

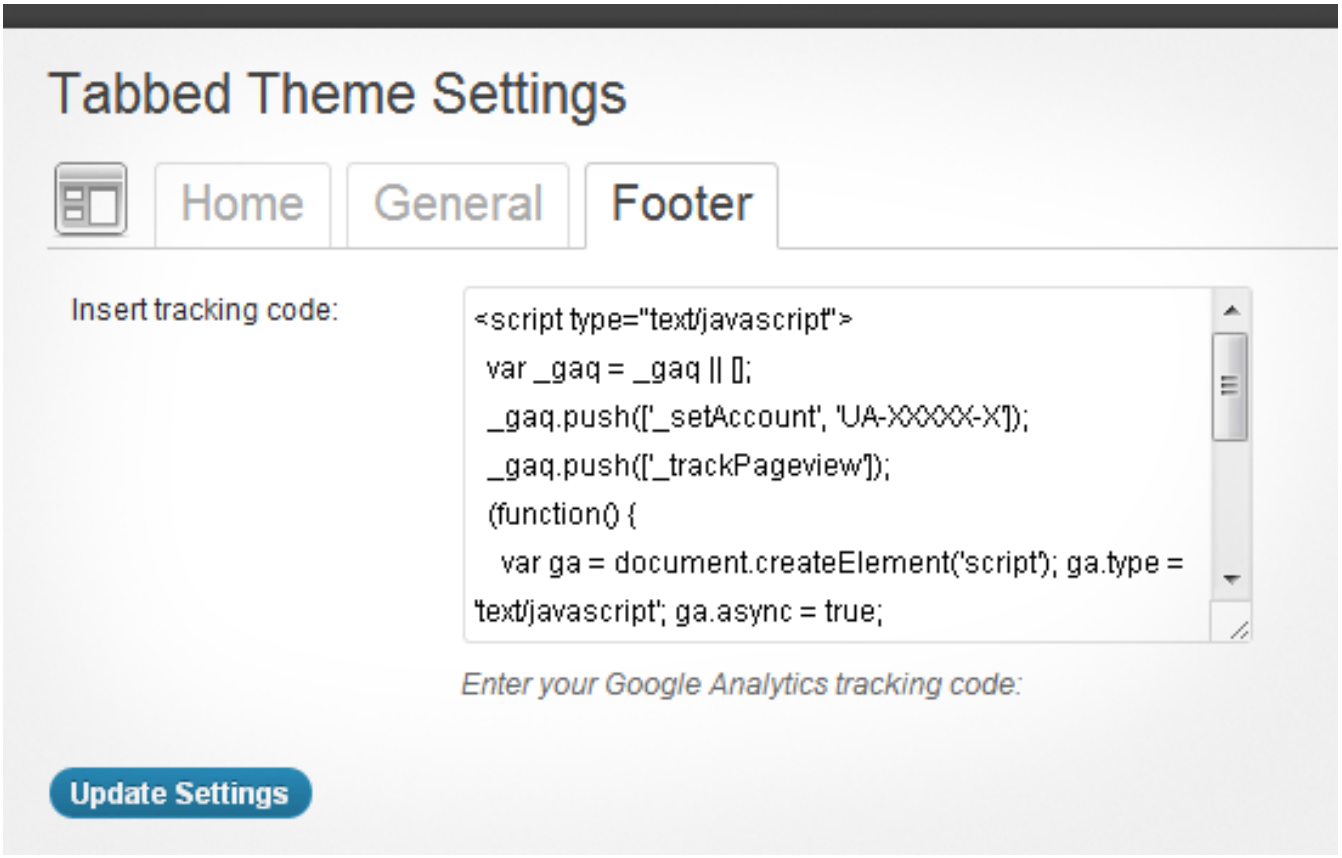
Creating The Tabs

The first snippet we will inspect is the code that produces the tabs:

```
function ilc_admin_tabs( $current = 'homepage' ) {
    $tabs = array( 'homepage' => 'Home Settings', 'general' =>
'General', 'footer' => 'Footer' );
    echo '<div id="icon-themes" class="icon32"><br></div>';
    echo '<h2 class="nav-tab-wrapper">';
    foreach( $tabs as $tab => $name ){
        $class = ( $tab == $current ) ? ' nav-tab-active' :
'';
        echo "<a class='nav-tab$class' href='?page=theme-
settings&tab=$tab'>$name</a>";

    }
    echo '</h2>';
}
```

This function will be called later in the content for the settings page. We first define an array that contains all of our tabs. The first tab, which is displayed first by default, is `homepage`, where we can set up some option for the appearance of the home page. Then we have `general`, which could be a page containing options used throughout the website, and, finally, `footer`, to include a tracking code in the footer.



We then set up the URL links for each tab and output them. Notice that if the tab is open, an additional class, `nav-tab-active`, is added.

Displaying The Tabbed Content

The content for the settings page is displayed in the callback function for `add_theme_page` (which is an abstraction of `add_submenu_page`, with the parent slug set to themes.php), which in our theme will be named `ilc_settings_page`. This is where you will call the function that we just went over.

```
function ilc_settings_page() {
   global $pagenow;
   $settings = get_option( "ilc_theme_settings" );

//generic HTML and code goes here

if ( isset ( $_GET['tab'] ) ) ilc_admin_tabs($_GET['tab']);
else ilc_admin_tabs('homepage');
```

If the tab is the default one, then `$_GET['tab']` is not defined, in which case the current tab will be homepage and, thus, the highlighted one. Otherwise, the highlighted tab will be the one defined in `$_GET['tab']`.

Following the same function, we now need to display the right set of fields. Depending on the value of `$tab`, we would display the fields for the settings tab for the home page or for one of the other tabs:

```
<form method="post" action="<?php admin_url( 'themes.php?
page=theme-settings' ); ?>">
<?php
wp_nonce_field( "ilc-settings-page" );

if ( $pagenow == 'themes.php' && $_GET['page'] == 'theme-
settings' ){

   if ( isset ( $_GET['tab'] ) ) $tab = $_GET['tab'];
   else $tab = 'homepage';

   echo '<table class="form-table">';
   switch ( $tab ){
      case 'general' :
         ?>
         <tr>
            <th>Tags with CSS classes:</th>
            <td>
               <input id="ilc_tag_class" name="ilc_tag_class"
type="checkbox" <?php if ( $settings["ilc_tag_class"] ) echo
'checked="checked"'; ?> value="true" />
               <label for="ilc_tag_class">Checking this will
output each post tag with a specific CSS class based on its
slug.</label>
```

```
            </td>
         </tr>
         <?php
      break;
      case 'footer' :
         ?>
         <tr>
            <th><label for="ilc_ga">Insert tracking code:</
label></th>
            <td>
               Enter your Google Analytics tracking code:
               <textarea id="ilc_ga" name="ilc_ga" cols="60"
rows="5"><?php echo
esc_html( stripslashes( $settings["ilc_ga"] ) ); ?></
textarea><br />

            </td>
         </tr>
         <?php
      break;
      case 'homepage' :
         ?>
         <tr>
            <th><label for="ilc_intro">Introduction</label></
th>
            <td>
               Enter the introductory text for the home page:
               <textarea id="ilc_intro" name="ilc_intro"
cols="60" rows="5" ><?php echo
esc_html( stripslashes( $settings["ilc_intro"] ) ); ?></
textarea>
            </td>
         </tr>
         <?php
      break;
   }
   echo '</table>';
}

?>
   <p class="submit" style="clear: both;">
      <input type="submit" name="Submit"  class="button-
primary" value="Update Settings" />
```

```
      <input type="hidden" name="ilc-settings-submit"
value="Y" />
   </p>
</form>
```

All of the settings will be stored in a single array in order to prevent several queries from being made.

Saving The Tabbed Fields

Now we need to know which slots of the array to save. Depending on the tab being displayed, certain options stored in the settings array will be displayed. If we just save all of the array slots, then we would overwrite some of the positions not shown in the current tab and thus not meant to be saved.`function ilc_save_theme_settings() {`

```
   global $pagenow;
   $settings = get_option( "ilc_theme_settings" );

   if ( $pagenow == 'themes.php' && $_GET['page'] == 'theme-
settings' ){
      if ( isset ( $_GET['tab'] ) )
          $tab = $_GET['tab'];
       else
          $tab = 'homepage';

       switch ( $tab ){
          case 'general' :
         $settings['ilc_tag_class'] = $_POST['ilc_tag_class'];
      break;
          case 'footer' :
         $settings['ilc_ga'] = $_POST['ilc_ga'];
      break;
      case 'homepage' :
         $settings['ilc_intro'] = $_POST['ilc_intro'];
      break;
       }
   }
   //code to filter html goes here
```

```
    $updated = update_option( "ilc_theme_settings",
$settings );
}
```

We've used a `switch` conditional again to query the value of `$tab` and store the right values in the array. After that, we've updated the option in the WordPress database.

Redirecting The User To The Right Tab

Now that the contents are saved, we need WordPress to redirect the user back to the appropriate tab on the settings page.

```
function ilc_load_settings_page() {
  if ( $_POST["ilc-settings-submit"] == 'Y' ) {
   check_admin_referer( "ilc-settings-page" );
   ilc_save_theme_settings();

   $url_parameters = isset($_GET['tab'])? 'updated=true&tab='.
$_GET['tab'] : 'updated=true';
   wp_redirect(admin_url('themes.php?page=theme-settings&'.
$url_parameters));
   exit;
  }
}
```

Depending on whether the `$_GET['tab']` variable is set, we use `wp_redirect` to send the user either to the default tab or to one of the other tabs.

Now our tabs are working, displaying the right set of fields, saving the right fields, and then redirecting the user to the correct tab.

Download The Theme

Almost any theme with a moderate number of options would benefit from tabs on the settings page. Just remember that this is one approach. Another approach would be to add several collapsable meta boxes, as seen on the page for writing posts, and to automatically collapse the boxes that are not frequently used. However, tabs enable you to better separate each set of options.

Finally, here is the theme, so that you can take a closer look:

- Theme with tabbed settings page

The theme also implements the function whereby each tag is outputted with a unique CSS class, so you can check that out, too.

Create Native Admin Tables In WordPress The Right Way

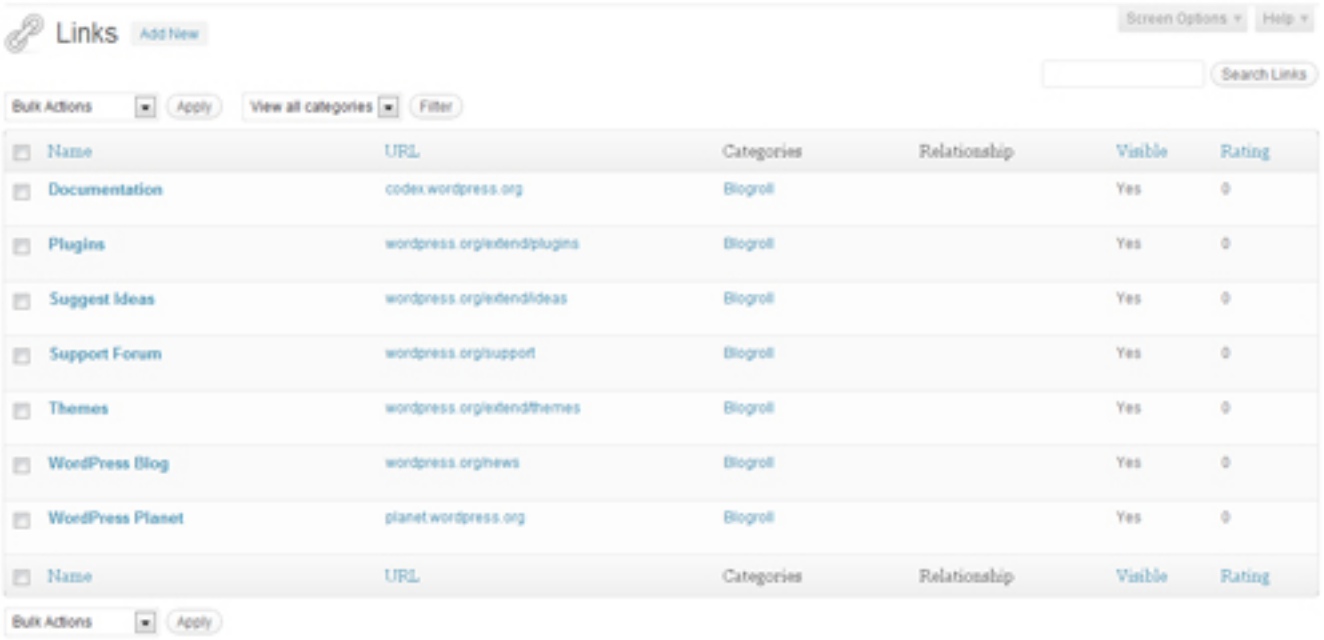
Jeremy Desvaux de Marigny

List tables are a common element in WordPress' administration interface. They are used on nearly all default admin pages with lists, and developers often integrate them into their plugins. But creating one of these tables is not really intuitive if you haven't done it before, and I've seen people try to replicate it by using WordPress CSS classes in custom markup and even by replicating the CSS from scratch.

In this chapter, we'll see how WordPress provides functionality that can be used to generate native admin tables. We'll look at a typical WordPress table and its different components and show how to implement it the right way.

Presentation Of A WordPress Table

To better understand the various elements we'll be talking about, let's take the default link manager that you see when you click "Links" in the admin menu. Here's what you see:



The default page for managing links in WordPress 3.2.

As you can see, a few different elements precede the table that enable you to perform actions on the table. Then we have the table's header, the rows, the table's footer and, finally, some more actions.

BEFORE AND AFTER THE TABLE

WordPress' admin interface is consistent, so you'll get used to finding elements in certain places as you navigate.

Before and after the admin tables, for example, are where you would usually find options to take action on the table. These include bulk actions, which enable you to edit and delete multiple posts and to filter the list based on a certain criteria.

We'll see in the second part of this chapter how to interact with these two areas and how to display options there.

HEADER AND FOOTER

Speaking of consistency, every admin table in WordPress has a header and footer.

Following the same logic, they display the the same information: the titles of the columns. Some of the titles are simple and some are linked (meaning that the table can be ordered according to that column).

THE CONTENT

Obviously, the reason you would create a table is to put some content in it. This content would go in the rows between the header and footer.

How Is This Done In WordPress?

As we've just seen, a WordPress table has three families of elements. Let's see how to achieve this, using a concrete example.

OUR EXAMPLE TABLE

Most of the time, the data we want to display will be in the form of a SQL table. We'll use the default links table in WordPress as our example, but the concepts apply to any database table and could easily be adapted to your needs. Our table will have the following structure:

	Field	Type	Collation	Attributes	Null	Default	Extra	Action
	link_id	bigint(20)		UNSIGNED	No		auto_increment	
	link_url	varchar(255)	utf8_general_ci		No			
	link_name	varchar(255)	utf8_general_ci		No			
	link_image	varchar(255)	utf8_general_ci		No			
	link_target	varchar(25)	utf8_general_ci		No			
	link_description	varchar(255)	utf8_general_ci		No			
	link_visible	varchar(20)	utf8_general_ci		No	Y		
	link_owner	bigint(20)		UNSIGNED	No	1		
	link_rating	int(11)			No	0		
	link_updated	datetime			No	0000-00-00 00:00:00		
	link_rel	varchar(255)	utf8_general_ci		No			
	link_notes	mediumtext	utf8_general_ci		No			
	link_rss	varchar(255)	utf8_general_ci		No			

This table contains some default data that will be perfect for testing.

USING THE LIST TABLE CLASS

To create an HTML table in WordPress, we don't have to write a lot of HTML. Instead, we can rely on the precious work of the `WP_List_Table` class. As explained by the WordPress Codex, this class is a powerful tool for generating tables.

It is tailored for back-end developers, so we can concentrate on the most essential task (the treatment of the data), leaving the other tasks (such as HTML rendering) to WordPress.

The `WP_List_Table` class is essentially a little framework whose functionality we can rely on to prepare our table. This is an object-oriented approach, because we'll be creating an object that extends

`WP_List_Table` and using that, instead of using `WP_List_Table` directly.

Let's create a class `Link_List_Table` with a simple constructor:

```
class Link_List_Table extends WP_List_Table {

  /**
   * Constructor, we override the parent to pass our own
arguments
   * We usually focus on three parameters: singular and plural
labels, as well as whether the class supports AJAX.
   */
   function __construct() {
      parent::__construct( array(
     'singular'=> 'wp_list_text_link', //Singular label
     'plural' => 'wp_list_test_links', //plural label, also
this well be one of the table css class
     'ajax' => false //We won't support Ajax for this table
     ) );
   }

}
```

This is the starting point of our table. We now have an object that has access to the properties and methods of its parent, and we'll customize it to suit our needs.

Keeping in mind the three types of elements we saw earlier, let's see now what to add to our class to get the same result.

HOW TO ADD ELEMENTS BEFORE AND AFTER THE TABLE

To display content before or after the table, we need to add a method named `extra_tablenav` to our class. This method can be implemented as follows:

```
/**
 * Add extra markup in the toolbars before or after the list
 * @param string $which, helps you decide if you add the
markup after (bottom) or before (top) the list
 */
function extra_tablenav( $which ) {
  if ( $which == "top" ){
      //The code that goes before the table is here
      echo"Hello, I'm before the table";
  }
  if ( $which == "bottom" ){
      //The code that goes after the table is there
      echo"Hi, I'm after the table";
  }
}
```

The interesting thing here is that the `extra_tablenav` method takes one parameter, named `$which`, and this function is called twice by `Link_List_Table`, (once before the table and once after). When it's called before, the value of the $which parameter is `top`, and when it's called a second time, after the table, its value is `bottom`.

You can then use this to position the various elements that you'd like to appear before and after the table.

This function exists in the parent `WP_List_Table` class in WordPress, but it doesn't return anything, so if you don't override it, nothing bad will happen; the table just won't have any markup before or after it.

HOW TO PREPARE THE TABLE'S HEADER AND FOOTER

In the header and footer, we have the column's headers, and some of them are sortable.

We'll add to our class a method named `get_columns` that is used to **define the columns**:

```
/**
 * Define the columns that are going to be used in the table
 * @return array $columns, the array of columns to use with
the table
 */
function get_columns() {
  return $columns= array(
      'col_link_id'=>__('ID'),
      'col_link_name'=>__('Name'),
      'col_link_url'=>__('Url'),
      'col_link_description'=>__('Description'),
      'col_link_visible'=>__('Visible')
  );
}
```

The code above will build an array in the form of `'column_name'=>'column_title'`. This array would then be used by your class to display the columns in the header and footer, in the order you've written them, so defining that is pretty straightforward.

Plenty of fields are in the links table, but not all of them interest us. With our `get_columns` method, we've chosen to display only a few of them: the ID, the name, the URL, the description of the link, as well as whether the link is visible.

Unlike the `extra_tablenav` method, the `get_columns` is a parent method that must be overridden in order to work. This makes sense, because if you don't declare any columns, the table will break.

To specify the columns to which to add **sorting functionality**, we'll add the `get_sortable` columns method to our class:

```
/**
 * Decide which columns to activate the sorting functionality
on
 * @return array $sortable, the array of columns that can be
sorted by the user
 */
public function get_sortable_columns() {
  return $sortable = array(
      'col_link_id'=>'link_id',
      'col_link_name'=>'link_name',
      'col_link_visible'=>'link_visible'
  );
}
```

Here again, we've built a PHP array. The model for this one is `'column_name'=>'corresponding_database_field'`. In other words, the column_name must be the same as the column name defined in the get_columns method, and the corresponding_database_field must be the same as the name of the corresponding field in the database table.

The code we have just written specifies that we would like to add sorting functionality to three columns ("ID," "Name" and "Visible"). If you don't want the user to be able to sort any columns or if you just don't want to implement this method, WordPress will just assume that no columns are sortable.

At this point, our class is ready to handle quite a few things. Let's look now at how to display the data.

HOW TO DISPLAY THE TABLE'S ROWS

The first steps in preparing the list table are very quick. We just have to tackle a few more things in the treatment of data.

To make the list table display your data, you'll need to prepare your items and assign them to the table. This is handled by the `prepare_items` method:

```
/**
 * Prepare the table with different parameters, pagination,
columns and table elements
 */
function prepare_items() {
  global $wpdb, $_wp_column_headers;
  $screen = get_current_screen();

  /* -- Preparing your query -- */
        $query = "SELECT * FROM $wpdb->links";

  /* -- Ordering parameters -- */
      //Parameters that are going to be used to order the
result
      $orderby = !empty($_GET["orderby"]) ?
mysql_real_escape_string($_GET["orderby"]) : 'ASC';
      $order = !empty($_GET["order"]) ?
mysql_real_escape_string($_GET["order"]) : '';
      if(!empty($orderby) & !empty($order)){ $query.=' ORDER
BY '.$orderby.' '.$order; }

  /* -- Pagination parameters -- */
        //Number of elements in your table?
        $totalitems = $wpdb->query($query); //return the total
number of affected rows
        //How many to display per page?
        $perpage = 5;
        //Which page is this?
        $paged = !empty($_GET["paged"]) ?
mysql_real_escape_string($_GET["paged"]) : '';
        //Page Number
        if(empty($paged) || !is_numeric($paged) || $paged<=0 )
{ $paged=1; }
        //How many pages do we have in total?
        $totalpages = ceil($totalitems/$perpage);
        //adjust the query to take pagination into account
      if(!empty($paged) && !empty($perpage)){
          $offset=($paged-1)*$perpage;
```

```
            $query.=' LIMIT '.(int)$offset.','.(int)$perpage;
        }

    /* -- Register the pagination -- */
        $this->set_pagination_args( array(
            "total_items" => $totalitems,
            "total_pages" => $totalpages,
            "per_page" => $perpage,
        ) );
        //The pagination links are automatically built according
to those parameters

    /* -- Register the Columns -- */
        $columns = $this->get_columns();
        $_wp_column_headers[$screen->id]=$columns;

    /* -- Fetch the items -- */
        $this->items = $wpdb->get_results($query);
}
```

As you can see, this method is a bit more complex than the previous ones we added to our class. So, let's see what is actually happening here:

1. **Preparing the query**
   The first thing to do is specify the general query that will return the data. Here, it's a generic `SELECT` on the links table.

2. **Ordering parameters**
   The second section is for the ordering parameters, because we have specified that our table can be sorted by certain fields. In this section, we are getting the field (if any) by which to order our record (`$_GET['order']`) and the order itself (`$_GET['orderby']`). We then adjust our query to take those into account by appending an `ORDER BY` clause.

3. **Pagination parameters**
   The third section deals with pagination. We specify how many items are

in our database table and how many to show per page. We then get the current page number (`$_GET['paged']`) and then adapt the SQL query to get the correct results based on those pagination parameters.

4. **Registration**
   This part of the function takes all of the parameters we have prepared and assigns them to our table.

5. **Ready to go**
   Our list table is now set with all of the information it needs to display our data. It knows what query to execute to get the records from the database; it knows how many records will be returned; and all the pagination parameters are ready. This is an essential method of your list table class. If you don't implement it properly, WordPress won't be able to retrieve your data. If the method is missing in your class, WordPress will return an error telling you that the prepare_items method must be overridden.

6. **Displaying the rows**
   This is it! Finally, we get to the method responsible for displaying the records of data. It is named `display_rows` and is implemented as follows.

```
/**
 * Display the rows of records in the table
 * @return string, echo the markup of the rows
 */
function display_rows() {

  //Get the records registered in the prepare_items method
  $records = $this->items;

  //Get the columns registered in the get_columns and
get_sortable_columns methods
  list( $columns, $hidden ) = $this->get_column_info();
```

```
  //Loop for each record
  if(!empty($records)){foreach($records as $rec){

      //Open the line
         echo '< tr id="record_'.$rec->link_id.'">';
      foreach ( $columns as $column_name =>
$column_display_name ) {

          //Style attributes for each col
          $class = "class='$column_name column-$column_name'";
          $style = "";
          if ( in_array( $column_name, $hidden ) ) $style = '
style="display:none;"';
          $attributes = $class . $style;

          //edit link
          $editlink  = '/wp-admin/link.php?
action=edit&link_id='.(int)$rec->link_id;

          //Display the cell
          switch ( $column_name ) {
              case "col_link_id":   echo '< td '.
$attributes.'>'.stripslashes($rec->link_id).'< /td>'; break;
              case "col_link_name": echo '< td '.
$attributes.'><strong><a href="'.$editlink.'"
title="Edit">'.stripslashes($rec->link_name).'</a></strong>< /
td>'; break;
              case "col_link_url": echo '< td '.
$attributes.'>'.stripslashes($rec->link_url).'< /td>'; break;
              case "col_link_description": echo '< td '.
$attributes.'>'.$rec->link_description.'< /td>'; break;
              case "col_link_visible": echo '< td '.
$attributes.'>'.$rec->link_visible.'< /td>'; break;
          }
      }

      //Close the line
      echo'< /tr>';
  }}
}
```

This function gets the data prepared by the `prepare_items` method and loops through the different records to build the markup of the corresponding table row.

With this method, you have great control over how to display the data. If you do not wish to add this method to your class, then the class will use the parent's method to render the data in WordPress' default style.

Your list table class is now finished and ready to be used on one of your pages.

All of the methods we've added to our class already exist in the parent `WP_List_Table` class. But for your child class to work, you must override at least two of them: `get_columns` and `prepare_items`.

Implementation

Now that our list table class is ready, let's see how we can use it on a page of our choice.

WHERE DO WE WRITE IT?

The code that we'll cover in this section has to be written on the page where you want to display the admin table.

We'll create a very simple demonstration plugin, named "Test WP List Table." Basically, this plugin will add a link in the WordPress "Plugins" sub-menu. Our code will, therefore, be written in the plugin file.

BEFORE WE BEGIN

Important: the `WP_List_Table` class is not available in plugins by default. You can use the following snippet to check that it is there:

```
//Our class extends the WP_List_Table class, so we need to
make sure that it's there
if(!class_exists('WP_List_Table')){
  require_once( ABSPATH . 'wp-admin/includes/class-wp-list-
table.php' );
}
```

Also, if you have created your `Links_List_Table` class in an external file, make sure to include it before you start instantiating.

INSTANTIATE THE TABLE

The first step is to create an instance of our list table class, then call the `prepare_items` method to fetch the data to your table:

```
//Prepare Table of elements
$wp_list_table = new Links_List_Table();
$wp_list_table->prepare_items();
```

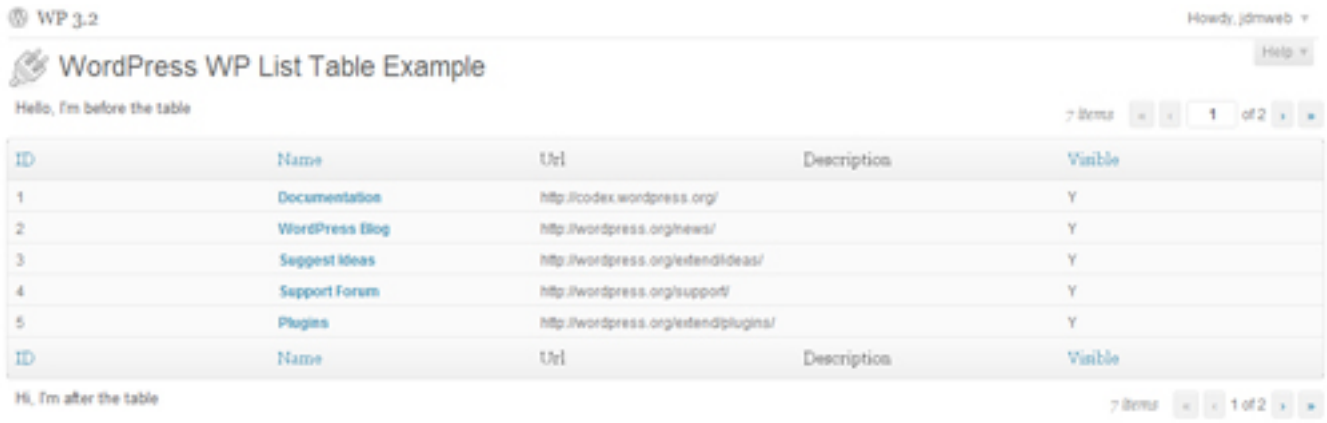
DISPLAY IT

The `$wp_list_table` object is now ready to display the table wherever you want.

Build your page's markup, and wherever you decide to display the table, make a call to the display() method:

```
//Table of elements
$wp_list_table->display();
```

Calling the display() method will generate the full markup of the list table, from the before-and-after area to the table's content, and including the table's header and footer. It also automatically generates all pagination links for you, so the result should look like this:



In the download accompanying this article, you'll find the complete PHP file containing the class definition and the example of its implementation. It is named testWPListTable.php, and it is written in the form of a simple plugin that you can put in your WordPress plugin folder and activate if you want to see what it does.

Conclusion

Creating a PHP class merely to display a table of data might seem like overkill. But this class is very easy to create and customize. And once it's done, you'll be happy that the parts of tables that are difficult to implement, such as pagination and reordering, are now taken care of.

Also, because the generated markup is exactly what WordPress supports, if an update is released one day, your tables will remain in good shape.

The PHP code we've used is clean and easy to understand. And mastering the default functionality won't take a long time.

What we've seen today is the basic implementation of a WordPress list table, but you can add other supported methods to the class for extra functionality.

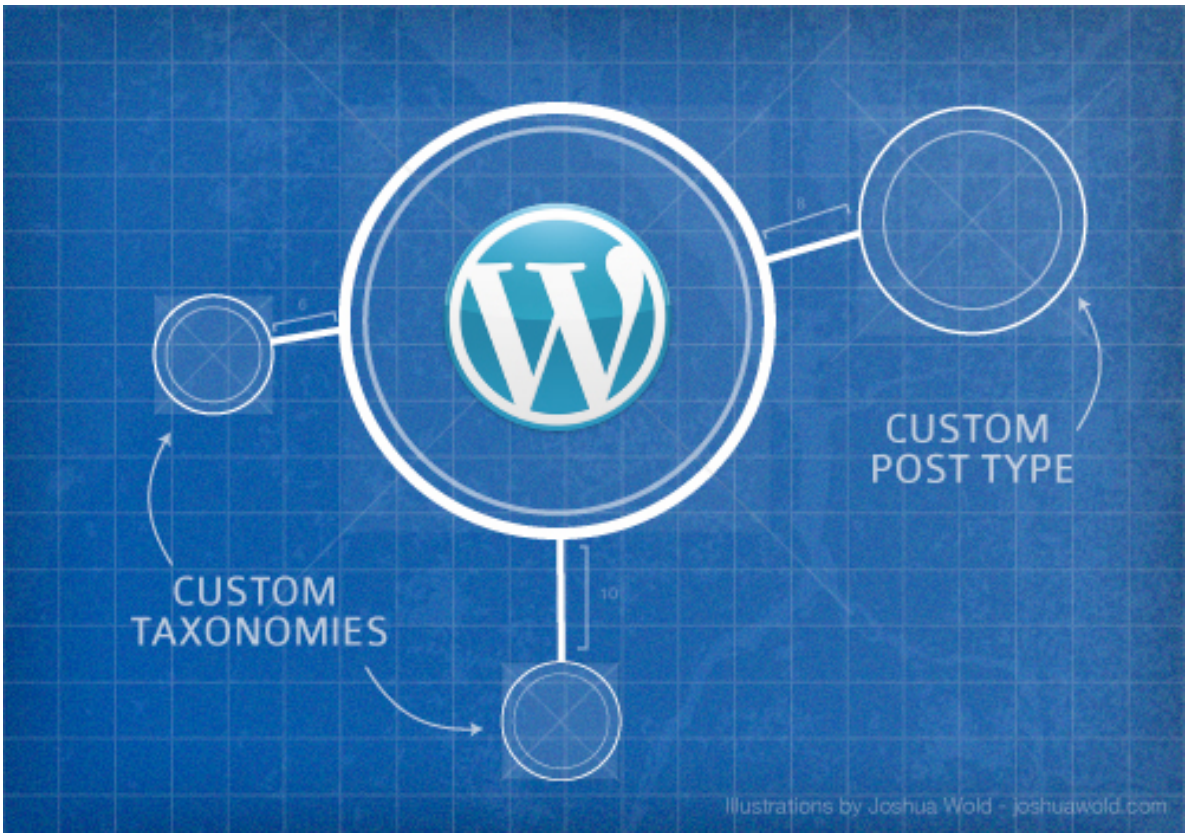
For more information, read the Codex page dedicated to WordPress list tables, and have a look at another custom list table example.

I hope you've found this article useful, and I wish you good luck with list tables!

How To Build A Media Site On WordPress – Part 1

Jonathan Wold

WordPress is amazing. With its growing popularity and continual development, it is becoming the tool of choice for many designers and developers. WordPress projects, though, are pushing well beyond the confines of mere “posts” and “pages”. How do you go about adding and organizing media and all its complexities? With the introduction of WordPress 3.1, several new features were added that make using WordPress to manage media even more practical and in this tutorial, we’re going to dive in and show you how.



In part 1, we're going to setup custom post types and custom taxonomies, without plugins. After that, we'll build a template to check for and display media attached to custom posts. Then, in part two, we'll use custom taxonomy templates to organize and relate media (and other types of content).

As we focus on building a media centric site, I also want you to see that the principles taught in this series offer you a set of tools and experience to build interfaces for and organize many different types of content. Examples include:

- A "Media" center, of any type, added to an existing WordPress site
- A repository of videos, third party hosted (e.g. Vimeo, YouTube, etc), organized by topics and presenters

- A music site, with streaming and song downloads, organized by bands and associated by albums
- An author-driven Q&A site, with user submitted questions organized by topics and geographical location
- A recipe site with videos and visitor ratings, organized by category and shared ingredients

In a future tutorial, we will focus on customizing the WordPress backend (with clients especially in mind) to manage a media site and in another tutorial we will use the foundation laid to build a dynamic filtering interface that allows visitors to quickly sort their way through hundreds or even thousands of custom posts.

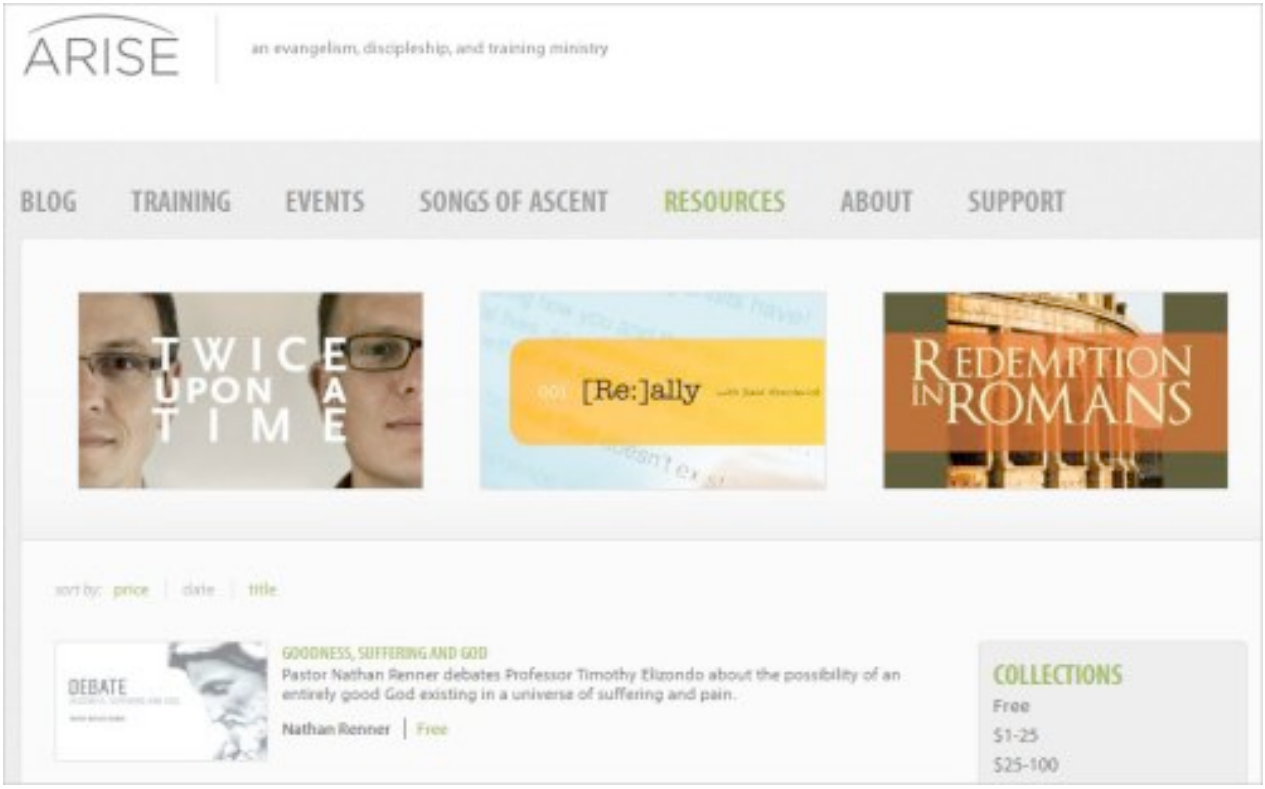
REQUIREMENTS

- **WordPress 3.1** – With the release of 3.1, several new features related to the use of custom post types and taxonomies were introduced that are essential to the techniques taught in this series.
- **Basic Familiarity with PHP** (or "No Fear") – To move beyond copying and pasting the examples I've given will require a basic familiarity with PHP or, at least, a willingness to experiment. If the code samples below are intimidating to you and you have the desire to learn, then I encourage you to tackle it and give it your best. If you have questions, ask in the comments.

WORKING EXAMPLE

In April, 2011 we (Sabramedia, of which I am a co-founder) worked with an organization in Southern California to develop a resource center on

WordPress to showcase their paid and free media products. On the front-end, we built a jQuery powered filtering interface to allow visitors to filter through media on-page. We'll cover the ins and outs of building a similar interface in part three.



The "Resource Center" on ARISE, with a custom taxonomy filter ("David Asscherick") pre-selected.

Working With Custom Post Types

By default, WordPress offers two different types of posts for content. First, you have the traditional "post", used most often for what WordPress is known best for – blogging. Second, you have "pages". Each of these, as far as WordPress is concerned, is a type of "post". A custom post type is a type of post that you define.

Note: You can learn more about post types on the WordPress Codex.

In this series, we are going to use custom post types to build a media based resource center. I will be defining and customizing a post type of "resource".

SETTING UP YOUR CUSTOM POST TYPE

You can setup your custom post types by code or by plugin. In these examples, I will be setting up the post type by code, storing and applying the code directly in the functions file on the default WordPress theme, Twenty Ten. You can follow along by using a plugin to setup the post types for you or by copying the code samples into the bottom of your theme's custom functions file (functions.php).

> *Note: As a best practice, unless you use an existing plugin to create the post types, you may want to consider creating your own WordPress plugin. Setting up custom post types and taxonomies separate from your theme becomes important if and when you want to make major changes to your theme or try a new theme out. Want to save some typing? Use the custom post code generator.*

Alright, let's setup our custom post type. Paste the following code into your theme's functions.php:

```
add_action('init', 'register_rc', 1); // Set priority to avoid
plugin conflicts

function register_rc() { // A unique name for our function
  $labels = array( // Used in the WordPress admin
      'name' => _x('Resources', 'post type general name'),
      'singular_name' => _x('Resource', 'post type singular
name'),
```

```
        'add_new' => _x('Add New', 'Resource'),
        'add_new_item' => __('Add New Resource'),
        'edit_item' => __('Edit Resource'),
        'new_item' => __('New Resource'),
        'view_item' => __('View Resource '),
        'search_items' => __('Search Resources'),
        'not_found' =>  __('Nothing found'),
        'not_found_in_trash' => __('Nothing found in Trash')
    );
    $args = array(
        'labels' => $labels, // Set above
        'public' => true, // Make it publicly accessible
        'hierarchical' => false, // No parents and children here
        'menu_position' => 5, // Appear right below "Posts"
        'has_archive' => 'resources', // Activate the archive
        'supports' =>
array('title','editor','comments','thumbnail','custom-
fields'),
    );
    register_post_type( 'resource', $args ); // Create the post
type, use options above
}
```

The code above tells WordPress to “register” a post type called “resource”. Then, we pass in our options, letting WordPress know that we want to use our own labels, that we want our post type to be publicly accessible, non-hierarchal, and that we want it to show up right below “posts” in our admin menu. Then, we activate the “archive” feature, new in WordPress 3.1. Finally, we add in “supports”: the default title field, the WordPress editor, comments, featured thumbnail, and custom fields (I’ll explain that later).

> *Note: For more information on setting up the post type and for details on all the options you have (there are quite a few available), refer to the register_post_type function reference on the WordPress Codex.*

If the code above was successful, you will see a new custom post type, appearing below “Posts” in the WordPress admin menu. It will look something like this:

A view of the WordPress Admin, after adding a custom post type

We’re in good shape! Next, let’s setup our custom taxonomies.

Working With Custom Taxonomies

A “taxonomy” is a way of organizing and relating information. WordPress offers two default taxonomies, categories and tags. Categories are hierarchal (they can have sub-categories) and are often used to organize content on a more broad basis. Tags, are non-hierarchal (no sub-tags) and are often used to organize content across categories.

A “term” is an entry within a taxonomy. For a custom taxonomy of “Presenters”, “John Smith” would be a term within that taxonomy.

In this series, we will be creating two different custom taxonomies to organize the content within our resource center.

- Presenters – Each media item in our resource center will have one or more presenters. For each presenter, we want to know their name and we want to include a short description. Presenters will be non-hierarchal.
- Topics – Our resource center will offer media organized by topics. Topics will be hierarchal, allowing for multiple sub-topics and even sub-sub-topics.

Note: Interested in working with more than the title and short description? Take a look at How To Add Custom Fields To Custom Taxonomies on the Sabramedia blog.

SETTING UP PRESENTERS

Our goal with presenters is to create a presenter profile, referenced on the respective media pages, that will give more information about the presenter and cross-reference other resources that they are associated with.

Add the following code to your theme's functions.php file:

```
$labels_presenter = array(
  'name' => _x( 'Presenters', 'taxonomy general name' ),
  'singular_name' => _x( 'Presenter', 'taxonomy singular
name' ),
  'search_items' =>  __( 'Search Presenters' ),
  'popular_items' => __( 'Popular Presenters' ),
  'all_items' => __( 'All Presenters' ),
  'edit_item' => __( 'Edit Presenter' ),
  'update_item' => __( 'Update Presenter' ),
  'add_new_item' => __( 'Add New Presenter' ),
  'new_item_name' => __( 'New Presenter Name' ),
  'separate_items_with_commas' => __( 'Separate presenters
with commas' ),
  'add_or_remove_items' => __( 'Add or remove presenters' ),
```

```
  'choose_from_most_used' => __( 'Choose from the most used
presenters' )
);

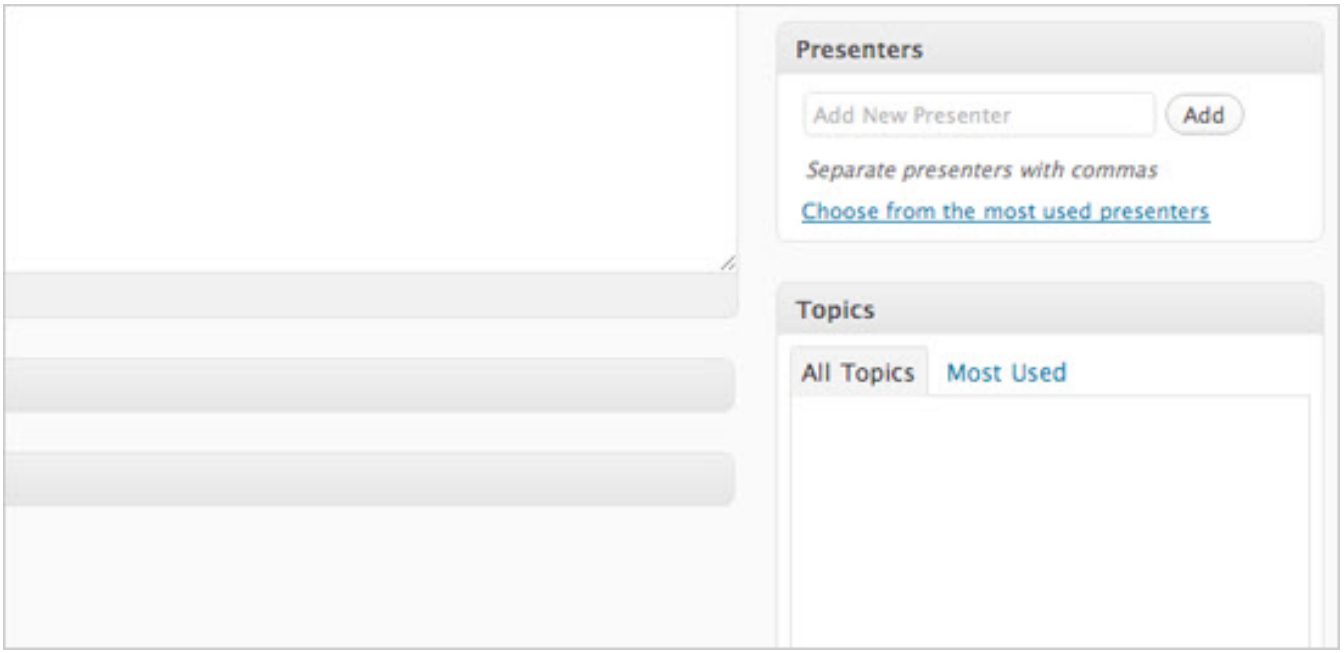
register_taxonomy(
  'presenters', // The name of the custom taxonomy
  array( 'resource' ), // Associate it with our custom post
type
  array(
      'rewrite' => array( // Use "presenter" instead of
"presenters" in the permalink
          'slug' => 'presenter'
          ),
      'labels' => $labels_presenter
      )
  );
```

Let's break that down. First, we setup the labels to be used when we "register"our taxonomy. Then, we give it a name, in this case "presenters", and assign it to the post type of "resource". If you had multiple post types, you would add them in with a comma, like this:

```
array( 'resource', 'other-type' ), // Associate it with our
custom post types
```

After that, we change the URL (or "permalink") to satisfy our desire for grammatical excellence. Rather than being "/presenters/presenter-name" we update the "slug" (what is a slug?) to remove the "s" so that the permalink will read "/presenter/presenter-name".

In our example, you should now notice a new menu option labeled "Presenters" under "Resources" in the admin sidebar. When you go to create a new resource you should also notice a meta box on the right side that looks like this:



My custom taxonomy of "Presenters" now shows up between the "Publish" box and "Featured Image".

> *Note: To learn more about setting up custom taxonomies and the options available, take a look at the register_taxonomy function reference on the WordPress Codex.*

SETTING UP TOPICS

Our goal with topics is to allow for a hierarchal set of topics and sub-topics, each with their own page, showing the resources that are associated with each respective topic.

Add the following code to your theme's functions.php file:

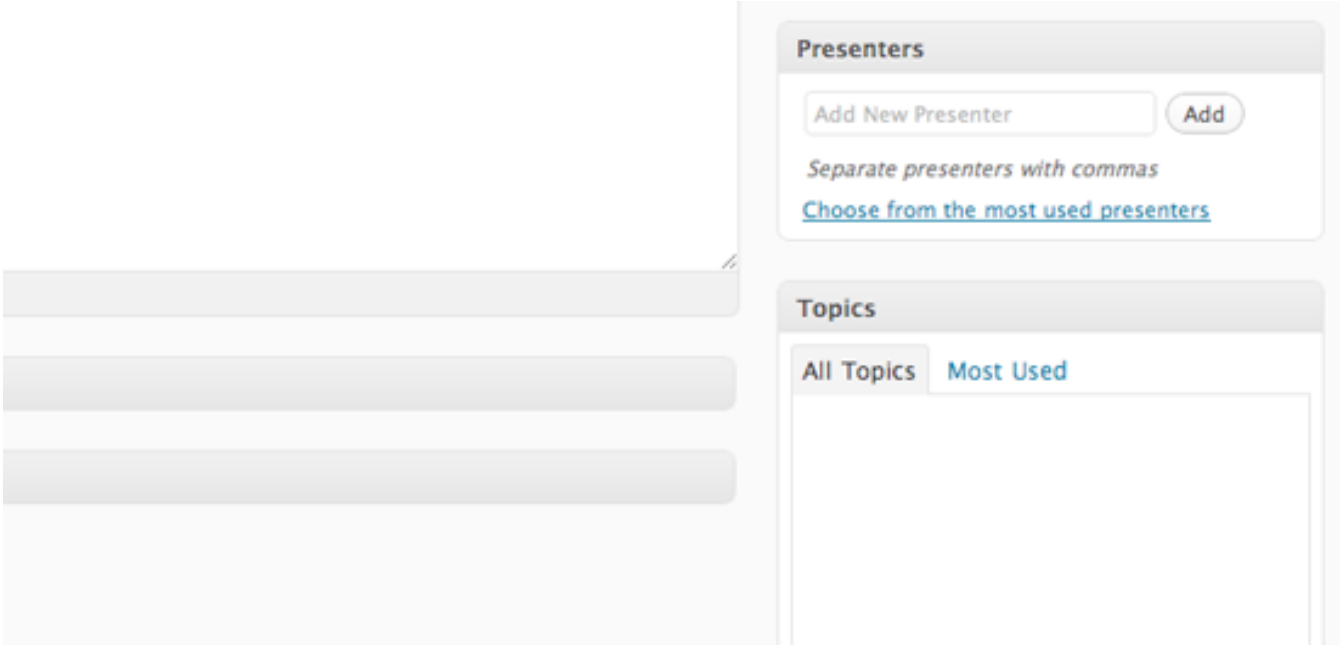
```
$labels_topics = array(
  'name' => _x( 'Topics', 'taxonomy general name' ),
  'singular_name' => _x( 'Topic', 'taxonomy singular name' ),
  'search_items' =>  __( 'Search Topics' ),
  'all_items' => __( 'All Topics' ),
  'parent_item' => __( 'Parent Topic' ),
```

```
  'parent_item_colon' => __( 'Parent Topic:' ),
  'edit_item' => __( 'Edit Topic' ),
  'update_item' => __( 'Update Topic' ),
  'add_new_item' => __( 'Add New Topic' ),
  'new_item_name' => __( 'New Topic Name' ),
);

register_taxonomy(
  'topics', // The name of the custom taxonomy
  array( 'resource' ), // Associate it with our custom post
type
  array(
      'hierarchical' => true,
      'rewrite' => array(
          'slug' => 'topic', // Use "topic" instead of "topics"
in permalinks
          'hierarchical' => true // Allows sub-topics to appear
in permalinks
          ),
      'labels' => $labels_topics
      )
  );
```

That was easy enough! The code above is similar to setting up presenters, except this time we are using a few different labels, specific to hierarchal taxonomies. We set hierarchal to true (it's set to "false" by default), we update the slug to be singular instead of plural, then, just before referencing our labels, we set the rewriting to be hierarchal. A hierarchal rewrite allows permalinks that look like this: /topic/topic-name/sub-topic-name.

With the above code implemented, you should notice another option below "Resources" in the WordPress admin and a new meta box that looks like this:



My custom taxonomy of "Topics" now shows up, albeit a bit empty looking, below "Presenters".

Adding Custom Fields To Custom Post Types

In many cases, the "title" and "editor" (the default content editor in WordPress) aren't going to be enough. What if you want to store extra information about a particular custom post? Examples might include:

- Duration of a media file – HH:MM:SS format, useful to pre-populate your media player with the duration on page load.
- Original recording date – Stored as a specific date with day, month, and year.

We call this "meta" information and it is a set of details that are specific to the individual item and usually make the most sense to store as meta data, as opposed to terms within a custom taxonomy. While you could put all

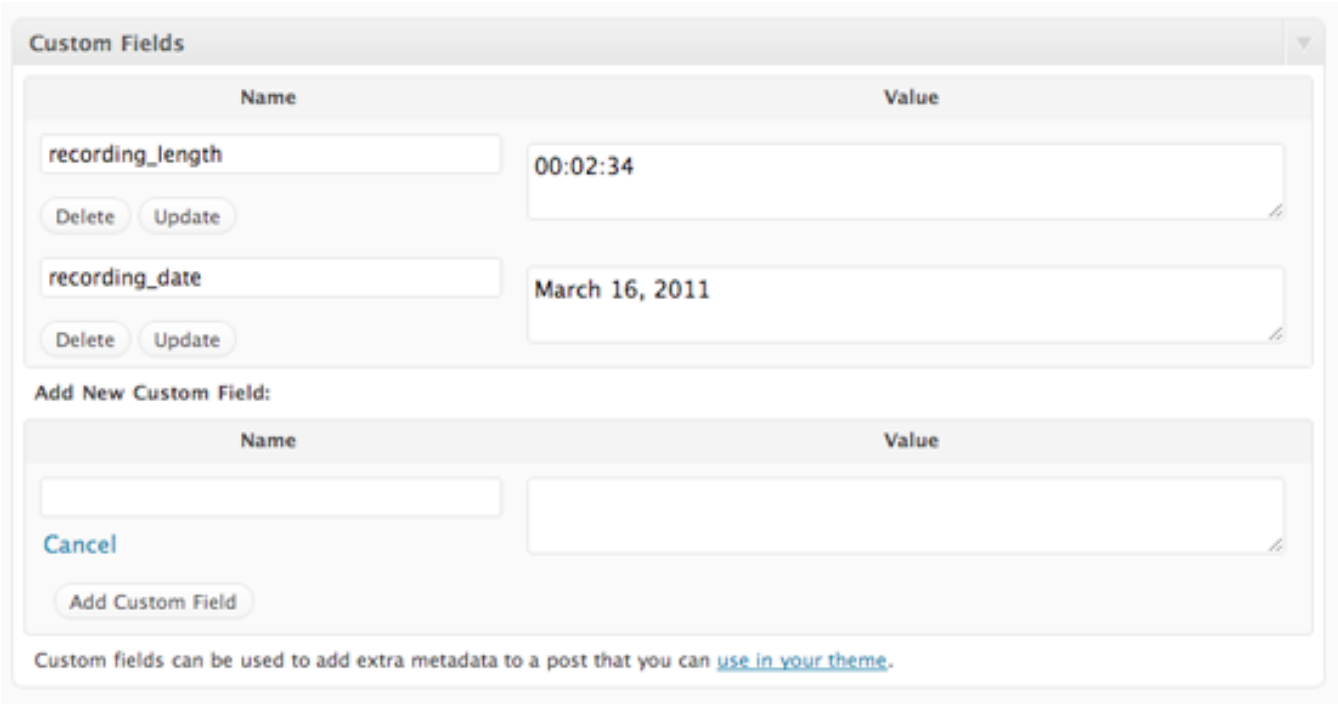
these details in the “editor” field, it gives you very little flexibility with how this is displayed within your template.

So, let’s **setup some custom fields**. Use the custom fields interface at the bottom of an individual custom post to add some extra details about your custom post.

For our example, we’re going to add two fields. For each field, I will list the name, then an example value:

- **recording_length** - Example: 00:02:34
- **recording_date** – Example: March 16, 2011

Here’s how that looks after adding two custom fields:



An example of the custom fields interface after adding two “keys” and their respective “values”

> ***Note**: The default custom fields interface can be a bit limiting. If you'd like to make use of a plugin, try More Fields. The functionality is the same (just be mindful of what you name your custom fields) – a plugin typically offers you a better interface. If you want to build your own interface, take a look at WP Alchemy. To learn more about using custom fields, take a look at using custom fields on the WordPress Codex.*

CUSTOM TAXONOMIES VS. CUSTOM FIELDS

At this point, you may run into a situation where you're uncertain whether a particular piece of information should be stored as a custom taxonomy or as a custom field. Let's use the recording date as an example. If we were to log the complete date, then it would probably make the most sense to store it within a custom field on the individual item. If we were to just use the year, though, we could store it as a term within a custom taxonomy (we'd probably call it "year") and use it to show other resources recorded that same year.

The question is whether or not you want to relate content (in our case, "resources") by the information you're considering. If you don't see any need to relate content (and don't have plans to) then a custom field is the way to go. If you have a need to relate content or see a potential need down the road, then a custom taxonomy is the way to go.

Media Storage – WordPress vs. Third Party

Now that we have our custom post type and custom taxonomies in place, it's time to upload some media. Our goal is to make this as simple a process for the end-user as possible. There are two ways that we can manage the media, either directly within WordPress or via third party.

- **WordPress Managed** - WordPress has a media management system built-in. You can upload media directly from your post type interface or from the “media” section in the WordPress admin. If storage or bandwidth becomes an issue, you can use a plugin (such as WP Super Cache) to offload the storage of the media to a third party content delivery network (CDN) to optimize delivery speed and save on bandwidth.
- **Third Party** – Going this route, you can use a media hosting service like YouTube, Vimeo, Scribd (PDFs), Issuu (ebooks), or any media hosting service that offers you an embed option.

Going the internal route, the media is stored inside of WordPress and associated with the individual custom post. We then access it as an attachment within the template. Going the third party route, we get the embed code (or the media ID) and store it inside of WordPress within a custom field. We’ll look at examples of both options further on.

Preparing The Stage – Adding New Media

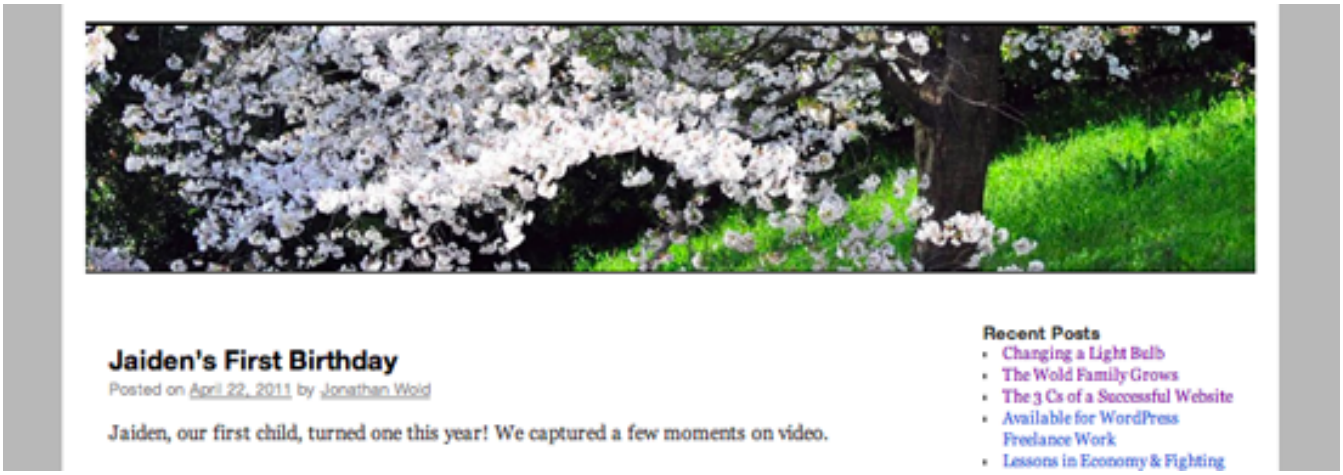
We’re about to start working with the templates. Before we do that, though, we need to have some media to work with within our new custom post type. Before proceeding, make sure you’ve done the following:

- Create a new “resource” post (or whatever your post type may be) and give it a title and a description in the main content editor.
- Associate your resource with a non-hierarchical custom taxonomy you’ve created (e.g. A presenter named “Jonathan Wold”).
- Associate your resource with a hierarchical custom taxonomy you’ve created (e.g. A topic of “Family” and a sub-topic of “Children”)

- Add one or more custom fields with a unique “key” and “value” (e.g. a key of “recording_duration” and a value of “00:02:34″).
- Upload a video file to your custom post using the WordPress media manager (click the “video” icon just below the title field and right above the editor).

***Note #1**: If you’re hosting your videos via third party, create a custom field to hold either the entire embed code or the ID of the video. I’ll give you an example using Vimeo a bit later that will use the video ID.*

***Note #2**: Depending on your hosting provider, you may run into trouble with a default upload limit, often 2MB or 8MB. Check out how to [increase the WordPress upload limit](#).After you’ve created a new post, previewing it should show you a screen, depending on your theme, will look something like this:*



A preview of my custom post, displaying title and description, on the Twenty Ten theme.

> ***Note**: If you preview your post and get a "404" error, you may need to update your Permalinks. From the WordPress Admin, Go to "Settings", then "Permalinks", and click "Save Changes". Refresh and you should be good to go.*

Displaying Our Media – Working With Custom Post Templates

If you previewed your custom post, you probably saw something similar to what I showed in my example – not much. Where are the custom taxonomy terms, custom fields, and videos? Missing – but not for long! In the following steps, we're going to create a custom template that tells WordPress what data to display and how to display it.

CREATING A CUSTOM POST TYPE TEMPLATE

The WordPress template engine has a hierarchy that it follows when deciding what theme template it uses to display data associated with a post. In the case of our "resource" post type, the WordPress hierarchy (as of 3.1) is as follows:

- **single-resource.php** – WordPress will check the theme folder for a file named single-resource.php, if it exists, it will use that file to display the content. For different post types, simple replace "resource" with the name of your custom post type.
- **single.php** – If no post type specific template is found, the default single.php is used. This is what you probably saw if you did an early preview.

- **index.php** – If no single template is found, WordPress defaults to the old standby – the index.

I'll be using minimal examples for each of the templates, modified to work with Twenty Ten. Each example will replace and build on the previous example. Expand to your heart's content or copy the essentials into your own theme.

To get started with our example, create a file called **single-resource.php** and upload it to your theme folder. Add the following code:

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

  <div id="container">
      <div id="content">
      <?php if ( have_posts() ) while ( have_posts() ) :
the_post(); ?>

          <div class="resource">
              <h1 class="entry-title"><?php the_title(); ?></
h1>
              <div class="entry-content">
                  <?php the_content();?>
              </div>
          </div>

      <?php endwhile; ?>
      </div>
  </div>

<?php get_sidebar(); ?>
<?php get_footer(); ?>
```

The code above will give you a rather unexciting, but working template that will display the title and content (drawn directly from the main editor). What about our custom fields? Let's add them in next.

Replace the code in single-resource.php with the following:

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

<?php // Let's get the data we need
  $recording_date = get_post_meta( $post->ID,
'recording_date', true );
  $recording_length = get_post_meta( $post->ID,
'recording_length', true );
?>

  <div id="container">
      <div id="content">
      <?php if ( have_posts() ) while ( have_posts() ) :
the_post(); ?>

          <div class="resource">
              <h1 class="entry-title"><?php the_title(); ?></
h1>
              <div class="entry-meta">
                  <span>Recorded: <?php echo $recording_date ?>
| </span>
                  <span>Duration: <?php echo $recording_length ?
> </span>
              </div>
              <div class="entry-content">
                  <?php the_content();?>
              </div>
          </div>

      <?php endwhile; ?>
      </div>
  </div>

<?php get_sidebar(); ?>
<?php get_footer(); ?>
```

We're making progress! Now, using the examples above, you should see the date your resource was published and the duration of the media file.

Let's take a look at how that works. In WordPress, data stored in custom fields can be accessed several ways. Here, we are using a function called get_post_meta. This function requires two parameters, the unique ID of the

post you want to get the data from and the name of the field (its “key”) whose data you’re after. Here’s the code again:

```
$recording_date = get_post_meta( $post->ID, 'recording_date',
true );
```

First, we set a variable with PHP – we name it “$recording_date”. Then, we use the “get_post_meta” function. Remember, it needs two parameters, ID and the “key” of the field we want. “$post->ID” tells WordPress to use the ID of the post it is currently displaying. If we wanted to target a specific post, we’d put its ID instead:

```
$recording_date = get_post_meta( 35, 'recording_date',
true ); // Get the date from post 35
```

The next parameter is the “key”, or “name” of our custom field. Be sure you get that right. The last parameter tells the function to return the result as a single “string” – something that we can use as text in our template below. To display our data in the template, we write:

```
<?php echo $recording_date ?>
```

Ok, let’s keep going and get our custom taxonomies showing up.

Replace the code in single-resource.php with the following:

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

<?php // Let's get the data we need
  $recording_date = get_post_meta( $post->ID,
'recording_date', true );
  $recording_length = get_post_meta( $post->ID,
'recording_length', true );
  $resource_presenters = get_the_term_list( $post->ID,
'presenters', '', ', ', '' );
  $resource_topics = get_the_term_list( $post->ID, 'topics',
'', ', ', '' );
?>
```

```
  <div id="container">
      <div id="content">
      <?php if ( have_posts() ) while ( have_posts() ) :
the_post(); ?>

          <div class="resource">
              <h1 class="entry-title"><?php the_title(); ?></
h1>
              <div class="entry-meta">
                  <span>Recorded: <?php echo $recording_date ?>
| </span>
                  <span>Duration: <?php echo $recording_length ?
> | </span>
                  <span>Presenters: <?php echo
$resource_presenters ?> | </span>
                  <span>Topics: <?php echo $resource_topics ?></
span>
              </div>
              <div class="entry-content">
                  <?php the_content();?>
              </div>
          </div>

      <?php endwhile; ?>
      </div>
  </div>

<?php get_sidebar(); ?>
<?php get_footer(); ?>
```

Now we're starting to get more dynamic. You should see your custom fields and, assuming that your custom post has "presenters" and "topics" associated with it, you should see a list of one or more custom taxonomy terms as links. If you clicked the link, you probably saw a page that didn't look quite what you expected – we'll get to that soon. Check out get_the_term_list on the WordPress Codex to learn more about how it works.

Adding A Media Player

Now that we have some basic data in place, it's time to add our media player. In this example, we will be working with the JW Media Player, a highly customizable open-source solution.

INSTALLING JW MEDIA PLAYER

You can access basic installation instructions here. I recommend the following steps:

- Download the player from the Longtail Video website.
- Create a folder within your theme to hold the player files – In this case, I've named the folder "jw".
- Upload **jwplayer.js** and **player.swf** to the JW Player folder within your theme.

JW Player is now installed and ready to be referenced.

Now, replace the code in single-resource.php with the following:

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

<?php // Let's get the data we need
  $recording_date = get_post_meta( $post->ID,
'recording_date', true );
  $recording_length = get_post_meta( $post->ID,
'recording_length', true );
  $resource_presenters = get_the_term_list( $post->ID,
'presenters', '', ', ', '' );
  $resource_topics = get_the_term_list( $post->ID, 'topics',
'', ', ', '' );

  $resource_video = new WP_Query( // Start a new query for our
videos
```

```
  array(
      'post_parent' => $post->ID, // Get data from the current
post
      'post_type' => 'attachment', // Only bring back
attachments
      'post_mime_type' => 'video', // Only bring back
attachments that are videos
      'posts_per_page' => '1', // Show us the first result
      'post_status' => 'inherit', // Attachments require
"inherit" or "all"
      )
  );
?>

  <div id="container">
      <div id="content">
      <?php if ( have_posts() ) while ( have_posts() ) :
the_post(); ?>

          <div class="resource">
              <h1 class="entry-title"><?php the_title(); ?></
h1>
              <div class="entry-meta">
                  <span>Recorded: <?php echo $recording_date ?>
| </span>
                  <span>Duration: <?php echo $recording_length ?
> | </span>
                  <span>Presenters: <?php echo
$resource_presenters ?></span>
              </div>
              <div class="entry-content">
                  <?php while ( $resource_video-
>have_posts() ) : $resource_video->the_post(); ?>
                      <p>Video URL: <?php echo $post->guid; ?></
p>
                  <?php endwhile; ?>

                  <?php wp_reset_postdata(); // Reset the loop ?
>

                  <?php the_content(); ?>
              </div>
          </div>
```

```
        <?php endwhile; ?>
        </div>
    </div>

<?php get_sidebar(); ?>
<?php get_footer(); ?>
```

> ***Note**: You may notice the somewhat mysterious reference to "wp_reset_postdata". We are creating a loop within a loop and, to prevent strange behavior with template tags like "the_content" (try removing "wp_reset_postdata" to see what happens), we need to run a reset after any new loops we add within the main loop. Learn more about the loop on the WordPress Codex.*

Now we're getting somewhere! If everything went as expected, you should see a direct, plain text URL to your video. That's not very exciting (yet), but we want to make sure we are getting that far before we add in the next step – the player.

If you're having trouble at this point, check back through your code and look for any mistakes that may have been made. If you are trying to vary widely from this example, simplify your variations and start as close to this example as you can – get that to work first then branch back out.

With the URL to our video available, we are ready to add in the player. Let's go!

Replace the code in single-resource.php with the following:

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

<?php // Let's get the data we need
```

```
  $recording_date = get_post_meta( $post->ID,
'recording_date', true );
  $recording_length = get_post_meta( $post->ID,
'recording_length', true );
  $resource_presenters = get_the_term_list( $post->ID,
'presenters', '', ', ', '' );
  $resource_topics = get_the_term_list( $post->ID, 'topics',
'', ', ', '' );

  $resource_video = new WP_Query( // Start a new query for our
videos
  array(
      'post_parent' => $post->ID, // Get data from the current
post
      'post_type' => 'attachment', // Only bring back
attachments
      'post_mime_type' => 'video', // Only bring back
attachments that are videos
      'posts_per_page' => '1', // Show us the first result
      'post_status' => 'inherit', // Attachments require
"inherit" or "all"
      )
  );
?>

  <div id="container">
      <div id="content">
      <?php if ( have_posts() ) while ( have_posts() ) :
the_post(); ?>

          <div class="resource">
              <h1 class="entry-title"><?php the_title(); ?></
h1>
              <div class="entry-meta">
                  <span>Recorded: <?php echo $recording_date ?>
| </span>
                  <span>Duration: <?php echo $recording_length ?
> | </span>
                  <span>Presenters: <?php echo
$resource_presenters ?> | </span>
                  <span>Topics: <?php echo $resource_topics ?></
span>
              </div>
```

```
            <div class="entry-content">
            <?php while ( $resource_video->have_posts() ) :
$resource_video->the_post(); // Check for our video ?>
                <div id="player">
                    <script type="text/javascript" src="<?php
bloginfo('stylesheet_directory'); ?>/jw/jwplayer.js"></script>
                    <div id="mediaspace">Video player loads
here.</div>
                    <script type="text/javascript">
                        jwplayer("mediaspace").setup({
                            flashplayer: '<?php
bloginfo( 'stylesheet_directory' ); ?>/jw/player.swf',
                            file: '<?php echo $post->guid; ?
>',
                            width: 640,
                            height: 360
                        });
                    </script>
                </div>
            <?php endwhile; ?>

            <?php wp_reset_postdata(); // Reset the loop ?>

            <?php the_content(); ?>
            </div>
        </div>

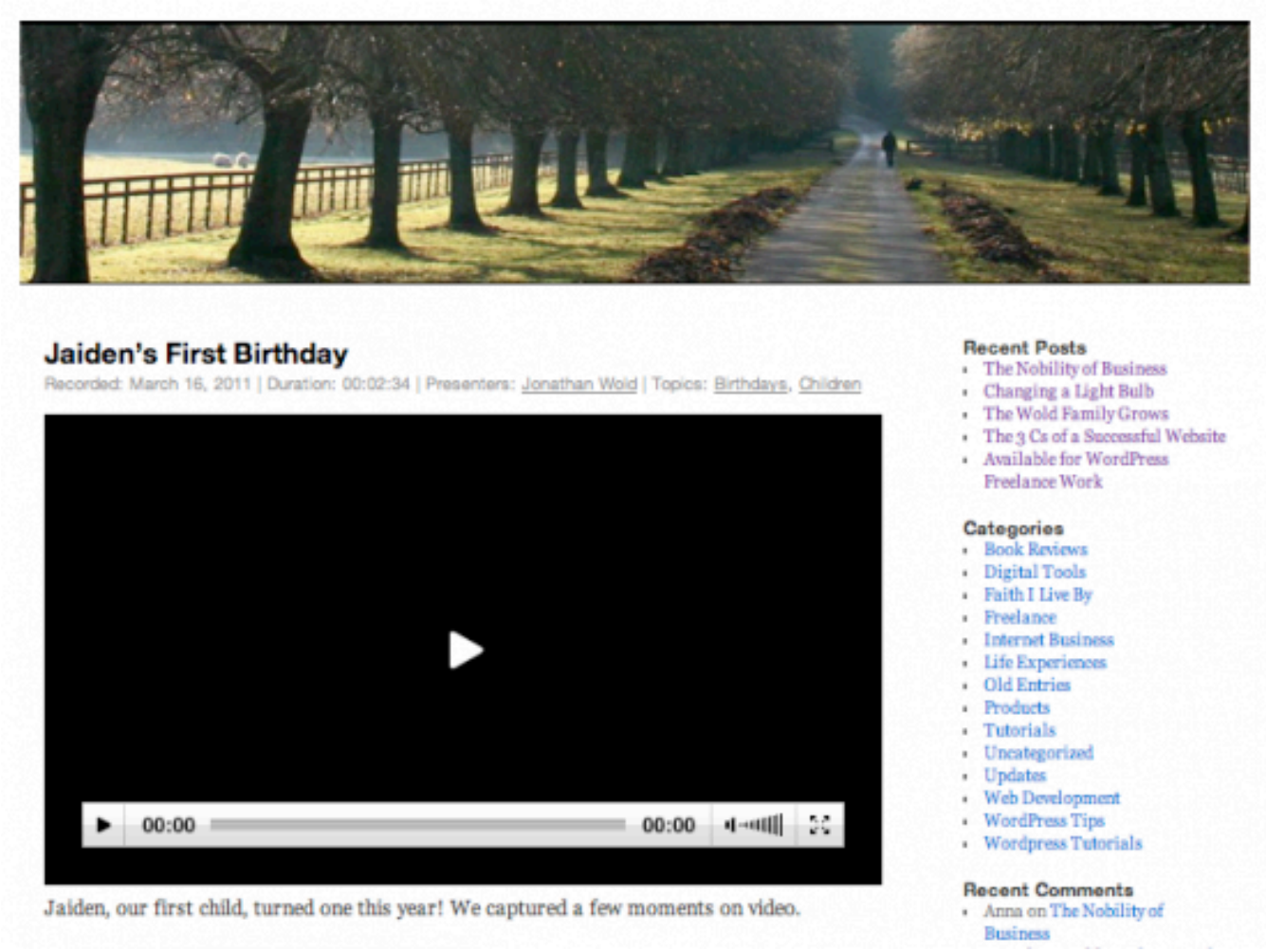
    <?php endwhile; ?>
    </div>
  </div>

<?php get_sidebar(); ?>
<?php get_footer(); ?>
```

Note carefully the assumptions I'm making in the code above. First, I am assuming that you are storing the JW player files in a folder called "jw" inside the WordPress theme folder of the currently activated theme. If you load the page and the player is not working (and you did have the video URL displaying in the previous step), view the source code on your page, copy the URLs that WordPress is generating to your respective JW player

files (jwplayer.js and player.swf) and try accessing them in your browser to make sure each is valid. If there is a problem, update your references accordingly.

Otherwise, there you have it! Your video details and the video itself is now displaying on the page and you should see something like this:



A view of the player, complete with title, description, custom field values and custom taxonomies terms.

> ***Note**: There is a lot that you can do to customize the appearance and behavior of the JW Player. A good place to start is the [JW Player Setup*

Wizard. Customize the player to your liking, then implement the code changes in your template accordingly.

USING VIMEO INSTEAD

Let's say you wanted to use Vimeo, instead of uploading the videos into WordPress. First, you need to add a custom field to store the ID of your Vimeo video. Assuming you've done that, and assuming that you've entered a valid Vimeo ID in your custom field (we named the field "vimeo_id" in our example), the following code will work:

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

<?php // Let's get the data we need
  $recording_date = get_post_meta( $post->ID,
'recording_date', true );
  $recording_length = get_post_meta( $post->ID,
'recording_length', true );
  $resource_presenters = get_the_term_list( $post->ID,
'presenters', '', ', ', '' );
  $resource_topics = get_the_term_list( $post->ID, 'topics',
'', ', ', '' );

  $vimeo_id = get_post_meta( $post->ID, 'vimeo_id', true );
?>

  <div id="container">
      <div id="content">
      <?php if ( have_posts() ) while ( have_posts() ) :
the_post(); ?>

          <div class="resource">
              <h1 class="entry-title"><?php the_title(); ?></
h1>
              <div class="entry-meta">
                  <span>Recorded <?php echo $recording_date ?> |
</span>
                  <span>Duration: <?php echo $recording_length ?
> | </span>
```

```
                <span>Presenters: <?php echo
$resource_presenters ?> | </span>
                <span>Topics: <?php echo $resource_topics ?></
span>
            </div>

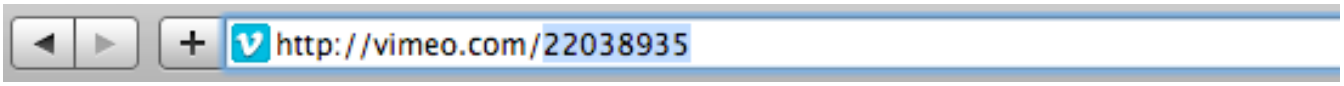
            <div class="entry-content">
                <?php if ($vimeo_id) { // Check for a video ?>
                    <iframe src="http://player.vimeo.com/
video/<?php echo $vimeo_id; ?>?byline=0&title=0&portrait=0"
width="640" height="360" frameborder="0" class="vimeo"></
iframe>
                <?php } ?>

                <?php the_content(); ?>
            </div>
        </div>

    <?php endwhile; ?>
    </div>
  </div>

<?php get_sidebar(); ?>
<?php get_footer(); ?>
```

We use “$vimeo_id” to retrieve and store the ID from our custom field (named “vimeo_id”, in this case) and then, in the code below, we first check to make sure the $vimeo_id field has data in it, then we use Vimeo’s iframe code (details here) to load the video.



In Vimeo’s case, the ID is a series of numbers (notice the selected text) after “vimeo.com/”.

Conclusion

And that concludes part one! You've learned how to setup custom post types and custom taxonomies without using plugins. You've also learned how to setup custom fields and display their data, along with a video player and custom taxonomy terms, within a custom post template. In part two, we'll look at how to customize the custom taxonomy templates and make them a whole lot more useful.

How To Build A Media Site On WordPress – Part 2

Jonathan Wold

The default “category” and “tag” taxonomies in WordPress offer a lot of flexibility to those with imagination and in my development experience I have seen a wide range of creative implementations.

With the introduction of custom taxonomies and their growing ease of use, though, we need no longer be bound to categories and tags. With the ability to create both hierarchical and non-hierarchical taxonomies and with the introduction of several new features in WordPress 3.1, now is the time, if you’re not already, to begin putting custom taxonomies to use.

In part one of this two part series, we learned how to setup custom post types and custom taxonomies. We also learned how to build a template to check for and display media attached to custom posts. Now, we’ll learn how to use custom taxonomy templates to organize and relate our media. Let’s get started!

Organizing Our Media – Working With Custom Taxonomy Templates

Now that we have our media displaying, it's time to work on how it's organized. If you tried clicking on one of the custom taxonomy terms, odds are the result weren't very exciting. You probably saw something like this:

A rather unhelpful default view of a term in the "presenter" taxonomy.

What we're going to do next is create a template that allows us to customize the results and offer a page that will be more useful.

CREATING A CUSTOM TAXONOMY TEMPLATE

As with custom posts, the WordPress template engine has a custom taxonomy template hierarchy that it follows to determine what template it uses to display data associated with a custom taxonomy term. We'll start with our "presenters" taxonomy. In our case, the WordPress hierarchy is as follows:

- **taxonomy-presenters.php** – WordPress will check the theme folder for a file named taxonomy-presenters.php. If it exists, it will use that template to display the content. For different custom taxonomies, simple replace "presenters" with the name of your custom taxonomy.
- **taxonomy.php** - If no custom taxonomy template is found, WordPress checks for a general taxonomy template.
- **archive.php** – If no general taxonomy template is used, the WordPress archive template is used.

- **index.php** – If no archive template is found, WordPress defaults to the old standby – the index.

> ***Note**: The WordPress template hierarchy structure also allows templates for specific terms. For instance, in a case where "Jonathan Wold" was a term in the "presenters" taxonomy, I could create a custom template called "taxonomy-presenters-jonathan-wold.php".*

Non-Hierarchical Custom Taxonomy Templates

We'll start with the non-hierarchical "presenters" custom taxonomy. As with the custom post type examples previously, I will be using minimal examples for each of the templates.

To get started with this example, create a file called **taxonomy-presenters.php** and upload it to your theme folder. Add the following code:

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

<?php // Get the data we need
  $presenter = get_term_by( 'slug', get_query_var( 'term' ),
get_query_var( 'taxonomy' ) );
?>

  <div id="container">
      <div id="content" class="presenter">
          <h1 class="entry-title"><?php echo $presenter-
>name; ?></h1>
          <p><?php echo $presenter->description; ?></p>
      </div>
  </div>

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

Previewing a term should now show you a rather empty page with the name of the term and the description (if you entered one when creating or editing the term). In my case, on Twenty Ten, accessing the term “Jonathan Wold” (/presenter/jonathan-wold) looks like this:

A rather basic, yet more useful, view of a custom taxonomy term template.

Before moving on, let’s review the code above to learn what it’s doing and what you can do with it.

```
$presenter = get_term_by( 'slug', get_query_var( 'term' ),
get_query_var( 'taxonomy' ) );
```

This piece of code may seem intimidating at first, but it’s rather simple. First, we are defining a variable called $presenter. Our goal is to have that variable store everything that WordPress knows about our term.

To do that, we are using the function get_term_by. That function requires three things:

1. **Field** – You can access a term by name, ID, or slug. In our case, we are using the slug, which is “jonathan-wold”.
2. **Value** – We’ve told WordPress that we want to get our term data by using the “slug” field. Now, it needs a slug to retrieve data. Since we want this to be dynamic, we are using another function called get_query_var. When you access a term in WordPress (e.g. viewing a term by its permalink), a query is run in order to generate the results for that term. Using “get_query_var” allows you to intercept that query and get the data for your own use.
3. **Taxonomy** – In addition to the term slug, WordPress also needs the taxonomy name (this is critical in cases where the same name is used across multiple taxonomies). We use “get_query_var” again to retrieve that for us.

If we wanted to access the term data for one specific term in a particular custom taxonomy, we would do it like this:

```
$presenter = get_term_by( 'slug', 'jonathan-wold',
'presenters');
```

In our example, we are adding code into our template telling WordPress to give us the data for the term a visitor is currently viewing. WordPress stores that data as an “object”.

To see what data is available to you in an object, add the following within your code:

```
<?php
  echo '<pre>';
  print_r( $presenter );
  echo '</pre>';
?>
```

Preview the term again and you should see a block of code that looks something like this:

```
stdClass Object
(
    [term_id] => 53
    [name] => Jonathan Wold
    [slug] => jonathan-wold
    [term_group] => 0
    [term_taxonomy_id] => 52
    [taxonomy] => presenters
    [description] => An Internet Entrepreneur who believes that faith and works are inseparable.
    [parent] => 0
    [count] => 1
)
```

An easily readable view of the object attributes and values.

That block of code lets you see what WordPress knows about your particular object and what information you have available to use within your template.

> ***Note**: In part one, I referenced a technique for adding custom fields to your custom taxonomies and giving you access to more data within your templates. Just incase you missed the reference, take a look at the advanced tutorial, [How To Add Custom Fields To Custom Taxonomies], on the Sabramedia blog.*

DISPLAYING OBJECT DATA IN TEMPLATES

Now, let's look at how we took the data from that object and actually displayed it in the template. Let's start with the first example:

```
<?php echo $presenter->name; ?>
```

In English, we are telling PHP to “echo”, or display, the “name” value of the $presenter object. We would know that the object created with “get_term_by” contains the value for “name” by either looking up the return values for get_term_by in the Codex or by using “print_r” to see for ourselves. We’ll explore this in more detail once we look at the “topics” taxonomy.

To get our description, we do the same thing, changing the “name” value to “description”:

```
<?php echo $presenter->description; ?>
```

DISPLAYING TERM RESULTS IN CUSTOM TAXONOMY TEMPLATES

Now that we have our term name and description displaying, it’s time to show some actual custom post results.

We are continuing our example with taxonomy-presenters.php. Replace the existing code with the following:

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

<?php // Get the data we need
  $presenter = get_term_by( 'slug', get_query_var( 'term' ),
get_query_var( 'taxonomy' ) );

  $resources = new WP_Query(
      array(
          'post_type' => 'resource', // Tell WordPress which
post type we want
          'posts_per_page' => '3', // Show the first 3
          'tax_query' => array( // Return only resources where
presenter is listed
              array(
                  'taxonomy' => 'presenters',
                  'field' => 'slug',
                  'terms' => $presenter->slug,
```

```
                )
            )
        )
    );
?>

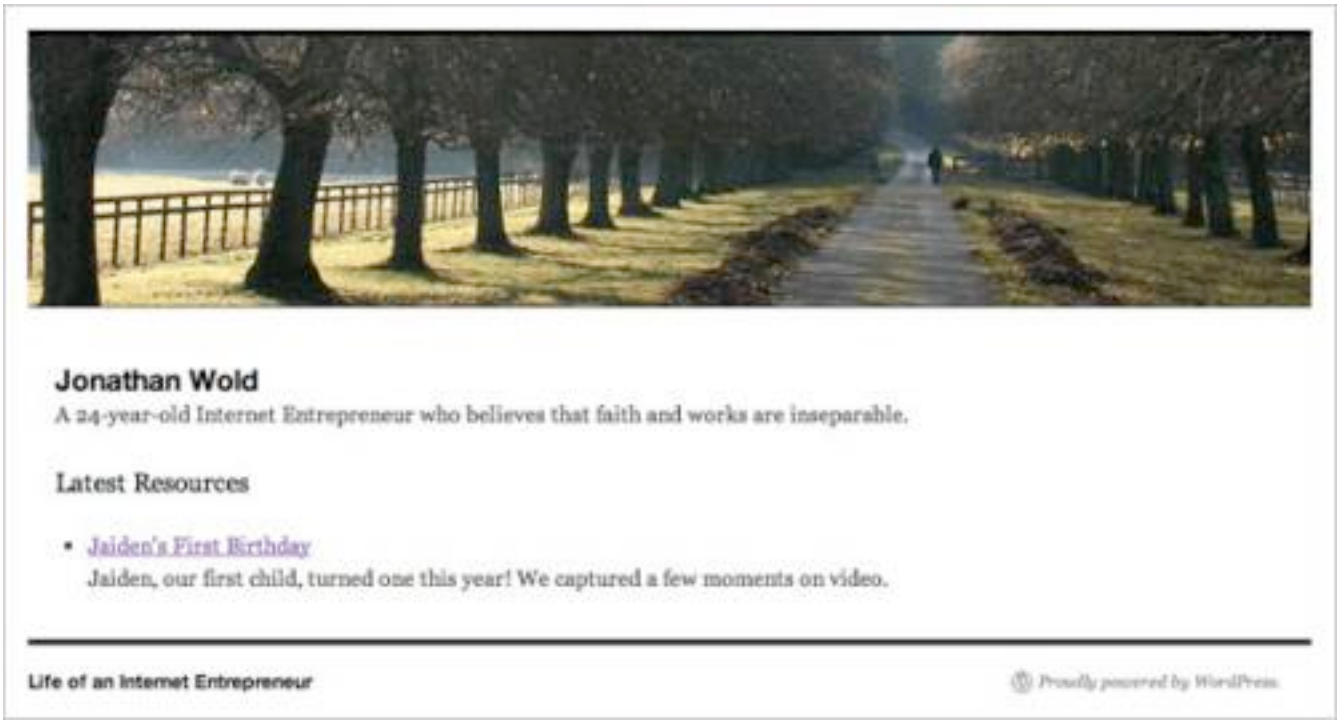
  <div id="container">
      <div id="content" class="presenter">
          <h1 class="entry-title"><?php echo $presenter-
>name; ?></h1>
          <p><?php echo $presenter->description; ?></p>

          <div class="resources">
              <h3>Latest Resources</h3>
              <ul id="resource-list">
                  <?php while ( $resources->have_posts() ) :
$resources->the_post(); ?>
                      <li id="resource-<?php the_ID(); ?>"
class="resource">
                          <a href="<?php the_permalink(); ?>"><?
php the_title(); ?></a>
                          <span><?php the_excerpt(); ?></span>
                      </li>
                  <?php endwhile; ?>
              </ul>
          </div>

      </div>
  </div>

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

Previewing one of your terms should now display the name and description of the term along with a list of custom posts associated with that term. In our case, the results look like this:



With a customizable list of related custom posts, the term results template is looking much more useful.

The update to this code block is the addition of our "$resources" query. Let's take a look at that more closely:

```
$resources = new WP_Query(
  array(
      'post_type' => 'resource', // Tell WordPress which post
type we want
      'posts_per_page' => '3', // Show the first 3
      'tax_query' => array( // Return resources associated with
presenter
          array(
              'taxonomy' => 'presenters',
              'field' => 'slug',
              'terms' => $presenter->slug,
              )
          )
      )
  );
```

For our variable of $resources, we are creating a new instance of the WordPress class, WP_Query. Then, we're setting values on several parameters, post_type, post_per_page, and tax_query.

The first two are straight forward. With "post_type", you let WordPress know which types of content you're wanting to return. We used that in our media example to retrieve attachments. To display multiple posts types, replace the "post_type" line with this:

```
'post_type' => array( 'resource', 'other_post_type',
'another_post_type' ),
```

For "posts_per_page", you are letting WordPress know how many posts to return before triggering pagination. If you want to return all posts, use "-1" for the value, like this:

```
'posts_per_page' => '-1', // Show all the posts
```

Now, "tax_query" is a new parameter added in WordPress 3.1. It is a powerful parameter that lets you return results associated with multiple taxonomies and custom fields.

Let's take a closer look at it:

```
'tax_query' => array( // Return resources associated with
presenter
  array(
      'taxonomy' => 'presenters',
      'field' => 'slug',
      'terms' => $presenter->slug,
      )
  )
```

First, we choose our custom taxonomy. In our case, we are hardcoding in "presenters". If we wanted to make it more dynamic and build, for instance, a general taxonomy template (taxonomy.php) to handle multiple taxonomies in a similar way, we would use "get_query_var" again, like so:

```
'taxonomy' => get_query_var( 'taxonomy' ),
```

> ***Note**: The "tax_query" function works with one taxonomy at a time. To query multiple taxonomies, simply duplicate the code above (be sure to add the appropriate comma at the end) and change the parameters accordingly.*

Next, we have the "field" parameter. This lets WordPress know what field we will be returning our terms by. WordPress accepts "slug" or "id". I am using "slug" because I prefer recognizing posts by words over numbers.

Then, we have "terms". In our case, we are using the $presenter variable to pass in the "slug" in the same way we added data directly into our custom post template. If we wanted to make it more dynamic, we could use "get_query_var" again:

```
'term' => get_query_var( 'term' ),
```

If we want to return results for multiple terms, we add an array, like this:

```
'term' => array( 'term_1', 'term_2', 'random_other_term' ),
```

To modify our results further, we can use an optional "operator" parameter that allows us to specify whether our results are "IN", "NOT IN", or "OR". A simple example, appropriate for use in a single taxonomy, is "NOT IN".

To modify the query to return results that are "NOT IN" the custom taxonomy and terms that you've listed, add the following within your tax_query array:

```
'operator' => 'NOT IN',
```

> ***Note**: To experiment with results queried against multiple custom taxonomies, take a look at "Multiple Taxonomy Handling" under [Taxonomy Parameters](#) on the Codex reference for WP_Query.*

Now that we've gone through that, we reference our newly created query with a loop. Here's the code again:

```
<div class="resources">
  <h3>Latest Resources</h3>
  <ul id="resource-list">
      <?php while ( $resources->have_posts() ) : $resources-
>the_post(); ?>
          <li id="resource-<?php the_ID(); ?>"
class="resource">
              <a href="<?php the_permalink(); ?>"><?php
the_title(); ?></a>
              <span><?php the_excerpt(); ?></span>
          </li>
      <?php endwhile; ?>
  </ul>
</div>
```

This is another basic instance of The Loop, customized to return results from our $resources query and, in this case, the results returned are "the_ID", "the_permalink", "the_title", and "the_excerpt".

CHECKING FOR EMPTY RESULTS

In our example above, we have some code (like the <UL>) that appears outside of our loop. If there were no results, the "container" HTML would still show up in the template. To prevent that, we can preface it with a conditional statement like this:

```
<?php if ( $resources->post_count > 0 ) { // Check to make
sure there are resources ?>
// Display your results
<?php } ?>
```

Replace “$resources” with the name of your custom query and return your results within the conditional statement. If the “post_count” is greater than zero (“> 0″), then the code will appear in your template – otherwise, the page remains free of extra HTML.

Hierarchical Custom Taxonomy Templates

Alright, now that we have a non-hierarchical taxonomy under our belt, let’s move on and tackle hierarchy. We covered the basics in setting up “presenters”, so let’s pick up there where we left off.

Create a file called **taxonomy-topics.php** and add the following code:

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

<?php // Get the data we need
  $topic = get_term_by( 'slug', get_query_var( 'term' ),
get_query_var( 'taxonomy' ) );

  $resources = new WP_Query(
      array(
          'post_type' => 'resource', // Tell WordPress which
post type we want
          'posts_per_page' => '3', // Show the first 3
          'tax_query' => array( // Return only resources where
presenter is listed
              array(
                  'taxonomy' => 'topics',
                  'field' => 'slug',
                  'terms' => $topic->slug,
                  )
              )
          )
```

```
    );
?>

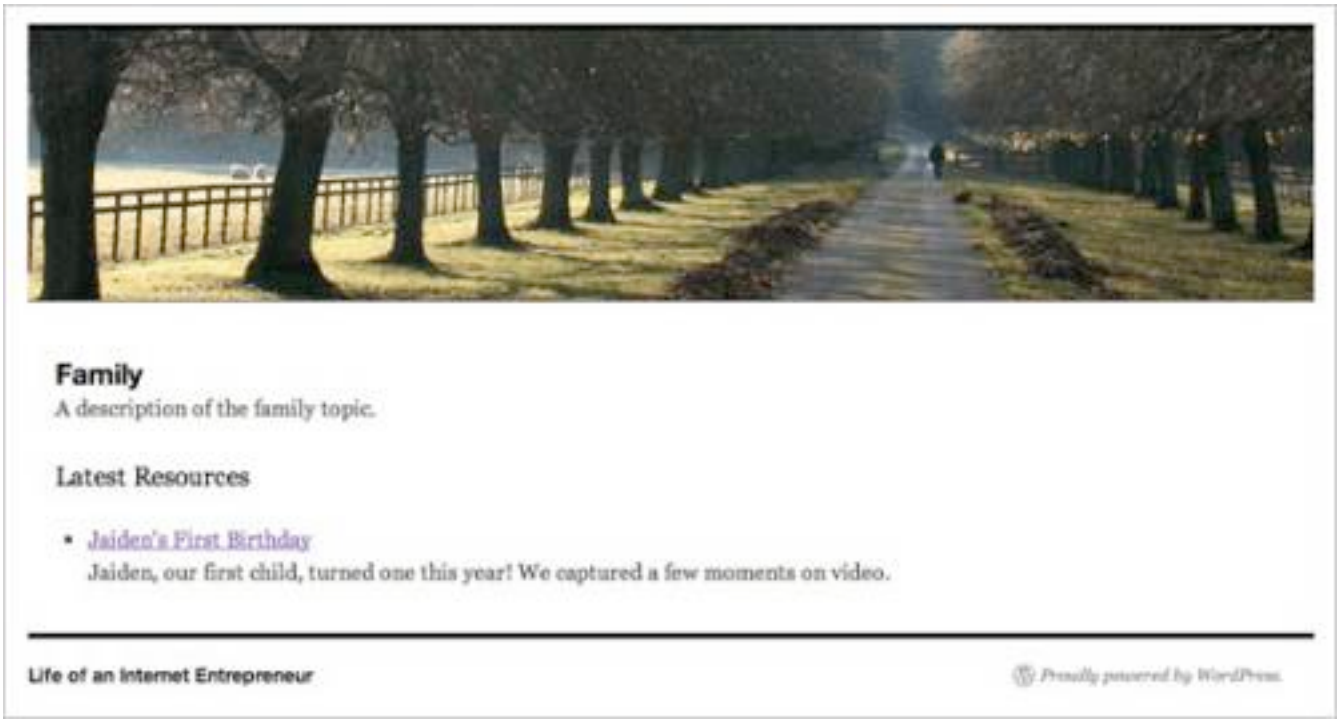
  <div id="container">
      <div id="content" class="presenter">
          <h1 class="entry-title"><?php echo $topic->name; ?></
h1>
          <p><?php echo $topic->description; ?></p>

          <?php if ( $resources->post_count > 0 ) { // Check to
make sure there are resources ?>
              <div class="resources">
                  <h3>Latest Resources</h3>
                  <ul id="resource-list">
                      <?php while ($resources->have_posts()) :
$resources->the_post(); ?>
                          <li id="resource-<?php the_ID(); ?>"
class="resource">
                              <a href="<?php the_permalink(); ?
>"><?php the_title(); ?></a>
                              <span><?php the_excerpt(); ?></
span>
                          </li>
                      <?php endwhile; ?>
                  </ul>
              </div>
          <?php } ?>

      </div>
  </div>

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

Previewing a “topic” should now give you a familiar plain template that looks something like this:



A basic, yet useful view of another custom taxonomy term.

CREATING PARENT AND CHILDREN LINKS

Now, the thing that is different with this taxonomy is that it can have both “parents” and “children”. What we want to do is check for a parent topic and, if it exists, display a link to it. We also want to check for sub-topics and if they exist, display links to them.

> ***Note**: For these examples to work, be sure that you’re working with a post example that has multiple levels of a hierarchical custom taxonomy associated with it. In my example, I have created topics 3 levels deep and associated all of them with the post.*

So let's get started. First, within the PHP section at the top of our template, add the following code:

```
if ( $topic->parent > 0 ) { // Check to make sure the topic
has a parent
  $topic_parent = get_term( $topic->parent, 'topics' ); // Get
the object for the topic's parent
}

$topic_children = get_terms( 'topics', 'child_of='.$topic-
>term_id );
$last_topic = end( array_keys( $topic_children ) ); // Mark
the last topic
```

Alright, what do we have going on here? First, we're checking to make sure the topic has a parent. If a topic does not have a parent, WordPress gives the "parent" attribute a value of zero ("0"). So, the first thing we do is a conditional to check and make sure that the parent has a value greater than zero. If it does, we define the variable $topic_parent and use the get_term function to retrieve the parent topic based on its ID.

Next, we define another variable called $topic_children. This time, we use the get_terms function, which has a special attribute called "child_of". We pass in the value of the current topic and tell WordPress, in English, to "take the current topic and bring me back a list of all its sub-topics or children".

Then, we define a variable called $last_topic. The data that $topic_children gives us is in the form of an array. Our $last_topic variable counts to the "end" of the array and keeps track of it. We're going to use that later to put a comma after each of our sub-topics and then do nothing for the last sub-topic.

Now, to show the results, add the following code within your template:

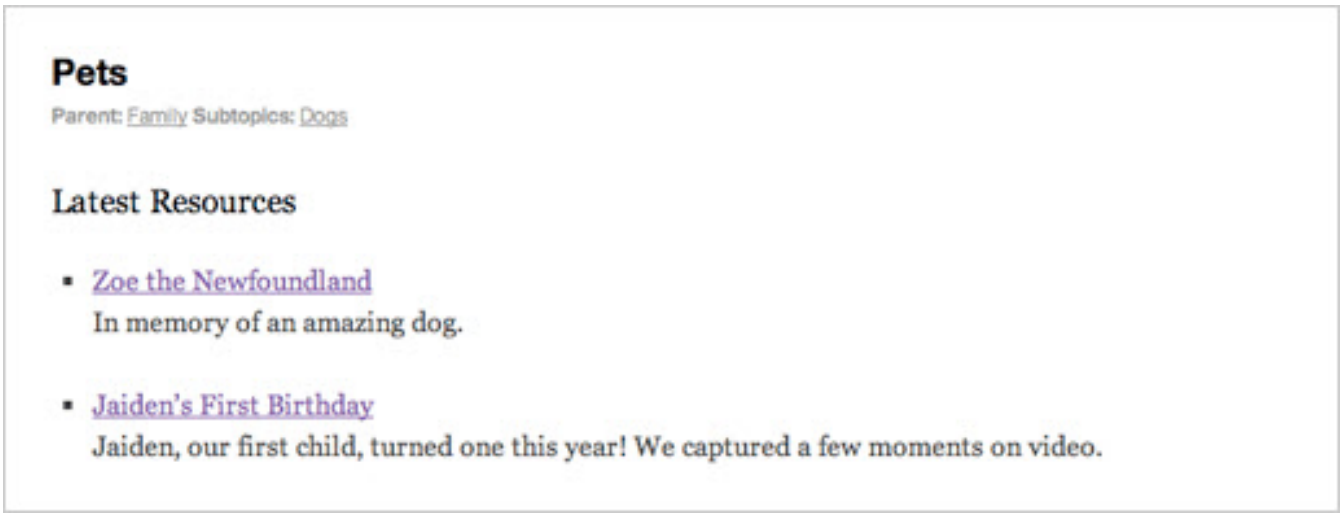
```
<?php if ( $topic->parent > 0  ) { ?>
<strong>Parent:</strong> <a href="<?php echo
get_term_link( $topic_parent->slug, 'topics' ); ?>"><?php echo
$topic_parent->name; ?></a>
<?php } ?>

<?php if ( $topic_children ) { ?>
  <strong>Subtopics: </strong>
  <?php foreach ( $topic_children as $key =>
$topic_single ) : ?>
      <span><a href="<?php echo get_term_link( $topic_single-
>slug, 'topics' ); ?>"><?php echo $topic_single->name; ?></
a></span><?php if ( $key !== $last_topic ) echo ', '; ?>
<?php endforeach; ?>
<?php } ?>
```

Each block of code first checks to make sure that a parent topic or sub-topic(s) exists, respectively. Then, in the case of the "parent", we use the get_term_link function to retrieve the link by the "slug" of the $topic_parent.

For our sub-topics, we create a "foreach" loop to output a list of all sub-topics. At the end, we do a conditional check on the $last_topic in our array using the variable we created earlier. If it is not the last topic, we echo a comma after the close – otherwise, we do nothing.

And there you have it! The result using the Twenty Ten theme will look something like this:



A view of the "topics" taxonomy, with the parent topic and sub-topics listed.

RELATING TAXONOMIES BY POSTS

Now, this is where we get a bit fancy. Let's say we're working on our template for the "topics" taxonomy and we wanted to show a list of "presenters" who covered that particular topic. How would we do that? In the code that follows, we're going to use the custom posts themselves as our reference point and bring back the related custom taxonomies.

The rationale is simple. If we had 10 posts associated with a particular term in a given custom taxonomy, those 10 posts will likely have other terms from other custom taxonomies associated with them as well. So, we use the posts themselves to retrieve and compile the term data that would otherwise not be related to our particular term. Here are some examples where this might be especially useful:

- **Events** - An "event" taxonomy where we want to show a list of "presenters" at that same event.
- **Movies** - A "genre" taxonomy where we want to show a list of "directors" who make that same genre of film.

- **Recipes** - A “category” taxonomy where we want to show related “ingredients”.

Alright, let’s dive into the code:

```
// Retrieve all the IDs for resources associated with the
current term
$post_ids = array();
foreach ( $resources->posts as $post ) {
  array_push( $post_ids, $post->ID );
}

// Get presenter data based on the posts associated with the
current term
$presenters_by_posts = wp_get_object_terms( $post_ids,
"presenters" );

$topic_presenters = array();
foreach ( $presenters_by_posts as $presenter ){
  $topic_presenters[$presenter->term_id] = $presenter;
}
$last_presenter = end( array_keys( $topic_presenters ) );
```

First, we define an empty array called $post_ids. Then, we create a loop through each of the “resources” associated with our current term using the $resource query we created earlier. We take that loop and “push” each of the post IDs for our resources back into the previously empty $post_ids array.

Next, we define a new variable, $presenters_by_posts. We use the wp_get_object_terms function, which accepts either a single ID or an array of IDs (which we just created) to return a list of terms. In our case, we’re using this function to check all the custom posts associated with this term and bring back a list of all the “presenters”.

Next, we define another empty array called $topic_presenters. We now loop through our $presenters_by_posts and then redefine our $presenter

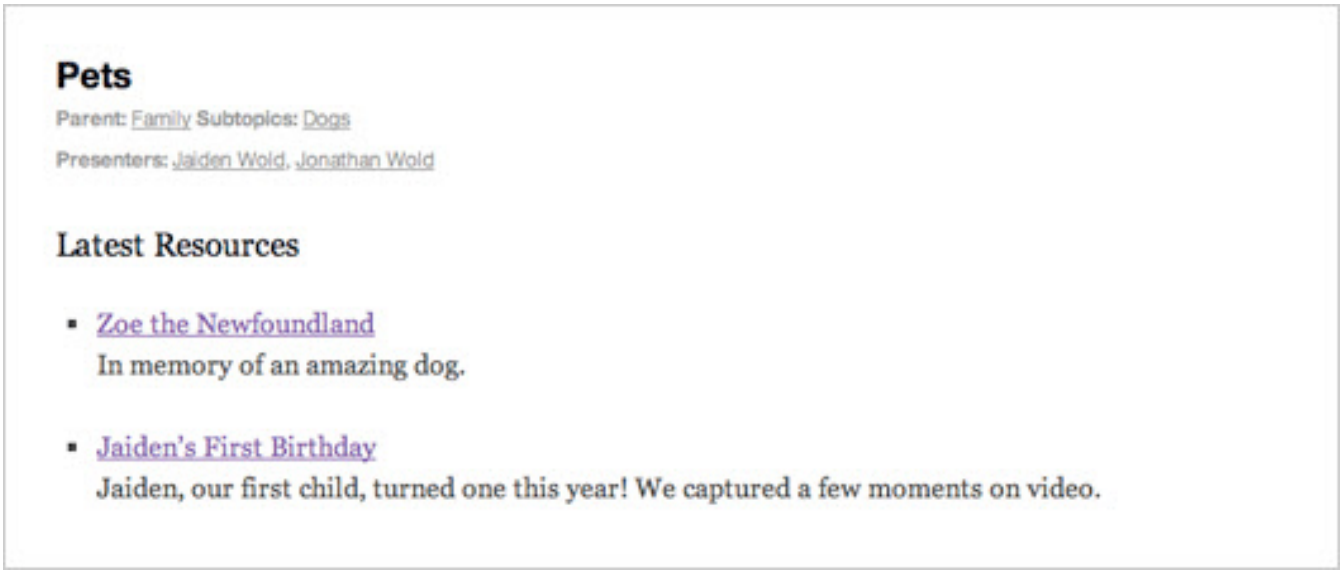
variable to hold the term_id of each $presenter that we returned using our $presenters_by_posts function.

Now, let's make use of that in the template. Add the following the code:

```
<?php if ( $topic_presenters ) { ?>
  <strong>Presenters:</strong>
  <?php foreach ( $topic_presenters as $key =>
$presenter ) : ?>
      <span><a href="<?php echo get_term_link( $presenter-
>slug, 'presenters' ); ?>"><?php echo $presenter->name; ?></
a></span><?php if ( $key !== $last_presenter ) echo ', '; ?>
  <?php endforeach; ?>
<?php } ?>
```

Now, we simply loop through each of our $topic_presenters using our redefined $presenter. We then access the attribute values of our $presenter object to echo the "slug" for the term link and the term "name". Finally, we do a check for the $last_presenter and if it is not the last one, we echo a comma.

Here's how that looks in my example:



The updated view of "Topics", with a list of presenters related by custom posts.

Conclusion

And that's a wrap! With part one and part two under your belt you have taken some solid steps above and beyond the basics of WordPress theme development. My goal has been to give you some solid examples that you can follow and to explain what's been done along the way so you can apply what you've learned to your own projects. You've learned a lot about custom post types and custom taxonomies and I am looking forward to seeing what you build.

Getting Started With bbPress

Thord Daniel Hedengren

Forums have been around forever, so it should come as no surprise that several plugins for the popular publishing platform WordPress provide this feature, as well as support for integrating other forum software. One project, however, has a special place in the WordPress community, and that is bbPress. This is the software created by WordPress founder, Matt Mullenweg, as a lightweight system for the WordPress.org support forums. In true open-source fashion, the bbPress project was born (at bbpress.org, of course) as a lightweight standalone alternative for forums.

The problem is that the project never really kept up the pace; and while the WordPress community wanted to use it, and bbPress saw some promising

spurts of development, it never really caught up to the alternatives. Most of us who needed a forum went either with a plugin alternative that integrated perfectly or with forum software such as Vanilla.

The Facebook-inspired community plugin BuddyPress changed all that. BuddyPress, which adds groups and other membership functionality to a blog, started to ship with bbPress integrated in it. Perhaps unknowingly, some WordPress bloggers who had community features powered by BuddyPress were actually running a version of bbPress, which is enabled in the BuddyPress interface. It worked — and continues to work — great actually; because although bbPress, as standalone forum software, is way behind the competition in terms of features, sometimes all you need is a lightweight alternative, which was the idea behind bbPress all along.

bbPress 2.0 changed it all again, because bbPress has now been officially reborn as a plugin for WordPress, something that had been in the works for quite some time. This is where we stand today, with a fresh release of the first version of the bbPress plugin. In the coming weeks (or right now, depending on when you're reading this), the plugin will get proper documentation and more support for cool functionality. That shouldn't stop you from giving it a go right away, because getting started and taking advantage of its core functionality is easy enough.

Before we move on, we need to clear up some nomenclature:

- bbPress is a plugin for WordPress, and is sometimes referred to as bbPress 2.0 for clarity.
- bbPress 1.0 is a standalone forum that integrates with WordPress (and the BuddyPress plugin) but does not reside in WordPress' core.
- BuddyPress is a separate plugin for WordPress that integrates with the bbPress plugin.

- BuddyPress still ships with bbPress, but you can connect to your bbPress plugin forums if you want to.

Yes, it's all a bit messy.

Getting bbPress Up And Running

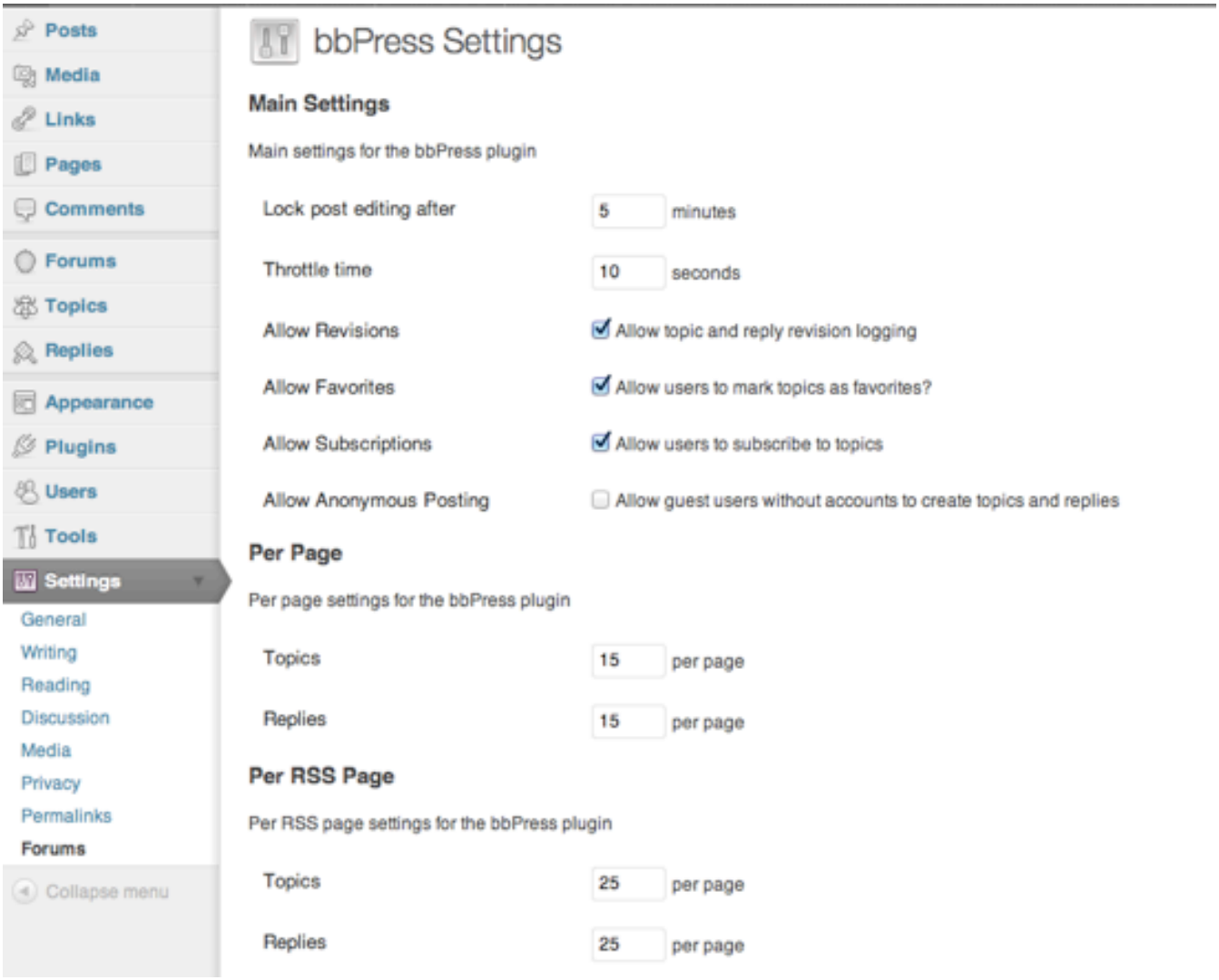
Installing bbPress is easy, because it's available in WordPress' plugin directory. Either install it from within WordPress, using the "Add new plugin" feature, or via FTP if you prefer to (or must) upload plugins. Then, activate the plugin, and you're all set!

Well, not quite. You'll want to look at some settings before starting to use the forums.

You'll notice a new "Forums" menu under "Settings" in the admin area, along with the brand new sections "Forums," "Topics" and "Replies," all sporting bee-inspired icons.

Let's look at the "Forums" settings pane first, shown above. Here you have an assortment of settings for your forums, such as whether to allow anonymous posts, how long posters should be able to edit their posts, and how many topics to show per page.

The "Archive" and "Single Slugs" settings are important. These define the URLs of your forums, the posts, and the tags for posts. Choose something that fits your set-up; if you're running an English-language website, then the default settings will probably do, but you can fine tune to your needs.

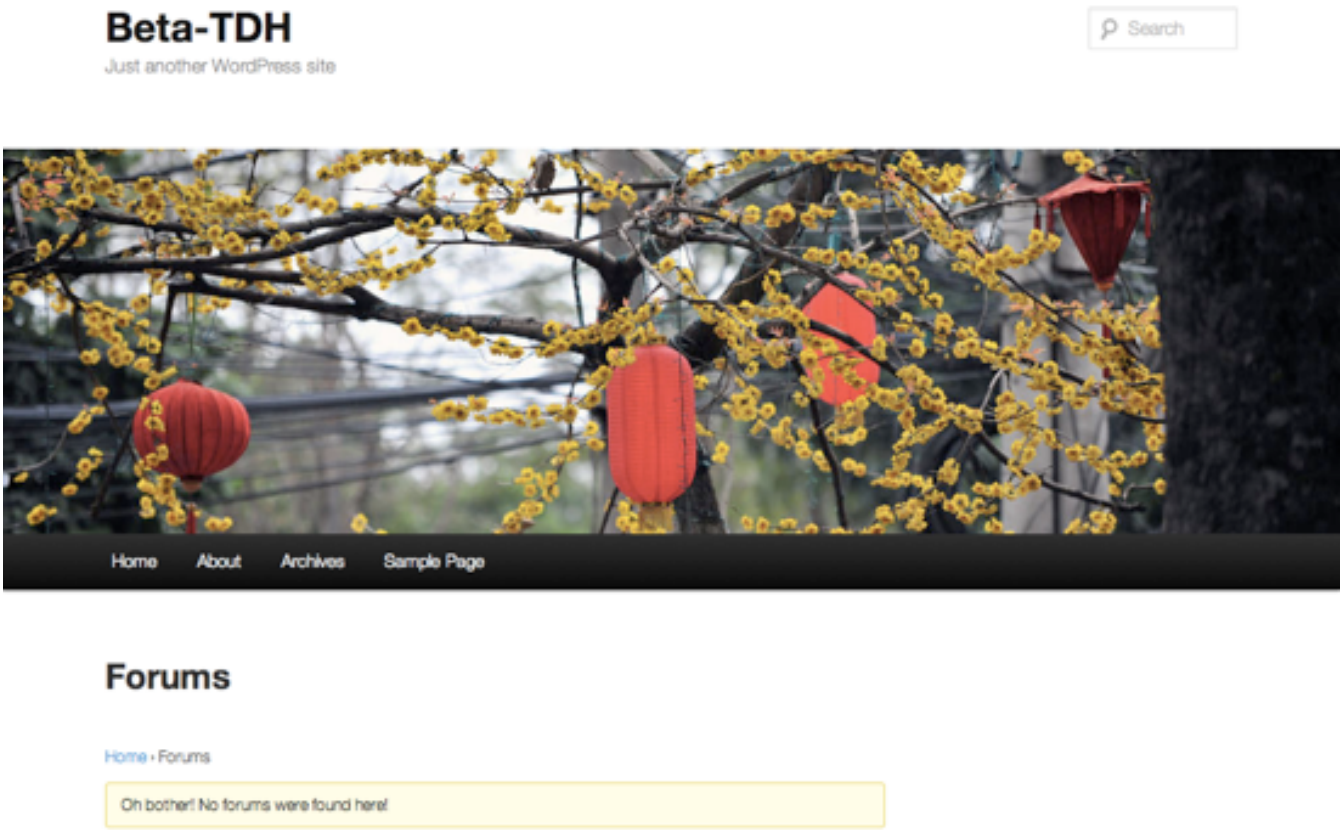


bbPress settings.

Remember to go to Settings ➜ Permalinks after making any change to the slugs, and rebuild the permalink structure by clicking the “Save Changes” button on that page. If you ever have problems viewing the forums, give this a shot because it might be an issue with the permalinks, and rebuilding them might help. Also, make sure to press the “Save” button in Settings ➜ Forums.

Where are your forums, then? Well, you’ll already know that from the Settings ➜ Forums page, because they are located at the base slug

assigned for the `forums`. By default, it would be forums, so you'd find them at `yourdomain.com/forums/`. Do yourself a favor and use pretty permalinks, because although bbPress will work without them, the URLs will look so much better if they're pretty. That Google will thank you is just a bonus (note: an actual thank-you from Google is not guaranteed).

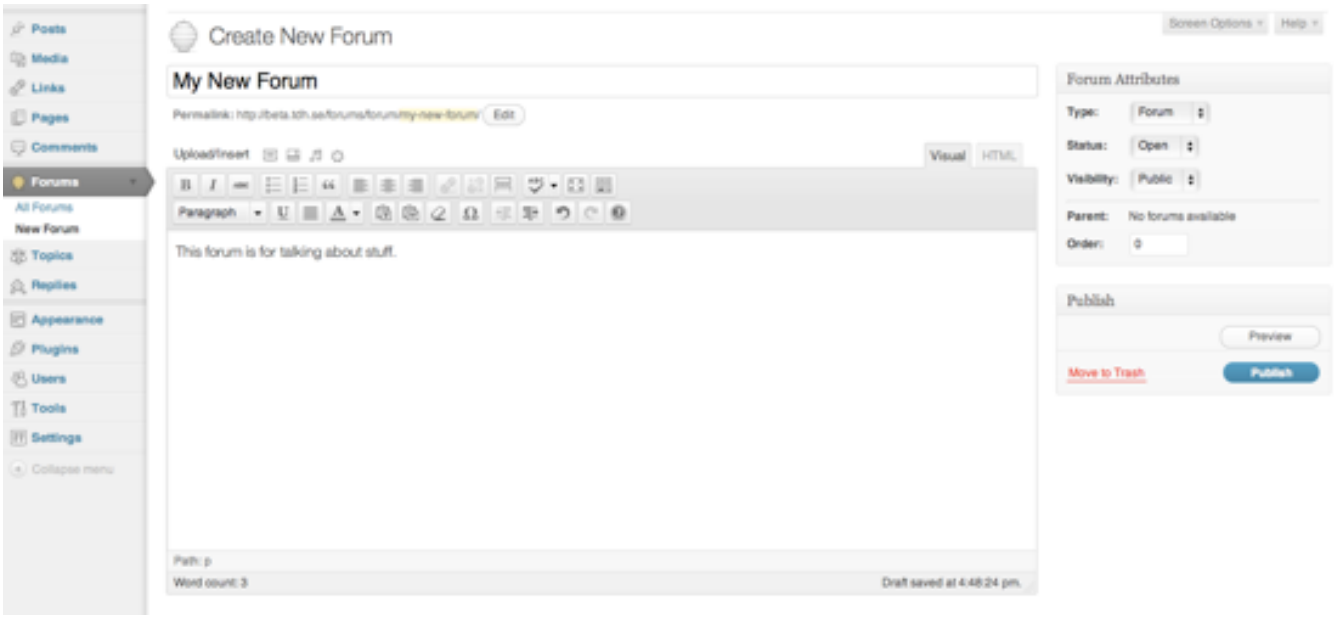


The forums page, without any forums unfortunately.

There we go: all set up and ready to go. Too bad there aren't any forums, nor posts… yet!

Managing bbPress Forums

Getting bbPress set up and ready to go is a breeze, but if you actually want some action in your brand new forums, then you'll need to create a forum. This is easily done under "Forums" in the admin area. Just click "New Forum," and you'll get a familiar-looking screen to create a forum.

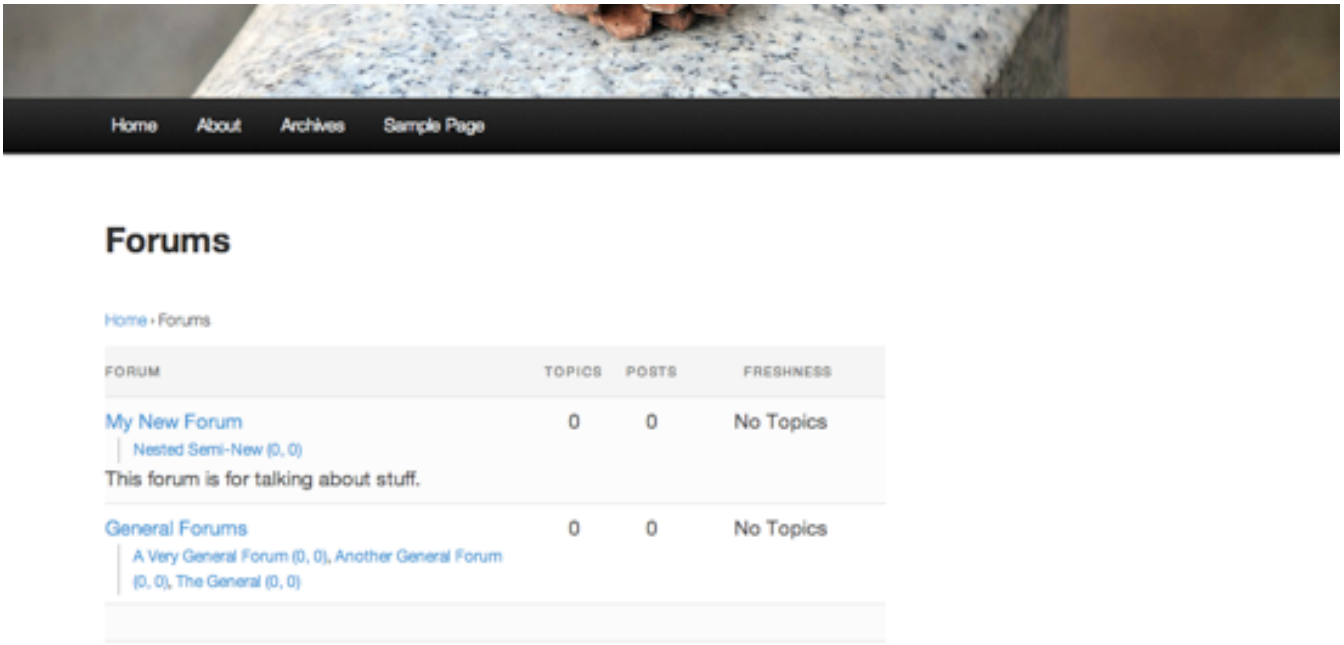


Create a forum.

This is pretty self-explanatory. The one thing you'll need to be careful with is the box in the top-right corner. These are the settings that enable you to control whether a forum is open or closed, whether it is a forum or a category, and who should see it. When you have created multiple forums, the "Parent" and "Order" options will show up, allowing you to nest forums (much like "Pages") and sort them (also like Pages).

To make a long story short, with a few forums created, users will soon be able to post in your forums. Depending on your settings, they may need to

sign up, but that's a different matter and depends on what kind of website you're running.



A forums page.

Managing "topics," which are new posts, and "replies," which are replies to topics, is easy enough. These show up under their respective sections in the WordPress admin area, and they behave much like posts and comments. That's no surprise because bbPress has the same model as standard posts and Pages, using custom post types. This will also make it easy to style the bbPress forum should you want to, something we'll look at more closely later.

Finally, one thing to know when running bbPress on a non-English website: localization projects are on GlotPress, and you can get a translation by using the options at the bottom of the entry for your selected language. You'll need to upload these to the `wp-content/plugins/bbpress/bbp-languages/` folder, and the file should be called bbpress-sv_SE.mo, where

sv_SE should be swapped for your language of choice. Hopefully, we'll be able to store these files in the `wp-content/languages/` folder later, but this doesn't work for me right now.

EXTENDING BBPRESS

Although bbPress is now a WordPress plugin and not a standalone system, you'll find plugins that extend its functionality. Quite a few actually: for displaying the latest posts in widgets, adding signatures and whatnot.

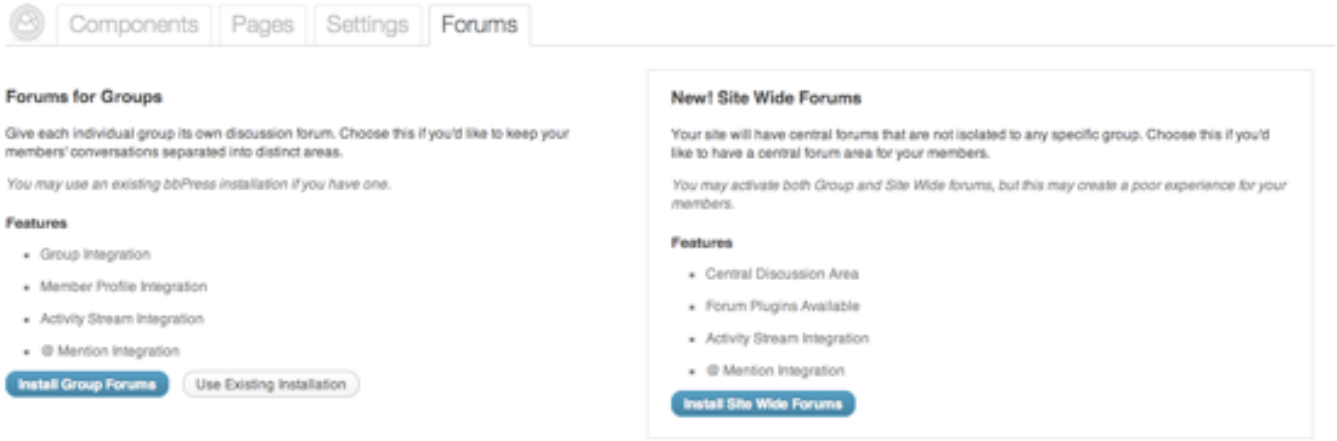
Your starting point for bbPress-related plugins is the plugins section of the bbPress website and, of course, the WordPress plugin directory (begin with a search).

One thing, though: make sure any plugin you choose is made for bbPress 2.0 (i.e. the plugin version). Older plugins made for the 1.x branch will not work.

BuddyPress And bbPress

BuddyPress, the plugin that enables you to create your own Facebook-like community on a WordPress website, work just great with bbPress. That should come as no surprise because the plugin still ships with the forum component (bbPress) built in. But this forum component is for enabling forums for your BuddyPress groups. Groups are exactly what they sound like: members can join them, even create their own (depending on your settings), and discuss various topics in them. With forums enabled for groups, every group will get a forum. This is still true with BuddyPress 1.5, despite there being a standalone bbPress plugin now. If you want forums for your BuddyPress-powered groups, then either choose an existing bbPress

installation or install one in the BuddyPress settings. And yes, this is a bit confusing.



The settings page for BuddyPress forums.

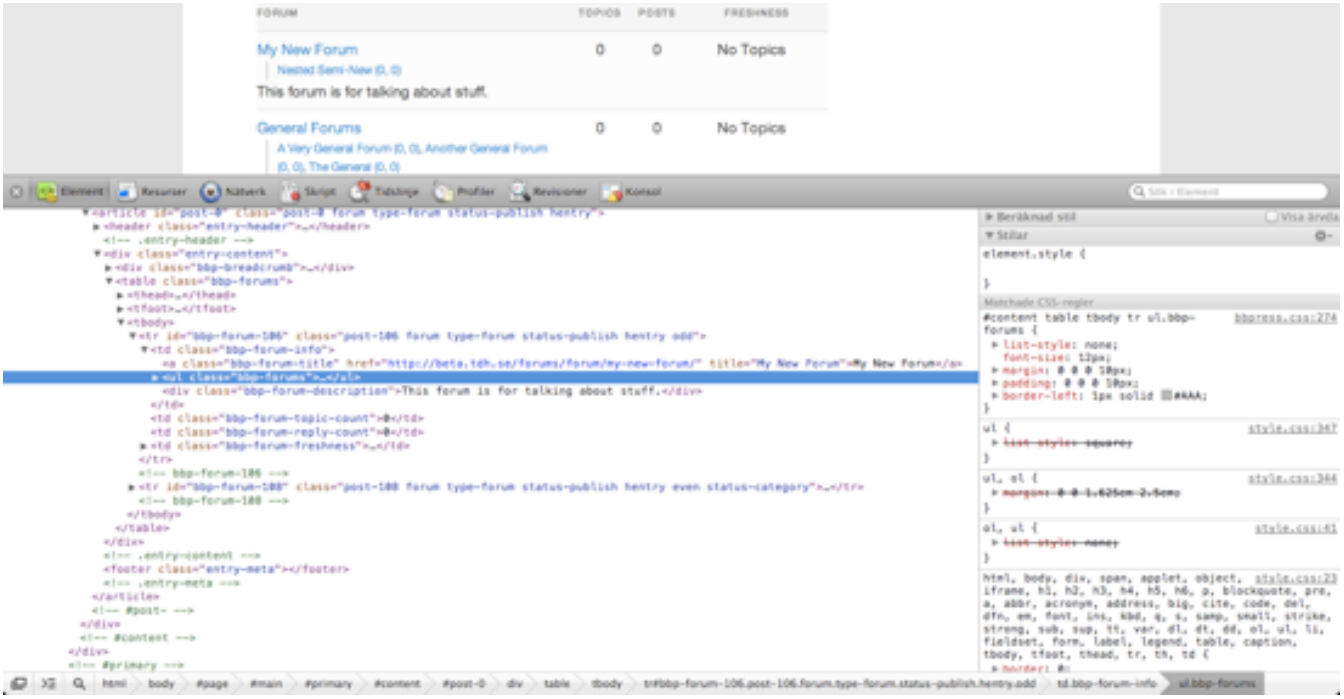
With bbPress 2.0 and the shift from standalone forum software to WordPress plugin, you can rest assured that BuddyPress and bbPress still work well enough together. The option for installing forums site-wide is on the settings page for the BuddyPress forum, and it actually installs the bbPress plugin, rather than rely on the built-in forum component in the BuddyPress plugin. BuddyPress and the bbPress plugin integrate nicely out of the box, but not for group forums. Instead, your posts in the forums will show up in the BuddyPress activity stream; surely we'll see some cool plugins in the future that leverage both BuddyPress and the bbPress plugin, tying the two even closer together.

All in all, there is no reason not to combine bbPress with BuddyPress if you need more community features than just a forum on your website.

Making bbPress Look Good

While your forums will work well out of the box, as you no doubt have gathered from the screenshots earlier in this article, you might want to make bbPress better suit the look of your website. You've already seen the default styles of bbPress, which you can tweak easily enough: just add CSS to your theme's style sheet!

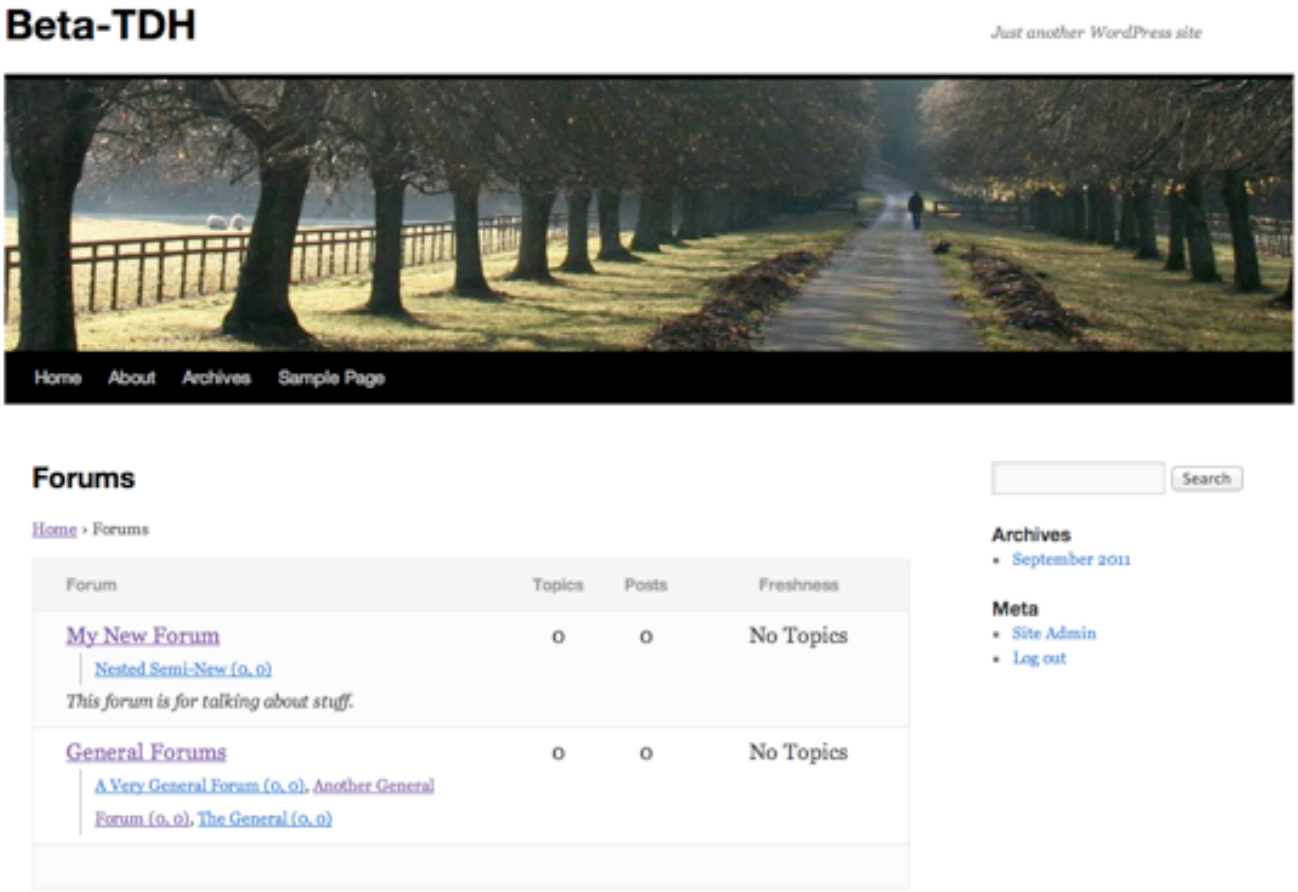
Doing this is easy: just inspect the code of the forums with your favorite Web inspector (such as Firebug or the built-in inspector in Chrome or Safari), and find the classes that you'll need to style the forums.



The `ul.bbp-forums` *class gives you control.*

If you want more control, perhaps to break from the default layout of the forums, you can add additional template files to your WordPress theme. The bbPress plugin is already compatible with Twenty Ten, the previous default theme. In the `bbpress` folder, look at the files in `bbp-themes/bbp-`

`twentyten/` and you'll get an idea what you can do. Simply changing the theme to Twenty Ten (instead of Twenty Eleven, which was shown earlier in this article) will give us something different and more attuned to our theme.



We get a different look when using Twenty Ten.

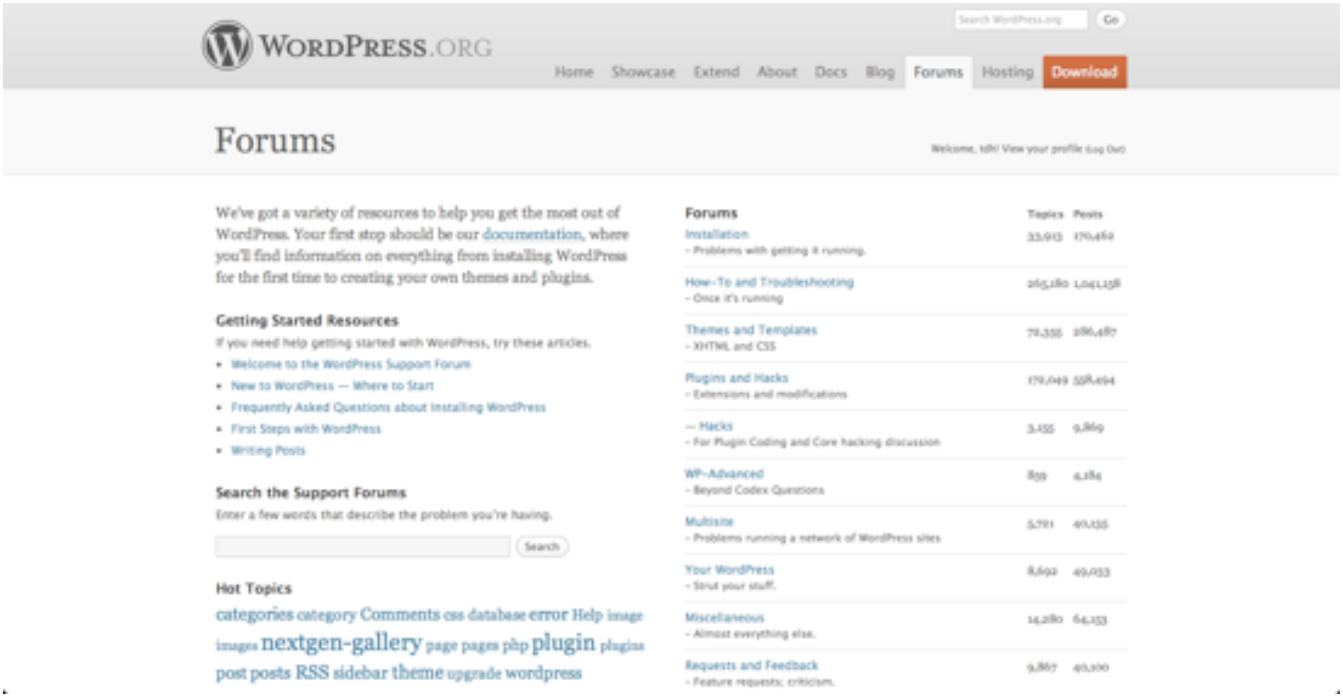
How you style the forums will depend on how much you want to deviate from the default look and feel. If everything is where it should be, then you'll be able to make the forums looks good and fit your theme just by adding styles to your theme's style sheet. But if you want to move things around a lot, then you'll probably have to create your own template files. Consult the files in the `bbpress/bbp-themes/bbp-twentyten/` folder to get an idea of what can be done, while we wait for bbPress to publish proper

documentation. Because forums are really just a custom post type, you'll likely be able to find your way around if you've worked with them before.

Three Websites That Use bbPress

Want to see some bbPress forums in action, other than bbPress.org itself or Twenty Ten and Twenty Eleven themes with the plugin activated?

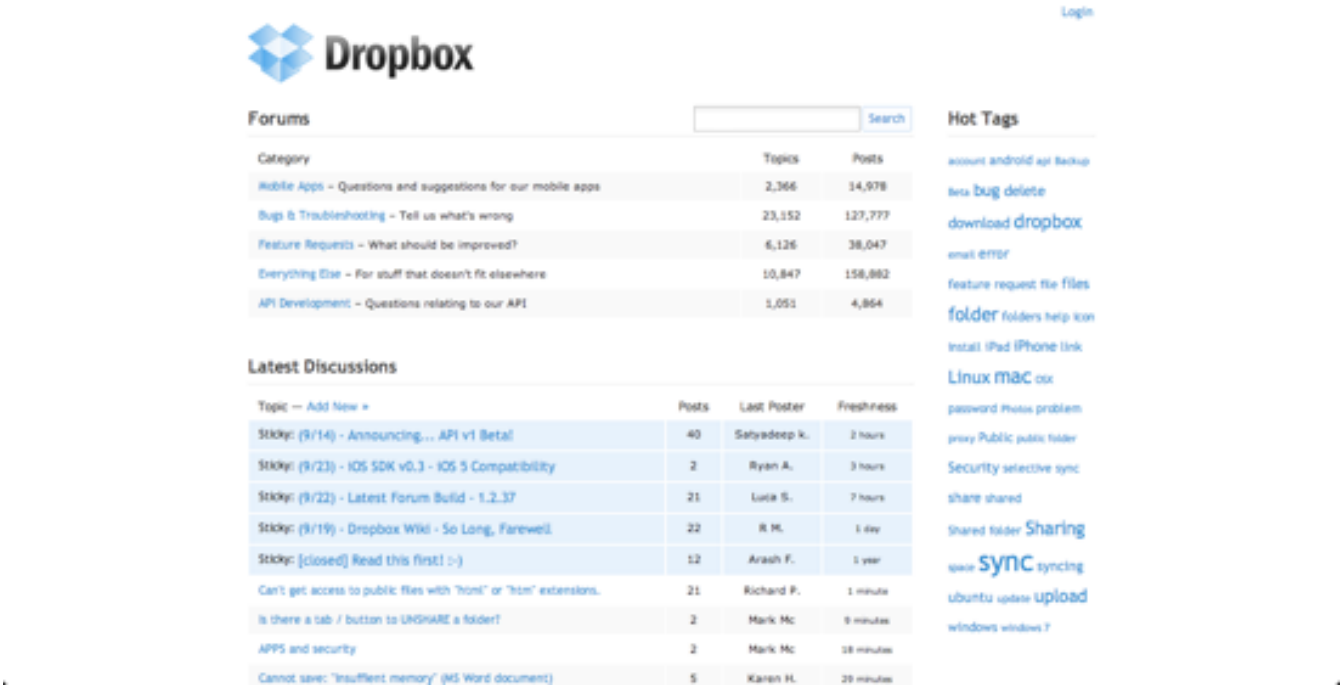
WORDPRESS.ORG



WordPress Forums

While using bbPress on WordPress.org might not exactly qualify as eating one's own dog food, this is where it started after all.

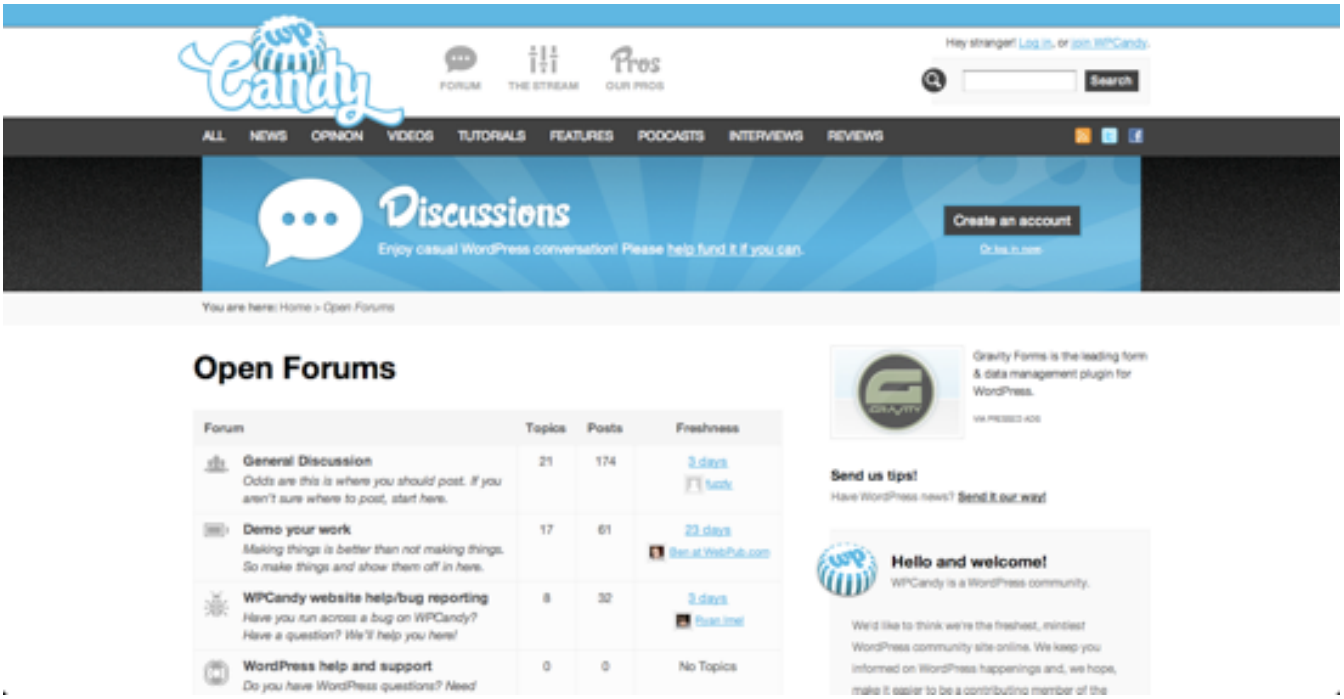
DROPBOX



Dropbox Forums

The syncing service Dropbox has been using bbPress forums for quite some time, with a pretty simple, standard look. This is just the standalone version, and it shows that bbPress is ready for prime time.

WPCANDY



Discussions on WPCandy

The forums section of WPCandy is a great example of how bbPress can be easily integrated in an existing WordPress theme.

What's Next?

Personally, I'm thrilled to see bbPress become a WordPress plugin. We've seen plugins that add forum features to WordPress in the past, but I haven't been comfortable running any of them, to be honest. Whenever I've needed forums, I've used software such as the excellent Vanilla. Some people have suggested the BuddyPress plugin, but that's a bit much if all you need is a simple forum for discussions.

With bbPress 2.0, this isn't an issue anymore, and although documentation isn't available yet, getting started is easy enough. You'll probably want to

add features to your forums, and that's easy with additional plugins. And because bbPress is really just a custom post type for your WordPress website, using actual registered users, you can use existing plugins to achieve things such as moderator privileges and whatnot. We can anticipate a boom of bbPress-compatible plugins in the near future that will make our forums even better and more interesting.

For now, let's play with what we have, which is usually more than enough.

WordPress Multisite: Practical Functions And Methods

Kevin Leary

Multisite is a powerful new feature that arrived with the release of WordPress 3.0. It allows website managers to host multiple independent websites with a single installation of WordPress. Although each “website” in a network is independent, there are many ways to share settings, code and content throughout the entire network.

Since the beginning of the year, I've been developing themes and plugins for a WordPress Multisite-powered content network. During that time I've learned many powerful tips and tricks unique to Multisite. This guide will introduce you to a few Multisite-specific functions, along with real-world programming examples that you can begin using today. Hopefully, it will open your eyes to a few of the new possibilities available in Multisite.

Why Use Multisite?

Multisite is a great option for freelancers, businesses and organizations that manage multiple WordPress websites. Whether you're a freelancer who wants to provide hosting and maintenance to clients, a college organization looking to centralize the management of your websites, or a large news publisher trying to isolate silos for different departments, Multisite is the answer.

Managing multiple websites with a single installation of WordPress enables you to easily upgrade the core, plugins and themes for every website in a network. You can share functionality across multiple websites with network plugins, as well as standardize design elements across multiple websites using a parent theme.

OVERVIEW OF BENEFITS

- Users are able to easily access and manage multiple websites with a single user account and profile.
- Users can access a particular website or every website using the same account.
- Information from one website can be completely isolated from others.

- Information from one website can be easily shared with others.
- Theme functionality can be shared across multiple websites using a parent-child theme relationship or a functionality plugin.
- Updates and upgrades can be rolled out across multiple websites in less time, reducing overhead and maintenance costs.
- Customizations to WordPress can be efficiently distributed in a centralized, cascading method using network-wide plugins.

I won't explain how to install and configure Multisite. If you need help, plenty of great articles are available in the WordPress Codex.

Working With Multisite Functions

Multisite-enabled WordPress installations contain additional functions and features that theme developers can use to improve the experience of a website. If you find yourself developing themes and plugins for WordPress Multisite, consider the following tips to customize and improve the connectivity of the network.

DISPLAYING INFORMATION ABOUT A NETWORK

You might find yourself in a situation where you would like to display the number of websites or users in your network. Providing a link to the network's primary website would also be nice, so that visitors can learn more about your organization.

Multisite stores global options in the `wp_sitemeta` database table, such as the network's name (`site_name`), the **administrator's email address**

(`admin_email`) and the **primary website's URL** (`siteurl`). To access these options, you can use the `get_site_option()` function.

In this example, I've used the get_site_option() function along with `get_blog_count()` and `get_user_count()` to display a sentence with details about a network.

```
<?php if( is_multisite() ): ?>

   The <a href="<?php echo
esc_url( get_site_option( 'siteurl' ) ); ?>"><?php echo
esc_html( get_site_option( 'site_name' ) ); ?> network</a>
currently powers <strong><?php echo get_blog_count(); ?></
strong> websites and <strong><?php echo get_user_count(); ?></
strong> users.

<?php endif; ?>
```

This small snippet of code will display the following HTML:

```
The <a href="http://www.smashingmagazine.com">Smashing
Magazine network</a> currently powers <strong>52</strong>
websites and <strong>262</strong> users.
```

Many useful Multisite functions can be found in the `/wp-includes/ms-functions.php` file. I highly suggest browsing the Trac project yourself. It's a great way to find new functions and to become familiar with WordPress coding standards.

BUILD A NETWORK NAVIGATION MENU

Many networks have consistent dynamic navigation that appears on all websites, making it easy for visitors to browse the network. Using the $wpdb database class, along with the `get_site_url()`, `home_url()`, `get_current_blog_id()`, `switch_to_blog()` and `restore_current_blog()` functions, we can create a fully dynamic

network menu, including a class (`.current-site-item`) to highlight the current website.

The SQL query we've created in this example has the potential to become very large, possibly causing performance issues. For this reason, we'll use the Transients API, which enables us to temporarily store a cached version of the results as network website "transients" in the `sitemeta` table using the `set_site_transient()` and `get_site_transient()` functions.

Transients provide a simple and standardized way to store cached data in the database for a set period of time, after which the data expires and is deleted. It's very similar to storing information with the Options API, except that it has the added value of an expiration time. Transients are also sped up by caching plugins, whereas normal options aren't. Due to the nature of the expiration process, never assume that a transient is in the database when writing code.

The SQL query will run every two hours, and the actual data will be returned from the transient, making things much more efficient. I've included two parameters, `$size` and `$expires`, allowing you to control the number of posts returned and the expiration time for the transient.

One of the most powerful elements of this example is the use of `switch_to_blog()` and `restore_current_blog()`. These two Multisite functions enable us to temporarily switch to another website (by ID), gather information or content, and then switch back to the original website.

Add the following to your theme's functions.php file:

```
/**
 * Build a list of all websites in a network
 */
```

```
function wp_list_sites( $expires = 7200 ) {
   if( !is_multisite() ) return false;

   // Because the get_blog_list() function is currently
flagged as deprecated
   // due to the potential for high consumption of resources,
we'll use
   // $wpdb to roll out our own SQL query instead. Because the
query can be
   // memory-intensive, we'll store the results using the
Transients API
   if ( false === ( $site_list =
get_transient( 'multisite_site_list' ) ) ) {
      global $wpdb;
      $site_list = $wpdb->get_results( $wpdb->prepare('SELECT
* FROM wp_blogs ORDER BY blog_id') );
      // Set the Transient cache to expire every two hours
      set_site_transient( 'multisite_site_list', $site_list,
$expires );
   }

   $current_site_url = get_site_url( get_current_blog_id() );

   $html = '<ul>' . "\n";

   foreach ( $site_list as $site ) {
      switch_to_blog( $site->blog_id );
      $class = ( home_url() == $current_site_url ) ? '
class="current-site-item"' : '';
      $html .= "\t" . '<li>blog_id . '"' . $class . '><a
href="' . home_url() . '">' . get_bloginfo('name') . '</a></
li>' . "\n";
      restore_current_blog();
   }

   $html .= '</ul><!--// end #network-menu -->' . "\n\n";

   return $html;
}
```

(**Please note**: The `get_blog_list()` function is currently deprecated due to the potential for a high consumption of resources if a network contains

more than 1000 websites. Currently, there is no replacement function, which is why I have used a custom `$wpdb` query in its place. In future, WordPress developers will probably release a better alternative. I suggest checking for a replacement before implementing this example on an actual network.)

This function first verifies that Multisite is enabled and, if it's not, returns `false`. First, we gather a list of IDs of all websites in the network, sorting them in ascending order using our custom `$wpdb` query. Next, we iterate through each website in the list, using `switch_to_blog()` to check whether it is the current website, and adding the `.current-site-item` class if it is. Then, we use the name and link for that website to create a list item for our menu, returning to the original website using `restore_current_blog()`. When the loop is complete, we return the complete unordered list to be outputted in our theme. It's that simple.

To use this in your theme, call the `wp_list_sites()` function where you want the network menu to be displayed. Because the function first checks for a Multisite-enabled installation, you should verify that the returned value is not `false` before displaying the corresponding HTML.

```
<?php
// Multisite Network Menu
$network_menu = wp_list_sites();
if( $network_menu ):
?>
<div>
   <?php echo $network_menu; ?>
</div><!--// end #network-menu -->
<?php endif; ?>
```

LIST RECENT POSTS ACROSS AN ENTIRE NETWORK

If the websites in your network share similar topics, you may want to create a list of the most recent posts across all websites. Unfortunately, WordPress

does not have a built-in function to do this, but with a little help from the `$wpdb database class`, you can create a custom database query of the latest posts across your network.

This SQL query also has the potential to become very large. For this reason, we'll use the Transients API again in a method very similar to what is used in the `wp_list_sites()` function.

Start by adding the `wp_recent_across_network()` function to your theme's functions.php file.

```
/**
 * List recent posts across a Multisite network
 *
 * @uses get_blog_list(), get_blog_permalink()
 *
 * @param int $size The number of results to retrieve
 * @param int $expires Seconds until the transient cache
expires
 * @return object Contains the blog_id, post_id, post_date and
post_title
 */
function wp_recent_across_network( $size = 10, $expires =
7200 ) {
   if( !is_multisite() ) return false;

   // Cache the results with the WordPress Transients API
   // Get any existing copy of our transient data
   if ( ( $recent_across_network =
get_site_transient( 'recent_across_network' ) ) === false ) {

      // No transient found, regenerate the data and save a
new transient
      // Prepare the SQL query with $wpdb
      global $wpdb;

      $base_prefix = $wpdb->get_blog_prefix(0);
      $base_prefix = str_replace( '1_', '' , $base_prefix );
```

```
      // Because the get_blog_list() function is currently
flagged as deprecated
      // due to the potential for high consumption of
resources, we'll use
      // $wpdb to roll out our own SQL query instead. Because
the query can be
      // memory-intensive, we'll store the results using the
Transients API
      if ( false === ( $site_list =
get_site_transient( 'multisite_site_list' ) ) ) {
         global $wpdb;
         $site_list = $wpdb->get_results( $wpdb-
>prepare('SELECT * FROM wp_blogs ORDER BY blog_id') );
         set_site_transient( 'multisite_site_list',
$site_list, $expires );
      }

      $limit = absint($size);

      // Merge the wp_posts results from all Multisite
websites into a single result with MySQL "UNION"
      foreach ( $site_list as $site ) {
         if( $site == $site_list[0] ) {
            $posts_table = $base_prefix . "posts";
         } else {
            $posts_table = $base_prefix . $site->blog_id .
"_posts";
         }

         $posts_table = esc_sql( $posts_table );
         $blogs_table = esc_sql( $base_prefix . 'blogs' );

         $query .= "(SELECT $posts_table.ID,
$posts_table.post_title, $posts_table.post_date,
$blogs_table.blog_id FROM $posts_table, $blogs_table\n";
         $query .= "\tWHERE $posts_table.post_type =
'post'\n";
         $query .= "\tAND $posts_table.post_status =
'publish'\n";
         $query .= "\tAND $blogs_table.blog_id = {$site-
>blog_id})\n";

         if( $site !== end($site_list) )
```

```
            $query .= "UNION\n";
         else
            $query .= "ORDER BY post_date DESC LIMIT 0,
$limit";
      }

      // Sanitize and run the query
      $query = $wpdb->prepare($query);
      $recent_across_network = $wpdb->get_results( $query );

      // Set the Transients cache to expire every two hours
      set_site_transient( 'recent_across_network',
$recent_across_network, 60*60*2 );
   }

   // Format the HTML output
   $html = '<ul>';
   foreach ( $recent_across_network as $post ) {
      $html .= '<li><a>blog_id, $post->ID ) . '">' . $post-
>post_title . '</a></li>';
   }
   $html .= '</ul>';

   return $html;
}
```

Using this function in your theme is simple. Be certain to check the `return` value before outputting HTML to avoid conflicts with non-Multisite installations.

```
<?php
// Display recent posts across the entire network
$recent_network_posts = wp_recent_across_network();
if( $recent_network_posts ):
?>
<div class="recent-accross-network">
   <?php echo $recent_network_posts; ?>
</div>
<?php endif; ?>
```

RETRIEVE A SINGLE POST FROM ANOTHER WEBSITE IN THE NETWORK

In certain situations, you may find it useful to refer to a single page, post or post type from another website in your network. The `get_blog_post()` function makes this process simple.

For example, you may want to display `the_content()` from an "About" page on the primary website in your network.

```
<?php
// Display "About" page content from the network's primary
website
$about_page = get_blog_post( 1, 317 );
if( $about_page ):
?>
<div class="network-about entry">
   <?php echo $about_page->post_content; ?>
</div>
<?php endif; ?>
```

Did you notice that the entire `$post` object is returned? In this example, we've used only `the_content()`, but far more information is available for other circumstances.

SET UP GLOBAL VARIABLES ACROSS A NETWORK

Starting any WordPress project in a solid local development environment is always important. You might find it handy to have a global variable that determines whether a website is "live" or "staging." In Multisite, you can achieve this using a network-activated plugin that contains the following handy function, assuming that your local host contains `localhost` in the URL:

```
/**
 * Define network globals
```

```
 */
function ms_define_globals() {
   global $blog_id;
   $GLOBALS['staging'] = ( strstr( $_SERVER['SERVER_NAME'],
'localhost' ) ) ? true : false;
}
add_action( 'init', 'ms_define_globals', 1 );
```

When would you use this `$staging` variable? I use it to display development-related messages, notifications and information to improve my workflow.

DISPLAY THE PAGE REQUEST INFORMATION IN A LOCAL ENVIRONMENT

I use the `$staging` global variable to display the number of queries and page-request speed for every page across a network in my local environment.

```
/**
 * Display page request info
 *
 * @requires $staging
 */
function wp_page_request_info() {
   global $staging;
   if ( $staging ): ?>
      <?php echo get_num_queries(); ?> queries in <?php
timer_stop(1); ?> seconds.
   <?php endif;
}
add_action( 'wp_footer', 'wp_page_request_info', 1000 );
```

This is only one of many ways you can use the `ms_define_globals()` function. I've used it to define, find and replace URLs in the content delivery network, to detect mobile devices and user agents, and to filter local attachment URLs.

Conclusion

There is tremendous value in the simplicity of managing multiple websites in a single installation of WordPress. Leveraging WordPress Multisite is quickly becoming a requisite skill among WordPress developers. These techniques should provide a solid foundation for you to build on, so that you can be the next WordPress Multisite theme rock star!

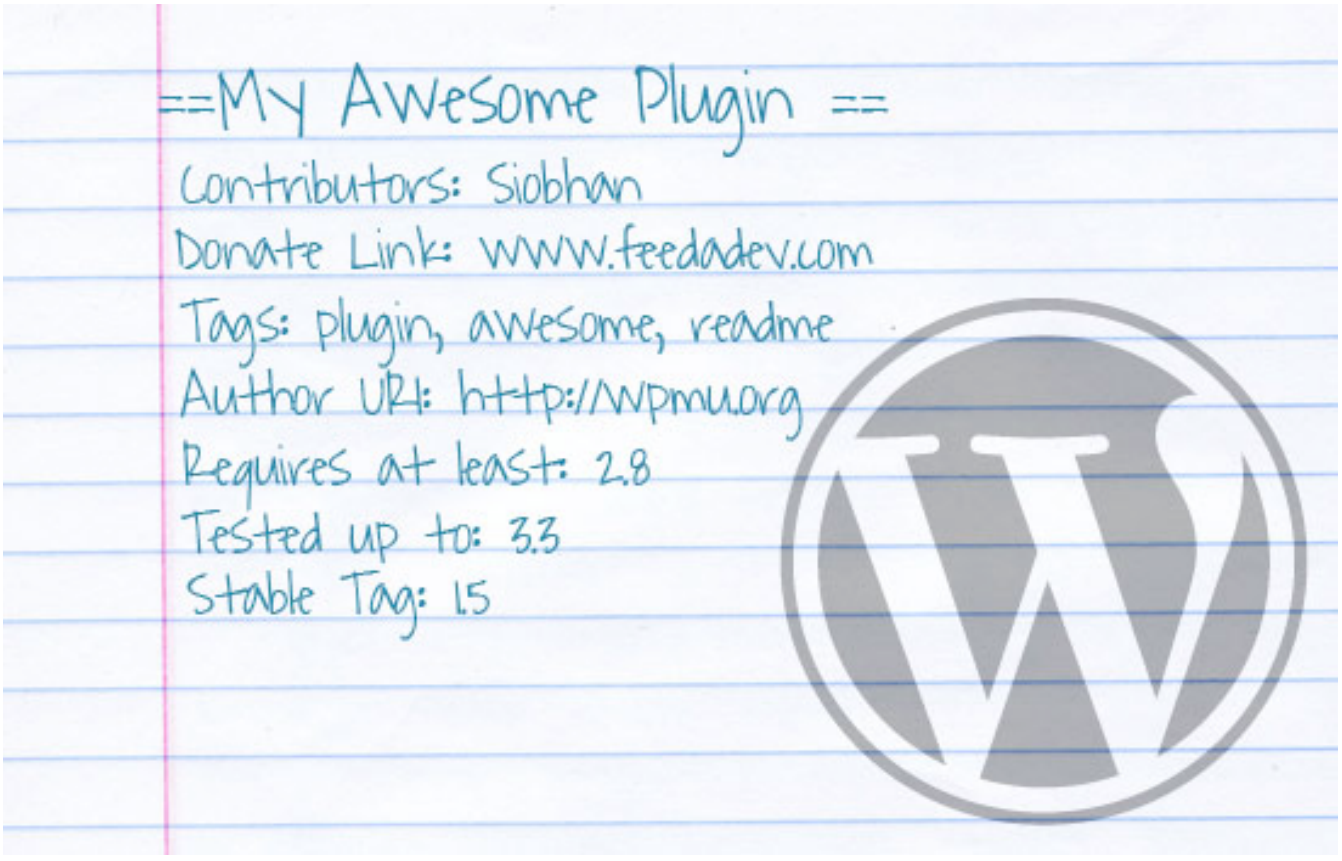
OTHER RESOURCES

- Mastering WordPress Multisite series, WPCandy
- "Create a [Multisite] Network," WordPress Codex
- "WordPress MultiSite Subdomains on MAMP," Perishable Press
- "Migrating Multiple Blogs Into WordPress 3.0 Multisite," WordPress Codex
- OpenView Venture Partners, a live example of a WordPress Multisite network
- "Transients, Caching and the Complexities of Multisite," The Otto and Nacin Show, WordCamp San Francisco 2011

How To Improve Your WordPress Plugin's Readme.txt

Siobhan McKeown

If you're a plugin developer and you just love to write code, then writing a readme.txt file for a plugin in WordPress' repository might be your idea of hell. When you've written all of that lovely code, why must you spend time writing about how to use it?



Unfortunately, some plugin developers view writing a readme.txt file as the least important part of their job. So, we end up with things like the following:

- Descriptions with only one line,
- Descriptions with hardly any information,
- No translation into English,
- Typos,
- Confusion.

Why You Should Care About Your Readme.txt

A poorly written readme.txt does not necessarily indicate that the plugin is poorly written; the code could be mind-blowingly good. But it does give the impression of an overall lack of attention to detail and a lack of care for end users. You see, no one will notice if a readme file is particularly awesome, but they **will notice if it's bad**.

You needn't view the readme file as pain-in-the-butt homework that you've got to do after all the fun you've had with coding. A readme benefits you. Here's how:

- Gives you an opportunity to say **why your plugin is so good**,
- Shows off your plugin's **features**,
- Makes it **easy for users** to install and use the plugin,
- **Anticipates** support questions with a thorough FAQ,
- **Links** to your website and other products,
- **Upsells** your commercial services.

In this article, we'll look at how to improve your WordPress readme file so that it benefits both your users and you.

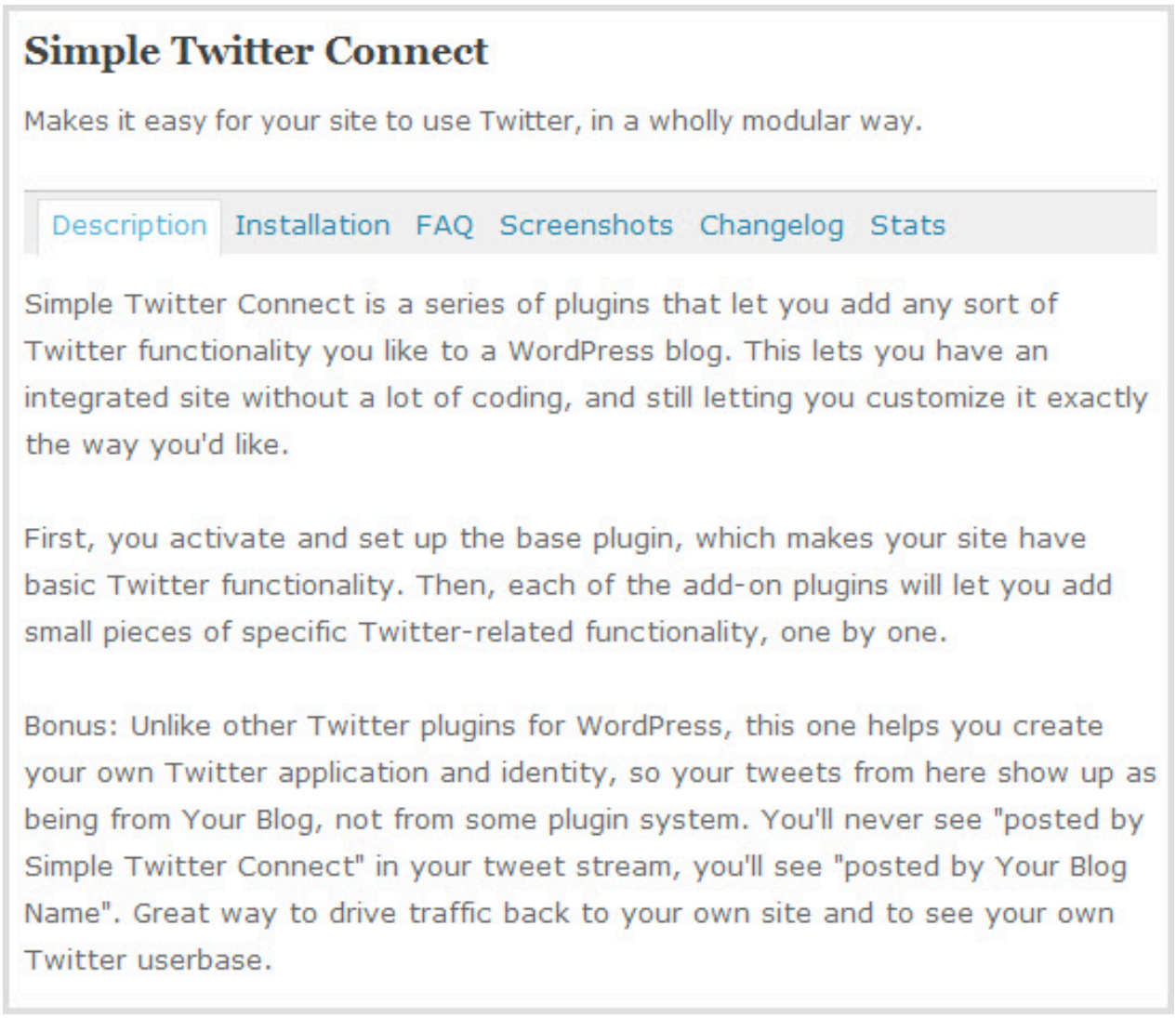
But first, the mechanics.

Using (Quasi-)Markdown

Markdown is a text-to-HTML conversion tool, developed by John Gruber and Aaron Schwartz, that is used for readme files of plugins in the WordPress repository.

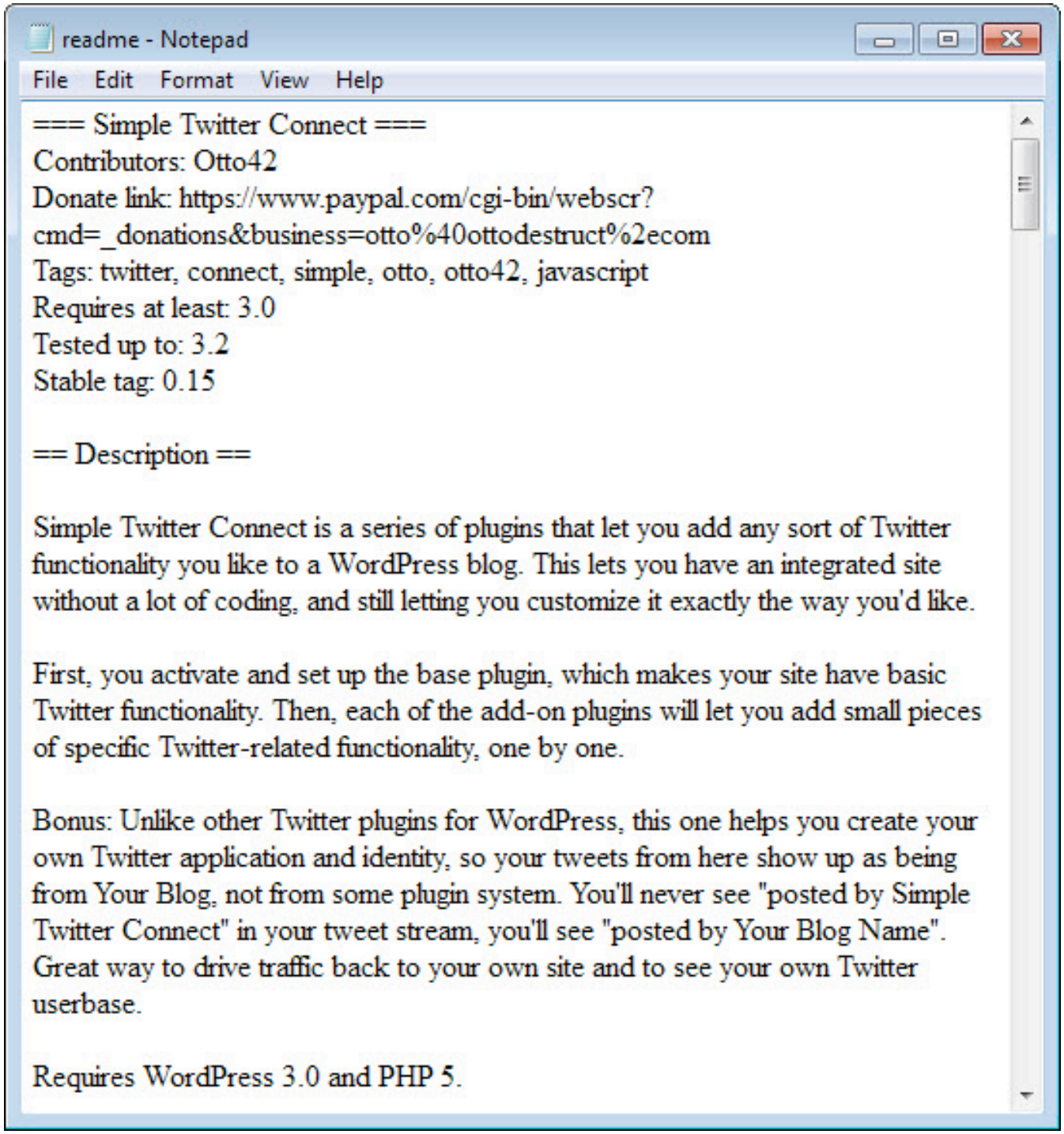
You write the readme in Markdown, and Markdown will convert it to HTML for the WordPress directory page, but it will still look good in WordPress' back end and when you peek at it in your favorite text editor.

Take a look at the directory page for Simple Twitter Connect:



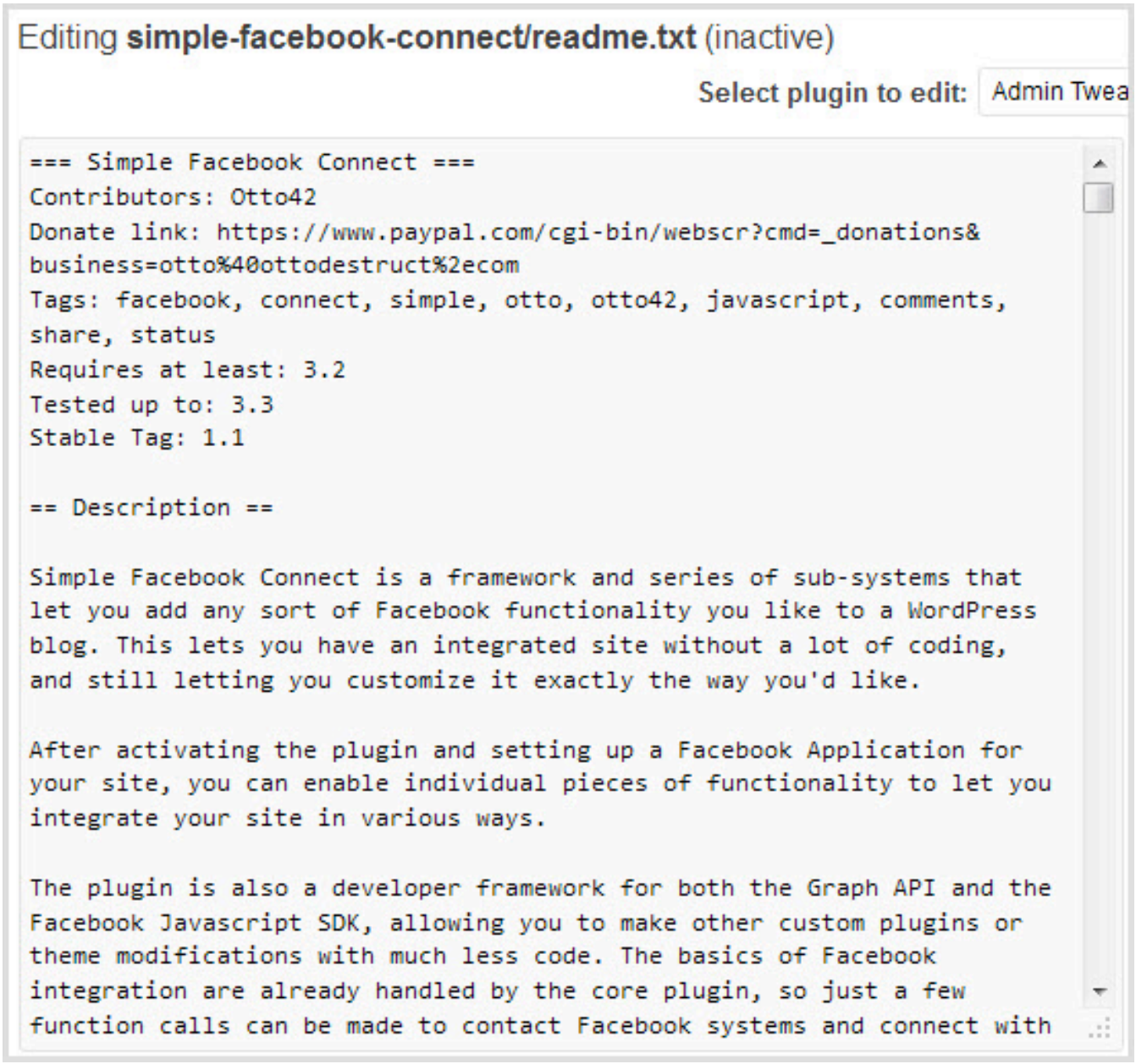
Simple Twitter Connect's listing in the WordPress plugin directory.

Here's how it appears when I download and extract the readme:



Simple Twitter Connect's readme file in Notepad.

And here's how it appears in WordPress' back end:



Simple Twitter Connect's readme on the "Edit Plugin" page in the WordPress Dashboard.

It is readable no matter what medium it is viewed in. Imagine how difficult reading the readme in a text editor would be if it was marked up in HTML. Markdown makes your readme clear, readable and semantic.

WordPress.org uses what Mark Jacquith describes as quasi-markdown. It basically works like markdown, and much of the syntax is the same, but there are a few important differences, particularly in the headers.

To get you started with your WordPress readme, here's a table containing the syntax that you'll need.

Syntax	Example	Outputs as	Use for
=== foo ==	=== Plugin Name ===	Plugin Name	Plugin name
== foo ==	==Description==	Description	Tabbed section headings
= foo =	=Inline Heading=	Inline heading	Headings in the body of the readme
* foo	* list item, *list item	 - list item - list item 	Unordered list
1. foo, 2. foo	1. list item, 2. list item	 - list item - list item 	Ordered list
foo bar	You talking to me?	<blockquote>You talking to me?</blockquote>	block quotes
foo	*emphasis*	emphasis	Italics
foo	**bold**	bold	Bold
foo’	wp-config.php’	wp-config.php	Any code you want to display (including shortcodes)
[link] (http://foobar.com “foobar”)	[WordPress](http://wordpress.org/ “Your favorite software”)	WordPress	Links
<http://foobar.com>	<http://wordpress.org>	http://wordpress.org	Links

[youtube http://foobarvideo.com]	[youtube http://www.youtube.com/watch?v=CVmGBoPx6Ms]	<object width='532' height='325'><param name='movie' value='http://www.youtube.com/v/CVmGBoPx6Ms?fs=1'></param><param name='allowFullScreen' value='true'></param><param name='allowscriptaccess' value='never'></param><embed src='http://www.youtube.com/v/CVmGBoPx6Ms?fs=1' type='application/ x-shockwave-flash' allowscriptaccess='never' allowfullscreen='true' width='532' height='325'></embed></object>	Video

Now that you know how to write in WordPress-flavored Markdown, let's get started writing a readme.

Writing Your Readme.txt

Writing a readme is not hard, but putting a bit of **thought** into it beforehand does pay off.

- What do you think end users will need in order to get optimal use from your plugin?
- Do you need to include any special instructions or code snippets?
- What problems do you anticipate users running into?
- How can you make your plugin as attractive as possible?

- In a saturated market, what makes your plugin stand out?

With a bit of thought, you can produce a useful readme that shows off the best aspects of your plugin.

When writing the readme, remember that if people are able to use your plugin properly, then they will be less likely to come knocking on your door asking for help.

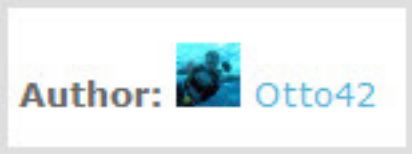
Let's look at each of the main readme sections to see what you can do to improve it. As a reference point, we'll use the readme for Simple Twitter Connect, which is concise, useful and well-formed.

Plugin Header

The plugin's header contains basic information about your plugin that the user will take in at a glance. This information will be displayed in the plugin's directory as a quick reference for users.

Here's what you'll need to complete it:

- **Contributors**
  Anyone who has contributed to the plugin. You should use a contributor's user name from WordPress.org, linking it to their WordPress.org profiles, which list all of the contributor's plugins. Unfortunately, many developers don't mention their WordPress user name, which means they miss out on the opportunity to showcase the rest of their work.

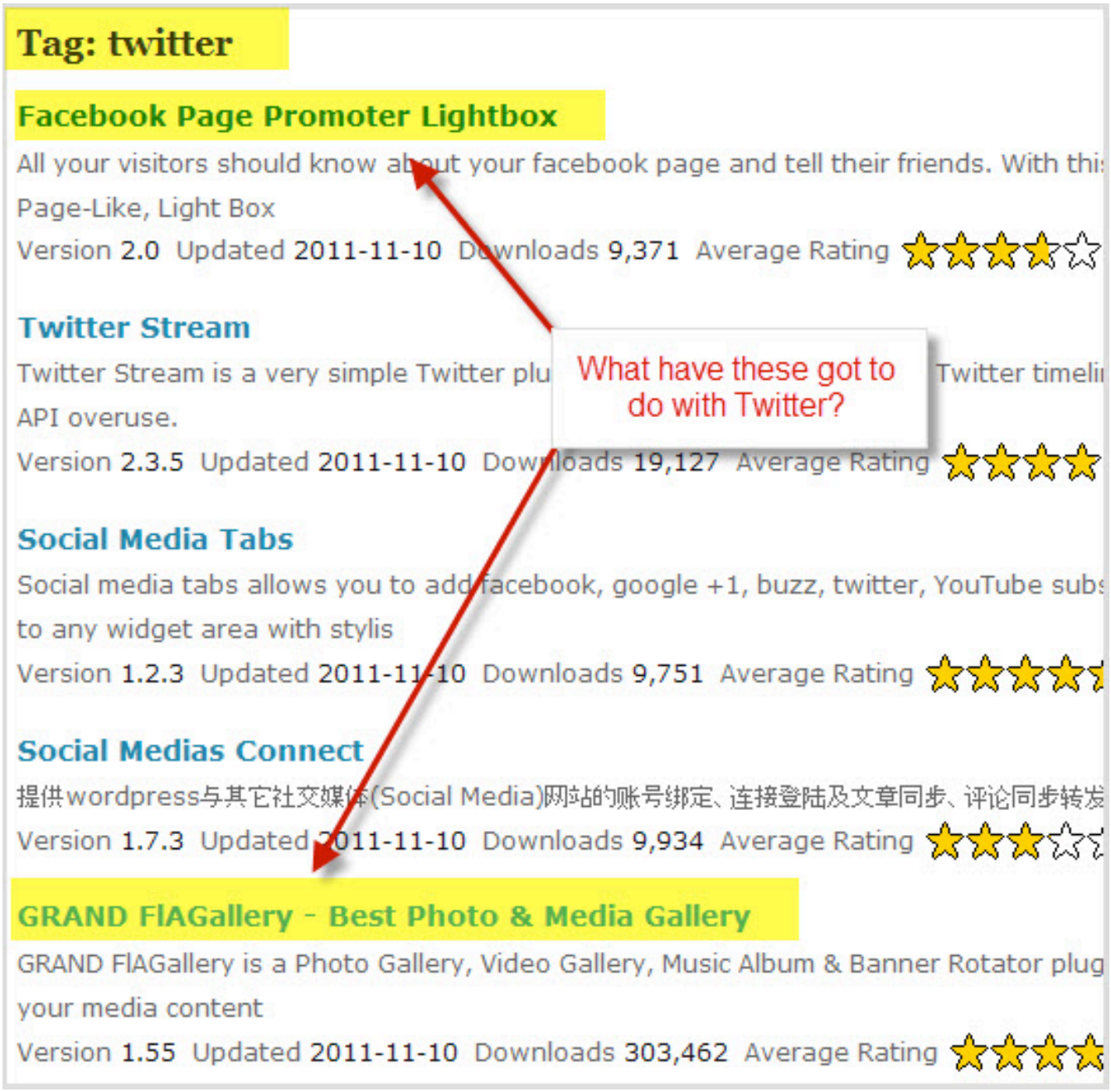
  A contributor tag done right.

  A contributor tag done wrong.

- **Donate link**
  This is your chance to get a little something back from the WordPress community. If you don't have a donate link, you aren't giving people the opportunity to thank you!

- **Tags**
  As on a blog, tags are used to categorize content. In a directory of 15,000+ WordPress plugins, **using tags effectively** makes it easy for people to find your plugin. Keep the tags relevant, and think of what people who need your plugin might be searching for. Some developers tag their plugins with words that are only tangentially related.



This listing for the Twitter tag isn't helpful because it's bloated with irrelevant content. As WordPress community members and contributors, we have a responsibility to keep the directory of plugins as intuitive as possible.

- **Author URI**
  Your home page.

- **Plugin URI**
  The plugin's home page. Note that the Author URI and Plugin URI aren't included in the readme template provided by WordPress. Including them is definitely a good thing because they direct people to your respective home pages.
- **Requires at least**
  This is the version of WordPress that your plugin requires. Some people don't upgrade WordPress (some for valid reasons and some for dumb reasons). Tell them whether WordPress will work with their version.
- **Tested up to**
  Another piece of important information. This tells people which version of WordPress your plugin has been tested up to.
- **Stable tag**
  The stable tag tells WordPress which version of the plugin should appear in the directory. This should be in numeric format, which is much easier for WordPress to deal with. Aim for numbers like 1.5, 0.5 or whatever version you're at. If your stable version is in the trunk in Subversion, then you can specify "trunk," but that is the only time you should use words instead of numbers.

Here's what the header information for Simple Twitter Connect looks like:

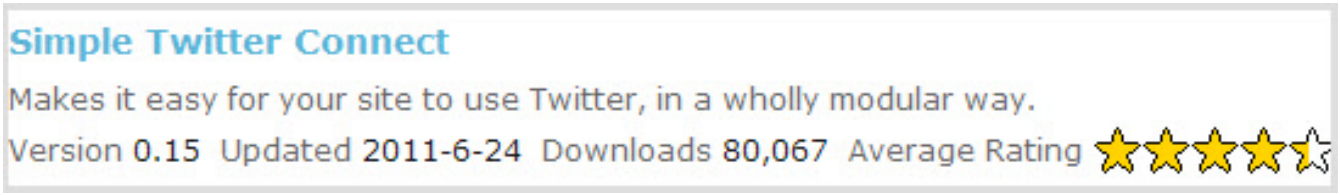
```
=== Simple Twitter Connect ===
Contributors: Otto42
Donate link: https://www.paypal.com/cgi-bin/webscr?
cmd=_donations&business=otto%40ottodestruct%2ecom
Tags: twitter, connect, simple, otto, otto42, javascript
Requires at least: 3.0
Tested up to: 3.2
Stable tag: 0.15
```

TIPS FOR WRITING HEADER INFORMATION

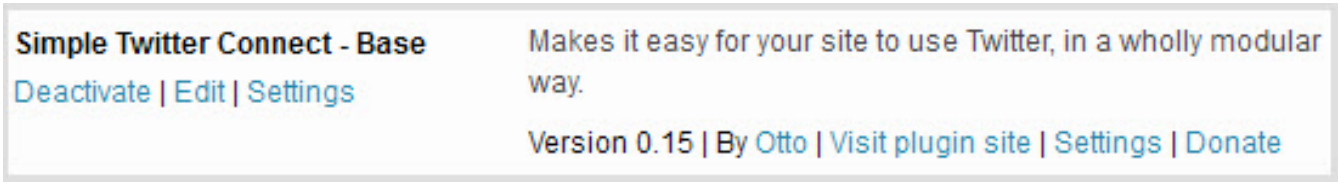
- Use WordPress.org user names for **contributors**.
- Use a **number** for the stable tag.
- Include **donation** link to make some cash.
- Keep tags **relevant**.
- **Test** with the most recent WordPress version.

The Short Description

The short description is what appears on the WordPress.org page that lists plugins.

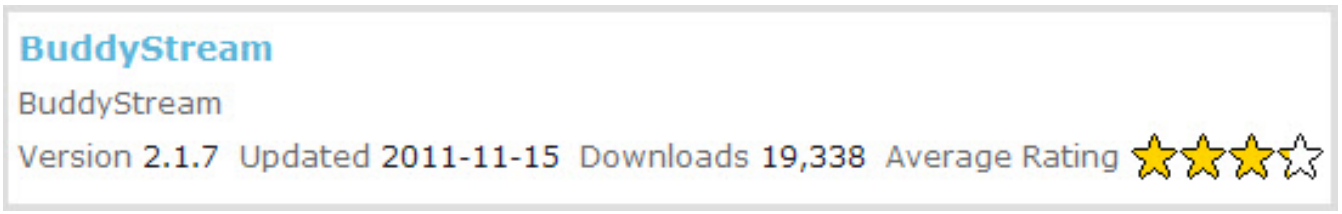


It also appears in the user's list of installed plugins:



It's your chance to craft a punchy 150-character description that will make people want to install your plugin.

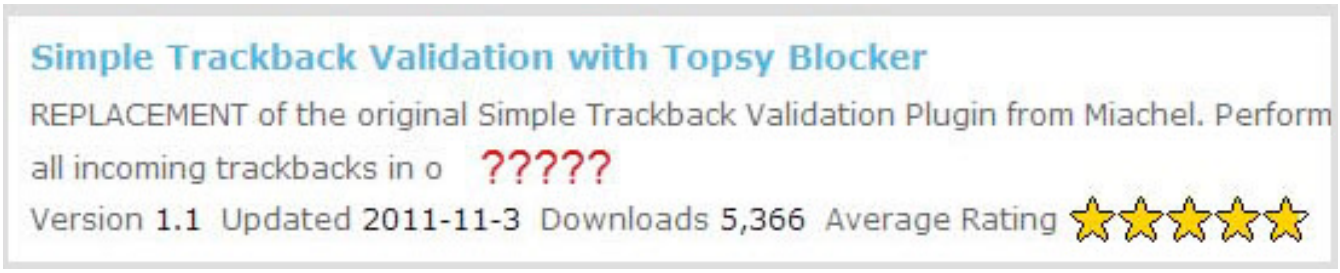
Unfortunately, some people miss the mark:



I'm not sure what this one is all about:

None of these short descriptions tell you anything about what the plugin does. They're totally useless in providing information to WordPress users.

Another problem is when developers do not add a short description at all. In that case, the 150 characters are excerpted from the long description. Check out the short description for [Simple TrackbackSimple Trackback Validation with Topsy Blocker]:



The short description cuts off in the middle of a word. We know that the plugin performs a test on trackbacks, but why? What for? How?

SHORT DESCRIPTIONS THAT GET IT RIGHT

- VideoPress
  "Manage and embed videos hosted on VideoPress. Requires a WordPress.com blog with the VideoPress premium upgrade."
- Yet Another Related Posts Plugin
  "Display a list of related entries on your site and feeds based on a unique algorithm. Templating allows customization of the display."
- BuddyPress
  "Social networking in a box. Build a social network for your company, school, sports team or niche community."
- The Google+ Button
  "Adds the Google +1 button to your site so your visitors can vote to tell the world how great your site is!"

TIPS FOR WRITING A SHORT DESCRIPTION

- Stick to **under 150** characters.
- Don't rely on the text being pulled from the **long description**.

- Tell people what the plugin **will do** for them.
- **Don't worry about technical details**. You can provide them in the long description.
- Tell people whether the plugin has any **requirements** to work, such as a subscription.

Long Description

The long description is where you really go to town. You've hooked people with the short description; **now it's time to sell your plugin**. Let's check out the long description for Simple Twitter Connect:

```
== Description ==
Simple Twitter Connect is a series of plugins that let you add
any sort of Twitter functionality you like to a WordPress
blog. This lets you have an integrated site without a lot of
coding, and still letting you customize it exactly the way
you'd like.

First, you activate and set up the base plugin, which makes
your site have basic Twitter functionality. Then, each of the
add-on plugins will let you add small pieces of specific
Twitter-related functionality, one by one.

Bonus: Unlike other Twitter plugins for WordPress, this one
helps you create your own Twitter application and identity, so
your tweets from here show up as being from Your Blog, not
from some plugin system. You'll never see "posted by Simple
Twitter Connect" in your tweet stream, you'll see "posted by
Your Blog Name". Great way to drive traffic back to your own
site and to see your own Twitter userbase.

Requires WordPress 3.0 and PHP 5.

**Current add-ons**
* Login using Twitter
* Comment using Twitter credentials
```

```
* Users can auto-tweet their comments
* Tweet button (official one from twitter)
* Tweetmeme button
* Auto-tweet new posts to an account
* Manual Tweetbox after Publish
* Full @anywhere support
* Auto-link all twitter names on the site (with optional
hovercards)
* Dashboard Twitter Widget

**Coming soon**
* More direct retweet button (instead of using Tweetmeme)
* (Got more ideas? Tell me!)

If you have suggestions for a new add-on, feel free to email
me at otto@ottodestruct.com.
Want regular updates? Become a fan of my sites on Facebook!

http://www.facebook.com/apps/application.php?id=116002660893


http://www.facebook.com/ottopress

Or follow my sites on Twitter!

http://twitter.com/ottodestruct
```

What makes this a good example of a long description?

1. The opening description is **clear and useful**. It tells you what the plugin does and how it works.
2. The **key features stand out** (you can tell that the plugin creates your own Twitter app).
3. The plugin's **requirements** are listed.
4. **Markdown** is used to break up the text into sections and lists, making it easier to read.

5. All **current features** are listed.
6. **Planned features** are listed.
7. The developer has included **links** to his Facebook page and Twitter account.

A good long description is **useful and relevant**. It tells the user what the plugin does and how it does it, and it provide anything else that the developer feels is important. You can also use it to send users to your home page or social-media profiles.

Here are some other plugins with great long descriptions:

- EG Attachments
- Google Maps All-in-One
- Next Gen Gallery

TIPS FOR WRITING A LONG DESCRIPTION

- Use Markdown to bring some **variety to the formatting**. For example, add headings to break up sections, use lists, and put important points in bold.
- Be clear about the plugin's **features**.
- Include any **requirements**, such as other plugins that your plugin relies on, or specific PHP versions.
- Link to any **extended documentation** on your website.
- Don't be afraid to **link** to your website or social profiles.
- If you've got one, a **video** is a great demonstration tool.

Installation

The installation section should include everything that a person needs to get the plugin up and running correctly. Often, you can keep this brief. Check out the instructions for Simple Twitter Connect:

```
1. Upload 'plugin-name.php' to the '/wp-content/plugins/'
directory,
2. Activate the plugin through the 'Plugins' menu in
WordPress.
```

For many plugins, this will suffice. But some plugins need detailed instructions. Check out the ones for these:

- WP Super Cache,
- WordPress MU Domain Mapping.

Users need mammoth instructions to set up both of these plugins. WP Super Cache requires you to look in your `.htaccess` file, and Domain Mapping needs you to move files to `/wp-content/` and to edit wp-`config.php`.

TIPS FOR WRITING INSTRUCTIONS

- Be **brief**, a few lines if possible.
- Include **everything the user needs** to install the plugin.
- Make sure each step **logically** follows the preceding one.
- Include any **code snippets** that the user might need.
- Be **clear and concise**. This is the place for instructions, not sales.

FAQ

The FAQ (frequently asked questions) section is your opportunity to **troubleshoot problems** before they even get posted to WordPress' support forums. OK, so a few people will probably never bother looking at the FAQ to see whether their question is addressed there, which is annoying, but at least you will be able to direct them to the FAQ.

Some FAQs, however, are pretty lousy. In fact, some plugins, even popular ones, lack any FAQ at all. This is a problem: if too many plugin developers don't include an FAQ, then some WordPress users will assume that plugins normally don't have one and will run straight to the support forums. This wastes the time of people who have taken the trouble to include an FAQ.

Let's check out the FAQ for Simple Twitter Connect:

```
== Frequently Asked Questions ==

= Whoa, what's with all these plugins? =
The principle behind this plugin is to enable small pieces of
Twitter functionality, one at a time.

Thus, you have the base plugin, which does nothing except to
enable your site for Twitter OAuth in general. It's required
by all the other plugins.

Then you have individual plugins, one for each piece of
functionality. One for enabling comments, one for adding
Login, etc. These are all smaller and simpler, for the most
part, because they don't have to add all the Twitter
connections stuff that the base plugin adds.

= The comments plugin isn't working! =

You have to modify your theme to use the comments plugin.

In your comments.php file (or wherever your comments form is),
you need to do the following.
```

1. Find the three inputs for the author, email, and url information. They need to have those ID's on the inputs (author, email, url). This is what the default theme and all standardized themes use, but some may be slightly different. You'll have to alter them to have these ID's in that case.

2. Just before the first input, add this code:
[div id="comment-user-details"]
[?php do_action('alt_comment_login'); ?]
(Replace the []'s with normal html greater/less than signs).

3. Just below the last input (not the comment text area, just the name/email/url inputs, add this:
[/div]

That will add the necessary pieces to allow the script to work.

Hopefully, a future version of WordPress will make this simpler.

= Twitter Avatars look wrong. =

Twitter avatars use slightly different code than other avatars. They should style the same, but not all themes will have this working properly, due to various theme designs and such.

However, it is almost always possible to correct this with some simple CSS adjustments. For this reason, they are given a special "twitter-avatar" class, for you to use to style them as you need. Just use .twitter-avatar in your CSS and add styling rules to correct those specific avatars.

= Why can't I email people who comment using Twitter? =

Twitter offers no way to get a valid email address for a user. So the comments plugin uses a fake address of the twitter's username @fake.twitter.com. The "fake" is the giveaway here.

= When users connect using Twitter on the Comments section, there's a delay before their info appears. =

```
Yes. In order to make the plugin more compatible with caching
plugins like WP-Super-Cache, the data for a Twitter connect
account is retrieved from the server using an AJAX request.
This means that there will be a slight delay while the data is
retrieved, but the page has already been loaded and displayed.
Most of the time this will not be noticeable.

= Why does the settings screen warn me that I don't have a URL
shortener plugin? =

Simple Twitter Connect does not implement a URL shortening
service in favor of letting other plugins implement one for
it. WordPress 3.0 includes a new shortlink method for plugins
to implement this properly.

A shortener plugin should implement the "get_shortlink" filter
for it to be detected. WordPress 3.0 will be required for this
to work.

The WordPress.com stats plugin implements this, and it
provides shortlinks to "wp.me" URLs. If you wish to use
another shortener plugin, tell that plugin's author to
implement this same standard, and the plugin will
automatically be detected and used by Simple Twitter Connect.
```

That's a pretty thorough FAQ! It's useful because it anticipates any questions a user might have. It's a **pre-emptive strike**: get in there with the answers before you see anxiety explosions in the support forums.

When writing support documentation such as an FAQ, aim at the lowest common denominator. A developer who is using your plugin likely won't need an FAQ, and if they do, they'll probably get the gist of it quickly and be able to implement it in an instant. There's a rumor going around that WordPress is easy to use, which means that a lot of people who don't know a whole lot about code will be fiddling around with it and break stuff. They are the people to whom you should be addressing your documentation, because they'll be giving you the biggest headaches in the support forums.

TIPS FOR WRITING AN FAQ

- **Anticipate** issues and answer them in advance.
- Deal with any **known conflicts** with old WordPress versions or other plugins.
- If a problem recurs in the support forums, add it to your FAQ.
- Use **Markdown** to highlight each question and follow it with an answer.
- Include any **code snippets or shortcodes** that the user might need.
- The FAQ is a **tool to help you**. Use it properly, and save some money on Aspirin.

Screenshots

Unfortunately, many developers think that screenshots are the realm of theme designers. Perhaps you assume that there's nothing exciting or sexy to see in your plugin, so why bother providing screenshots? Check out the screenshots for Simple Twitter Connect. They show both the UI and the front end of the plugin, giving you an idea what it will be like before installing it.

Screenshots help users decide whether a plugin is right for them. As a developer, you might not think your UI is particularly interesting, but as an end user, I like to look at the settings before installing a plugin. It might not be a deal-breaker, but it does help me decide.

Here are some plugins with nice screenshots:

- Social Profiles Sidebar Widget
- WP Page Navi

- Next Gen Gallery

TIPS FOR PUTTING TOGETHER SCREENSHOTS

- Name screenshots as screenshot-1.png, screenshot-2.png, etc.
- Include at least one screenshot of the **UI**, and one of what it does in the **front end** (if it does anything there).
- Keep the size of the screenshot file **small**.

Other Notes

Including “Other Notes” is not essential. Many plugins don’t have them; Simple Twitter Connect doesn’t, for instance. But the section is useful if you have **additional information** that doesn’t necessarily fit in any other section.

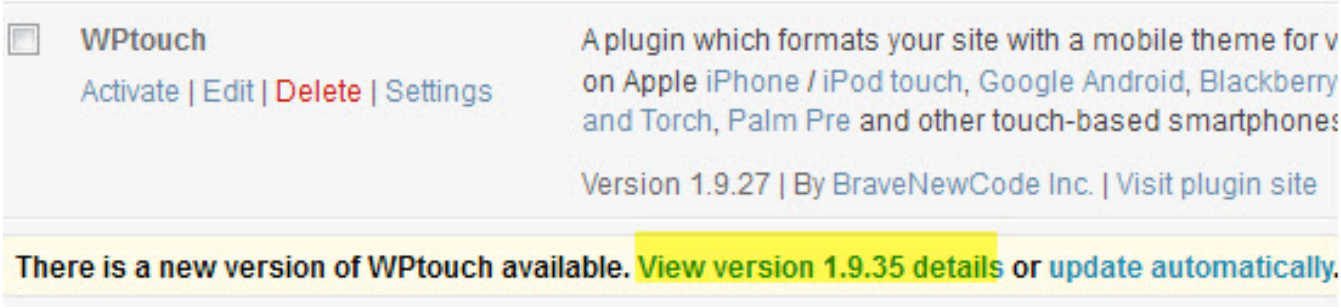
THINGS YOU CAN INCLUDE IN “OTHER NOTES”

- Credits
- To-do list
- Localizations
- Filters
- Set-up guide
- Requirements
- External links
- Upgrading information
- Licence

Changelog

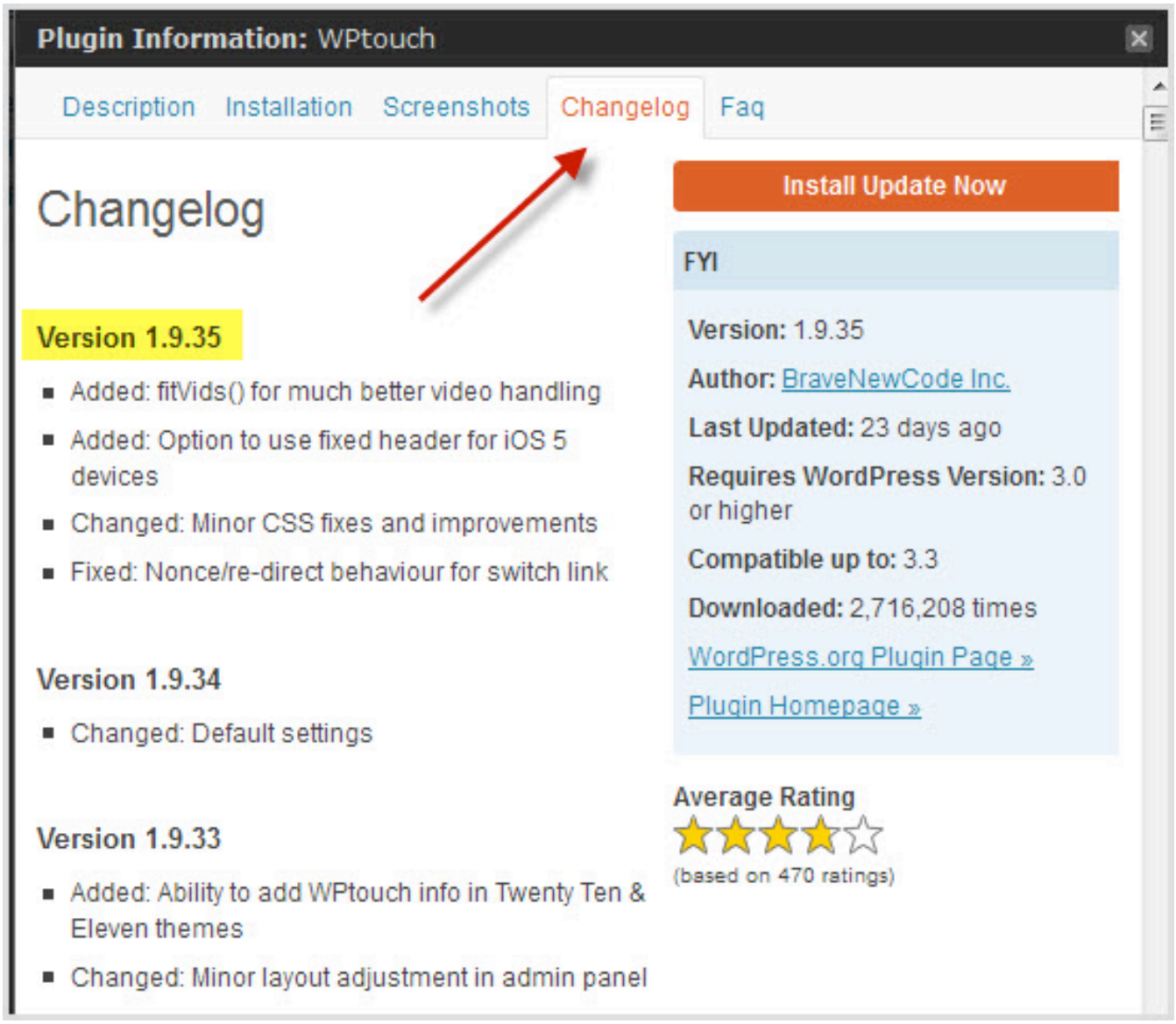
A changelog, as you would guess, is a **log of all changes** you've made to the plugin. It is an important part of the documentation and shouldn't be ignored. A changelog may not make for the most exciting reading, but it is helpful. It tells a user what changes have been made to a plugin, and it makes clear why the user should update. It's also useful for tracking the development of the plugin over time. A changelog gives a plugin c**ontext and history**, and that can be important for people running big projects or important client websites.

Here's a scenario: a user sees an alert in the admin area telling them that they need to update a plugin.



This user needs to update WP Touch!

The changelog shows the changes, so that the user can decide whether to update.



The changelog helps us decide.

Maintaining some websites is a balancing act, and some administrators don't want to upgrade unless absolutely necessary (say, for security updates or for new must-have features). Give them the information they need to be able to decide for themselves. If the update is minor, they might decide to save themselves the hassle and wait until something major arrives.

The changelog for Simple Twitter Connect is lengthy, not surprisingly. You'll see all of the minor and major changes that have been made. Being able to see this plugin's history and development is so useful.

TIPS FOR WRITING THE CHANGELOG

- **No change is insignificant**. Make sure to add every single change.
- The changelog must have the most recent changes first.

Conclusion

Underestimating the importance of the readme file does a disservice to your code and to the WordPress community; so, seeing so many plugin developers do just that is sad. Don't fight the readme: embrace it. It's an important part of your documentation. A good readme is useful, helping users get set up to use the plugin as effectively as possible. And it's useful to you, the developer. The readme demonstrates your plugin and anticipates problems, and it'll save you time and headaches with support requests.

Here are some useful resources to get your plugin ready for the WordPress repository:

- WordPress readme.txt template
  The readme.txt template provided by WordPress.org. Download it and stick your readme text in it to get started.
- WordPress readme.txt validator
  Validating your readme before submitting it is helpful. This tool will flag any issues so that you can get it just right.

- WordPress Plugin Readme File Generator
  Feeling lazy? If you would prefer to fill in a form than write up a readme.txt file, try this generator.
- "How to Publish to the WordPress Plugin Repository"
  There's more to publishing in the repository than writing a readme. This tutorial takes you through the steps to getting published.
- "How to Write a WordPress Plugin: 12 Essential Guides and Resources"
  Haven't even started writing your plugin yet? Check out this collection of plugin development resources.

Integrating Amazon S3 With WordPress

Sameer Borate

Computing is full of buzzwords, “cloud computing” being the latest one. But unlike most trends that fizzle out after the initial surge, cloud computing is here to stay. This article goes over Amazon’s S3 cloud storage service and guides you to implementing a WordPress plugin that backs up your WordPress database to Amazon’s S3 cloud. Note that this is not a tutorial on creating a WordPress plugin from scratch, so some familiarity with plugin development is assumed.

The reason for using Amazon S3 to store important data follows from the “3-2-1” backup rule, coined by Peter Krogh. According to the 3-2-1 rule, you

would keep three copies of any critical data: the original data, a backup copy on removable media, and a second backup at an off-site location (in our case, Amazon's S3 cloud).

Cloud Computing, Concisely

Cloud computing is an umbrella term for any data or software hosted outside of your local system. Cloud computing is categorized into three main service types: infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS).

- **Infrastructure as a service**
  IaaS provides virtual storage, virtual machines and other hardware resources that clients can use on a pay-per-use basis. Amazon S3, Amazon EC2 and RackSpace Cloud are examples of IaaS.
- **Platform as a service**
  PaaS provides virtual machines, application programming interfaces, frameworks and operating systems that clients can deploy for their own applications on the Web. Force.com, Google AppEngine and Windows Azure are examples of PaaS.
- **Software as a service**
  Perhaps the most common type of cloud service is SaaS. Most people use services of this type daily. SaaS provides a complete application operating environment, which the user accesses through a browser rather than a locally installed application. SalesForce.com, Gmail, Google Apps and Basecamp are some examples of SaaS.

For all of the service types listed above, the service provider is responsible for managing the cloud system on behalf of the user. The user is spared the

tedium of having to manage the infrastructure required to operate a particular service.

Amazon S3 In A Nutshell

Amazon Web Services (AWS) is a bouquet of Web services offered by Amazon that together make up a cloud computing platform. The most essential and best known of these services are Amazon EC2 and Amazon S3. AWS also includes CloudFront, Simple Queue Service, SimpleDB, Elastic Block Store. In this article, we will focus exclusively on Amazon S3.

Amazon S3 is cloud-based data-storage infrastructure that is accessible to the user programmatically via a Web service API (either SOAP or REST). Using the API, the user can store various kinds of data in the S3 cloud. They can store and retrieve data from anywhere on the Web and at anytime using the API. But S3 is nothing like the file system you use on your computer. A lot of people think of S3 as a remote file system, containing a hierarchy of files and directories hosted by Amazon. Nothing could be further from the truth.

Amazon S3 is a flat-namespace storage system, devoid of any hierarchy whatsoever. Each storage container in S3 is called a “bucket,” and each bucket serves the same function as that of a directory in a normal file system. However, there is no hierarchy within a bucket (that is, you cannot create a bucket within a bucket). Each bucket allows you to store various kinds of data, ranging in size from 1 B to a whopping 5 GB.

A file stored in a bucket is referred to as an object. An object is the basic unit of stored data on S3. Objects consist of data and meta data. The meta data is a set of name-value pairs that describe the object. Meta data is optional but often adds immense value, whether it’s the default meta data

added by S3 (such as the date last modified) or standard HTTP meta data such as `Content-Type`.

So, what kinds of objects can you store on S3? Any kind you like. It could be a simple text file, a style sheet, programming source code, or a binary file such as an image, video or ZIP file. Each S3 object has its own URL, which you can use to access the object in a browser (if appropriate permissions are set — more on this later).

You can write the URL in two formats, which look something like this:

The bucket's name here is deliberately simple, `codediesel`. It can be more complex, reflecting the structure of your application, like `codediesel.wordpress.backup` or `codediesel.assets.images`.

Every S3 object has a unique URL, formed by concatenating the following components:

1. Protocol (`http://` or `https://`),
2. S3 end point (`s3.amazonaws.com`),

3. Bucket's name,
4. Object key, starting with `/`.

In order to be able to identify buckets, the S3 system requires that you assign a name to each bucket, which must be unique across the S3 bucket namespace. So, if a user has named one of their buckets `company-docs`, you cannot create a bucket with that name anywhere in the S3 namespace. Object names in a bucket, however, must be unique only to that bucket; so, two different buckets can have objects with the same name. Also, you can describe objects stored in buckets with additional information using meta data.

Bucket names must comply with the following requirements:

- May contain lowercase letters, numbers, periods (`.`), underscores (_) and hyphens (–);
- Must begin with a number or letter;
- Must be between 3 and 255 characters long;
- May not be formatted as an IP address (e.g. `265.255.5.4`).

In short, Amazon S3 provides a highly reliable cloud-based storage infrastructure, accessible via a SOAP or REST API. Some common usage scenarios for S3 are:

- Backup and storage
  Provide data backup and storage services.
- Host applications
  Provide services that deploy, install and manage Web applications.

- Host media
  Build a redundant, scalable and highly available infrastructure that hosts video, photo or music uploads and downloads.
- Deliver software
  Host your software applications that customers can download.

Amazon S3's Pricing Model

Amazon S3 is a paid service; you need to attach a credit card to your Amazon account when signing up. But it is surprisingly low priced, and you pay only for what you use; if you use no resources in your S3 account, you pay nothing. Also, as part of the AWS "Free Usage Tier," upon signing up, new AWS customers receive 5 GB of Amazon S3 storage, 20,000 `GET` requests, 2,000 `PUT` requests, and 15 GB of data transfer out each month free for one year.

So, how much do you pay after the free period. As a rough estimate, if you stored 5 GB of data per month, with data transfers of 15 GB and 40,000 `GET` and `PUT` requests a month, the cost would be around $2.60 per month. That's lower than the cost of a burger — inexpensive by any standard. The prices may change, so use the calculator on the S3 website.

Your S3 usage is charged according to three main parameters:

- The total amount of data stored,
- The total amount of data transferred in and out of S3 per month,
- The number of requests made to S3 per month.

Your S3 storage charges are calculated on a unit known as a gigabyte-month. If you store 1 GB for one month, you'll be charged for one gigabyte-month, which is $0.14.

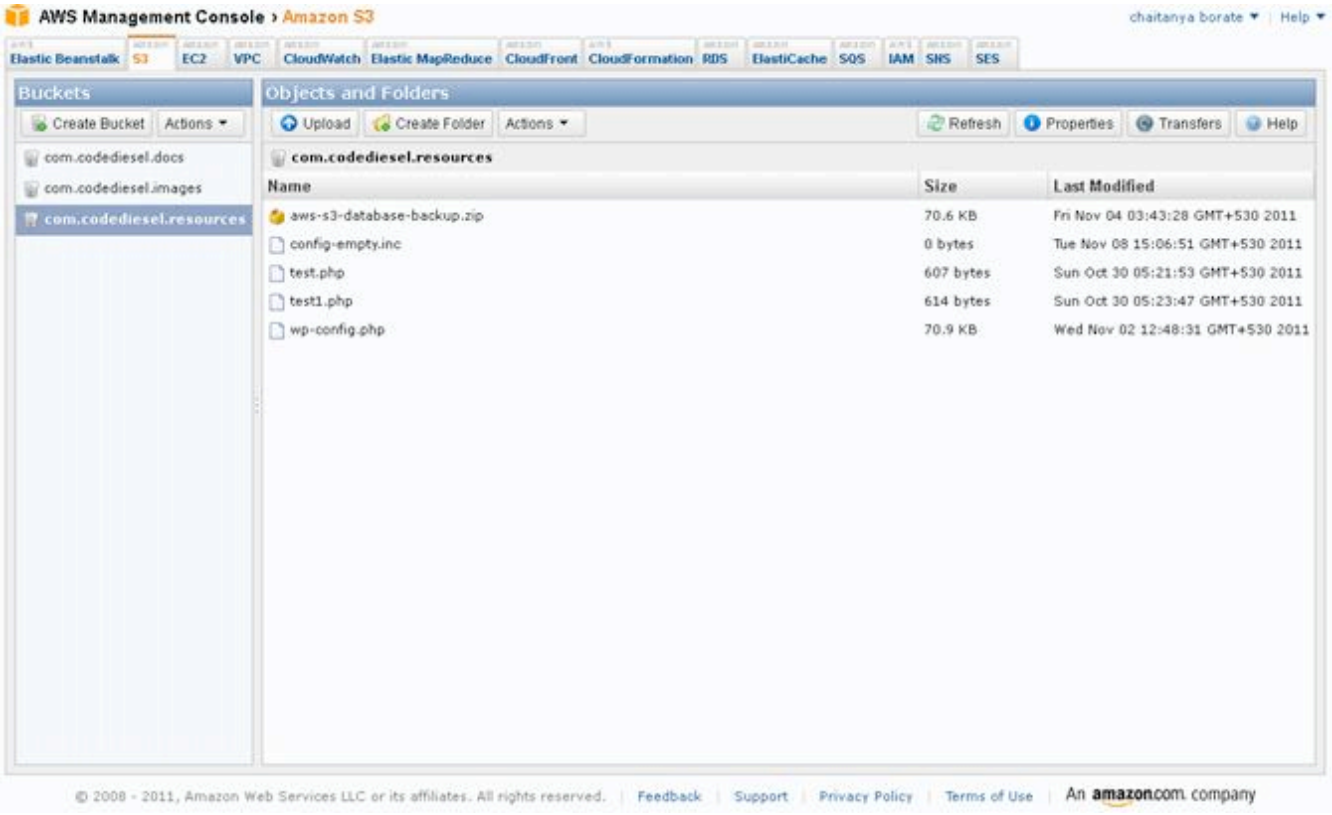
Your data transfer charges are based on the amount of data uploaded and downloaded from S3. Data transferred out of S3 is charged on a sliding scale, starting at $0.12 per gigabyte and decreasing based on volume, reaching $0.050 per gigabyte for all outgoing data transfer in excess of 350 terabytes per month. Note that there is no charge for data transferred within an Amazon S3 "region" via a COPY request, and no charge for data transferred between Amazon EC2 and Amazon S3 within the same region or for data transferred between the Amazon EC2 Northern Virginia region and the Amazon S3 US standard region. To avoid surprises, always check the latest pricing policies on Amazon.

Introduction To The Amazon S3 API And CloudFusion

Now with the theory behind us, let's get to the fun part: writing code. But before that, you will need to register with S3 and create an AWS account. If you don't already have one, you'll be prompted to create one when you sign up for Amazon S3.

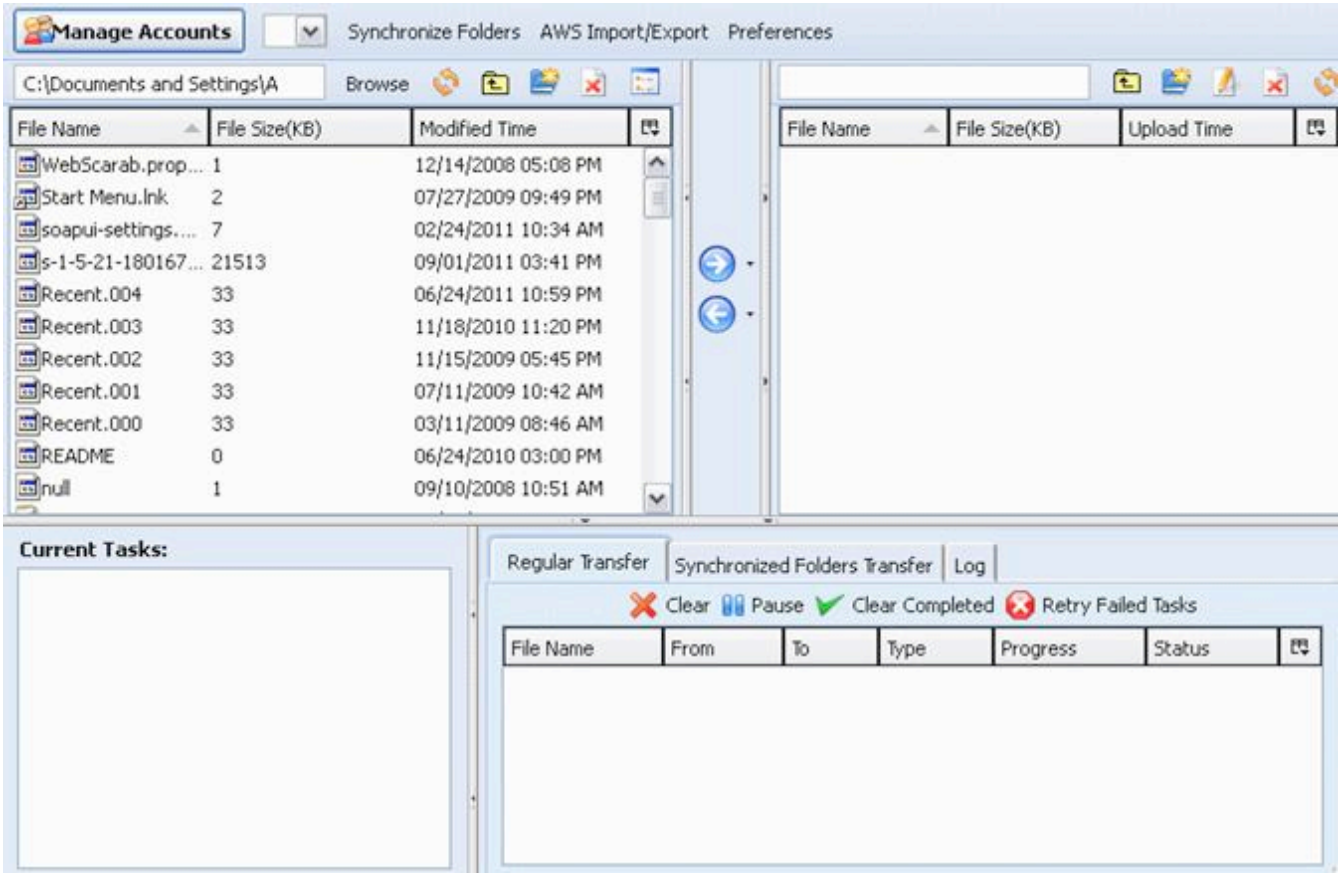
Before moving on to the coding part, let's get acquainted with some visual tools that we can use to work with Amazon S3. Various visual and command-line tools are available to help you manage your S3 account and the data in it. Because the visual tools are easy to work with and user-friendly, we will focus on them in this article. I prefer working with the AWS Management Console for security reasons.

AWS MANAGEMENT CONSOLE



The Management Console is a part of the AWS. Because it is a part of your AWS account, no configuration is necessary. Once you've logged in, you have full access to all of your S3 data and other AWS services. You can create new buckets, create objects, apply security policies, copy objects to different buckets, and perform a multitude of other functions.

S3FOX ORGANIZER



The other popular tool is S3Fox Organizer. S3Fox Organizer is a Firefox extension that enables you to upload and download files to and from your Amazon S3 account. The interface, which opens in a Firefox browser tab, looks very much like a regular FTP client with dual panes. It displays files on your PC on the left, files on S3 on the right, and status messages and information in a panel at the bottom.

Onto The Coding

As stated earlier, AWS is Amazon's Web service infrastructure that encompasses various cloud services, including S3, EC2, SimpleDB and

CloudFront. Integrating these varied services can be a daunting task. Thankfully, we have at our disposal an SDK library in the form of CloudFusion, which enables us to work with AWS effortlessly. CloudFusion is now the official AWS SDK for PHP, and it encompasses most of Amazon's cloud products: S3, EC2, SimpleDB, CloudFront and many more. For this post, I downloaded the ZIP version of the CloudFusion SDK, but the library is also available as a PEAR package. So, go ahead: download the latest version from the official website, and extract the ZIP to your working directory or to your PHP `include` path. In the extracted directory, you will find the `config-sample.inc.php` file, which you should rename to `config.inc.php`. You will need to make some changes to the file to reflect your AWS credentials.

In the `config` file, locate the following lines:

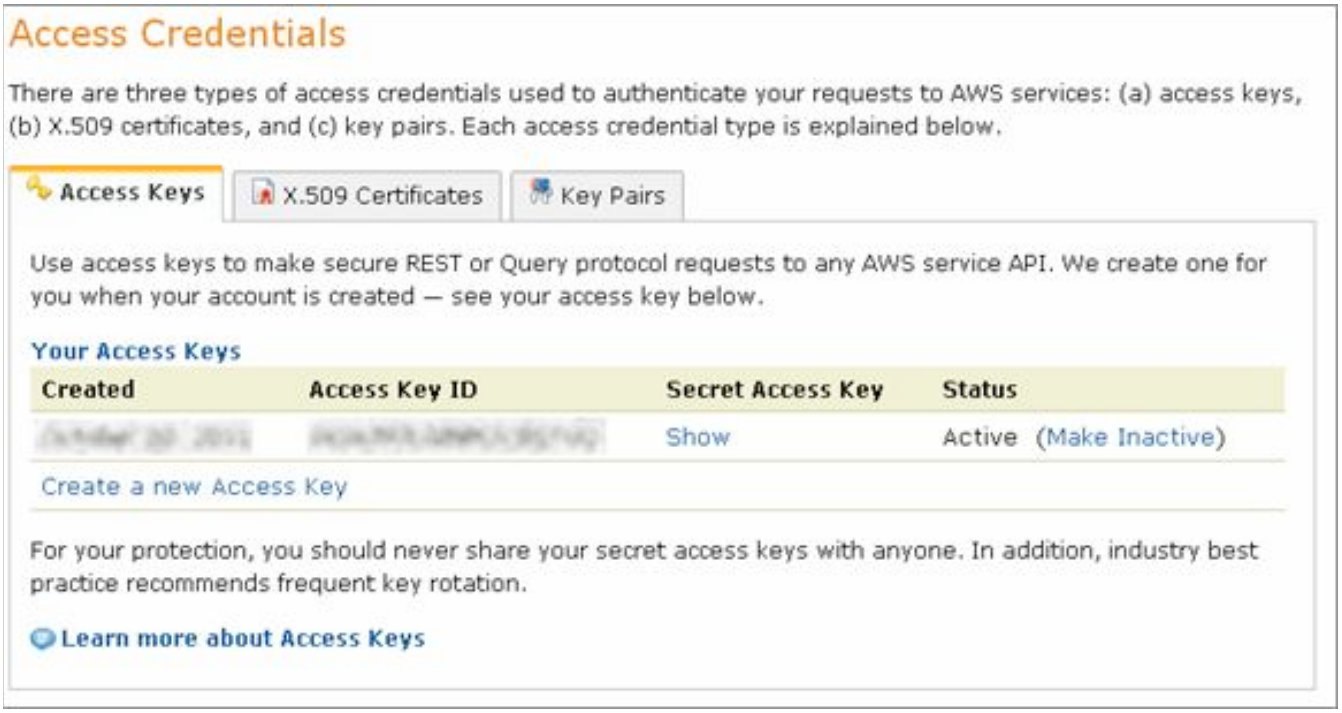
```
define('AWS_KEY', '');
define('AWS_SECRET_KEY', '');
```

Modify the lines to mirror your Amazon AWS' security credentials. You can find the credentials in your Amazon AWS account section, as shown below.

Get the keys, and fill them in on the following lines:

```
define('AWS_KEY', 'your_access_key_id');
define('AWS_SECRET_KEY', 'your_secret_access_key');
```

You can retrieve your access key and secret key from your Amazon account page:



With all of the basic requirements in place, let's create our first bucket on Amazon S3, with a name of your choice. The following example shows a bucket by the name of `com.smashingmagazine.images`. (Of course, by the time you read this, this name may have already be taken.) Choose a structure for your bucket's name that is relevant to your work. For each bucket, you can control access to the bucket, view access logs for the bucket and its objects, and set the geographical region where Amazon S3 will store the bucket and its contents.

```
/* Include the CloudFusion SDK class */
require_once( 'sdk-1.4.4/sdk.class.php');

/* Our bucket name */
$bucket = 'com.smashingmagazine.images';

/* Initialize the class */
$s3 = new AmazonS3();

/* Create a new bucket */
```

```
$resource = $s3->create_bucket($bucket,
AmazonS3::REGION_US_E1);

/* Check if the bucket was successfully created */
if ($resource->isOK()) {
    print("'${bucket}' bucket created\n");
} else {
    print("Error creating bucket '${bucket}'\n");
}
```

Let's go over each line in the example above. First, we included the CloudFusion SDK class in our file. You'll need to adjust the path depending on where you've stored the SDK files.

```
require_once( 'sdk-1.4.4/sdk.class.php');
```

Next, we instantiated the Amazon S3 class:

```
$s3 = new AmazonS3();
```

In the next step, we created the actual bucket; in this case, `com.smashingmagazine.images`. Again, your bucket's name must be unique across all existing bucket names in Amazon S3. One way to ensure this is to prefix a word with your company's name or domain, as we've done here. But this does not guarantee that the name will be available. Nothing prevents anyone from creating a bucket named `com.microsoft.apps` or `com.ibm.images`, so choose wisely.

```
$bucket = 'com.smashingmagazine.images';
$resource = $s3->create_bucket($bucket,
AmazonS3::REGION_US_E1);
```

To reiterate, bucket names must comply with the following requirements:

- May contain lowercase letters, numbers, periods (.), underscores (_) and hyphens (-);
- Must start with a number or letter;

- Must be between 3 and 255 characters long;
- May not be formatted as an IP address (e.g. 265.255.5.4).

Also, you'll need to select a geographical location for your bucket. A bucket can be stored in one of several regions. Reasons for choosing one region over another might be to optimize for latency, to minimize costs, or to satisfy regulatory requirements. Many organizations have privacy policies and regulations on where to store data, so consider this when selecting a location. Objects never leave the region they are stored in unless you explicitly transfer them to another region. That is, if your data is stored on servers located in the US, it will never be copied or transferred by Amazon to servers outside of this region; you'll need to do that manually using the API or AWS tools. In the example above, we have chosen the `REGION_US_E1` region.

Here are the permitted values for regions:

- `AmazonS3::REGION_US_E1`
- `AmazonS3::REGION_US_W1`
- `AmazonS3::REGION_EU_W1`
- `AmazonS3::REGION_APAC_SE1`
- `AmazonS3::REGION_APAC_NE1`

Finally, we checked whether the bucket was successfully created:

```
if ($resource->isOK()) {
    print("'${bucket}' bucket created\n");
} else {
    print("Error creating bucket '${bucket}'\n");
}
```

Now, let's see how to get a list of the buckets we've created on S3. So, before proceeding, create a few more buckets to your liking. Once you have a few buckets in your account, it is time to list them.

```
/* Include the CloudFusion SDK class */
require_once ('sdk-1.4.4/sdk.class.php');

/* Our bucket name */
$bucket = 'com.smashingmagazine.images;

/* Initialize the class */
$s3 = new AmazonS3();

/* Get a list of buckets */
$buckets = $s3->get_bucket_list();

if($buckets)  {
    foreach ($buckets as $b) {
        echo $b . "\n";
    }
}
```

The only new part in the code above is the following line, which gets an array of bucket names:

```
$buckets = $s3->get_bucket_list();
```

Finally, we printed out all of our buckets' names.

```
if($buckets)  {
    foreach ($buckets as $b) {
        echo $b . "\n";
    }
}
```

This concludes our overview of creating and listing buckets in our S3 account. We also learned about S3Fox Organizer and the AWS console tools for working with your S3 account.

Uploading Data To Amazon S3

Now that we've learned how to create and list buckets in S3, let's figure out how to put objects into buckets. This is a little complex, and we have a variety of options to choose from. The main method for doing this is `create_object`. The method takes the following format:

```
create_object ( $bucket, $filename, [ $opt = null ] )
```

The first parameter is the name of the bucket in which the object will be stored. The second parameter is the name by which the file will be stored on S3. Using only these two parameters is enough to create an empty object with the given file name. For example, the following code would create an empty object named config-empty.inc in the com.magazine.resources bucket:

```
$s3 = new AmazonS3();
$bucket = 'com.magazine.resources';
$response = $s3->create_object($bucket, 'config-empty.inc');

// Success?
var_dump($response->isOK());
```

Once the object is created, we can access it using a URL. The URL for the object above would be:

https://s3.amazonaws.com/com.magazine.resources/config-empty.inc

Of course, if you tried to access the URL from a browser, you would be greeted with an "Access denied" message, because objects stored on S3 are set to private by default, viewable only by the owner. You have to explicitly make an object public (more on that later).

To add some content to the object at the time of creation, we can use the following code. This would add the text “Hello World” to the `config-empty.inc` file.

```
$response = $s3->create_object($bucket,  config-empty.inc ‘,
    array(
        'body' => Hello World!'
));
```

As a complete example, the following code would create an object with the name simple.txt, along with some content, and save it in the given bucket. An object may also optionally contain meta data that describes that object.

```
/* Initialize the class */
$s3 = new AmazonS3();

/* Our bucket name */
$bucket = 'com.magazine.resources’;

$response = $s3->create_object($bucket, 'simple.txt',
    array(
    'body' => Hello World!'
));

if ($response->isOK())
{
    return true;
}
```

You can also upload a file, rather than just a string, as shown below. Although many options are displayed here, most have a default value and may be omitted. More information on the various options can be found in the “AWS SDK for PHP 1.4.7.”

```
require_once( ‘sdk-1.4.4/sdk.class.php');

$s3 = new AmazonS3();
$bucket = 'com.smashingmagazine.images’;
```

```
$response = $s3->create_object($bucket, 'source.php',
    array(
    'fileUpload' => 'test.php',
    'acl' => AmazonS3::ACL_PRIVATE,
    'contentType' => 'text/plain',
    'storage' => AmazonS3::STORAGE_REDUCED,
    'headers' => array( // raw headers
        'Cache-Control' => 'max-age',
        'Content-Encoding' => 'text/plain',
        'Content-Language' => 'en-US',
        'Expires' => 'Thu, 01 Dec 1994 16:00:00 GMT',
    )
));

// Success?
var_dump($response->isOK());
```

Details on the various options will be explained in the coming sections. For now, take on faith that the above code will correctly upload a file to the S3 server.

Writing Our Amazon S3 WordPress Plugin

With some background on Amazon S3 behind us, it is time to put our learning into practice. We are ready to build a WordPress plugin that will automatically back up our WordPress database to the S3 server and restore it when needed.

To keep the article focused and at a reasonable length, we'll assume that you're familiar with WordPress plugin development. If you are a little sketchy on the fundamentals, read "How to Create a WordPress Plugin" to get on track quickly.

THE PLUGIN'S FRAMEWORK

We'll first create a skeleton and then gradually fill in the details. To create a plugin, navigate to the `wp-content/plugins` folder, and create a new folder named `s3-backup`. In the new folder, create a file named `s3-backup.php`. Open the file in the editor of your choice, and paste the following header information, which will describe the plugin for WordPress:

```
/*
Plugin Name: Amazon S3 Backup
Plugin URI: http://cloud-computing-rocks.com
Description: Plugin to back up WordPress database to Amazon S3
Version: 1.0
Author: Mr. Sameer
Author URI: http://www.codediesel.com
License: GPL2
*/
```

Once that's done, go to the plugin's page in the admin area, and activate the plugin.

Now that we've successfully installed a bare-bones WordPress plugin, let's add the meat and create a complete working system. Before we start writing the code, we should know what the admin page for the plugin will ultimately look like and what tasks the plugin will perform. This will guide us in writing the code. Here is the main settings page for our plugin:

The interface is fairly simple. The primary task of the plugin will be to back up the current WordPress database to an Amazon S3 bucket and to restore the database from the bucket. The settings page will also have a function for naming the bucket in which the backup will be stored. Also, we can specify whether the backup will be available to the public or accessible only to you.

Below is a complete outline of the plugin's code. We will elaborate on each section in turn.

```
/*
Plugin Name: Amazon S3 Backup
Plugin URI: http://cloud-computing-rocks.com
Description: Plugin to back up WordPress database to Amazon S3
Version: 1.0
Author: Mr. Sameer
Author URI: http://www.codediesel.com
License: GPL2
*/

$plugin_path = WP_PLUGIN_DIR . "/" .
dirname(plugin_basename(__FILE__));
```

```
/* CloudFusion SDK */
require_once($plugin_path . '/sdk-1.4.4/sdk.class.php');

/* WordPress ZIP support library */
require_once(ABSPATH . '/wp-admin/includes/class-pclzip.php');

add_action('admin_menu', 'add_settings_page');

/* Save or Restore Database backup */
if(isset($_POST['aws-s3-backup'])) {
…
}

/* Generic Message display */
function showMessage($message, $errormsg = false) {
…
}

/* Back up WordPress database to an Amazon S3 bucket */
function backup_to_AmazonS3() {
…
}

/* Restore WordPress backup from an Amazon S3 bucket */
function restore_from_AmazonS3() {
…
}

function add_settings_page() {
…
}

function draw_settings_page() {
…
}
```

Here is the directory structure that our plugin will use:

```
plugins (WordPress plugin directory)
---s3-backup (our plugin directory)
-------s3backup (restored backup will be stored in this
directory)
-------sdk-1.4.4 (CloudFusion SDK directory)
-------s3-backup.php (our plugin source code)
```

Let's start coding the plugin. First, we'll initialize some variables for paths and include the CloudFusion SDK. A WordPress database can get large, so to conserve space and bandwidth, the plugin will need to compress the database before uploading it to the S3 server. To do this, we will use the `class-pclzip.php` ZIP compression support library, which is built into WordPress. Finally, we'll hook the settings page to the admin menu.

```
$plugin_path = WP_PLUGIN_DIR . "/" .
dirname(plugin_basename(__FILE__));

/* CloudFusion SDK */
require_once($plugin_path . '/sdk-1.4.4/sdk.class.php');

/* WordPress ZIP support library */
require_once(ABSPATH . '/wp-admin/includes/class-pclzip.php');

/* Create the admin settings page for our plugin */
add_action('admin_menu', 'add_settings_page');
```

Every WordPress plugin must have its own settings page. Ours is a simple one, with a few buttons and fields. The following is the code for it, which will handle the mundane work of saving the bucket's name, displaying the buttons, etc.

```
function draw_settings_page() {
    ?>
    <div class="wrap">
        <h2><?php echo('WordPress Database Amazon S3
Backup') ?></h2>
            <form method="post" action="options.php">
```

```
            <input type="hidden" name="action"
value="update" />
            <input type="hidden" name="page_options"
value="aws-s3-access-public, aws-s3-access-bucket" />
            <?php
                wp_nonce_field('update-options');
                $access_options = get_option('aws-s3-access-
public');
            ?>
            <p>
                <?php echo('Bucket Name:') ?>
                <input type="text" name="aws-s3-access-bucket"
size="64" value="<?php echo get_option('aws-s3-access-
bucket'); ?>" />
            </p>
            <p>
                <?php echo('Public:') ?>
                <input type="checkbox" name="aws-s3-access-
public" <?php checked(1 == $access_options); ?> value="1" />
            </p>
            <p class="submit">
                <input type="submit" class="button-primary"
name="Submit" value="<?php echo('Save Changes') ?>" />
            </p>
        </form>
        <hr />
        <form method="post" action="">
            <p class="submit">
                <input type="submit" name="aws-s3-backup"
value="<?php echo('Backup Database') ?>" />
                <input type="submit" name="aws-s3-restore"
value="<?php echo('Restore Database') ?>" />
            </p>
        </form>
    </div>
    <?php
}
```

Setting up the base framework is essential if the plugin is to work correctly. So, double-check your work before proceeding.

DATABASE UPLOAD

Next is the main part of the plugin, its raison d'être: the function for backing up the database to the S3 bucket.

```
/* Back up WordPress database to an Amazon S3 bucket */
function backup_to_AmazonS3()
{
    global $wpdb, $plugin_path;

    /* Backup file name */
    $backup_zip_file = 'aws-s3-database-backup.zip';

    /* Temporary directory and file name where the backup file
will be stored */
    $backup_file = $plugin_path . "/s3backup/aws-s3-database-
backup.sql";

    /* Complete path to the compressed backup file */
    $backup_compressed = $plugin_path . "/s3backup/" .
$backup_zip_file;

    $tables = $wpdb->get_col("SHOW TABLES LIKE '" . $wpdb-
>prefix . "%'");
  $result = shell_exec('mysqldump --single-transaction -h ' .
                       DB_HOST . ' -u ' . DB_USER . ' --
password="' .
                       DB_PASSWORD . '" ' .
                       DB_NAME . ' ' . implode(' ',
$tables) . ' > ' .
                       $backup_file);

    $backups[] = $backup_file;

    /* Create a ZIP file of the SQL backup */
    $zip = new PclZip($backup_compressed);
    $zip->create($backups);

    /* Connect to Amazon S3 to upload the ZIP */
    $s3 = new AmazonS3();
    $bucket = get_option('aws-s3-access-bucket');
```

```
    /* Check if a bucket name is specified */
    if(empty($bucket)) {
        showMessage("No Bucket specified!", true);
        return;
    }

    /* Set backup public options */
    $access_options = get_option('aws-s3-access-public');

    if($access_options) {
        $access = AmazonS3::ACL_PUBLIC;
    } else {
        $access = AmazonS3::ACL_PRIVATE;
    }

    /* Upload the database itself */
    $response = $s3->create_object($bucket, $backup_zip_file,
                array(
                'fileUpload' => $backup_compressed,
                'acl' => $access,
                'contentType' => 'application/zip',
                'encryption' => 'AES256',
                'storage' => AmazonS3::STORAGE_REDUCED,
                'headers' => array( // raw headers
                    'Cache-Control' => 'max-age',
                    'Content-Encoding' => 'application/zip',
                    'Content-Language' => 'en-US',
                    'Expires' => 'Thu, 01 Dec 1994 16:00:00
GMT',
                )
            ));

    if($response->isOK()) {
        unlink($backup_compressed);
        unlink($backup_file);
        showMessage("Database successfully backed up to Amazon
S3.");
    } else {
        showMessage("Error connecting to Amazon S3", true);
    }
}
```

The code is self-explanatory, but some sections need a bit of explanation. Because we want to back up the complete WordPress database, we need to somehow get ahold of the MySQL dump file for the database. There are multiple ways to do this: by using MySQL queries within WordPress to save all tables and rows of the database, or by dropping down to the shell and using `mysqldump`. We will use the second method. The code for the database dump shown below uses the shell_exec function to run the `mysqldump` command to grab the WordPress database dump. The dump is further saved to the `aws-s3-database-backup.sql` file.

```
$tables = $wpdb->get_col("SHOW TABLES LIKE '" . $wpdb-
>prefix . "%'");
$result = shell_exec('mysqldump --single-transaction -h ' .
                        DB_HOST . ' -u ' . DB_USER .
                        ' --password="' . DB_PASSWORD . '"
' .
                        DB_NAME . ' ' . implode(' ',
$tables) .
                        ' > ' . $backup_file);

$backups[] = $backup_file;
```

The SQL dump will obviously be big on most installations, so we'll need to compress it before uploading it to S3 to conserve space and bandwidth. We'll use WordPress' built-in ZIP functions for the task. The `PclZip` class is stored in the `/wp-admin/includes/class-pclzip.php` file, which we have included at the start of the plugin. The `aws-s3-database-backup.zip` file is the final ZIP file that will be uploaded to the S3 bucket. The following lines will create the required ZIP file.

```
/* Create a ZIP file of the SQL backup */
    $zip = new PclZip($backup_compressed);
    $zip->create($backups);
```

The `PclZip` constructor takes a file name as an input parameter; aws-s3-database-backup.zip, in this case. And to the create method we pass an

array of files that we want to compress; we have only one file to compress, `aws-s3-database-backup.sql`.

Now that we've taken care of the database, let's move on to the security. As mentioned in the introduction, objects stored on S3 can be set as private (viewable only by the owner) or public (viewable by everyone). We set this option using the following code.

```
/* Set backup public options */
    $access_options = get_option('aws-s3-access-public');

    if($access_options) {
        $access = AmazonS3::ACL_PUBLIC;
    } else {
        $access = AmazonS3::ACL_PRIVATE;
    }
```

We have listed only two options for access (`AmazonS3::ACL_PUBLIC` and `AmazonS3::ACL_PRIVATE`), but there are a couple more, as listed below and the details of which you can find in the Amazon SDK documentation.

- `AmazonS3::ACL_PRIVATE`
- `AmazonS3::ACL_PUBLIC`
- `AmazonS3::ACL_OPEN`
- `AmazonS3::ACL_AUTH_READ`
- `AmazonS3::ACL_OWNER_READ`
- `AmazonS3::ACL_OWNER_FULL_CONTROL`

Now on to the main code that does the actual work of uploading. This could have been complex, but the CloudFusion SDK makes it easy. We use the `create_object` method of the S3 class to perform the upload. We got a short glimpse of the method in the last section.

```
/* Upload the database itself */
$response = $s3->create_object($bucket, $backup_zip_file,
array(
            'fileUpload' => $backup_compressed,
            'acl' => $access,
            'contentType' => 'application/zip',
            'encryption' => 'AES256',
            'storage' => AmazonS3::STORAGE_REDUCED,
            'headers' => array( // raw headers
                'Cache-Control' => 'max-age',
                'Content-Encoding' => 'application/zip',
                'Content-Language' => 'en-US',
                'Expires' => 'Thu, 01 Dec 1994 16:00:00 GMT',
            )
        ));
```

Let's go over each line in turn. But bear in mind that the method has quite a many options, so refer to the original documentation itself.

- `$backup_zip_file`
  The name of the object that will be created on S3.
- `'fileUpload' => $backup_compressed`
  The name of the file, whose data will be uploaded to the server. In our case, `aws-s3-database-backup.zip`.
- `'acl' => $access`
  The access type for the object. In our case, either public or private.
- `'contentType' => 'application/zip'`
  The type of content that is being sent in the body. If a file is being uploaded via fileUpload, as in our case, it will attempt to determine the correct MIME type based on the file's extension. The default value is `application/octet-stream`.
- `'encryption' => 'AES256'`
  The algorithm to use for encrypting the object. (Allowed values: AES256)

- `'storage' => AmazonS3::STORAGE_REDUCED`
  Specifies whether to use “standard” or “reduced redundancy” storage. Allowed values are `AmazonS3::STORAGE_STANDARD` and `AmazonS3::STORAGE_REDUCED`. The default value is `STORAGE_STANDARD`.
- ```
  'headers' => array( // raw headers
  'Cache-Control' => 'max-age',
  'Content-Encoding' => 'application/zip',
  'Content-Language' => 'en-US',
  'Expires' => 'Thu, 01 Dec 1994 16:00:00 GMT',
  )
  ```
  The standard HTTP headers to send along with the request. These are optional.

Note that the plugin does not provide a function to create a bucket on Amazon S3. You need use Amazon’s AWS Management Console or S3Fox Organizer to create a bucket before uploading objects to it.

DATABASE RESTORE

Merely being able to back up data is insufficient. We also need to be able to restore it when the need arises. In this section, we’ll lay out the code for restoring the database from S3. When we say “restore,” keep in mind that the database’s ZIP file from S3 will simply be downloaded to the specified folder in our plugin directory. The actual database on our WordPress server is not changed in any way; you will have to restore the database yourself manually. We could have equipped our plugin to also auto-restore, but that would have made the code a lot more complex.
  ```

Here is the complete code for the restore function:

```
/* Restore WordPress backup from an Amazon S3 bucket */
function restore_from_AmazonS3()
{
    global $plugin_path;

    /* Backup file name */
    $backup_zip_file = 'aws-s3-database-backup.zip';

    /* Complete path to the compressed backup file */
    $backup_compressed = $plugin_path . "/s3backup/" .
$backup_zip_file;

    $s3 = new AmazonS3();
    $bucket = get_option('aws-s3-access-bucket');

    if(empty($bucket)) {
        showMessage("No Bucket specified!", true);
        return;
    }

    $response = $s3->get_object($bucket, $backup_zip_file,
array(
                'fileDownload' => $backup_compressed
                 ));

    if($response->isOK()) {
        showMessage("Database successfully restored from
Amazon S3.");
    } else {
        showMessage("Error connecting to Amazon S3", true);
    }

}
```

As you can see, the code for restoring is much simpler than the code for uploading. The function uses the `get_object` method of the SDK, the definition of which is as follows:

```
get_object ( $bucket, $filename, [ $opt = null ] )
```

The details of the method's parameters are enumerated below:

- `$bucket`
  The name of the bucket where the backup file is stored. The bucket's name is stored in our WordPress settings variable `aws-s3-access-bucket`, which we retrieve using the `get_option('aws-s3-access-bucket')` function.
- `$backup_zip_file`
  The file name of the backup object. In our case, `aws-s3-database-backup.zip`.
- `'fileDownload' => $backup_compressed`
  The file system location to download the file to, or an open file resource. In our case, the `s3backup` directory in our plugin folder. It must be a server-writable location.

At the end of the function, we check whether the download was successful and inform the user.

In addition to the above functions, there are some miscellaneous support functions. One is for displaying a message to the user:

```
/* Generic message display */
function showMessage($message, $errormsg = false) {
  if ($errormsg) {
     echo '<div id="message" class="error">';
  } else {
     echo '<div id="message" class="updated">';
  }

  echo "<p><strong>$message</strong></p></div>";
}
```

Another is to add a hook for the settings page to the admin section.

```
function add_settings_page() {
    add_options_page('Amazon S3 Backup', 'Amazon S3 Backup',8,
                     's3-backup', 'draw_settings_page');
}
```

The button commands for backing up and restoring are handled by this simple code:

```
/* Save or Restore Database backup */
if(isset($_POST['aws-s3-backup'])) {
    backup_to_AmazonS3();
} elseif(isset($_POST['aws-s3-restore'])) {
    restore_from_AmazonS3();
}
```

This rounds out our tutorial on creating a plugin to integrate Amazon S3 with WordPress. Although we could have added more features to the plugin, the functionality was kept bare to maintain focus on Amazon S3's basics, rather then on the calisthenics of plugin development.

In Conclusion

This has been a relatively long article, the aim of which was to acquaint you with Amazon S3 and how to use it in your PHP applications. Together, we've developed a WordPress plugin to back up our WordPress database to the S3 cloud and to retrieve it when needed. To reiterate, S3 is a complex service with many features and functions, which we have not covered here due to space considerations. Perhaps in future articles we'll elaborate on the other features of S3 and add more features to the plugin we developed here.

The Authors

Elio Rivero

Elio Rivero is a web designer and developer who works daily with WordPress. He runs I Love Colors where he shares tutorials about jQuery, WordPress and blogs about typography, design and illustration. He also sells WordPress plugins on CodeCanyon.

Kevin Leary

Kevin Leary is a WordPress Web Designer & Developer located in Boston, Massachusetts. He specializes in customizing themes and plugins for businesses.

Jeremy Desvaux de Marigny

Passionate web developer from Lyon. Jeremy likes to build sweet & strong web apps, usually based on PHP. WordPress nerd, he loves jQuery, tweets about web development, blogs at jdmweb, and works at NOE interactive.

Jonathan Wold

Jonathan Wold is the husband of a beautiful redhead named Joslyn and the father of a baby boy named Jaiden. He works at Sabramedia, a web design and development company that specializes in WordPress powered media sites. He is also a core developer and evangelist for Newsroom, a paywall, newspaper CMS, and accounting system built for the newspaper industry.

Justin Tadlock

I like to break WordPress in as many ways as possible. I even co-wrote a book about it: Pro WP Plugin Development. Outside of that, I own and run Theme Hybrid (a theme club) and co-own DevPress (a themes, plugins, and tutorials community). I also play Halo sometimes.

Sameer Borate

Sameer Borate is a freelance web developer, working primarily in PHP and MySQL. He blogs about his various interests and web topics at www.codediesel.com. When away from blogging or writing code, he spends his free time reading non-fiction books. He is of the staunch opinion that the Semantic Web is the next frontier of the Internet and would like to make some time to delve into its mysteries.

Siobhan McKeown

Siobhan McKeown is a writer at WPMU.org, one of the leading WordPress news and tutorial sites around. She is also the founder of Words for WP, a copywriting service just for the WordPress community. She is a big fan of words, and of WordPress, which works out pretty well. You can find her on twitter and occasionally hanging out on G+.

Thord Daniel Hedengren

Thord Daniel Hedengren is addicted to words, which is probably why he keep writing books and never shuts up about writing. He is a contributor to numerous magazines and sites, as well as the owner of the Odd Alice web

agency in Stockholm. You should follow this crazy Sweden on Twitter where he mixes wisdom with nonsense and the occasional jab as @tdh.