

Mastering WordPress

Advanced Techniques

A large, stylized, blue 'W' character with a smaller '2' at the bottom right corner, set against a white circular background.

Imprint

Published in November 2011

Smashing Media GmbH, Freiburg, Germany

Cover Design: Ricardo Gimenes

Editing: Andrew Rogerson, Talita Telma

Proofreading: Andrew Lobo, Iris Ljesnjanin

Reviewing: Jeff Starr

Idea and Concept: Sven Lennartz, Vitaly Friedman

Founded in September 2006, [Smashing Magazine](#) delivers useful and innovative information to Web designers and developers. Smashing Magazine is a well-respected international online publication for professional Web designers and developers. Our main goal is to support the Web design community with useful and valuable articles and resources, written and created by experienced designers and developers.

ISBN: 9783943075182

Version: December 16, 2011

Table of Contents

[Preface](#)

[The Definitive Guide To WordPress Hooks](#)

[Custom Fields Hacks For WordPress](#)

[Power Tips For WordPress Template Developers](#)

[Advanced Power Tips For WordPress Template Developers](#)

[Advanced Power Tips for WordPress Template Developers: Reloaded](#)

[Ten Things Every WordPress Plugin Developer Should Know](#)

[Create Perfect Emails For Your WordPress Website](#)

[Writing WordPress Guides for the Advanced Beginner](#)

[Advanced Layout Templates In WordPress' Content Editor](#)

[The Authors](#)

Preface

WordPress has many facets to show for those who like to explore its possibilities. Well known as a free source blog-publishing platform, WordPress also gained popularity because of its flexibility and development options through plugins, hooks and custom fields – just to name a few.

If you are already familiar with the fundamentals of WordPress, you might be searching for specific knowledge in order to become a better expert. This Smashing eBook #11: Mastering WordPress, which is packed with exclusive advanced techniques, is probably what you are looking for. After reading this eBook, you will be able to implement various hints which WordPress has to offer for your upcoming projects.

This eBook contains 9 articles which brings topics that will help you work with custom fields, understand coding with hooks, maintain plugins, create emails for your website, write WordPress guides and administrate layout content editor templates. Three whole chapters are destined to advise you with power tips for WordPress template developers. For instance, you will get tips which address common CMS implementation challenges without plugin dependence and cover a handful of API calls and integration with PHP codes. Also, these tips explain how to customize basic content administration and add features to the post and page editor in WordPress.

The articles have been published on Smashing Magazine in 2010 and 2011, and they have been carefully edited and prepared for this eBook.

We hope that you will find this eBook useful and valuable. We are looking forward to your feedback on [Twitter](#) or via our [contact form](#).

— Andrew Rogerson, Smashing eBook Editor

The Definitive Guide To WordPress Hooks

Daniel Pataki

If you're into WordPress development, you can't ignore hooks for long before you have to delve into them head on. Modifying WordPress core files is a big no-no, so whenever you want to change existing functionality or create new functionality, you will have to turn to hooks.



In this chapter, I would like to dispel some of the confusion around hooks, because not only are they the way to code in WordPress, but they also teach us a great design pattern for development in general. Explaining this

in depth will take a bit of time, but bear with me: by the end, you'll be able to jumble hooks around like a pro.

Why Hooks Exist

I think the most important step in grasping hooks is to understand the need for them. Let's create a version of a WordPress function that already exists, and then evolve it a bit using the "hooks mindset."

```
function get_excerpt($text, $length = 150) {  
    $excerpt = substr($text, $length)  
    return $excerpt;  
}
```

This function takes two parameters: a string and the length at which we want to cut it. What happens if the user wants a 200-character excerpt instead of a 150-character one? They just modify the parameter when they use the function. No problem there.

If you use this function a lot, you will notice that the parameter for the text is usually the post's content, and that you usually use 200 characters instead of the default 150. Wouldn't it be nice if you could set up new defaults, so that you didn't have to add the same parameters over and over again? Also, what happens if you want to add some more custom text to the end of the excerpt?

These are the kinds of problems that hooks solve. Let's take a quick look at how.

```
function get_excerpt($text, $length = 150) {  
  
    $length = apply_filters("excerpt_length", $length);  
  
    $excerpt = substr($text, $length)  
    return $excerpt;  
}
```

As you can see, the default excerpt length is still 150, but we've also applied some filters to it. A filter allows you to write a function that modifies the value of something—in this case, the excerpt's length. The name (or tag) of this filter is `excerpt_length`, and if no functions are attached to it, then its value will remain 150. Let's see how we can now use this to modify the default value.

```
function get_excerpt($text, $length = 150) {  
  
    $length = apply_filters("excerpt_length");  
  
    $excerpt = substr($text, $length)  
    return $excerpt;  
}  
  
function modify_excerpt_length() {  
    return 200;  
}  
  
add_filter("excerpt_length", "modify_excerpt_length");
```

First, we have defined a function that does nothing but return a number. At this point, nothing is using the function, so let's tell WordPress that we want to hook this into the `excerpt_length` filter.

We've successfully changed the default excerpt length in WordPress, without touching the original function and without even having to write a custom excerpt function. This will be extremely useful, because if you always want excerpts that are 200 characters long, just add this as a filter and then you won't have to specify it every time.

Suppose you want to tack on some more text, like "Read on," to the end of the excerpt. We could modify our original function to work with a hook and then tie a function to that hook, like so:

```
function get_excerpt($text, $length = 150) {  
  
    $length = apply_filters("excerpt_length");  
  
    $excerpt = substr($text, $length)  
    return apply_filters("excerpt_content", $excerpt);  
}  
  
function modify_excerpt_content($excerpt) {  
    return $excerpt . "Read on...";  
}  
add_filter("excerpt_content", "modify_excerpt_content");
```

This hook is placed at the end of the function and allows us to modify its end result. This time, we've also passed the output that the function would normally produce as a parameter to our hook. The function that we tie to this hook will receive this parameter.

All we are doing in our function is taking the original contents of \$excerpt and appending our "Read on" text to the end. But if we choose, we could also return the text "Click the title to read this article," which would replace the whole excerpt.

While our example is a bit redundant, since WordPress already has a better function, hopefully you've gotten to grips with the thinking behind hooks. Let's look more in depth at what goes on with filters, actions, priorities, arguments and the other yummy options available.

Filters And Actions

Filters and actions are two types of hooks. As you saw in the previous section, a filter modifies the value of something. An action, rather than modifying something, calls another function to run beside it.

A commonly used action hook is `wp_head`. Let's see how this works. You may have noticed a function at the bottom of your website's head section named `wp_head()`. Diving into the code of this function, you can see that it contains a call to `do_action()`. This is similar to `apply_filters()`; it means to run all of the functions that are tied to the `wp_head` tag.

Let's put a copyright meta tag on top of each post's page to test how this works.

```
add_action("wp_head", "my_copyright_meta");

function my_copyright_meta() {
    if(is_singular()) {
        echo "<meta name='copyright' content='© Me, 2011'">";
    }
}
```

The Workflow Of Using Hooks

While hooks are better documented nowadays, they have been neglected a bit until recently, understandably so. You can find some good pointers in the

Codex, but the best thing to use is [Adam Brown's hook reference](#), and/or look at the [source code](#).

Say you want to add functionality to your blog that notifies authors when their work is published. To do this, you would need to do something when a post is published. So, let's try to find a hook related to publishing.

Can we tell whether we need an action or a filter? Sure we can! When a post is published, do we want to modify its data or do a completely separate action? The answer is the latter, so we'll need an action. Let's go to the [action reference](#) on Adam Brown's website, and search for "Publish."

The first thing you'll find is `app_publish_post`. Sounds good; let's click on it. The details page doesn't give us a lot of info (sometimes it does), so click on the "View hook in source" link next to your version of WordPress (preferably the most recent version) in the table. This website shows only a snippet of the file, and unfortunately the beginning of the documentation is cut off, so it's difficult to tell if this is what we need. Click on "View complete file in SVN" to go to the complete file so that we can search for our hook.

In the file I am viewing, the hook can be found in the `_publish_post_hook()` function, which—according to the documentation above it—is a "hook to schedule pings and enclosures when a post is published," so this is not really what we need.

With some more research in the action list, you'll find the `publish_post` hook, and this is what we need. The first thing to do is write the function that sends your email. This function will receive the post's ID as an argument, so you can use that to pull some information into the email. The second task is to hook this function into the action. Look at the finished code below for the details.

```
function authorNotification($post_id) {  
    global $wpdb;  
    $post = get_post($post_id);  
    $author = get_userdata($post->post_author);  
  
    $message = "  
        Hi ".$author->display_name.",  
        Your post, ".$post->post_title." has just been published. Well  
done!  
    ";  
    wp_mail($author->user_email, "Your article is online", $message);  
}  
add_action('publish_post', 'authorNotification');
```

Notice that the function we wrote is usable in its own right. It has a very specific function, but it isn't only usable together with hooks; you could use it in your code any time. In case you're wondering, `wp_mail()` is an awesome mailer function—have a look at the WordPress Codex for more information.

This process might seem a bit complicated at first, and, to be totally honest, it does require browsing a bit of documentation and source code at first, but as you become more comfortable with this system, your time spent researching what to use and when to use it will be reduced to nearly nothing.

Priorities

The third parameter when adding your actions and filters is the priority. This basically designates the order in which attached hooks should run. We haven't covered this so far, but attaching multiple functions to a hook is, of course, possible. If you want an email to be sent to an author when their

post is published and to also automatically tweet the post, these would be written in two separate functions, each tied to the same tag (`publish_post`).

Priorities designate which hooked function should run first. The default value is 10, but this can be changed as needed. Priorities usually don't make a huge difference, though. Whether the email is sent to the author before the article is tweeted or vice versa won't make a huge difference.

In rarer cases, assigning a priority could be important. You might want to overwrite the actions of other plugins (be careful, in this case), or you might want to enforce a specific order. I recently had to overwrite functionality when I was asked to optimize a website. The website had three to four plugins, with about nine JavaScript files in total. Instead of disabling these plugins, I made my own plugin that overwrote some of the JavaScript-outputting functionality of those plugins. My plugin then added the minified JavaScript code in one file. This way, if my plugin was deactivated, all of the other plugins would work as expected.

Specifying Arguments

The fourth argument when adding filters and actions specifies how many arguments the hooked function takes. This is usually dictated by the hook itself, and you will need to look at the source to find this information.

As you know from before, your functions are run when they are called by `apply_filters()` or `do_action()`. These functions will have the tag as their first argument (i.e. the name of the hook you are plugging into) and then passed arguments as subsequent arguments.

For example, the filter `default_excerpt` receives two parameters, as seen in `includes/post.php`.

```
$post->post_excerpt = apply_filters( 'default_excerpt',
$post_excerpt, $post );
```

The arguments are well named — `$post_excerpt` and `$post` — so it's easy to guess that the first is the excerpt text and the second is the post's object. If you are unsure, it is usually easiest either to look further up in the source or to output them using a test function (make sure you aren't in a production environment).

```
function my_filter_test($post_excerpt, $post) {
    echo "<pre>";
    print_r($post_excerpt);
    print_r($post);
    echo "</pre>";
}
add_filter("default_excerpt", "my_filter_test");
```

Variable Hook Names

Remember when we looked at the `publish_post` action? In fact, this is not used anymore; it was renamed in version 2.3 to `{$new_status}_{$post->post_type}`. With the advent of custom post types, it was important to make the system flexible enough for them. This new hook now takes an arbitrary status and post type (they must exist for it to work, obviously).

As a result, `publish_post` is the correct tag to use, but in reality, you will be using `{$new_status}_{$post->post_type}`. A few of these are around; the naming usually suggests what you will need to name the action.

Who Is Hooked On Who?

To find out which function hooks into what, you can use the neat script below. Use this function without arguments to get a massive list of everything, or add a tag to get functions that are hooked to that one tag. This is a great one to keep in your debugging tool belt!

```
function list_hooked_functions($tag=false){  
    global $wp_filter;  
    if ($tag) {  
        $hook[$tag]=$wp_filter[$tag];  
        if (!is_array($hook[$tag])) {  
            trigger_error("Nothing found for '$tag' hook", E_USER_WARNING);  
            return;  
        }  
    }  
    else {  
        $hook=$wp_filter;  
        ksort($hook);  
    }  
    echo '<pre>';  
    foreach($hook as $tag => $priority){  
        echo "<br /><strong>$tag</strong><br />";  
        ksort($priority);  
        foreach($priority as $priority => $function){  
            echo $priority;  
            foreach($function as $name => $properties) echo "\t$name<br />";  
        }  
    }  
    echo '</pre>';  
    return;  
}
```

Creating Your Own Hooks

A ton of hooks are built into WordPress, but nothing is stopping you from creating your own using the functions we've looked at so far. This may be beneficial if you are building a complex plugin intended for wide release; it will make your and other developers' jobs a lot easier! In the example below, I have assumed we are building functionality for users to post short blurbs on your website's wall. We'll write a function to check for profanity and hook it to the function that adds the blurbs to the wall. Look at the full code below. The explanation ensues.

```
function post_blurb($user_id, $text) {  
  
    $text = apply_filters("blurb_text", $text);  
  
    if(!empty($text)) {  
        $wpdb->insert('my_wall', array("user_id" => $user_id, "date" =>  
date("Y-m-d H:i:s"), "text" => $text), array("%d", "%s", "%s"));  
    }  
}  
  
function profanity_filter($text) {  
    $text_elements = explode(" ", $text);  
    $profanity = array("badword", "naughtyword",  
"inappropriatelanguage");  
  
    if(array_intersect($profanity, $text_elements)) {  
        return false;  
    }  
    else {  
        return $text;  
    }  
}
```

```
add_filter("blurb_text", "profanity_filter");
```

The first thing in the code is the designation of the function that adds the blurb. Notice that I included the `apply_filters()` function, which we will use to add our profanity check.

Next up is our profanity-checking function. This checks the text as its argument against an array of known naughty words. By using `array_intersect()`, we look for array elements that are in both arrays—these would be the profane words. If there are any, then `return false;` otherwise, return the original text.

The last part actually hooks this function into our blurb-adding script.

Now other developers can hook their own functions into our script. They could build a spam filter or a better profanity filter. All they would need to do is hook it in.

Mixing And Matching

The beauty of this system is that it uses functions for everything. If you want, you can use the same profanity filter for other purposes, even outside of WordPress, because it is just a simple function. Already have a profanity-filter function? Copy and paste it in; all you'll need to do is add the one line that actually hooks it in. This makes functions easily reusable in various situations, giving you more flexibility and saving you some time as well.

That's All

Hopefully, you now fully understand how the hooks system works in WordPress. It contains an important pattern that many of us could use even outside of WordPress.

This is one aspect of WordPress that does take some time getting used to if you're coming to it without any previous knowledge. The biggest problem is usually that people get lost in all of the filters available or in finding their arguments and so on, but with some patience this can be overcome easily. Just start using them, and you'll be a master in no time!

Custom Fields Hacks For WordPress

Jean-Baptiste Jung

The incredible flexibility of WordPress is one of the biggest reasons for its popularity among bloggers worldwide. **Custom fields** in particular, which let users create variables and add custom values to them, are one of the reasons for WordPress' flexibility.

In this chapter, we've compiled a list of **10 useful things that you can do with custom fields in WordPress**. Among them are setting expiration time for posts, defining how blog posts are displayed on the front page, displaying your mood or music, embedding custom CSS styles, disabling search engine indexing for individual posts, inserting a "Digg this" button only when you need it and, of course, displaying thumbnails next to your posts.

1. Set An Expiration Time For Posts

The problem. Sometimes (for example, if you're running a contest), you want to be able to publish a post and then automatically stop displaying it after a certain date. This may seem quite hard to do but in fact is not, using the power of custom fields.

The solution. Edit your theme and replace your current WordPress loop with this "hacked" loop:



Image source: Richard Vantielcke

```
<?php  
if (have_posts()) :  
    while (have_posts()) : the_post(); ?>  
        $expirationtime = get_post_custom_values('expiration');  
        if (is_array($expirationtime)) {  
            $expirestring = implode($expirationtime);  
        }  
  
        $secondsbetween = strtotime($expirestring)-time();  
        if ( $secondsbetween > 0 ) {  
            // For example...  
            the_title();  
            the_excerpt();  
        }  
    }  
endif;  
wp_reset_postdata();
```

```
endwhile;  
endif;  
?>
```

To create a post set to expire at a certain date and time, just create a custom field. Specify expiration as a key and your date and time as a value (with the format **mm/dd/yyyy 00:00:00**). The post will not show up after the time on that stamp.

Code explanation. This code is simply a custom WordPress loop that automatically looks to see if a custom field called expiration is present. If one is, its value is compared to the current date and time.

If the current date and time is equal to or earlier than the value of the custom expiration field, then the post is not displayed.

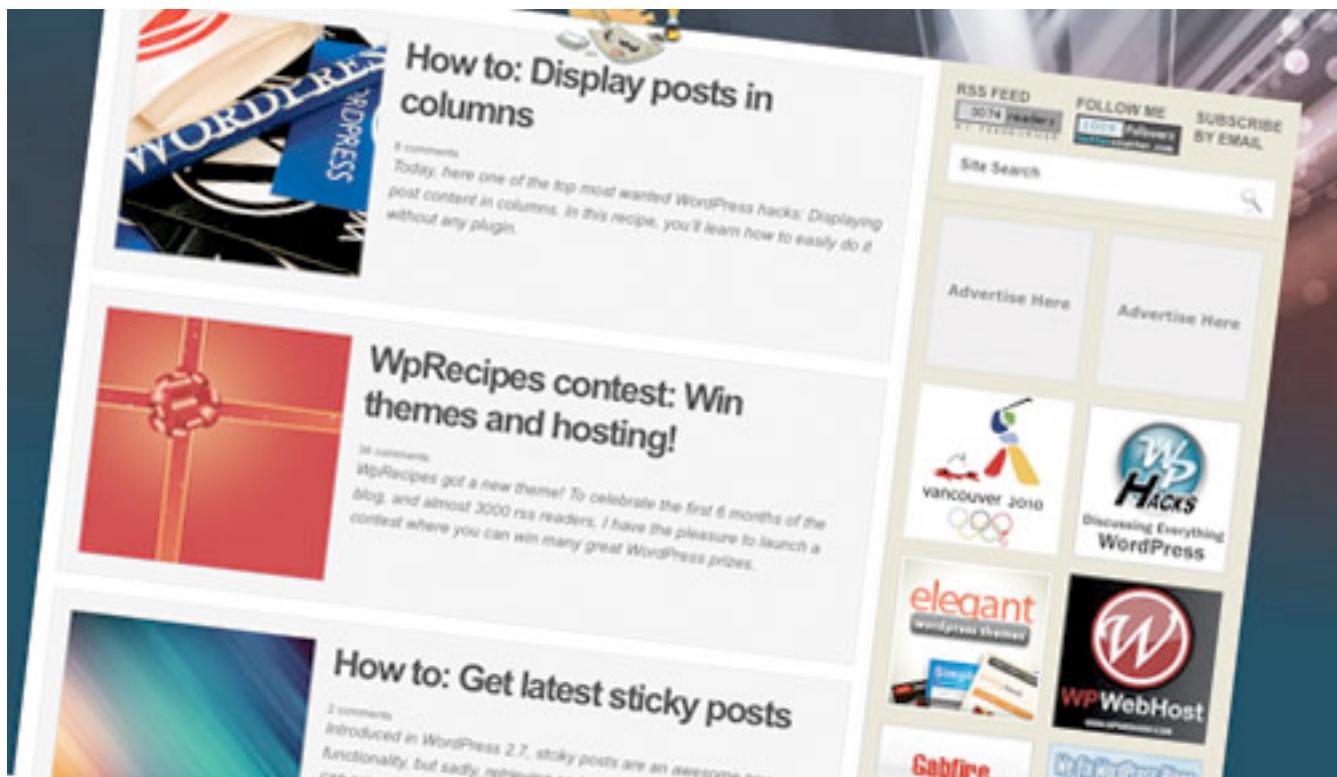
Note that this code does not remove or unpublish your post, but just prevents it from being displayed in the loop.

2. Define How Blog Posts Are Displayed On The Home Page

The problem I've always wondered why 95% of bloggers displays all of their posts the same way on their **home** page. Sure, WordPress has no built-in option to let you define how a post is displayed. But wait: with custom fields, we can do it easily.

The solution. The following hack lets you define how a post is displayed on your home page. Two values are possible:

- Full post
- Post excerpt only



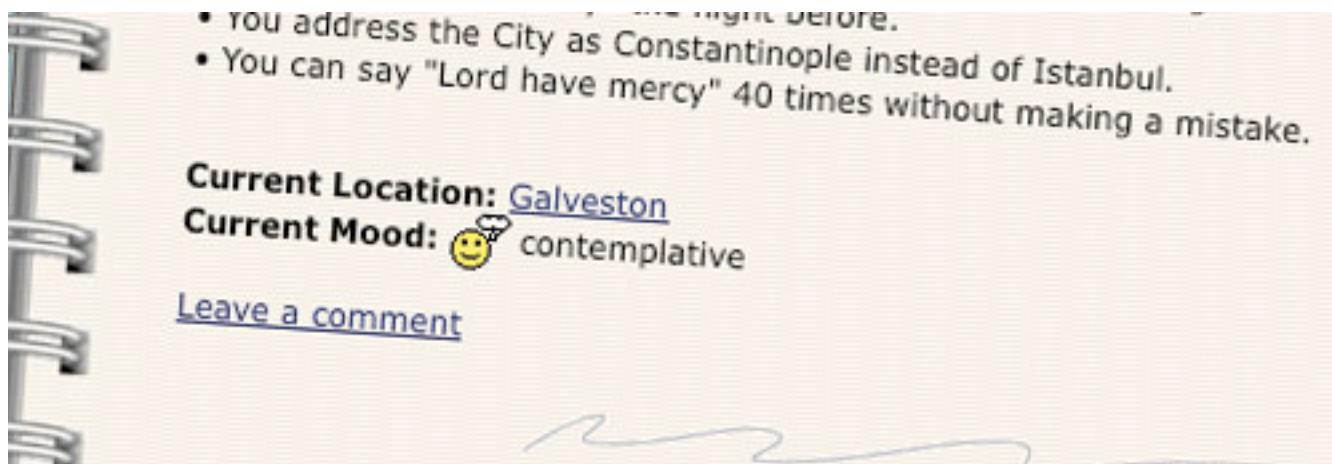
Once more, we'll use a custom WordPress loop. Find the loop in your index.php file and replace it with the following code:

```
<?php if (have_posts()) :  
    while (have_posts()) : the_post();  
        $customField = get_post_custom_values("full");  
        if (isset($customField[0])) {  
            //Custom field is set, display a full post  
            the_title();  
            the_content();  
        } else {  
            // No custom field set, let's display an excerpt  
            the_title();  
            the_excerpt();  
        }  
    endwhile;  
endif;  
?>
```

In this code, excerpts are displayed by default. To show full posts on your home page, simply edit the post and create a custom field called full and give it any value.

Code explanation. This code is rather simple. The first thing it does is look for a custom field called full. If this custom field is set, full posts are displayed. Otherwise, only excerpts are shown.

3. Display Your Mood Or The Music You're Listening To



The problem. About five or six years ago, I was blogging on a platform called LiveJournal. Of course it wasn't great as WordPress, but it had nice features that WordPress doesn't have. For example, it allowed users to display their current mood and the music they were listening to while blogging.

Even though I wouldn't use this feature on my blog, I figure many bloggers would be interested in knowing how to do this in WordPress.

The solution. Open your single.php file (or modify your index.php file), and paste the following code anywhere within the loop:

```
$customField = get_post_custom_values("mood");
if (isset($customField[0])) {
    echo "Mood: ".$customField[0];
}
```

Save the file. Now, when you write a new post, just create a custom field called mood, and type in your current mood as the value.

Code explanation. This is a very basic use of custom fields, not all that different from the well-known hack for displaying thumbnails beside your posts' excerpts on the home page. It looks for a custom field called mood. If the field is found, its value is displayed.

4. Add Meta Descriptions To Your Posts



The problem. WordPress, surprisingly, does not use meta description tags by default.

Sure, for SEO, meta tags are not as important as they used to be. Yet still, they can enhance your blog's search engine ranking nevertheless.

How about using custom fields to create meta description tags for individual posts?

The solution. Open your header.php file. Paste the following code anywhere within the <head> and </head> tags:

```
<meta name="description" content=""  
<?php if ( (is_home()) || (is_front_page()) ) {  
    echo ('Your main description goes here');  
} elseif(is_category()) {  
    echo category_description();  
} elseif(is_tag()) {  
    echo '-tag archive page for this blog' . single_tag_title();  
} elseif(is_month()) {  
    echo 'archive page for this blog' . the_time('F, Y');  
} else {  
    echo get_post_meta($post->ID, "Metadescription", true);  
} ?>">
```

Code explanation. To generate meta descriptions, this hack makes extensive use of WordPress conditional tags to determine which page the user is on.

For category pages, tag pages, archives and the home page, a static meta description is used. Edit lines 3, 7 and 9 to define your own. For posts, the code looks for a custom field called Metadescription and use its value for the meta description.

5. Link To External Resources



The problem. Many bloggers have asked me the following question: “How can I link directly to an external source, rather than creating a post just to tell visitors to visit another website?”

The solution to this problem is to use custom fields. Let’s see how we can do that.

The solution. The first thing to do is open your `functions.php` file and paste in the following code:

```
function print_post_title() {  
    global $post;  
    $thePostID = $post->ID;  
    $post_id = get_post($thePostID);  
    $title = $post_id->post_title;  
    $perm = get_permalink($post_id);  
    $post_keys = array(); $post_val = array();
```

```
$post_keys = get_post_custom_keys($thePostID);

if (!empty($post_keys)) {
foreach ($post_keys as $pkey) {
    if ($pkey=='url1' || $pkey=='title_url' || $pkey=='url_title')
{
    $post_val = get_post_custom_values($pkey);
}
}

if (empty($post_val)) {
    $link = $perm;
} else {
    $link = $post_val[0];
}

} else {
    $link = $perm;
}

echo '<h2><a href="'. $link .'" rel="bookmark" title="'.$title.'">' .
$title.'</a></h2>';
}
```

Once that's done, open your `index.php` file and replace the standard code for printing titles...

```
<h2><a href=<?php the_permalink() ?>" rel="bookmark"
title="Permanent Link to <?php the_title(); ?>"><?php the_title(); ?
></a></h2>
```

... with a call to our newly created `print_post_title()` function:

```
<?php print_post_title() ?>
```

Now, whenever you feel like pointing one of your posts' titles somewhere other than your own blog, just scroll down in your post editor and create or

select a custom key called `url1` or `title_url` or `url_title` and put the external URL in the value box.

Code explanation. This is a nice custom replacement function for the `the_title()` WordPress function.

Basically, this function does the same thing as the good old `the_title()` function, but also looks for a custom field. If a custom field called `url1` or `title_url` or `url_title` is found, then the title link will lead to the external website rather than the blog post. If the custom field isn't found, the function simply displays a link to the post itself.

6. Embed Custom CSS Styles



The problem. Certain posts sometimes require additional CSS styling. Sure, you can switch WordPress' editor to HTML mode and add inline styling to your post's content. But even when inline styling is useful, it isn't always the cleanest solution.

With custom fields, we can easily create new CSS classes for individual posts and make WordPress automatically add them to the blog's header.

The solution. First, open your header.php file and insert the following code between the <head> and </head> HTML tags:

```
<?php if (is_single()) {  
    $css = get_post_meta($post->ID, 'css', true);  
    if (!empty($css)) { ?>  
        <style type="text/css">  
            <?php echo $css; ?>  
        </style>  
        <?php }  
    } ?>
```

Now, when you write a post or page that requires custom CSS styling, just create a custom field called css and paste in your custom CSS styling as the value. As simple as that!

Code explanation. First, the code above makes sure we're on an actual post's page by using WordPress' conditional tag `is_single()`. Then, it looks for a custom field called `css`. If one is found, its value is displayed between `<style>` and `</style>` tags.

7. Re-Define The <title> Tag



The problem. On blogs, as on every other type of website, content is king. And SEO is very important for achieving your goals with traffic. By default, most WordPress themes don't have an optimized <title> tag.

Some plug-ins, such as the well-known “All in One SEO Pack,” override this, but you can also do it with a custom field.

The solution. Open your header.php file for editing. Find the <title> tag and replace it with the following code:

```
<title>
<?php if (is_home ()) {
    bloginfo('name');
} elseif (is_category()) {
    single_cat_title(); echo ' - ' ; bloginfo('name');
} elseif (is_single()) {
    $customField = get_post_custom_values("title");
    if (isset($customField[0])) {
        echo $customField[0];
    } else {
        single_post_title();
    }
} elseif (is_page()) {
    bloginfo('name'); echo ': ' ; single_post_title();
} else {
    wp_title(' ',true);
} ?>
</title>
```

Then, if you want to define a custom title tag, simply create a custom field called title, and enter your custom title as a value.

Code explanation. With this code, I have used lots of template tags to generate a custom <title> tag for each kind of post: home page, page, category page and individual posts.

If the active post is an individual post, the code looks for a custom field called title. If one is found, its value is displayed as the title. Otherwise, the code uses the standard `single_post_title()` function to generate the post's title.

8. Disable Search Engine Indexing For Individual Posts



The problem. Have you ever wanted to create semi-private posts, accessible to your regular readers but not to search engines? If so, one easy solution is to... you guessed it! Use a custom field.

The solution. First, get the ID of the post that you'd not like to be indexed by search engines. We'll use a post ID of 17 for this example.

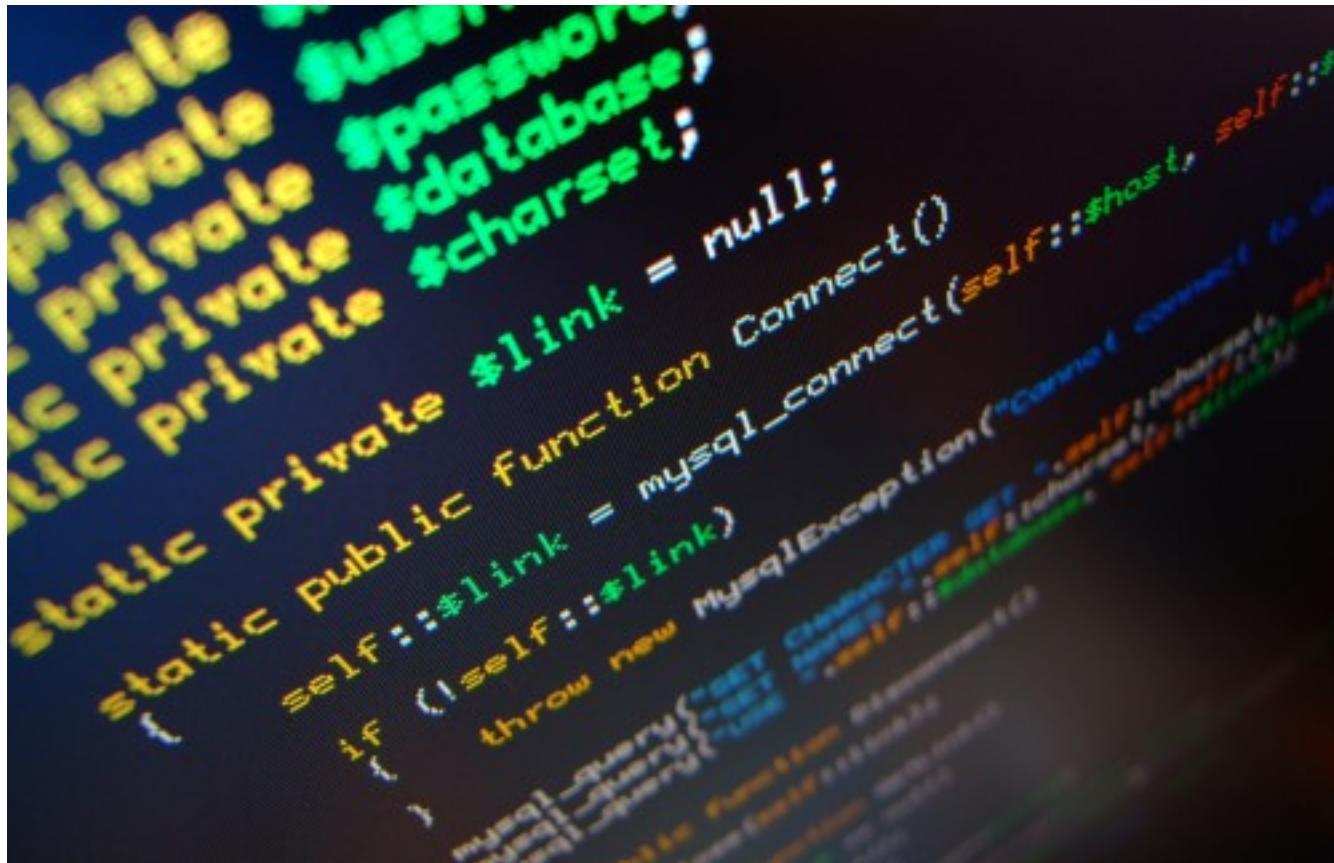
Open your header.php file and paste the following code between the <head> and </head> tags:

```
<?php $cf = get_post_meta($post->ID, 'noindex', true);
if (!empty($cf)) {
echo '<meta name="robots" content="noindex"/>';
}
?>
```

That's all. Pretty useful if you want certain info to be inaccessible to search engines!

Code explanation. In this example, we used the `get_post_meta()` function to retrieve the value of a custom field called `noindex`. If the custom field is set, then a `<meta name="robots" content="noindex" />` tag is added.

9. Get Or Print Any Custom Field Value Easily With A Custom Function



The problem. Now that we've shown you lot of great things you can do with custom fields, how about an automated function for easily getting custom fields values?

Getting custom field values isn't hard for developers or those familiar with PHP, but can be such a pain for non-developers. With this hack, getting any custom field value has never been easier.

The solution. Here's the function. Paste it into your theme's `functions.php` file. If your theme doesn't have this file, create it.

```
function get_custom_field_value($szKey, $bPrint = false) {  
    global $post;
```

```
$szValue = get_post_meta($post->ID, $szKey, true);
if ( $bPrint == false ) return $szValue; else echo $szValue;
}
```

Now, to call the function and get your custom field value, use the following code:

```
<?php if ( function_exists('get_custom_field_value') ) {
    get_custom_field_value('featured_image', true);
} ?>
```

Code explanation. First, we use the PHP `function_exists()` function to make sure the `get_custom_field_value` function is defined in our theme. If it is, we use it. The first argument is the custom field name (here, `featured_image`), and the second lets you echo the value (true) or call it for further PHP use (false).

10. Insert A “Digg This” Button Only When You Need It

About the Author

Muhammad Usama Alam is a software engineer and Web developer with over designing and developing business solutions for the enterprise. He also blogs at [\(al\)](#)



Published in [How-To](#), May 9th, 2009
Tags: [...](#)

The problem. To get traffic from well-known Digg.com, a good idea is to integrate its “Digg this” button into your posts so that readers can contribute to the posts’ success.

But do all of your posts need this button? Definitely not. For example, if you write an announcement telling readers about improvements to your website, submitting the post to Digg serves absolutely no value.

The solution. Custom fields to the rescue once again. Just follow these steps to get started:

1. Open your single.php file and paste these lines where you want your “Digg this” button to be displayed:

```
<?php $cf = get_post_meta($post->ID, 'digg', true);  
if (!empty($cf)) {  
echo 'http://digg.com/tools/digghis.js " type="text/javascript">' } ?>
```

- Once you've saved the single.php file, you can create a custom field called digg and give it any value. If set, a Digg button will appear in the post.

Code explanation. This code is very simple. Upon finding a custom field called digg, the code displays the “Digg this” button. The JavaScript used to display the “Digg this” button is provided by Digg itself.

Bonus: Display Thumbnails Next To Your Posts



The problem Most people knows this trick and have implemented it successfully on their WordPress-powered blogs. But I figure some people still may not know how to display nice thumbnails right next to the posts on their home page.

The solution.

1. Start by creating a default image in Photoshop or Gimp. The size in my example is 200×200 pixels but is of course up to you. Name the image `default.gif`.
2. Upload your `default.gif` image to the image directory in your theme.
3. Open the `index.php` file and paste in the following code where you'd like the thumbnails to be displayed:

```
<?php $postimageurl = get_post_meta($post->ID, 'post-img', true);  
if ($postimageurl) {  
?>  
    <a href="php the_permalink(); ?&gt;" rel="bookmark"&gt;&lt;img src="<?php<br/echo $postimageurl; ?>" alt="Post Pic" width="200" height="200" /></  
a>  
<?php } else { ?>  
    <a href="php the_permalink(); ?&gt;" rel="bookmark"&gt;&lt;img src="<?php<br/bloginfo('template_url'); ?>/images/wprecipes.gif" alt="Screenshot"  
width="200" height="200" /></a>  
<?php } ?>
```

4. Save the file.
5. In each of your posts, create a custom field called `post-img`. Set its value as the URL of the image you'd like to display as a thumbnail.

Code explanation The code looks for a custom field called `post-img`. If found, its value is used to display a custom thumbnail.

In case a `post-img` custom field is not found, the default image is used, so you'll never have any posts without thumbnails.

More Custom Field Resources

- [Add Thumbnails to WordPress with Custom Fields](#)

A very detailed article about adding thumbnails to your posts with custom fields. A great follow-up to the last hack we showed!

- [How to Use WordPress Custom Fields](#)

Want to know more about custom fields? Then this article is definitely for you.

- [Creating Custom Write Panels in WordPress](#)

A very detailed tutorial on creating custom write panels in WordPress using custom fields.

- [Custom Shortcodes](#)

A cool WordPress plug-in for managing custom fields using the insert shortcodes.

- [More Fields](#)

The More Fields plug-in allows you to create more user-friendly custom fields. Definitely interesting for when you create WordPress-powered websites for clients!

Power Tips For WordPress Template Developers

Jacob Goldman

With its latest releases, WordPress has extended its potential well beyond blogging, moving toward an advanced, robust and very powerful content management solution. By default, WordPress delivers a very lightweight, minimal system that offers only basic functionalities. But where the WordPress core falls short, there are a wealth of plug-ins that extend its limitations.

Plug-ins often offer simple solutions, but they are not always elegant solutions: in particular, they can add a noticeable overhead, e.g. if they offer more functionality than needed. In fact, some general and frequently needed WordPress-functionalities can be added to the engine without bloated plugins, using the software itself.

This article presents **8 tips for WordPress template developers that address common CMS implementation challenges**, with little to no plug-in dependence. These examples are written for WordPress 2.7+ and should also work in the latest WordPress-version.

1. Associating pages with post categories

WordPress enables administrators to identify any page as the posts page: this is ideal for CMS implementations featuring a single news or blog feed. However, **WordPress provides no simple, out-of-the-box mechanism to configure a site with multiple, independent feeds**.

Here's a common use case: a company wants a simple and casual blog, and a separate and more formal feed for press releases inside their "About Us" section. Let's list a few requirements mandated by a sample client seeking just that:

- At no point should these two feeds be displayed as one.
- Links to these feeds need to appear in page navigation.
- The Press Release page needs to have static, maintainable content above its feed.
- For SEO purposes, these feeds should have a page-like permalink structure: in other words, "mysite.com/category/press-releases" is unacceptable; the goal is "mysite.com/about-us/press-releases".

mysite.com	
Home	<i>static front page</i>
Services	
Specialization	
Case Studies	
About Us	
History	
Press Releases	<i>press feed, w/ RSS</i>
Our Blog	<i>simple blog</i>

As is often the case, there are several approaches one can take. Major considerations used to gauge the best approach include the number of standalone feed pages (one, in this case: Press Releases) and the necessity for a "primary" feed requiring multiple category support. For this example, let us assume that "Our Blog" does need to behave like a full featured blog with categories.

PREPARING THE SITE

This approach to “**pages with single feeds**” is built upon an association created between a page and a post category. The “primary” blog will simply be the “posts page” with a few template adjustments that will exclude posts from the “Press Releases” feed. To meet the SEO requirement for a logical and consistent URL structure, we will need to carefully configure and set permalinks.

- In the “Reading” settings, ensure that the “Front page displays” option is set to “A static page”, and that the “Posts page” is set to “Our Blog”.
- In the “Permalinks” settings for WordPress, ensure that “Custom Structure” is selected. The structure should be: **/%category%/
%postname%/.**
- In the page list, identify the the permalink (or slug) for the “About Us” page (using our example sitemap: let’s say “about-us”). Identify the slug for the Press Releases page (“press-releases”).
- Two corresponding categories must be added: an “About Us” category with a matching permalink (“about-us”), and a “Press Releases” category with a matching permalink (“press-releases”) and its parent category set to “About Us”.
- Create a post in the “Press Releases” category for testing purposes.

Establish permalink structure allowing posts to appear as if under a page

QUICK EDIT

Title	Press Releases	Parent	About Us
Slug	press-releases	Order	0
Date	Mar 27, 2009 @ 20:38	Template	Default Template
Password		<input type="checkbox"/> Allow Comments <input type="checkbox"/> Allow Pings	
Status	Published		

Cancel **Update Page**

Name	Description	Slug	Posts
About Us — Press Releases		press-releases	0
About Us		about-us	0

Establish matching category hierarchy and page hierarchy to single feed page

EXCLUDING A CATEGORY FROM THE BLOG PAGE

To exclude a category from the main blog page (which shows all posts across categories), the post query used for the [blog page template](#) must be modified.

The WordPress codex [outlines the solution](#). Simply identify the category ID for the “Press Releases” category (hovering the mouse over the category name in the admin panel and looking at the URL in the status bar is an easy way to find the ID – let’s use 5 for the example), and insert the following code above the post loop:

```
query_posts ("cat=-5");
```

Note that many templates also include a list of categories in the sidebar, recent post lists, and other components that may not exclude posts from the “press releases” category. These will also need to be modified to exclude the category; this is easily supported by most WordPress calls.

ENABLING THE INDIVIDUAL FEED PAGE

The feed page will require a [custom page template](#). For this example, we named the template “Press Release Feed”, and used the generic “page.php” template as a starting point (copying it and renaming it “page_press.php”).

Since the requirements mandate static, editable page content above the feed, the first post loop – that drops in the page content – will be left as is.

Below the code for page content output, another post query and loop will be executed. Once completed, the query should be reset using “wp_reset_query” so that items appearing after the loop – such as side navigation – can correctly reference information stored in the original page query.

The general framework for the code is below. The [query posts documentation on the WordPress codex](#) provides insight into great customization.

```
query_posts('category_name=Press Releases');
if ( have_posts() ) : while ( have_posts() ) : the_post();
    //post output goes here... index.php typically provides a good
    //template
endwhile; endif;
wp_reset_query();
```

Of course, be certain to assign the “Press Releases” page the new template, in the page editor.

THE DEVIL IS IN THE DETAILS

Depending on the characteristics of the individual site, many additional template customizations – beyond those outlined above – will probably be necessary. In particular, this “power tip” does not cover specific strategies for handling individual post views within these isolated feeds. At the high

level, using conditional [`in_category`](#) checks within the “single.php” template (used for output of individual posts) should provide the foundation for customizing post views based on their category. If you are interested, a more detailed article may explore these strategies in greater detail (please let us know in the comments!).

ALTERNATIVE SCENARIOS

Creating individual page templates for each standalone feed is an efficient solution for a site with only a couple of such feeds. There are, however, WordPress-powered sites like [m62 visual communications](#) that extend the idea of category and even tag association with pages much more deeply. m62 features dozens of pages associated with individual blog categories, parent categories, and tags, seamlessly mixed in with standard, “feed-less” pages. In these instances, a smarter solution would involve specialized templates that **match tag and category permalinks against page permalinks to dynamically create associations.**

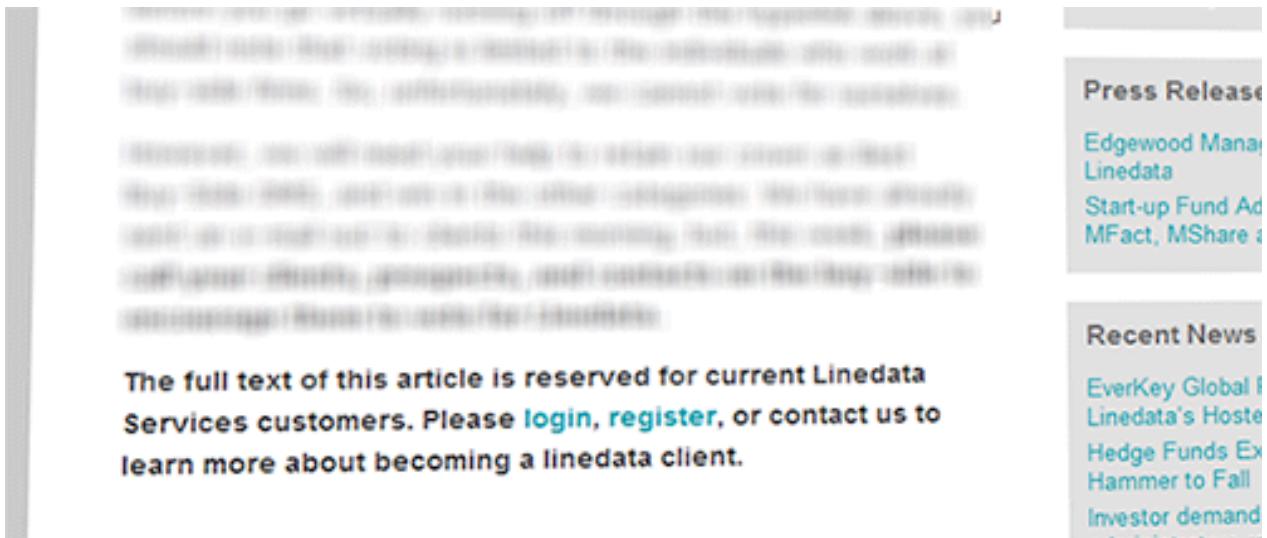
This approach can also facilitate sites that require more than one “primary” (multi-category) blog, through the use of hierarchical categories / parent categories.

Again, if there is interest, a future article can discuss these methods in detail.

2. “Friendly” member only pages

Out-of-the-box, WordPress includes an option to designate any page or post as private. By default, these items do not show up in page or post lists (including navigation) and generate 404 errors when visited directly – unless the visitor is logged in. While utilitarian, more often than not, this is not ideal for usability.

Often times, sites intentionally make public visitors aware of pages or posts whose full content is only visible to members. A friendly message alerting visitors that they have reached a members-only page, with a prompt to log in, may be a better solution. Content-centric websites may tease the public with “above the fold” – or abbreviated – content for the entire audience, while enticing the visitor to log in or sign up to read the entire article.



This example offers a framework for these “hybrid” member / public pages using the latter scenario as an example. Content featured “above the fold” – or above the “more” separator – will be visible to the general public.

Content below the fold will only be available to members. In place of content below the fold, public visitors will be prompted to log in.

This approach to “hybrid” pages is built upon public, published pages with a custom field used to identify the page content as “member exclusive”.

1. Create a page or post.
2. Start with a paragraph or two visible to the general public.
3. Insert the “more tag” at the end of the public content.
4. Enter content visible only to logged in members below the more tag.

5. Add a custom field named “member_content”. Set its value to 1.
6. Publish the page with public visibility (the default).

The screenshot shows the WordPress visual editor interface. At the top, there's a toolbar with various icons for upload, insert, and styling. Below the toolbar, the content area is divided into two sections by a horizontal line:

- Visual** tab is selected.
- HTML** tab is also present.
- Content Above the Fold:** "This content, above the fold, is visible to the public."
- Content Below the Fold:** "This content, which is below the fold, will only be visible to members who are logged in." A "More..." button is visible next to this section.

At the bottom of the editor, there are status bars for "Path: p", "Word count: 12", and "Draft Saved at 4:08:25 pm".

Custom Fields panel:

Name	Value
member_content	1

Buttons for "Delete" and "Update" are located at the bottom of the panel.

The next step involves editing the applicable template files. Typically, this will be “page.php” (pages) and “single.php” (posts). Note that if these hybrid views will only apply to pages, a developer can create a “member content” page template as an alternative to using a custom field. Doing so will eliminate the need for the custom field check and alternative outputs inside the same template.

For this example, we shall assume that we created a post (not page) with member exclusive content. Therefore, we will need to edit “single.php”. Inside the template, find the `the_content` call used to drop in page and post content. Here’s what this often looks like before the changes:

```
the_content();
```

Here is the new code with the alternative “public” view:

```
if(!get_post_meta($post->ID, 'member_content', true) ||  
is_user_logged_in()) {  
    the_content('<p class="serif">Read the rest of this entry ></p>');  
} else {  
    global $more; // Declare global $more (before the loop).  
    $more = 0; // Set (inside the loop) to display content above the  
more tag.  
    the_content(""); //pass empty string to avoid showing "more" link  
    echo "<p><em>The complete article is only available to members.  
Please log in to read the article in its entirety.</em></p>";  
}
```

Combine this with the next tip to include a login form that sends members right back to the current page or post.

3. Embedding a log-in form that returns to the current location

Sometimes, sending members to the standard WordPress login form is not ideal. It may, for instance, not be consistent with the look and feel a client is seeking. There may also be instances where embedding a login form in a page – as in tip 7 – offers superior usability compared to clicking a link for the login page.

Staff Area - Login Required

This section is restricted to MIP Committee Members only.

Username

Password

Remember my information

If you have any questions, please email info@museumsinthepark.org

The code below drops the WordPress login form into the template, and sends the user back to the page they logged in from.

```
<?php if(!is_user_logged_in()) { ?>
<form action=<?php echo wp_login_url(get_permalink()); ?>
method="post">

    <label for="log"><input type="text" name="log" id="log" value=<?
    php echo wp_specialchars(stripslashes($user_login), 1) ?>">
    size="22" /> User</label><br />

    <label for="pwd"><input type="password" name="pwd" id="pwd"
    size="22" /> Password</label><br />

    <input type="submit" name="submit" value="Send" class="button" />

    <label for="rememberme"><input name="rememberme" id="rememberme"
    type="checkbox" checked="checked" value="forever" /> Remember me</
    label>

</form>
<?php } ?>
```

Be aware of a pitfall of this easy-to-implement power tip: if the user fails to login with the proper credentials, the log-in error will appear on the standard WordPress login form. The visitor will, however, still be redirected back to the original page upon successful log-in.

4. IDENTIFYING THE TOP LEVEL PAGE

The “top level page” is the highest level page within the current branch of the sitemap. For example, if you consider the page below, you’ll find that “Support & Resources”, “Finding Support”, and “Support for Patients” all share the top level page “Support & Resources”.

The screenshot shows a navigation bar at the top with three items: "Support & Resources", "Healthcare Professionals", and "Newsroom". Below the navigation bar is a sidebar titled "Support & Resources" which contains a list of links:

- ▶ [Overview](#)
- ▶ [Managing Health Care](#)
- ▶ [**Finding Support**](#)
 - ▶ [Caregiver Support Groups](#)
 - ▶ [Day Programs & Respite](#)
 - ▶ [Connecting with Others](#)
 - ▶ [Support for Patients](#)
- ▶ [Caregiving Challenges](#)

There are some relatively new plug-ins that make “section”, or “top level page” WordPress navigation a cinch, such as [Simple Section Navigation](#). However, there are plenty of instances (outside of navigation) where the template may need to be aware of the current top level page.

For instance, you many want to be able to style certain design elements, such as the navigation bar’s background image, depending on the currently chosen section. This can be achieved by checking the page ID of the top

level page inside the header, and dropping in additional styles when the IDs for those top level pages were found.



Here is how it works. Although WordPress offers no built in call to determine the **top level page**, it can be found with a single line of code in the template:

```
$top_level = ($post->post_parent) ? end(get_post_ancestors($post)) : $post->ID;
```

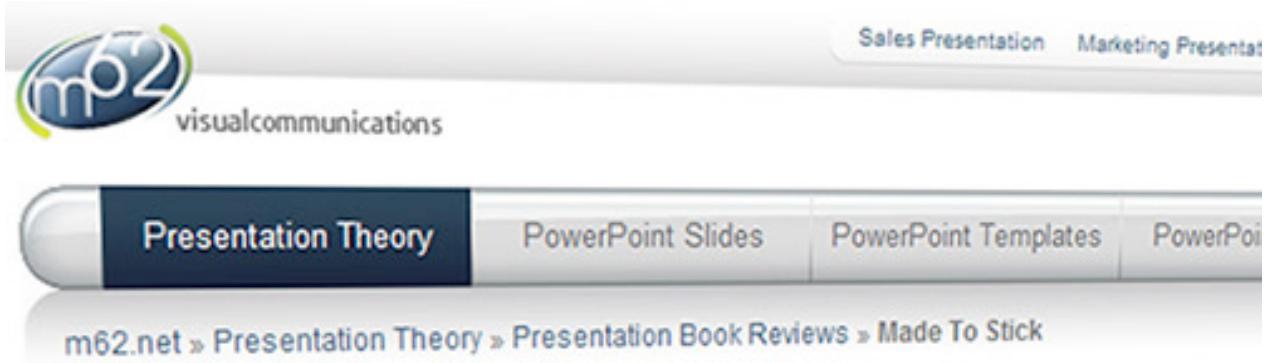
Using a [ternary conditional](#), this line of code checks the value of \$post->post_parent, which returns the current page's parent page ID, if one exists. If the conditional is evaluated as "true" (as any positive integer will), than the current page has some "ancestry"; in other words, it is inside of a page hierarchy or branch in the sitemap. If the conditional fails, the page is either a top level page, or not in any page ancestry (i.e. a post on a site without a blog "page" assigned).

If the current page has ancestry, an array containing the hierarchy "above" the page (parents, grandparents, etc) can be retrieved using the get_post_ancestors function. The last value in the array that function returns is always the ID of the top level page. Jump right to the last value using PHP's end function. If the conditional fails (no ancestry), the code simply grabs the current page ID.

Keep in mind that, in many instances, this information is only useful when WordPress is working with an actual page (as opposed to a post, 404 page, etc). Therefore, this function and code that uses the top_level variable, may

need to be wrapped inside a check that confirms that WordPress is loading a page: see the [is_page\(\) function](#).

5. Breadcrumb Navigation – without a plug-in



Made To Stick

There are plenty of WordPress extensions that generate breadcrumb navigation. But you can actually **create custom breadcrumb navigation with only a handful of lines of code** in the template, opening up greater control and, potentially, less overhead. This approach to breadcrumbs builds on the `get_post_ancestors` function discussed in tip #4.

This tip won't review formatting of breadcrumbs; for this example, the breadcrumbs will be dropped in an unordered bullet list. As it happens, lists of links tend to be a good format for search engines, and you can format them almost any way you like.

To start with, here is a basic implementation of breadcrumbs that only deals with pages and includes a breadcrumb for "home" (the front page of the site) at the beginning of the list. Depending on the design of a particular template, some checks may need to be placed around this code. In this example, it will be assumed that this code will be placed in the `header.php` template file, that the crumbs should appear only on pages, and that it

should not show up on the front page. The current page and front page link will also be assigned special CSS classes for styling purposes.

```
if (is_page() && !is_front_page()) {
    echo '<ul id="breadcrumbs">';
    echo '<li class="front_page"><a href="'.get_bloginfo('url').'">Home</a></li>';
$post_ancestors = get_post_ancestors($post);
if ($post_ancestors) {
    $post_ancestors = array_reverse($post_ancestors);
    foreach ($post_ancestors as $crumb)
        echo '<li><a href="'.get_permalink($crumb).'">'.get_the_title($crumb).'</a></li>';
}
echo '<li class="current"><a href="'.get_permalink().'>'.get_the_title().'</a></li>';
echo '</ul>';
}
```

If the WordPress implementation has a static front page and has been assigned a “blog” page, one might want to show the breadcrumb path to the blog page. This can be accomplished by adding `is_home()` to the conditional check at the top:

```
if ((is_page() && !is_front_page()) || is_home()) {
    ...
}
```

The next evolution of this code involves the inclusion of breadcrumbs for individual category archives as well as individual posts. Note that WordPress allows posts to be assigned to multiple categories; to avoid making our breadcrumb trail unwieldily, the script will simply grab the first category assigned to the post. For the sake of simplicity, the example will be assumed that hierarchical categories are not in play.

```
if ((is_page() && !is_front_page()) || is_home() || is_category()
|| is_single()) {
    echo '<ul id="breadcrumbs">';
    echo '<li class="front_page"><a href="'.get_bloginfo('url').'">Home</a></li>';
    $post_ancestors = get_post_ancestors($post);
    if ($post_ancestors) {
        $post_ancestors = array_reverse($post_ancestors);
        foreach ($post_ancestors as $crumb)
            echo '<li><a href="'.get_permalink($crumb).'">'.get_the_title($crumb).'</a></li>';
    }
    if (is_category() || is_single()) {
        $category = get_the_category();
        echo '<li><a href="'.get_category_link($category[0]->cat_ID).'">'.$category[0]->cat_name.'</a></li>';
    }
    if (!is_category())
        echo '<li class="current"><a href="'.get_permalink().'>'.get_the_title().'</a></li>';
    echo '</ul>';
}
```

There are many ways to extend the breadcrumb navigation further. For instance, a developer might want breadcrumbs for different types of archives (tags, months, etc), or may incorporate hierarchical categories. While this article won't walk through every possible implementation, the samples above should provide you with a solid framework to work with.

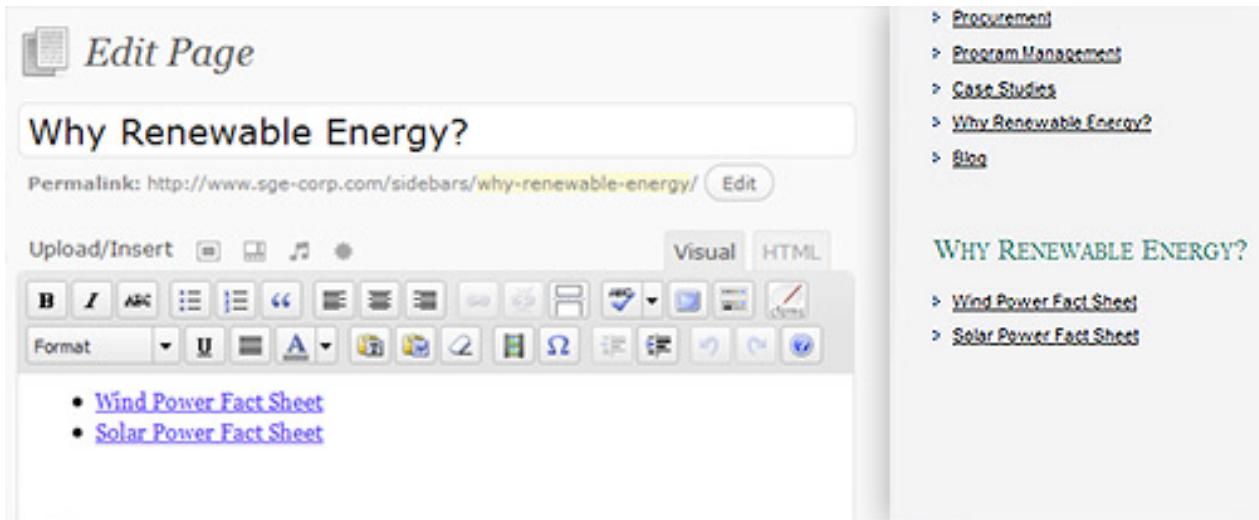
6. Creating sidebar content elements

Many websites feature distinct sidebars with common elements represented throughout the site, such as section navigation, contact information, and special badges (i.e. “Follow us on Twitter”). It is also common for sites to feature more basic HTML blocks in the sidebar that are associated with a single page, or several pages that may or may not be tied together in any logical way.

Some content management systems enable the idea of multiple content blocks out of the box. For instance, [CitySoft Community Enterprise](#) allows the editor to select from a variety of page layouts, some of which include multiple blocks of content that can be edited independently. This is convenient for some, though it does have some limitations: (1) it can be hard to integrate the prefabricated layout blocks into unusual areas in the overall site template, and (2), reusing some of these content blocks for multiple pages is not possible (without some additional, complicated custom development).

Here’s how to **implement reusable “sidebar elements” in WordPress**. For the sake of simplicity, this example assumes that only one sidebar element can be assigned to a page.

Fundamentally, these sidebar elements will simply be pages. Although non-essential, it can be a good organizational practice to create a page called “sidebars” that will contain all of the sidebar pages. Be careful to exclude this page in top level navigation and any other page lists or sitemaps. Sidebar elements are then constructed as “private” pages (so they cannot be searched or viewed independently by general visitors), with the “sidebar” container page set as the parent page. The title of the sidebar page will be used for the title of the sidebar element.



Once the sidebar has been created, the editor will need the ID of the sidebar page. The easiest way to find this is by rolling over the page title in the admin page list, and looking for the “id” in the URL (typically in the status bar).

To assign the sidebar to a page, a new [custom field](#) is assigned to the page that will hold the sidebar called “sidebar”. The value for this field is the page ID of the sidebar page.

Now, in the sidebar template file (or wherever the sidebar element should appear), some code is included that checks for the custom field, and – if found – drops in the referenced page. To make the process of dropping the sidebar page content a bit more simple, the example will use the light weight plug-in, [Improved Include Page](#). Here’s the code, which also drops “h2” tags around the page title:

```
$sidebar_pg = get_post_meta($post->ID, 'sidebar', true);
if (function_exists('iinclude_page')) && $sidebar_pg) {
    include_page($sidebar_pg, 'displayTitle=true&titleBefore=<h2>&titleAfter=</h2>
        &displayStyle=DT_FULL_CONTENT&allowStatus=publish, private');
}
```

7. Feature selected posts on the front page

Many CMS implementations feature some selected items from the blog feed on the home page, or even throughout the site in a sidebar or footer element. Content editors are, wisely, selective about what merits a front page mention. Here's how to **implement a selective blog feed** that can be placed on a front page template or anywhere else in the design.



A special category is needed to classify posts as “Featured”; a category named “Featured” or “Front Page” is a good convention. For the content editor, marking a post as “featured” is as simple as adding it to this category (remember: posts can have multiple categories). On the template side, the ID of the “featured” category will be needed. The easiest way to find the category ID is by rolling over the category “edit link” inside WordPress administration and noting the ID in the URL (typically in the status bar).

Using this ID (“4” in the example), and the number of posts to feature on the home page (let’s say three), the following code will list the featured posts beginning with the recent ones.

```
echo "<h3>Featured Blog Posts</h3>";
echo "<ul>";
$feat_posts = get_posts('numberposts=4&category=71');
foreach ($feat_posts as $feat) {
    echo '<li><a href="'.get_permalink($feat->ID).'">'.$feat->
post_title.</a></li>';
}
echo "</ul>";
```

As with the other examples, this code can be extended in a number of ways. For example, the [SGE Corporation](#) front page features the excerpt for the most recent item. The excerpt can be manually entered in the “excerpt” field or pulled automatically (if none is provided) by grabbing a certain number of characters from the beginning of the post’s content.

8. Highlight current post’s category

WordPress page lists assign special classes to each item, including classes that indicate whether a page is the current page, a parent page, an ancestor page, and so forth. Category lists do assign a special class (current-cat) to appropriate list items when the user is browsing a category’s archive.

Unfortunately, **categories do not, by default, get this special class assigned to them when the user is on a post inside the category.**

However, one can override this default limitation by grabbing the current category ID and passing it to the wp_list_categories function.

```
$category = get_the_category();
wp_list_categories('current_category='.$category[0]->cat_ID);
```

Note that there is one significant downside to this approach – only some category can be passed to the list categories function. So if a post is assigned to multiple categories, only the first category will be highlighted. However, if a site has distinct categories (say a news feed and an editorial

feed), this can help a template developer treat the category more like a page navigation item.

Advanced Power Tips For WordPress Template Developers

Jacob Goldman

In the [previous chapter](#), I presented 8 basic techniques for adding popular features to the front end of a WordPress-powered website. The premise was that WordPress has become an elegant, lightweight content management solution that offers the fundamentals out of the box, atop a modular core that offers incredible potential in the hands of a capable developer.

WordPress does not try to be an “everything to everyone” CMS right out of the box. Many systems do an average job incorporating 99% of what the potential CMS market might need, even if the last 15-20% is used only by a fraction of the market and adds considerably to the system’s overall “heft” (or bloat). At the other end of the spectrum are completely custom solutions that are finely tailored to exact needs, at the cost of reinventing wheels like polished content editing with media management and version control.

The self-proclaimed WordPress “code poets” have, alternatively, focused on doing an A+ job with the “fat middle”: the 80-85% of features that almost everyone needs, and coupling those with a first rate framework and API that enables capable developers to add in almost any niche or “long tail” feature. In fact, the core WordPress framework is so capable that a handful of “intermediary” frameworks that sit on top of it have already emerged.

That previous “Power Tips” entry scratched the surface, covering a handful of API calls mixed in with some simple PHP code and configuration tips intended to help beginner WordPress template developers kick their game up a notch. This chapter takes power tips to the next level, expanding on

some of the topics in the first chapter, and **introducing more advanced techniques and methods for customizing not only the front end, but the content management (or back end) experience.**

Multiple Column Content Techniques

The average blog or website has a single, clearly defined block of space for a given page's or post's unique content. But **there are plenty of creative websites that don't conform to this simple notion of “one unique block” per page.** A creative online portfolio layout might feature a screenshot and project description in a left column, and a list of technologies used in a right column. Both the left and right column are unique to each portfolio page.

Here's a screenshot from an in-development website project, built on WordPress. The “projects” area features portfolio-like layouts of green building projects throughout the state. In addition to a specially designed gallery visualization, note that the individual project profile has two distinct columns.

marketplace

 SEARCH

PROJECTS

SERVICE PROVIDERS

RESOURCES

[Home](#) > [LEED Project Profiles](#) > **Fort Adams Redoubt & Jail**

 Commercial
Government
Institutional
Residential
LEED Facts »

Gold	42
Sustainable Sites	9/14
Water Efficiency	4/5
Energy & Atmosphere	9/17
Materials & Resources	9/13
Indoor Environ. Quality	10/14
Innovation & Design	1/5

Architect/Interior Designer: Newport Collaborative Architects, Inc.

Civil Engineer: Fuss & O'Neill

Commissioning Agent: Synergy

Consulting Contractor: The Damon Company

Landscape Architect: Martin Van Hot, Island Garden Shop

LEED Consultant: Newport Collaborative Architects, Inc.

Lighting Designer: Newport Collaborative Architects, Inc.

Structural Engineer: Structures North, Inc.

Mechanical and Plumbing Engineer: Donovan and Sons

Project Size: 4,181 square feet

Total Project Cost: \$1.3 million

Cost per square foot: \$310

Photographer: Kendra Cliness
Content Block 2
Project Tags: [Historic](#), [multi-use](#), [test](#)


photo 1/5

[PREV](#) | [NEXT](#)

The Redoubt Jail is one of the most visible and prominent historic defensive structures at Fort Adams. The original single story structure dates from the first half of the 19th century, with a later 19th century first floor addition. A second story addition, also dating from the later 19th century, was later removed and the building sat as a disused single story storage shed from 1948 to 2008.

The existing masonry on the first floor was completely restored and the jail cells are now part of the historic museum.

The former Guard Room is now used as a multi-use assembly space, while the former Officer of the Guard's office is now storage and the lift location.

The second floor was restored based on the evidence from historic photographs, plans from the National Archives and on-site evidence. The upper story now houses the main administrative office for the Fort Adams Trust.

Content Block 1

A more commonplace layout might feature an obvious, primary block of page content, but also feature a sidebar element that is unique to the current page: maybe a quote from a customer about a specific product or service. The “Power Tips” article offered a method to associate sidebar elements with multiple pages using custom fields and page IDs (tip #6). That approach isn’t very effective or efficient for designs with a 1:1 relationship between sidebars and pages (where each page has a unique sidebar element).

The screenshot shows the WordPress 'Edit Page' screen. At the top left is a 'Edit Page' button. Below it is a title 'Why Renewable Energy?' and a permalink. A toolbar above the editor includes 'Upload/Insert' and 'Visual/HTML' buttons. The main editor area contains a bulleted list: 'Wind Power Fact Sheet' and 'Solar Power Fact Sheet'. To the right of the editor is a sidebar with a navigation menu:

- > [Procurement](#)
- > [Program Management](#)
- > [Case Studies](#)
- > [Why Renewable Energy?](#)
- > [Blog](#)

Below the menu is a section titled 'WHY RENEWABLE ENERGY?' with two links:

- > [Wind Power Fact Sheet](#)
- > [Solar Power Fact Sheet](#)

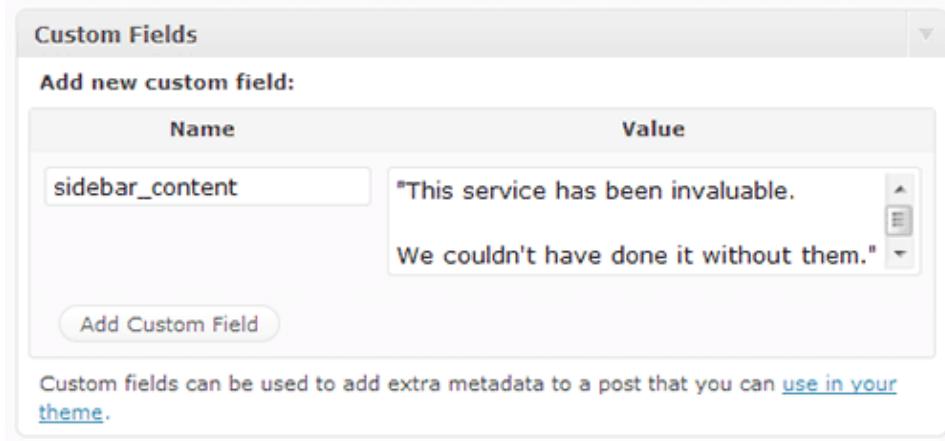
Yes, the developer could add table buttons to the WordPress editor, and let content authors fend for themselves: a solution prone to problematic layouts and bad output relied upon far too often. Here are a few simple options that keep layout in the hands of the template developer while making content management easier and problem-free.

SHORT, SIMPLE, AND HTML FREE? NO WORRIES.

Before we delve into solutions that assume a need for HTML formatting in this second content block, let's review a more basic solution. If the second column does not need to be formatted – or maybe should not be formatted by the editor for design reasons – then a simple custom field will do the trick. In the case of a simple sidebar element, like a customer quote, this may be just the trick.

There are already great tutorials and useful custom fields hacks that walk through the WordPress custom fields feature, so if you are not familiar with the basic idea behind custom fields, start there. Let's go ahead and create a custom field named "sidebar_content" (also known as the "key"), and put some simple content in there. Just to shake things up, let's assume we do need a very basic HTML feature for our content authors, who know nothing about HTML: line and paragraph breaks. Let's also assume that we want to

format this sidebar content on the front end with some of the basic automatic niceties we get when we output post content, like curly quotation marks.



The screenshot shows the 'Custom Fields' screen in the WordPress admin. A single custom field is listed with the name 'sidebar_content' and the value containing two lines of text: '\"This service has been invaluable.' and 'We couldn't have done it without them.''. There is also a link to 'use in your theme'.

Here's how we can output this in any template file, using [the "the_content"](#) filter to apply the WordPress content filter to our custom field. That filter converts single line breaks to break tags, double line breaks to paragraphing tags, and even transforms simple quotation marks to curly quotes!

```
$sidebar_content = get_post_meta($post->ID, "sidebar_content", true);  
  
if ($sidebar_content) {  
    echo '<div id="sidebar_content">';  
    echo apply_filters("the_content", $sidebar_content);  
    echo '</div>';  
}
```

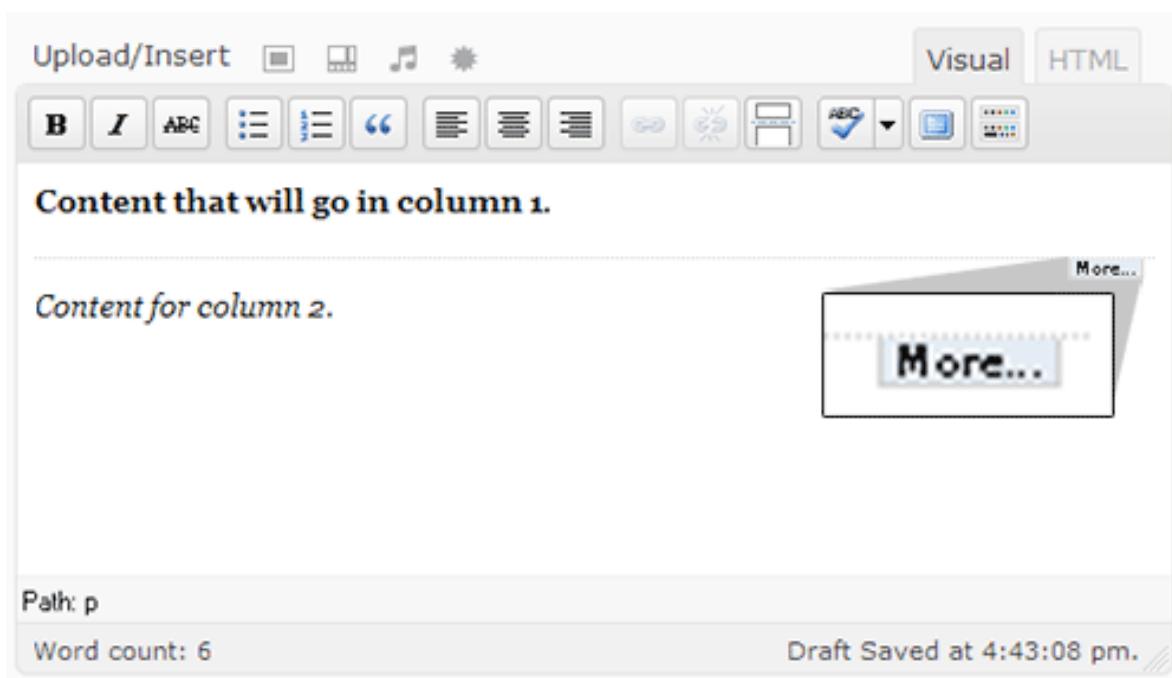
Of course, we can make this even more intuitive for the content authors by creating a new meta field box for sidebar content instead of relying on the generic "custom fields" box... which will be covered later in this article!

USING THE MORE TAG FOR... MORE

The WordPress editor has a button “more tag” button that is primarily intended to separate “above the fold” content from “below the fold” content. If you are not already familiar with the “more” divider, [read up on that first](#).

If the pages or posts that need a two column layouts also rely on traditional more separation, this tip will most likely not be effective, unless one of the columns is also the intended “above the fold” content. However, most instances where a two column layout is desirable don’t overlap with a traditional above / below the fold need. It is fairly rare, for instance, for pages (vs. posts) to actually make any use of the more tag. So let’s start taking advantage of that feature!

The basic idea is that content above the more divider will represent one block of HTML content, while content below the divider will represent a second block (be it a sidebar element or column).



Here is how to retrieve content above and below the more divider as separate blocks of HTML content in the corresponding page template file.

```
global $more;

$more = 0;
echo '<div id="column_one">';
the_content('');
echo '</div>';

$more = 1;
echo '<div id="column_two">';
the_content('',true);
echo '</div>';
```

The global “more” variable lets WordPress know whether or not the content is being rendered in an “above the fold” (or “teaser”) only view. By passing an empty string to [“the_content”](#), we prevent a “read more” link from showing up below the HTML content. And, for column two, we pass a second parameter to “the_content” – true – which instructs WordPress to output the content without the teaser.

If the intent is to output the second block of content outside of the loop in another template element, such as a sidebar, this approach is a bit trickier. One option would be to store the second block of content in a uniquely named variable, declare it as a global variable in the sidebar, and – if there is any content inside the variable – output a new block. An alternative could involve checking which page template is in use with the [“is_page_template” function](#), and, if the two column template is in use, calling “the_content” with the second parameter set to true, as in the example above.

THE PLUG-IN SOLUTION: ADDING A SECOND HTML CONTENT BLOCK TO THE EDITOR

The ideal solution, of course, might be a second HTML editor field on the WordPress page or post editor. Unfortunately, no such plug-in existed... until recently! While writing this article, we decided it was time such a solution did

exist, and so the author of this article is happy to present a free, open source plug-in that combines some savvy understanding of how TinyMCE works (hint: it's as simple as a class name) with the custom meta box tutorial covered later in this article, and a little bit of extra customization and polish thrown into the mix.

[Secondary HTML Content](#) adds a second HTML editor to pages, posts, or both (customizable with a simple settings panel). You can output the content in a sidebar with an included widget, or integrate it more tightly with the template by using “the_content_2” and “get_the_content_2” functions.

The screenshot shows the WordPress admin interface for editing a page titled "Test page". The main content area contains the text "Main content area." and a secondary content block with the text "A secondary content block" and a bulleted list. The right sidebar displays publishing information: Status: Published, Visibility: Public, Published on: Oct 18, 2009 @ 3:16, and a "Update Page" button. Below the publishing info is an "Attributes" section with a "Parent" dropdown set to "Main Page (no parent)". The "Template" section indicates "Default Template". The "Order" section shows a value of 0. At the bottom of the screen, there are footer links for "Discussion" (Allow Comments, Allow Pings) and "Page Revisions".

Associating Pages with Post Content: Reloaded

The chapter on “[Power Tips](#)” covered the basic foundation for associating different WordPress pages with different post categories. The basic premise was that many sites require, effectively, different post “feeds” on different pages. For instance, there may be a company blog, but there may also be an independent news feed.

This continuation offers specific tips that extend the core concept introduced in part 1, making it easier to have multiple page / category associations, preventing entrance into the “real” category archive, and ensuring that individual post views retain a visual and architectural association with their parent “category page” layout.

Be sure to read part 1 before proceeding.

A REVIEW OF THE BASICS & THE TWO FUNDAMENTAL APPROACHES

At the heart of the category / page association (covered in part one) was:

- A matching of the “page slug” with the “category slug.”
- Using “[query_posts](#)” and the category parameter to exclude standalone page categories from the primary feed
- Using a dedicated page template with “query_posts” and the “category name” parameter to create a page featuring a feed for a single category.

The screenshot shows the 'Custom Structure' settings in WordPress. At the top, there's a radio button for 'Custom Structure' and a text input field containing the permalink structure '/%category%/%postname%/'. Below this is a section titled 'Establish permalink structure allowing posts to appear as if under a page'. The 'QUICK EDIT' form contains fields for Title ('Press Releases'), Slug ('press-releases'), Date ('Mar 27, 2009'), Password (''), Parent ('About Us'), Order ('0'), Template ('Default Template'), and Status ('Published'). There are also checkboxes for 'Allow Comments' and 'Allow Pings'. A 'Cancel' button and an 'Update Page' button are at the bottom. Below the quick edit form is a table with columns 'Name', 'Description', 'Slug', and 'Posts'. It lists two items: 'About Us — Press Releases' (slug: press-releases, posts: 0) and 'About Us' (slug: about-us, posts: 0). A note at the bottom says 'Establish matching category hierarchy and page hierarchy to single feed page'.

Before delving into the tips that extend those ideas, it is important to make a distinction between two common but fundamentally different use cases for page / category association. The more **typical use case, which the first part was tailored to, is a website that has a primary feed, like a blog, but also has one or two distinct feeds**, most often for a formal news or press feed.

The second use case is a bit more esoteric: there is no primary feed. The site has many pages, and many (but not all) of those top level pages are individual feeds of posts. The example, at the end of this power tip, [m62.net](#), is one such use case. Another common use case might be – again – a portfolio centric website.

Let's say we want to create "Joe's Portfolio", and Joe wants to feature 4 distinct areas of expertise. Each area of expertise should be a top level page, say, [joes-portfolio.com/web-design](#), [joes-portfolio.com/graphic-design](#), etc. Joe wants to have a little write-up about each service area at the top of the page, followed by a feed of case studies. Why a feed instead of sub-pages? Maybe Joe wants prospects to be able to subscribe to an RSS feed for each area of expertise; maybe he wants to easily cross-tag case studies based on industry; maybe he plans to update frequently and

doesn't want a huge page sitemap or wants visitors to page through a date-organized collection of case studies. There are many reasons to use posts instead of pages.

The following tips provide solutions for both use cases.

AUTOMATICALLY DETERMINING THE PAGE / CATEGORY ASSOCIATION

Part one suggested that a unique page template be created for any page associated with a category. That page template would then query for posts using a hardcoded category name or category ID. If there are only one or two standalone "category pages", this is an efficient and effective solution.

However, if there are many page / category associations, as in use case #2 (no primary feed), the process of manually creating page templates for each association is tedious to build and maintain, and not realistic if content editors who don't program need to be able to create more page / category associations on demand.

An alternative would be to [create a generic page template](#), let's say "template-category-connector.php", that is assigned to all pages associated with a category, and automatically determines the right category to query.

The following code performs the matching and executes the post query. The magic happens by taking advantage of our matching page and category slugs. Once again, if the website does not use permalinks, an alternative approach will be required (one permalink-free alternative could involve a custom field with the associated category ID).

```
$cat = get_category_by_slug($post->post_name);  
query_posts('cat='.$cat->term_id);
```

That's all there is to it... just proceed on with the [post loop](#) to output the applicable category's posts. Note that the template should probably check

for an actual return value from line 1, and output a graceful error in the event there is no match.

HANDLING ENTRY INTO THE “REAL” CATEGORY ARCHIVE

Now that there is a dedicated page layout that handles the category feed, we will want to be make certain that the visitor doesn’t land on WordPress’ default category “archive” view. For instance, when using permalinks with the default “category base” value, the archive view for a category with a top level category assigned a “web-design” slug would be: mysiteurl.com/category/web-design. However, the intent is for visitors to view this category at our top level page: mysiteurl.com/web-design.

By combining the WordPress category template file with some smart redirects, we can prevent entry into the default category archive. Out of the box, the [WordPress template system](#) allows developers to create global category archive templates as well as templates for individual category archives.

If we are in use case #1 – a site with a traditional blog feed and a standalone news feed on a “press releases” page – we will want to use the latter solution. Let’s say, as in part one, the category ID for “press releases” is 5. We create a template file in our theme folder named category-5.php. Under use case #2 (no primary feed), we will want to redirect all category archive traffic, in which case we need to work with the category.php template file.

A few lines of code in either template file will redirect visitors to the right place. We’ll also pass [HTTP error / redirect code “301”](#) – which will tell search engines to permanently redirect their link to the right location. Note that this particular code assumes we are using a permalink configuration. Line 2 can be modified to accomodate that situation.

```
| $destination = get_bloginfo('url');
```

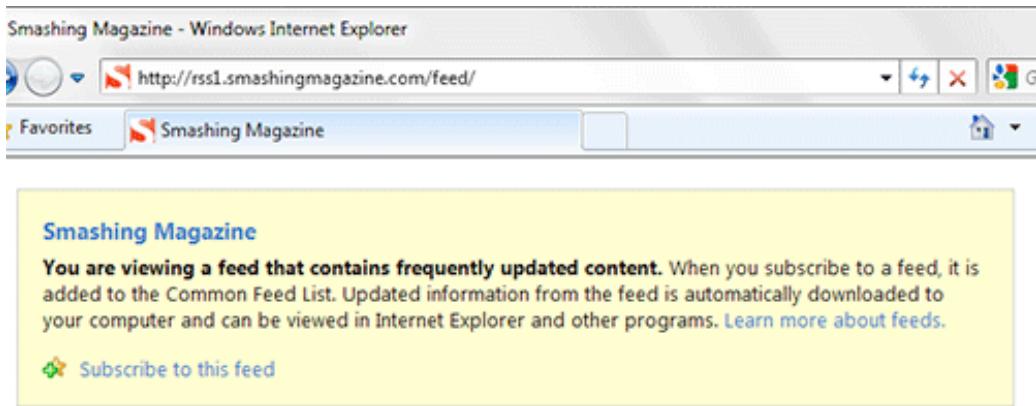
```
$destination .=  
str_replace('/'.get_option('category_base'). '/', '/',  
$_SERVER['REQUEST_URI']);  
wp_redirect($destination, 301);
```

In effect, that code removes the category base (“/category” by default) from the overall relative URL, and [safely redirects](#) the visitor to the page with the matching slug. Of course, if the site falls under use case #1 (one or two stand alone feeds), the line three could dropped into a specific category template (i.e. category-5.php) with a hardcoded absolute URL for the redirect destination.

HIDING STANDALONE CATEGORIES FROM THE CATEGORY LIST & PRIMARY SITE FEED

In the first use case (only isolating one or two categories from a primary feed), **it may be necessary to prevent isolated categories or the posts within those categories from appearing in some common theme elements that would traditionally include them.**

Consider the example from part one: a site with a traditional blog and a standalone press release feed. Assume the owners of the site want the RSS feed for the blog to be persistently available throughout the site (typically manifesting itself as an RSS icon in the browser location bar), but don’t want the press release items included in that primary feed. By default, the WordPress primary feed is available at “/feed”, and includes all published posts, regardless of category or any other post property.



To exclude categories from the primary RSS feed, we need to [filter](#) the WordPress function that retrieves the posts. Let's again assume that the category ID for Press Releases is 5. The following code should be placed in the template's "[functions.php](#)" file.

```
add_filter('pre_get_posts', 'exclude_press');

function exclude_press($query) {
    if($query->is_feed && !$query->is_category) $query-
>set('cat', '-5');
}
```

To summarize, we use the "pre_get_posts" filter to modify the post query before it executes. Within a new filter – named "exclude_press" – a conditional confirms that the post query is for a feed, and that the query is not for an individual category. If the check pans out, the query is modified to exclude category 5 before execution.

The notion of globally filtering the post query may have broader implications depending on the site's unique requirements. With some smart conditional checking, the filter could be extended to prevent the category from appearing anywhere except within the category or isolated post view. But be careful when extending the filter, and be sure to consider all possible views, including administrative views!

The category list is another frequently used site element that isolated categories should, in most cases, be excluded from. If the template [calls the category list in only one or two places by code](#) (as opposed to using the categories widget), excluding categories from the list is straight forward. However, if the categories widget is in use, or the category list is used throughout the template, an alternative approach is required. Enter the “list_terms_exclusions” filter. Again, the following code should be placed in the “functions.php” template file.

```
add_filter('list_terms_exclusions', 'filter_press');

function filter_press($exclusions) {
    $exclusions .= " AND t.term_id != 5 ";
    return $exclusions;
}
```

The return value of a “terms exclusions” filter is tacked onto the “where” clause in the SQL query that retrieves the terms. Without digging too deep here, the reason for discussing “terms” as opposed to, say, “categories” is because WordPress abstracts a variety of different taxonomies (link categories, post categories, tags, custom taxonomies, etc) into a unified database model that handles all taxonomies. Calls to “get categories”, “get tags”, and so forth, are all referring back to general “terms” behind the scenes. Ever wonder why category, tag, and other IDs tend to jump around? They are all being added to the same table. Assuming a fairly clean install, try adding a new post category, and note the ID. Then add a tag, and note its ID... one greater than the new post category.

term_taxonomy_id	term_id	taxonomy	description	parent	count
1	1	category		0	3
2	2	link_category		0	7
3	3	category		0	2
4	4	category		0	1
5	5	post_tag		0	0
6	6	post_tag		0	0

RETAINING THE PAGE LAYOUT FOR POST VIEWS WITHIN A CATEGORY PAGE

One of the most common challenges to tackle with page / category association is retaining a sense that the visitor is still within the “category page” hierarchy – and not a global feed hierarchy – when a visitor is reading an individual post. Part one hinted at this challenge under “The devil is in the details,” and started to suggest a path that incorporated using the “in_category” function. **We will explain how to use “in_category” within templates, as well as how to trick functions that reference the original query object into thinking that they are “within” the category page.**

Let’s start with case #1, and building on the example in the first article, assume we only need to contend with one isolated feed, “Press Releases” (category ID 5).

Say the theme has a sidebar template that lists post categories when rendering the blog part of the site, and when rendering a standalone page, shows a page list instead. Here’s an extremely simplified version of what that might look inside the sidebar template file.

```
if (is_page())
{
    wp_list_pages();
}
else
{
    wp_list_categories();
}
```

Of course, there may be alternative widget sets for pages or posts, and there is likely to be more than just one element in the sidebar. But the concept should hold. Now going back to the example, the theme should render posts in category 5 (Press Releases) as if the visitor were on a page (not the blog). Leveraging the “in_category” check, the code above would now like the following:

```
if (is_page() || in_category(5))
{
    wp_list_pages();
}
else
{
    wp_list_categories();
}
```

Note that if there are multiple categories whose posts should resemble page output, the “in_category” function should be passed an array of IDs, like so:

```
in_category(array(5, 7));
```

The need for a “in category” check is probably moot in case #2 (multiple page/category associations, without a primary feed): the template is probably structured to output the same elements on pages and posts from

the get go. In other words, everything is handled as if it is a page since there is no primary feed. However, the following tip – that dynamically looks up the faux parent page ID (the page associated with the category) – is necessary for the next part of this tip. Just amend the code to check if “faux_parent_page” has a valid value: if it does, then the post is inside an isolated category associated with a page.

Once again, this approach to dynamically seeking the faux parent page (the category page) depends on taking advantage of the matching permalink structure between post categories and pages that is at the heart of this association. If the site is unable to use permalinks, a more complex alternative look up of the faux parent page will be necessary.

```
foreach(get_the_category() as $category) {  
    $faux_parent_path = '/' .get_category_parents($category, FALSE, '/',  
TRUE);  
}  
$faux_parent_page = get_page_by_path($faux_parent_path)->ID;
```

Now that we have the ID of the category’s associated page, we can trick “black box” theme elements that determine page or post properties on their own (by referencing the post query) into thinking they are actually working with the category page.

The most common use case is page navigation. Whether its breadcrumbs, a top level page menu that should retain “current” (on) states, or a side navigation menu that should display the current section, there are many “black box” navigation functions that need to be tricked into rendering themselves as if on the category page.

Let’s use a simple top level page list, which should maintain proper “current_page”, “current_page_parent” (and so on) classes when on a post under a category page. Here’s what that simple function might look like before our changes:

```
wp_list_pages('depth=1');
```

Of course, posts do not normally have parent pages, so there will be no “current” classes assigned to that output when reading a post. Here is how to trick that function into thinking it is rendering the navigation for the “parent” category page.

```
//retrieve faux parent page dynamically... can skip and hard code in  
case 1  
  
foreach(get_the_category() as $category) {  
    $faux_parent_path = '/'.get_category_parents($category, FALSE,  
    '/', TRUE);  
}  
  
$faux_parent_page = get_page_by_path($faux_parent_path)->ID;  
  
//reset the post query as if on the faux parent page  
query_posts('page_id='.$faux_parent_page);  
  
//execute our "faked out" function  
wp_list_pages('depth=1');  
  
//reset the query back to the initial state  
wp_reset_query();
```

If there are multiple elements that need be “tricked,” a best practice would be to put the “faux parent page” retriever at the top of the template, and declare it a global in any template files that need it. This would avoid repeated look ups of the faux parent page.

AN EXAMPLE: SEEING IT ALL PUT TOGETHER

A great example of a WordPress-powered CMS that pushes use case #2 to its limits can be seen at the home of m62 visual communications, at <http://www.m62.net>.

Pharmaceutical Templates

Download free pharmaceutical PowerPoint templates. Our designers have developed small presentation toolkits for you to use in your own pharmaceutical PowerPoint presentations.

These templates are suitable for use when presenting on pharmaceutical issues. Templates in the medical section may also be of use.

If you want m62 to customise any of these PowerPoint templates, want your existing template refreshed, or would like a bespoke template designed, please contact us.



Useful Share: 

Presentations letting you down? Want a powerful presentation, that you actually **enjoy** presenting? Then [contact m62](#) now.

[Next Steps](#)  [Subscribe to article62](#)  [View Online Demonstration](#)  [Submit a Slide](#)  [Call Me](#)

Latest in Pharmaceutical Templates

[Get Updates:](#) 



Glass Bottles Template

15th January - Free glass bottles PowerPoint template for use in your pharmaceuticals



Atoms Template

15th January - This free PowerPoint template shows the model of a chain of atoms in the background. The rest of



Test Tube Template

10th January - Free PowerPoint template - test tubes in blue and green for use in pharmaceutical presentations.



Atomic Structure

5th September - This free PowerPoint template contains a very subtle pattern showing atomic structure in the

PowerPoint Templates

[Business Templates](#)

[Finance Templates](#)

[Technology Templates](#)

Pharmaceutical Templates

[Medical Templates](#)

[Education Templates](#)

[Transport and Logistics Templates](#)

[Travel Templates](#)

[Government Templates](#)

[Retail Templates](#)

[Industrial Templates](#)

[Christian Templates](#)

[Sport and Leisure Templates](#)

[Abstract Templates](#)

Contact m62

m62 Services

[m62 impress](#)

[m62 engage](#)

[m62 recall](#)

[m62 train](#)

 For the first time I didn't have to read the bullet, which

All of the navigation items across the top (Presentation Theory, PowerPoint Slides, etc) are pages associated with post categories. The sub-navigation on the right contains sub-pages that are also associated with sub-categories. For example, in the screenshot above ([available here](#)), the visitor is on the “Pharmaceutical Templates” page (faux category), which is a child of the “PowerPoint Templates” page (also a faux category). The content starting with “Download free” (below the page title) is the content from the “Pharmaceutical Templates” page. The posts below the “Next Steps” bar, titled “Latest in Pharmaceutical Templates”, are the posts inside that category. The applicable related category is automatically discovered by the WordPress template, populating the category name “Latest in X” and recent posts. Now let’s look at one of the posts inside that category.

Atoms Template

© Tuesday, January 13th, 2009 2 comments

This free PowerPoint template shows the model of a chain of atoms in the background. The rest of the template is clean and business-like.

This template may be useful for those presenting about particle Chemistry, biochemistry, or life sciences.

Download this PowerPoint template for use in your own presentations. This file is in .pot format, and may take a short while to download.



Useful Share: 



PowerPoint Templates

Business Templates

Finance Templates

Technology Templates

Pharmaceutical Templates

Medical Templates

Education Templates

Transport and Logistics
Templates

Travel Templates

Government Templates

Retail Templates

Industrial Templates

Christian Templates

Sport and Leisure Templates

Abstract Templates

Contact m62

m62 Services

m62 impress

m62 engage

m62 recall

m62 train

Using the tips outlined above, the individual post retains the feel of being within the “Pharmaceutical Templates” page, right down to the breadcrumb navigation and “current” states in the navigation.

But not only does m62.net use category / page associations for most top and second level navigation items, it actually extends the concept to tags. The 5 “tabs” on the top right actually represent post tags, and each has a “tag page.”

Advanced Power Tips for WordPress Template Developers: Reloaded

Jacob Goldman

In the previous chapter we covered multiple column content techniques and associating pages with post content and also discussed how to use the “More”-tag, hide standalone categories from the category list and retain the page layout for post views within a category page. This chapter presents techniques on how to customize basic content administration and add features to the post and page editor in WordPress.

Customizing Basic Content Administration

Many template developers have learned the art of making beautiful, highly customized front end templates for WordPress. But the real wizards know how to tailor the WordPress administrative console to create a tailored, customized experience for content managers.

CUSTOMIZING THE DASHBOARD WIDGETS

The dashboard is the first screen presented to registered visitors when they visit WordPress administration (/wp-admin). **Tailoring the dashboard to a client can be the difference between a great first impression and a confused one**, particularly if the theme customizes the administrative experience.

The dashboard is comprised of a number of widgets that can be repositioned and toggled using the “screen options” tab. WordPress has a hook – wp_dashboard_setup – that can be used to customize the

dashboard widgets, as well as a function – `wp_add_dashboard_widget` – that allows developers to easily add new widgets.

The WordPress codex [documents the process of adding and removing widgets](#).

Here is a practical use case based on that documentation: let's remove all of the default widgets that don't pertain to managing a typical site, and add one simple widget that welcomes the administrator and reminds them how to contact the developer for support.

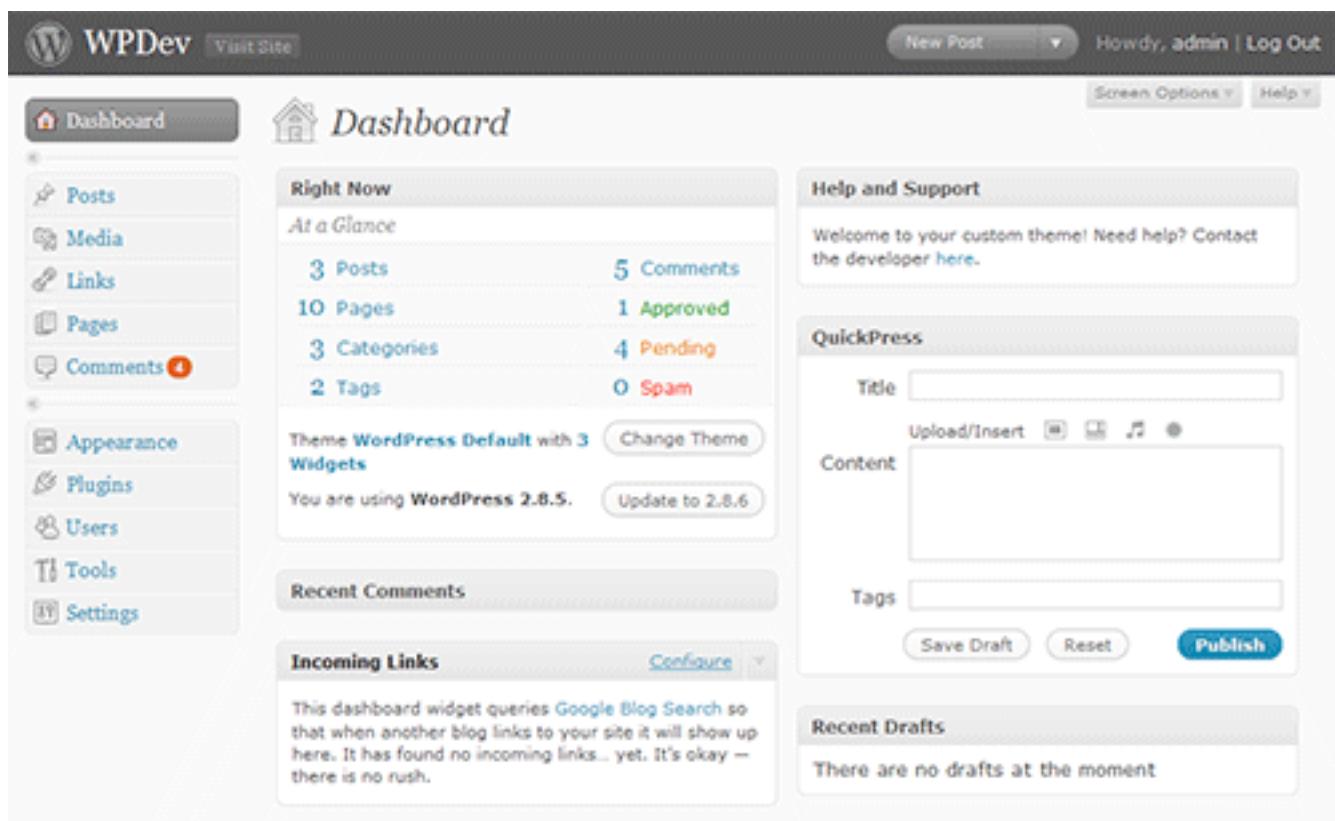
```
add_action('wp_dashboard_setup', 'my_custom_dashboard_widgets');

function my_custom_dashboard_widgets() {
    global $wp_meta_boxes;

    unset($wp_meta_boxes['dashboard']['normal']['core']
        ['dashboard_plugins']);
    unset($wp_meta_boxes['dashboard']['side']['core']
        ['dashboard_primary']);
    unset($wp_meta_boxes['dashboard']['side']['core']
        ['dashboard_secondary']);

    wp_add_dashboard_widget('custom_help_widget', 'Help and Support',
        'custom_dashboard_help');
}

function custom_dashboard_help() {
    echo '<p>Welcome to your custom theme! Need help? Contact the
developer <a href="http://mytemplates.com ">here</a>. <a
href=""http://mytemplates.com ""></a></p>';
}
```



CUSTOMIZING THE CONTEXTUAL HELP DROPODOWN

Throughout its administrative panel, WordPress has a small “Help” tab just below the administrative header. Clicking this tab rolls down contextual help for the current administrative page.

The screenshot shows the WordPress admin dashboard. On the left, there's a sidebar with links like Posts, Media, Links, Pages, Comments (with 4 notifications), Appearance, Plugins, Users, Tools, and Settings. The main content area has a heading 'Get help with "Dashboard"' followed by text about arranging modules in columns and moving them. Below that is another section about expanding and collapsing modules. There's also a 'Help' link at the bottom right. At the very bottom of the page, there's a footer bar with 'Right Now' and 'At a Glance' buttons, and a 'Help and Support' section.

If your theme has some special functionality that might not be intuitive, it's a good practice to add some additional contextual help. For example purposes, let's assume that the theme has been customized to use the “more divider” to separate content into two columns, as described in the first tip. That's probably not an obvious feature for your average content editor. To accomplish this, hook the contextual help text when on the “new page” and “edit page” administrative pages, and add a note about that feature.

```
//hook loading of new page and edit page screens
add_action('load-page-new.php','add_custom_help_page');
add_action('load-page.php','add_custom_help_page');

function add_custom_help_page() {
    //the contextual help filter
    add_filter('contextual_help','custom_page_help');
}

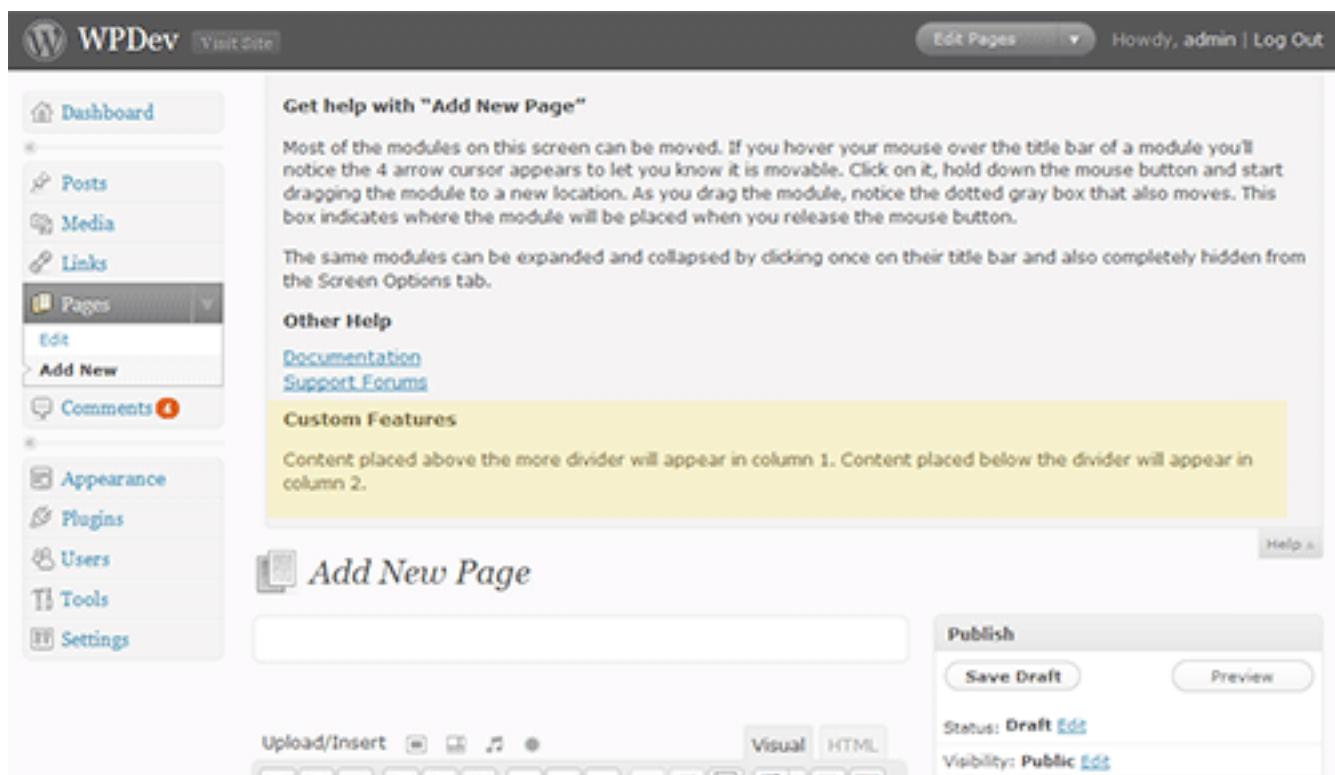
function custom_page_help($help) {
```

```

//keep the existing help copy
echo $help;

//add some new copy
echo "<h5>Custom Features</h5>";
echo "<p>Content placed above the more divider will appear in
column 1. Content placed below the divider will appear in column
2.</p>";
}

```



DROPPING IN YOUR OWN LOGO

Providing the client some administrative branding can be quick and easy. Here's **how to replace the default WordPress "W" logo in the administrative header with a custom alternative.**

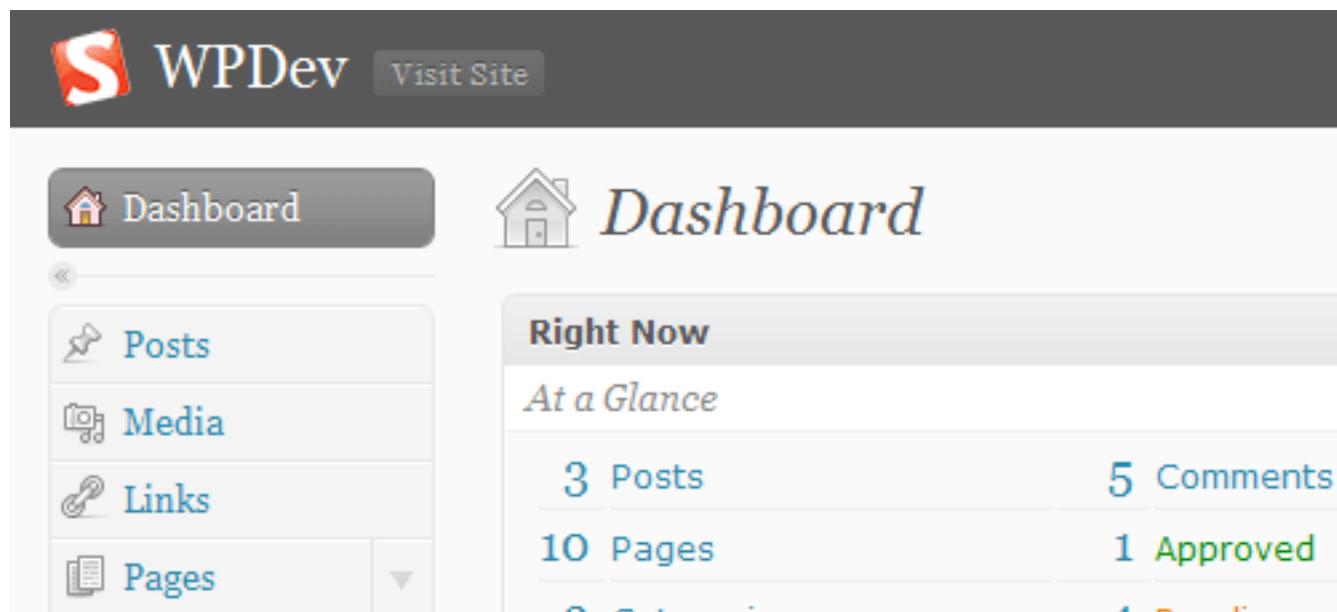
First, create an image that fits the allocated space. As of WordPress 2.8, the logo is a 30 pixels wide and 31 pixels tall transparent GIF. When using a

transparent GIF or 8-bit PNG, ensure that the image matte matches the header background color: hex value 464646.

A logo named “custom_logo.gif” inside the template directory’s image subfolder can substitute the default WordPress logo with the following code inside the theme’s “functions.php” file.

```
//hook the administrative header output
add_action('admin_head', 'my_custom_logo');

function my_custom_logo() {
    echo '
        <style type="text/css">
            #header-logo { background-image:
url('.get_bloginfo('template_directory').'/images/custom-
logo.gif) !important; }
        </style>
    ';
}
```



HIDING FIELDS BASED ON USER ROLE

Basic contributors might be confused or distracted by some of the boxes that surround the page or post editor, particularly if there are a handful of plug-ins that have added their own meta boxes. Alternatively, the content editor might simply want to keep author and contributor hands off of some special fields or features.

Let's say the content editor wants to keep authors and contributors way from the "custom fields" box. We can use the "remove_meta_box" function – regardless of user role – to remove that from all post editing screens like so:

```
//hook the admin init
add_action('admin_init', 'customize_meta_boxes');

function customize_meta_boxes() {
    remove_meta_box('postcustom', 'post', 'normal');
}
```

The "remove_meta_box" function takes three parameters. The first is the ID of the box. The easiest way to discover the ID of the meta box is to look for the ID attribute of the corresponding DIV "postbox" in the source code. The second parameter determines which the context the function applies to: page, post, or link. Finally, the context attribute determines the position within its context: normal, or advanced (in most cases, just setting this to "normal" will work fine).

The screenshot shows the 'Custom Fields' screen in the WordPress admin area. On the right, there's a sidebar with 'Template' and 'Default Template' sections, and an 'Order' section with a value of '0'. Below that is a note about pages having an order. At the bottom of the sidebar is a link to 'View in browser'.

The main area has a heading 'Add new custom field:' and a table with columns 'Name' and 'Value'. A dropdown menu is open under 'Name' with the option '- Select -'. Below the table is a text input field with placeholder 'Enter new'. At the bottom left is a button 'Add Custom Field'. A note at the bottom says: 'Custom fields can be used to add extra metadata to a post that you can use in your theme.'

Below the admin area is a browser developer tools interface. The 'HTML' tab is selected in the top bar. The left panel shows the page structure with an expanded node for the custom fields meta box, which is highlighted with a blue box. The right panel shows the CSS styles applied to this element, with the class '.widget, .postbox' defined as having a background color of #FFFFFF. A tooltip on the right side of the interface reads: 'The ID of the custom fields meta box on pages is "pagecustomdiv".'

The next step is to extend the “customize_meta_boxes” function so that the “custom fields” box – ID “postcustom” – is only hidden from users with author role or lower. We’ll use [get_currentuserinfo](#) to retrieve the user level. According to the WordPress codex, [authors represent level 2](#).

```
//hook the admin init
add_action('admin_init', 'customize_meta_boxes');

function customize_meta_boxes() {
    //retrieve current user info
    global $current_user;
    get_currentuserinfo();

    //if current user level is less than 3, remove the postcustom meta
    //box
    if ($current_user->user_level < 3)
        remove_meta_box('postcustom', 'post', 'normal');
}

}
```

Adding Features to the Post & Page Editor

WordPress provides a “custom fields” box that makes it quick and easy to start adding new metadata to your pages and posts. For a tech-savvy client or low budget customization, this is a great, inexpensive method to start [adding some unique fields for a custom implementation](#).

Custom Fields

Add new custom field:

Name	Value
sidebar_content	"This service has been invaluable. We couldn't have done it without them."

Add Custom Field

Custom fields can be used to add extra metadata to a post that you can [use in your theme](#).

But there are plenty of times when something more specialized than a generic “custom fields” box may be appropriate. A less savvy client may be confused by the generic fields that lack any documentation. A checkbox for a Boolean field may be more intuitive for a client than instructions to choose the custom field name from a drop down and type in “1” or “true” under the value. column Or maybe the field should be limited, in select box like fashion, to a few different choices.

The WordPress API can be used to add custom meta boxes to pages and / or posts. And with WordPress 2.8, adding new, tag-like taxonomies is a cinch.

ADDING A CUSTOM META BOX

Let’s say a hyper-local journalist has hired us to build a news blog that covers politics in New York City. The journalist has a few writers on her team, none of whom are particularly tech-savvy, but they will all be set up as authors and posting their reports directly in WordPress. Our imaginary client wants each article associated with a single borough, in addition to a “city-

wide” option. Articles will never be associated with 2 boroughs, and the staff is prone to typos.

A developer accustomed to basic WordPress administrative customization would probably go to “categories” first. Make a “city-wide” category, with sub-categories for each borough. However, categories are multi-select, and there’s no obvious way to prevent authors from selecting several. Furthermore, the client wants the borough named at the beginning of the article, and if categories are used in other ways (like news topics), extracting the borough name would be a bit tricky.

So how about a “custom field” for “borough”? The authors never remember to look in that generic custom fields box, and in their rush to meet deadlines, occasionally spell the borough wrong, breaking the “filter by borough” feature on the front end.

The right answer is a new custom “meta box,” with a drop down “Borough” field. The WordPress Codex [documents the “add_meta_box” function](#) in detail.

Let’s apply the code discussed in the codex to this use case, assuming we want the “Borough” field to only appear on posts (not pages), and be shown on the top-right of the post editor page.

```
/* Use the admin_menu action to define the custom boxes */
add_action('admin_menu', 'nyc_boroughs_add_custom_box');

/* Adds a custom section to the "side" of the post edit screen */
function nyc_boroughs_add_custom_box() {
    add_meta_box('nyc_boroughs', 'Applicable Borough',
    'nyc_boroughs_custom_box', 'post', 'side', 'high');
}

/* prints the custom field in the new custom post section */
function nyc_boroughs_custom_box() {
```

```
//get post meta value
global $post;
$custom = get_post_meta($post->ID, '_nyc_borough', true);

// use nonce for verification
echo '<input type="hidden" name="nyc_boroughs_noncename"
id="nyc_boroughs_noncename" value="'.wp_create_nonce('nyc-
boroughs').'" />';

// The actual fields for data entry
echo '<label for="nyc_borough">Borough</label>';
echo '<select name="nyc_borough" id="nyc_borough" size="1">';

//lets create an array of boroughs to loop through
$boroughs = array('Manhattan','Brooklyn','Queens','The
Bronx','Staten Island');

foreach ($boroughs as $borough) {
    echo '<option value="'. $borough .'">';
    if ($custom == $borough) echo ' selected="selected"';
    echo '>'. $borough . '</option>';
}

echo "</select>";
}

/* use save_post action to handle data entered */
add_action('save_post', 'nyc_boroughs_save_postdata');

/* when the post is saved, save the custom data */
function nyc_boroughs_save_postdata($post_id) {
    // verify this with nonce because save_post can be triggered at
other times
}
```

```

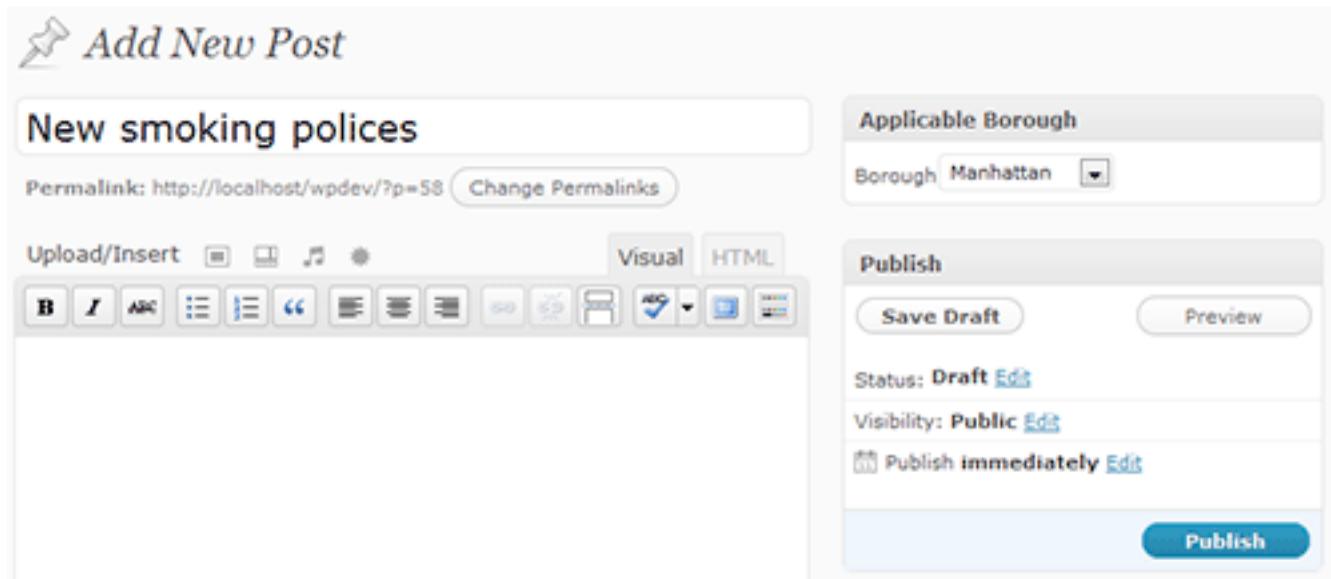
if (!wp_verify_nonce($_POST['nyc_boroughs_noncename'], 'nyc-boroughs')) return $post_id;

// do not save if this is an auto save routine
if (defined('DOING_AUTOSAVE') && DOING_AUTOSAVE) return $post_id;

$nyc_borough = $_POST['nyc_borough'];
update_post_meta($post_id, '_nyc_borough', $nyc_borough);
}

```

Take another look at the second to last line in that code block, where the post metadata is updated (update_post_meta will also add the meta if it does not exist). That function stores the field key and value (second and third parameters), assigned to the designated post (first parameter) in the same generic “way” that custom fields are stored. Notice that the field key name was prefaced with an underscore: “_nyc_borough”. **Meta fields with keys beginning with an underscore are not shown in the generic “custom fields” box.** All other meta fields are shown in that box.



We can use this field value in our template just as we would embed generic custom fields.

```
echo get_post_meta($post->ID, '_nyc_borough', true);
```

If we want to do a post query that only includes posts in the “Queens” borough, we can execute the query with the following code:

```
query_posts('meta_key=_nyc_borough&meta_value=Queens');
```

ADDING CUSTOM TAXONOMIES

A taxonomy, generically defined, is a “classification.” Post tags and categories in WordPress are both types of taxonomies, one of which – categories – has a “hierarchical” proprietary: categories can have child and parent categories. The ability to define new taxonomies has actually been around in some basic form since WordPress 2.3 – but WordPress 2.8 ups the ante, making it incredibly easy for template developers to add and manage tag-like taxonomies.

At the core API level, taxonomies may be hierarchical (or not, a la “tags”), associated with pages or posts, and have a few other more esoteric properties related to allowing post queries and permalink structures. The potential for custom taxonomies is considerable – posts could easily have two types of categories, pages could have multiple tags, and sites could have multiple tag clouds based on groupings more specific than a generic “tag.”

While the architecture for all of this is all there, the real magic of custom taxonomies – introduced in 2.8 – has only been enabled for posts and non-hierarchical types. But if those qualifications aren’t a show stopper, a developer can get a lot of value out of just a few lines of code: a new tag-like meta box added to posts, a new “posts menu” option for managing those values, and the ability to easily output clouds, filter by taxonomies, design taxonomy templates, and do just about anything one could do with generic “tags” on the front end.

The WordPress Codex outlines the [“register_taxonomy” function](#).

Let's go back to that hyper-local New York City politics blog. Say the editor wants authors to be able to "tag" articles with a distinct "people" taxonomy, but still wants to retain generic tagging. The new "people" taxonomy will highlight the names of political leaders mentioned in articles. On the front end the editor envisions a "tag cloud" that will help the most active politicians get recognized (for better or worse!). Clicking on a leader's name in the cloud should bring up a list of articles "tagged" with the given politician.

The following few lines of code will add the new "people" meta box to posts and add a new option to the "posts" menu where the taxonomy values can be managed.

```
//hook into the init action to add the taxonomy
add_action( 'init', 'create_nyc_people_taxonomy');

//create nyc people taxonomy
function create_nyc_people_taxonomy() {
    register_taxonomy('people', 'post', array('hierarchical' => false,
    'label' => 'people'));
}
```

The screenshot shows the WordPress admin interface for editing a post. The title of the post is "New smoking policies". In the left sidebar, under the "Posts" section, the "people" taxonomy is highlighted. On the right, there's a "Borough" dropdown set to "The Bronx". Under "Publish", the status is "Draft" and visibility is "Public". Below that, there's a "Delete" button and a "Publish" button. At the bottom right, there's a "people" taxonomy sidebar with a tag cloud.

To output a cloud for this custom taxonomy highlighting the 40 most-tagged politicians, the generic “[wp_tag_cloud](#)” function can be used with a few parameters.

```
wp_tag_cloud(array('taxonomy' => 'people', 'number' => 40));
```

To list the highlighted leaders in a single post

```
echo get_the_term_list($post->ID, 'people', 'People: ', ', ',');
```

Clicking on a person’s name will automatically take the visitor to an archive for that taxonomy. Custom template files can also be built for the custom taxonomy. A “`taxonomy.php`” template file in the theme folder can be used for all custom taxonomies. A “`taxonomy-people.php`” template file could be used for the “`people`” taxonomy in the example. As with all archives, if no taxonomy-specific template files are available, WordPress will fall back to the generic “archive” and “index” template files.

Ten Things Every WordPress Plugin Developer Should Know

Dave Donaldson

Plugins are a major part of why WordPress powers millions of blogs and websites around the world. The ability to extend WordPress to meet just about any need is a powerful motivator for choosing WordPress over other alternatives. Having written several plugins myself, I've come to learn many (but certainly not all) of the ins-and-outs of **WordPress plugin development**, and this chapter is a culmination of the things I think every WordPress plugin developer should know. Oh, and keep in mind everything you see here is compatible with WordPress 3.0+.

Don't Develop Without Debugging

The first thing you should do when developing a WordPress plugin is to enable debugging, and I suggest leaving it on the entire time you're writing plugin code. When things go wrong, WordPress raises warnings and error messages, but if you can't see them then they might as well have not been raised at all.

Enabling debugging also turns on WordPress notices, which is important because that's how you'll know if you're using any deprecated functions. Deprecated functions may be removed from future versions of WordPress, and just about every WordPress release contains functions slated to die at a later date. If you see that you are using a deprecated function, it's best to find its replacement and use that instead.



HOW TO ENABLE DEBUGGING

By default, WordPress debugging is turned off, so to enable it, open **wp-config.php** (tip: make a backup copy of this file that you can revert to later if needed) in the root of your WordPress installation and look for this line:

```
define('WP_DEBUG', false);
```

Replace that line with the following:

```
// Turns WordPress debugging on  
define('WP_DEBUG', true);
```

```
// Tells WordPress to log everything to the /wp-content/debug.log  
file  
define('WP_DEBUG_LOG', true);  
  
// Doesn't force the PHP 'display_errors' variable to be on  
define('WP_DEBUG_DISPLAY', false);  
  
// Hides errors from being displayed on-screen  
@ini_set('display_errors', 0);
```

With those lines added to your wp-config.php file, debugging is fully enabled. Here's an example of a notice that got logged to /wp-content/debug.log for using a deprecated function:

*[15-Feb-2011 20:09:14] PHP Notice: get_usermeta is **deprecated** since version 3.0! Use get_user_meta() instead. in C:\Code\Plugins\wordpress\wp-includes\functions.php on line 3237*

With debugging enabled, **keep a close eye on /wp-content/debug.log** as you develop your plugin. Doing so will save you, your users, and other plugin developers a lot of headaches.

HOW TO LOG YOUR OWN DEBUG STATEMENTS

So what about logging your own debug statements? Well, the simplest way is to use echo and see the message on the page. It's the quick-and-dirty-hack way to debug, but everyone has done it one time or another. A better way would be to create a function that does this for you, and then you can see all of your own debug statements in the debug.log file with everything else.

Here's a function you can use; notice that it only logs the message if WP_DEBUG is enabled:

```
function log_me($message) {  
    if (WP_DEBUG === true) {  
        if (is_array($message) || is_object($message)) {  
            error_log(print_r($message, true));  
        } else {  
            error_log($message);  
        }  
    }  
}
```

And then you can call the `log_me` function like this:

```
log_me(array('This is a message' => 'for debugging purposes'));  
log_me('This is a message for debugging purposes');
```

USE THE BLACKBOX DEBUG BAR PLUGIN

I only recently discovered this plugin, but it's already been a huge help as I work on my own plugins. The BlackBox plugin adds a thin black bar to the top of any WordPress post or page, and provides quick access to errors, global variables, profile data, and SQL queries.



Clicking on the Globals tab in the bar shows all of the global variables and their values that were part of the request, essentially everything in the `$_GET`, `$_POST`, `$_COOKIE`, `$_SESSION`, and `$_SERVER` variables:

Globals **Profiler (681.32 ms)** **SQL (9 queries in 30.77 ms)** **Errors (14)**

```

$_GET = array (
);

$_POST = array (
);

$_COOKIE = array (
    'wordpress_b99b4c5fe30c4336581c6d8f180dc196' => 'dave|1297886588|653cab3',
    'wp-settings-1' => 'm9=c&m5=c&m6=o&m4=o&m8=c&m10=c&editor=html&m1=c&m2=c',
    'wp-settings-time-1' => '1297800053',
    'wordpress_logged_in_b99b4c5fe30c4336581c6d8f180dc196' => 'dave|1297886588|653cab3',
    'wordpress_test_cookie' => 'WP Cookie check',
    'WRUID' => '0',
);
$_SESSION = NULL;

$_SERVER = array (
    'SERVER_SOFTWARE' => 'Microsoft-IIS/7.5',
    'REQUEST_URI' => '/maxfoundryplugins/wordpress/wp-admin/plugins.php',
    'PROCESSOR_ARCHITECTURE' => 'AMD64',
    '_FCGI_X_PIPE_' => '\\\\.\\pipe\\IISFCGI-8c99a8c9-90f6-456b-9514-ca3a7db'
);

```

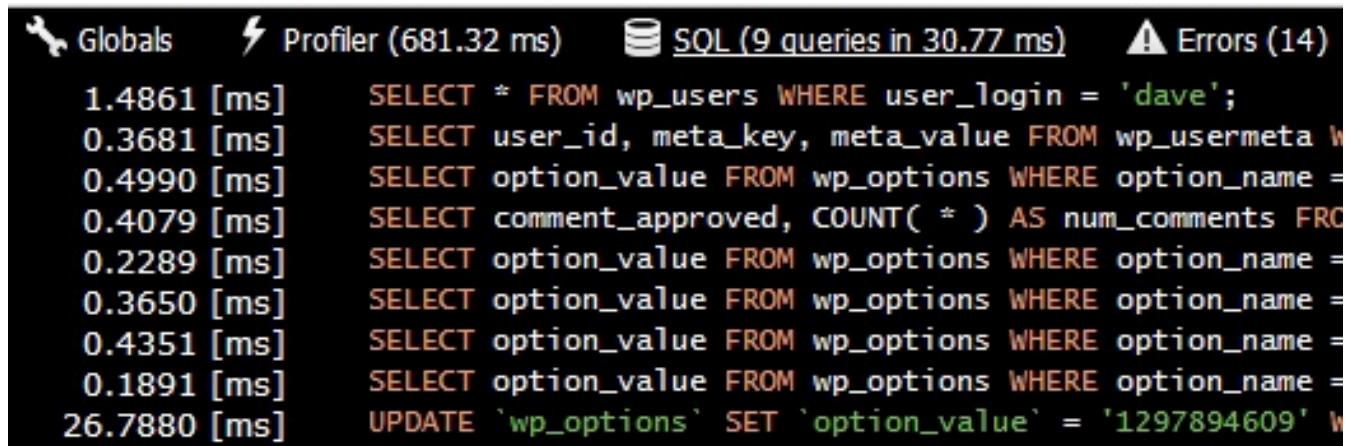
The next tab is the Profiler, which displays the time that passed since the profiler was started and the total memory WordPress was using when the checkpoint was reached:

Globals Profiler (681.32 ms) SQL (9 queries in 30.77 ms) Errors (14)		
Profiler Initiated	0.0000 ms	1804 kB
Profiler Noise	0.0639 ms	1805 kB
This is a checkpoint	6.0520 ms	1857 kB
Profiler Stopped	681.3169 ms	3150 kB

You can add your own checkpoints to the Profiler by putting this line of code anywhere in your plugin where you want to capture a measurement:

```
apply_filters('debug', 'This is a checkpoint');
```

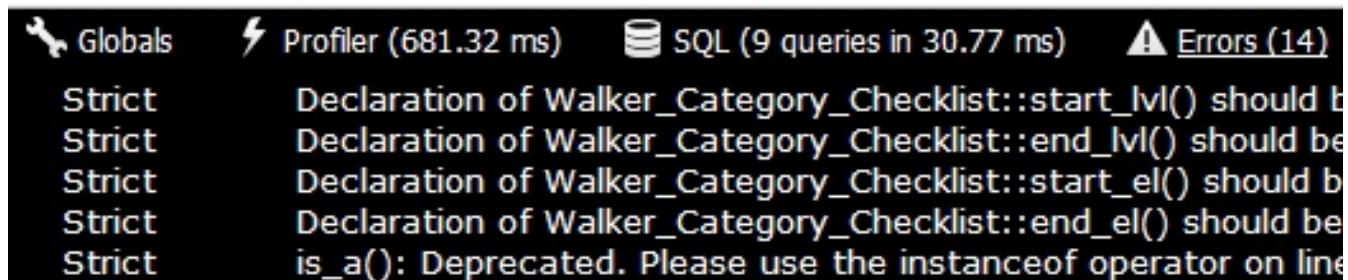
Perhaps the most valuable tab in the BlackBox plugin is the SQL tab, which shows you all of the database queries that executed as part of the request. Very useful for determining long-running database calls:



The screenshot shows the SQL tab of the BlackBox plugin. It displays a list of 9 database queries with their execution times in milliseconds. The queries are as follows:

- 1.4861 [ms] SELECT * FROM wp_users WHERE user_login = 'dave';
- 0.3681 [ms] SELECT user_id, meta_key, meta_value FROM wp_usermeta WHERE user_id = 1;
- 0.4990 [ms] SELECT option_value FROM wp_options WHERE option_name = 'comment_order';
- 0.4079 [ms] SELECT comment_approved, COUNT(*) AS num_comments FROM wp_posts WHERE post_status = 'publish' AND post_type = 'post' AND post_status = 'publish' AND post_type = 'post';
- 0.2289 [ms] SELECT option_value FROM wp_options WHERE option_name = 'comment_moderation';
- 0.3650 [ms] SELECT option_value FROM wp_options WHERE option_name = 'comment_notify_level';
- 0.4351 [ms] SELECT option_value FROM wp_options WHERE option_name = 'comment_notify_time';
- 0.1891 [ms] SELECT option_value FROM wp_options WHERE option_name = 'comment_whitelist';
- 26.7880 [ms] UPDATE `wp_options` SET `option_value` = '1297894609' WHERE option_name = 'comment_whitelist';

And finally we have the Errors tab, which lists all of the notices, warnings, and errors that occurred during the request:



The screenshot shows the Errors tab of the BlackBox plugin. It lists 14 errors, all of which are of type 'Strict'. The errors are as follows:

- Strict Declaration of Walker_Category_Checklist::start_lvl() should be able to return void
- Strict Declaration of Walker_Category_Checklist::end_lvl() should be able to return void
- Strict Declaration of Walker_Category_Checklist::start_el() should be able to return void
- Strict Declaration of Walker_Category_Checklist::end_el() should be able to return void
- Strict is_a(): Deprecated. Please use the instanceof operator on line 101

By providing quick access to essential debug information, the BlackBox plugin is a big-timer when it comes to debugging your WordPress plugin.

Prefix Your Functions

One of the first things that bit me when I started developing WordPress plugins was finding out that other plugin developers sometimes use the same names for functions that I use. For example, function names like `copy_file()`, `save_data()`, and `database_table_exists()` have a decent chance of being used by other plugins in addition to yours.

The reason for this is because when WordPress activates a plugin, PHP loads the functions from the plugin into the WordPress execution space, where all functions from all plugins live together. There is no separation or isolation of functions for each plugin, which means that **each function must be uniquely named**.

Fortunately, there is an easy way around this, and it's to name all of your plugin functions with a prefix. For example, the common functions I mentioned previously might now look like this:

```
function myplugin_copy_file() {  
}  
  
function myplugin_save_data() {  
}  
  
function myplugin_database_table_exists() {  
}
```

Another common naming convention is to use a prefix that is an abbreviation of your plugin's name, such as "My Awesome WordPress Plugin", in which case the function names would be:

```
function mawp_copy_file() {  
}  
  
function mawp_save_data() {  
}  
  
function mawp_database_table_exists() {  
}
```

There is one caveat to this, however. If you use PHP classes that contain your functions (which in many cases is a good idea), you don't really have to worry about clashing with functions defined elsewhere. For example, let's

say you have a class in your plugin named “CommonFunctions” with a `copy_file()` function, and another plugin has the same `copy_file()` function defined, but not in a class. Invoking the two functions would look similar to this:

```
// Calls the copy_file() function from your class
$common = new CommonFunctions();
$common->copy_file();

// Calls the copy_file() function from the other plugin
copy_file();
```

By using classes, the need to explicitly prefix your functions goes away. Just keep in mind that WordPress will raise an error if you use a function name that’s already taken, so keep an eye on the `debug.log` file to know if you’re in the clear or not.

Global Paths Are Handy

Writing the PHP code to make your plugin work is one thing, but if you want to make it look and feel good at the same time, you’ll need to include some images, CSS, and perhaps a little JavaScript as well (maybe in the form of a jQuery plugin). And in typical fashion, you’ll most likely organize these files into their own folders, such as “images”, “css”, and “js”.

That’s all well and good, but how should you code your plugin so that it can always find those files, no matter what domain the plugin is running under? The best way that I’ve found is to **create your own global paths** that can be used anywhere in your plugin code.

For example, I always create four global variables for my plugins, one each for the following:

- The path to the theme directory

-
- The name of the plugin
 - The path to the plugin directory
 - The url of the plugin

For which the code looks like this:

```
if (!defined('MYPLUGIN_THEME_DIR'))  
    define('MYPLUGIN_THEME_DIR',ABSPATH . 'wp-content/themes/' .  
get_template());  
  
if (!defined('MYPLUGIN_PLUGIN_NAME'))  
    define('MYPLUGIN_PLUGIN_NAME',  
trim(dirname(plugin_basename(__FILE__)), '/'));  
  
if (!defined('MYPLUGIN_PLUGIN_DIR'))  
    define('MYPLUGIN_PLUGIN_DIR', WP_PLUGIN_DIR . '/' .  
MYPLUGIN_PLUGIN_NAME);  
  
if (!defined('MYPLUGIN_PLUGIN_URL'))  
    define('MYPLUGIN_PLUGIN_URL', WP_PLUGIN_URL . '/' .  
MYPLUGIN_PLUGIN_NAME);
```

Having these global paths defined lets me write the code below in my plugin anywhere I need to, and I know it will resolve correctly for any website that uses the plugin:

```
$image = MYPLUGIN_PLUGIN_URL . '/images/my-image.jpg';  
$style = MYPLUGIN_PLUGIN_URL . '/css/my-style.css';  
$script = MYPLUGIN_PLUGIN_URL . '/js/my-script.js';
```

Store the Plugin Version for Upgrades

When it comes to WordPress plugins, one of the things you'll have to deal with sooner or later is upgrades. For instance, let's say the first version of your plugin required one database table, but the next version requires another table. How do you know if you should run the code that creates the second database table?

I suggest storing the plugin version in the WordPress database so that you can read it later to decide certain upgrade actions your plugin should take. To do this, you'll need to create a couple more global variables and invoke the `add_option()` function:

```
if (!defined('MYPLUGIN_VERSION_KEY'))
    define('MYPLUGIN_VERSION_KEY', 'myplugin_version');

if (!defined('MYPLUGIN_VERSION_NUM'))
    define('MYPLUGIN_VERSION_NUM', '1.0.0');

add_option(MYPLUGIN_VERSION_KEY, MYPLUGIN_VERSION_NUM);
```

I certainly could have simply called

`add_option('myplugin_version', '1.0.0');` without the need for the global variables, but like the global path variables, I've found these just as handy for using in other parts of a plugin, such as a Dashboard or About page.

Also note that `update_option()` could have been used instead of `add_option()`. The difference is that `add_option()` does nothing if the option already exists, whereas `update_option()` checks to see if the option already exists, and if it doesn't, it will add the option to the database using `add_option()`; otherwise, it updates the option with the value provided.

Then, when it comes time to check whether or not to perform upgrade actions, your plugin will end up with code that looks similar to this:

```
$new_version = '2.0.0';

if (get_option(MYPLUGIN_VERSION_KEY) != $new_version) {
    // Execute your upgrade logic here

    // Then update the version value
    update_option(MYPLUGIN_VERSION_KEY, $new_version);
}
```

Use dbDelta() to Create/Update Database Tables

If your plugin requires its own database tables, you will inevitably need to modify those tables in future versions of your plugin. This can get a bit tricky to manage if you're not careful, but WordPress helps alleviate this problem by providing the `dbDelta()` function.

A useful feature of the `dbDelta()` function is that it can be used for both creating and updating tables, but according to the WordPress codex page “Creating Tables with Plugins”, it’s a little picky:

- You have to put each field on its own line in your SQL statement.
- You have to have two spaces between the words PRIMARY KEY and the definition of your primary key.
- You must use the keyword KEY rather than its synonym INDEX and you must include at least one KEY.

Knowing these rules, we can use the function below to create a table that contains an ID, a name, and an email:

```

function myplugin_create_database() {
    global $wpdb;
    $table = $wpdb->prefix . 'myplugin_table_name';

    $sql = "CREATE TABLE " . $table . " (
        id INT NOT NULL AUTO_INCREMENT,
        name VARCHAR(100) NOT NULL DEFAULT '',
        email VARCHAR(100) NOT NULL DEFAULT '',
        UNIQUE KEY id (id)
    ) ;";

    require_once(ABSPATH . 'wp-admin/includes/upgrade.php');
    dbDelta($sql);
}

```

Important: The `dbDelta()` function is found in `wp-admin/includes/upgrade.php`, but it has to be included manually because it's not loaded by default.

So now we have a table, but in the next version we need to expand the size of the name column from 100 to 250. Fortunately `dbDelta()` makes this straightforward, and using our upgrade logic previously, the next version of the plugin will have code similar to this:

```

$new_version = '2.0.0';

if (get_option(MYPLUGIN_VERSION_KEY) != $new_version) {
    myplugin_update_database_table();
    update_option(MYPLUGIN_VERSION_KEY, $new_version);
}

function myplugin_update_database_table() {
    global $wpdb;
    $table = $wpdb->prefix . 'myplugin_table_name';
}

```

```
$sql = "CREATE TABLE " . $table . " (
    id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(250) NOT NULL DEFAULT '', // Bigger name column
    email VARCHAR(100) NOT NULL DEFAULT '',
    UNIQUE KEY id (id)
);";

require_once(ABSPATH . 'wp-admin/includes/upgrade.php');
dbDelta($sql);
}
```

While there are other ways to create and update database tables for your WordPress plugin, it's hard to ignore the flexibility of the `dbDelta()` function.

Know the Difference Between `include`, `include_once`, `require`, and `require_once`

There will come a time during the development of your plugin where you will want to put code into other files so that maintaining your plugin is a bit easier. For instance, a common practice is to create a `functions.php` file that contains all of the shared functions that all of the files in your plugin can use.

Let's say your main plugin file is named `myplugin.php` and you want to include the `functions.php` file. You can use any of these lines of code to do it:

```
include 'functions.php';
include_once 'functions.php';

require 'functions.php';
```

```
include 'functions.php';
include_once 'functions.php';

require 'functions.php';
```

But which should you use? It mostly depends on your expected outcome of the file not being there.

- **include:** Includes and evaluates the specified file, throwing a warning if the file can't be found.
- **include_once:** Same as include, but if the file has already been included it will not be included again.
- **require:** Includes and evaluates the specified file (same as include), but instead of a warning, throws a fatal error if the file can't be found.
- **require_once:** Same as require, but if the file has already been included it will not be included again.

My experience has been to always use `include_once` because a) how I structure and use my files usually requires them to be included once and only once, and b) if a required file can't be found I don't expect parts of the plugin to work, but it doesn't need to break anything else either.

Your expectations may vary from mine, but it's important to know the subtle differences between the four ways of including files.

Use `bloginfo('wpurl')` Instead of `bloginfo('url')`

By and large, WordPress is installed in the root folder of a website; it's standard operating procedure. However, every now and then you'll come across websites that install WordPress into a separate subdirectory under the root. Seems innocent enough, but the location of WordPress is critically

settings, and for sites where WordPress is installed into the root directory, they will have the exact same values:

WordPress address (URL)	<input type="text" value="http://mydomain.com"/>
Site address (URL)	<input type="text" value="http://mydomain.com"/>

But for sites where WordPress is installed into a subdirectory under the root (in this case a “wordpress” subdirectory), their values will be different:

WordPress address (URL)	<input type="text" value="http://mydomain.com/wordpress"/>
Site address (URL)	<input type="text" value="http://mydomain.com"/>

At this stage it's important to know the following:

- **bloginfo('wpurl')** equals the “WordPress address (URL)” setting
- **bloginfo('url')** equals the “Site address (URL)” setting

Where this matters is when you need to build URLs to certain resources or pages. For example, if you want to provide a link to the WordPress login screen, you could do this:

```
// URL will be http://mydomain.com/wp-login.php
<a href="<?php bloginfo('url') ?>/wp-login.php">Login</a>
```

But that won't resolve to the correct URL in the scenario such as the one above where WordPress is installed to the “wordpress” subdirectory. To do this correctly, you must use **bloginfo('wpurl')** instead:

```
// URL will be http://mydomain.com/wordpress/wp-login.php
<a href="<?php bloginfo('wpurl') ?>/wp-login.php">Login</a>
```

Using `bloginfo('wpurl')` instead of `bloginfo('url')` is the safest way to go when building links and URLs inside your plugin because it works in both scenarios: when WordPress is installed in the root of a website and also when it's installed in a subdirectory. Using `bloginfo('url')` only gets you the first one.

How and When to Use Actions and Filters

WordPress allows developers to add their own code during the execution of a request by providing various hooks. These hooks come in the form of actions and filters:

- **Actions:** WordPress invokes actions at certain points during the execution request and when certain events occur.
- **Filters:** WordPress uses filters to modify text before adding it to the database and before displaying it on-screen.

The number of actions and filters is quite large, so we can't get into them all here, but let's at least take a look at how they are used.

Here's an example of how to use the `admin_print_styles` action, which allows you to add your own stylesheets to the WordPress admin pages:

```
add_action('admin_print_styles', 'myplugin_admin_print_styles');

function myplugin_admin_print_styles() {
    $handle = 'myplugin-css';
    $src = MYPLUGIN_PLUGIN_URL . '/styles.css';

    wp_register_style($handle, $src);
    wp_enqueue_style($handle);
}
```

And here's how you would use the `the_content` filter to add a "Follow me on Twitter!" link to the bottom of every post:

```
add_filter('the_content', 'myplugin_the_content');

function myplugin_the_content($content) {
    $output = $content;
    $output .= '<p>';
    $output .= '<a href="http://twitter.com/username ">Follow me on
Twitter!</a>';
    $output .= '</p>';
    return $output;
}
```

It's impossible to write a WordPress plugin without actions and filters, and knowing what's available to use and when to use them can make a big difference. See the WordPress codex page "Plugin API/Action Reference" for the complete list of actions and the page "Plugin API/Filter Reference" for the complete list of filters.

***Tip:** Pay close attention to the order in which the actions are listed on its codex page. While not an exact specification, my experimentation and trial-and-error has shown it to be pretty close to the order in which actions are invoked during the WordPress request pipeline.*

Add Your Own Settings Page or Admin Menu

Many WordPress plugins require users to enter settings or options for the plugin to operate properly, and the way plugin authors accomplish this is by either adding their own settings page to an existing menu or by adding their own new top-level admin menu to WordPress.

HOW TO ADD A SETTINGS PAGE

A common practice for adding your own admin settings page is to use the `add_menu()` hook to call the `add_options_page()` function:

```
add_action('admin_menu', 'myplugin_admin_menu');

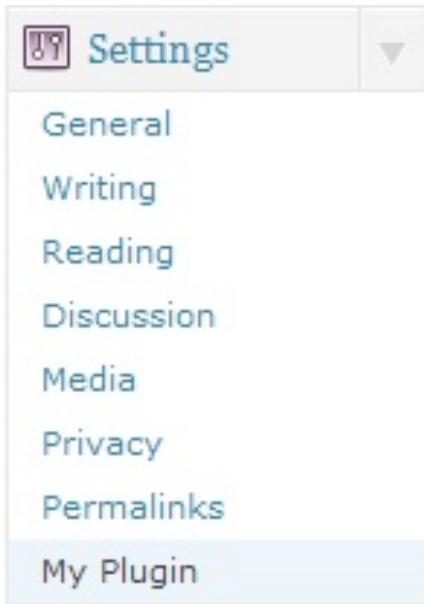
function myplugin_admin_menu() {
    $page_title = 'My Plugin Settings';
    $menu_title = 'My Plugin';
    $capability = 'manage_options';
    $menu_slug = 'myplugin-settings';
    $function = 'myplugin_settings';

    add_options_page($page_title, $menu_title, $capability,
    $menu_slug, $function);
}

function myplugin_settings() {
    if (!current_user_can('manage_options')) {
        wp_die('You do not have sufficient permissions to access this
page.');
    }

    // Here is where you could start displaying the HTML needed for
    // the settings
    // page, or you could include a file that handles the HTML
    // output for you.
}
```

By invoking the `add_options_page()` function, we see that the “My Plugin” option has been added to the built-in **Settings** menu in the WordPress admin panel:



The `add_options_page()` function is really just a wrapper function on top of the `add_submenu_page()` function, and there are other wrapper functions that do similar work for the other sections of the WordPress admin panel:

- `add_dashboard_page()`
- `add_posts_page()`
- `add_media_page()`
- `add_links_page()`
- `add_pages_page()`
- `add_comments_page()`
- `add_theme_page()`
- `add_plugins_page()`
- `add_users_page()`
- `add_management_page()`

HOW TO ADD A CUSTOM ADMIN MENU

Those wrapper functions work great, but what if you wanted to create your own admin menu section for your plugin? For example, what if you wanted to create a “My Plugin” admin section with more than just the Settings page, such as a Help page? This is how you would do that:

```
add_action('admin_menu', 'myplugin_menu_pages');

function myplugin_menu_pages() {
    // Add the top-level admin menu
    $page_title = 'My Plugin Settings';
    $menu_title = 'My Plugin';
    $capability = 'manage_options';
    $menu_slug = 'myplugin-settings';
    $function = 'myplugin_settings';

    add_menu_page($page_title, $menu_title, $capability, $menu_slug,
    $function);

    // Add submenu page with same slug as parent to ensure no
    // duplicates
    $sub_menu_title = 'Settings';
    add_submenu_page($menu_slug, $page_title, $sub_menu_title,
    $capability, $menu_slug, $function);

    // Now add the submenu page for Help
    $submenu_page_title = 'My Plugin Help';
    $submenu_title = 'Help';
    $submenu_slug = 'myplugin-help';
    $submenu_function = 'myplugin_help';

    add_submenu_page($menu_slug, $submenu_page_title, $submenu_title,
    $capability, $submenu_slug, $submenu_function);
}
```

```
function myplugin_settings() {
    if (!current_user_can('manage_options')) {
        wp_die('You do not have sufficient permissions to access this
page.');
    }

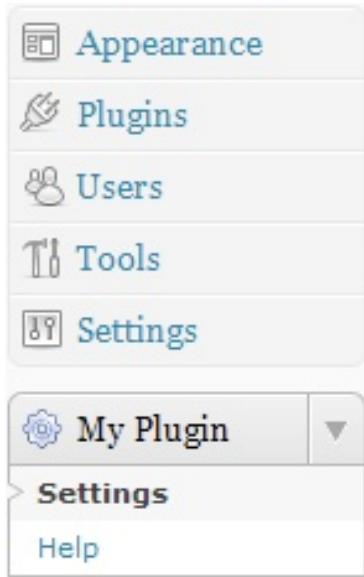
    // Render the HTML for the Settings page or include a file that
does
}

function myplugin_help() {
    if (!current_user_can('manage_options')) {
        wp_die('You do not have sufficient permissions to access this
page.');
    }

    // Render the HTML for the Help page or include a file that does
}

```

Notice that this code doesn't use any of the wrapper functions. Instead, it calls `add_menu_page()` (for the parent menu page) and `add_submenu_page()` (for the child pages) to create a separate "My Plugin" admin menu that contains the Settings and Help pages:



One advantage of adding your own custom menu is that **it's easier for users to find the settings for your plugin** because they aren't buried within one of the built-in WordPress admin menus. Keeping that in mind, if your plugin is simple enough to only require a single admin page, then using one of the wrapper functions might make the most sense. But if you need more than that, creating a custom admin menu is the way to go.

Provide a Shortcut to Your Settings Page with Plugin Action Links

In much the same way that adding your own custom admin menu helps give the sense of a well-rounded plugin, plugin action links work in the same fashion. So what are plugin action links? It's best to start with a picture:

<input type="checkbox"/>	Plugin	Description
<input checked="" type="checkbox"/>	My Plugin	This is not the greatest plugin in the world, it's just a tribute.
	Deactivate Edit	Version 1.0.0 By Dave Donaldson
<input type="checkbox"/>	Plugin	Description

See the “Deactivate” and “Edit” links underneath the name of the plugin? Those are plugin action links, and WordPress provides a filter named `plugin_action_links` for you to add more. Basically, **plugin action links are a great way to add a quick shortcut to your most commonly used admin menu page.**

Keeping with our Settings admin page, here’s how we would add a plugin action link for it:

```
add_filter('plugin_action_links', 'myplugin_plugin_action_links',
10, 2);

function myplugin_plugin_action_links($links, $file) {
    static $this_plugin;

    if (!$this_plugin) {
        $this_plugin = plugin_basename(__FILE__);
    }

    if ($file == $this_plugin) {
        // The "page" query string value must be equal to the slug
        // of the Settings admin page we defined earlier, which in
        // this case equals "myplugin-settings".
        $settings_link = '<a href="' . get_bloginfo('wpurl') . '/wp-
admin/admin.php?page=myplugin-settings">Settings</a>';
        array_unshift($links, $settings_link);
    }
}

return $links;
}
```

With this code in place, now when you view your plugins list you’ll see this:

<input type="checkbox"/> Plugin	Description
<input type="checkbox"/> My Plugin	This is not the greatest plugin in the world, it's just a tribute. Settings Deactivate Edit Version 1.0.0 By Dave Donaldson
<input type="checkbox"/> Plugin	Description

Here we provided a plugin action link to the Settings admin page, which is the same thing as clicking on Settings from our custom admin menu. The benefit of the plugin action link is that **users see it immediately after they activate the plugin**, thus adding to the overall experience.

Create Perfect Emails For Your WordPress Website

Daniel Pataki

Whatever type of website you operate, its success will probably hinge on your interaction with your audience. If executed well, one of the most effective tools can be a **simple email**.

WordPress users are in luck, since WordPress already has easy-to-use and extendable functions to give you a lot of power and flexibility in handling your website's emails.

In order to create our own system, we will be doing four things. First, we will create a nice email template to use. We will then modify the mailer function so that it uses our new custom template. We will then modify the actual text of some of the built-in emails. Then we will proceed to hook our own emails into different events in order to send some custom emails. Let's get started!

How WordPress Sends Emails

WordPress has a handy function built in called `wp_mail()`, which handles the nitty-gritty of email sending. It is able to handle almost anything you throw at it, from custom HTML emails to modifications to the “From” field.

WordPress itself uses this function, and you can, too, by using WordPress hooks. You can read all about [how hooks work in WordPress](#), but here is the nutshell version, and we will be working with them in this chapter so much that you'll learn it by the end.

Hooks enable you to add your own functions to WordPress without modifying core files. Without hooks, if you wanted to send a publication notice to the author of a post, you would have to find the function that published the post and add your own code directly to it. With hooks, you write the function for sending the email, and then hook it into the function that publishes the post. Basically, you are telling WordPress to run your custom function whenever the function for publishing posts runs.

Setting Up Shop

The first thing we'll have to do is create a plugin. We could get away without it and just use our theme's functions file, but this would become clunky in the long run. Don't worry: setting up a plugin is super-easy.

Go to your website's plugins folder, which can be found under wp-content. Create a new folder named `my_awesome_email_plugin`. If you want a different name, use something unique, not `email` or `email_plugin`; otherwise, conflicts might arise with other plugins.

Create a file named `my_awesome_email_plugin.php` in the new folder. The name of the file (without the extension) must be the same as the name of the folder.

Edit the contents of `my_awesome_email_plugin.php` by copying and pasting the code below and modifying it where necessary. This is just some default information that WordPress uses to show the plugin in the plugins menu in the admin area.

```
<?php
/*
Plugin Name: My Awesome Email Plugin
Plugin URI: http://myawesomewebsite.com
Description: I created this plugin to rule the world via awesome
WordPress email goodness
Version: 1.0
Author: Me
Author URI: http://myself.me
*/
?>
```

Once that's done, save the file, go to the WordPress admin section, and activate your new plugin. If you're new to this, then congratulations! You have just created your first working WordPress plugin! It doesn't really do anything yet, but don't let that bother you. Just read on, because we'll be adding some functionality after the next section.

Creating An Email Template

Creating good email templates is worth a chapter on its own. I will just share the method that I use, which does not mean that doing it differently is not allowed. Feel free to experiment!

I am not a big fan of using images in emails, so we will be building an HTML template using only CSS. Our goal is to come up with a template to which we can add a header and footer section. We will send our emails in WordPress by pulling in the header, putting the email text under that and then pulling in the footer. This way, you can change the design of your emails very easily just by modifying the templates.

Without further ado, here's the code for the email template that I made. Or you can [download it as an HTML file](#) (right-click, and then select “Save as”). If you want a quick preview of what it looks like, just click the link.

```
<html>
  <head>

    <title>The Subject of My Email</title>

  </head>
  <body>

    <div id="email_container" style="background:#444">
      <div style="width:570px; padding:0 0 0 20px; margin:50px auto 12px auto" id="email_header">
        <span style="background:#585858; color:#fff; padding:12px;font-family:trebuchet ms; letter-spacing:1px;
-moz-border-radius-topleft:5px; -webkit-border-top-left-radius:5px;
border-top-left-radius:5px;moz-border-radius-topright:5px; -webkit-border-top-right-radius:5px;
border-top-right-radius:5px;">
          MyAwesomeWebsite.com
        </span>
      </div>
      </div>

      <div style="width:550px; padding:0 20px 20px 20px; background:#fff; margin:0 auto; border:3px #000 solid;
moz-border-radius:5px; -webkit-border-radius:5px; border-radius:5px; color:#454545;line-height:1.5em;" id="email_content">

        <h1 style="padding:5px 0 0 0; font-family:georgia;font-weight:500;font-size:24px;color:#000;border-bottom:1px solid #bbb">
          The subject of this email
        </h1>

        <p>
          Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa
          <strong>strong</strong>. Cum sociis natoque penatibus
```

et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut.

</p>

<p>

Imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede link mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi.

</p>

<p style="">

Warm regards,

The MyAwesomeWebsite Editor

</p>

<div style="text-align:center; border-top:1px solid #eee;padding:5px 0 0 0;" id="email_footer">

<small style="font-size:11px; color:#999; line-height:14px;">

You have received this email because you are a member of MyAwesomeSite.com.

If you would like to stop receiving emails from us, feel free to

unregister from our mailing list

</small>

```
</div>

</div>
</div>
</body>
</html>
```

Remember that this is an email, so the HTML won't be beautiful. The safest styling method is inline, so the fewer frills you can get away with, the better.

Let's split this into two parts. The header part of the email is everything from the top right up to and including the `h1` heading on row 23 (i.e. lines 01 to 23). Copy that bit and paste it into a new file in your `my_email_plugin` folder, and name it `email_header.php`. The footer part of the email is everything from the paragraph tag before "Warm regards" right until the end (i.e. lines 48 to 64). The text between the header and footer is just a placeholder so that you can see what the finished product will look like. We will fill it with whatever content we need to send at the time.

Preparing The WordPress System For Our Emails

By default, WordPress sends plain-text emails. In order to accommodate our fancy HTML email, we need to tell the `wp_mail()` function to use the HTML format. We will also set up a custom "From" name and "From" address in the process, so that the email looks good in everyone's inbox. To accomplish this, we'll be using the previously mentioned hooks. Let's look at the code; explanation follows.

```
add_filter ("wp_mail_content_type", "my_awesome_mail_content_type");
function my_awesome_mail_content_type() {
    return "text/html";
}

add_filter ("wp_mail_from", "my_awesome_mail_from");
```

```
function my_awesome_mail_from() {
    return "hithere@myawesomesite.com";
}

add_filter ("wp_mail_from_name", "my_awesome_email_from_name");
function my_awesome_email_from_name() {
    return "MyAwesomeSite";
}
```

On line 01, we have defined that we are adding a filter to the WordPress function `wp_mail_content_type()`. Our filter will be called `my_awesome_mail_content_type`. A filter is nothing more than a function, so we need to create the function `my_awesome_mail_content_type()`.

Remember that actions are functions called from within other functions? We add an action to the `wp_insert_user()` function, and the action is performed whenever `wp_insert_user()` runs. Filters are specified in much the same way; but, instead of running alongside the function that it is called from, it modifies the value of the entity that it is called on.

In our case, this means that somewhere inside the `wp_mail()` function is a variable that holds the email type, which is by default `text/plain`. The filter hook `wp_mail_content_type` is called on this variable, which means that all attached filters will be run. We happen to have attached a filter to it on line 01, so our function will perform its task. All we need to do is return the value `text/html`, which will modify the value of the variable in the `wp_mail` function to `text/html`.

The logic behind the rest of the code is exactly the same. Adding a filter to `wp_mail_from` enables us to change the sender's address to `hithere@myawesomewebsite.com`, and adding a filter to `wp_mail_from_name` enables us to change the sender's name.

Modifying Existing WordPress System Emails

WELCOMING NEW USERS

Username: danipataki
Password: HS))EmQxh1\$f
<http://webtastique.net/wp-login.php>

This is the content of the default WordPress email.

As mentioned, WordPress has a bunch of built-in emails that can be easily controlled (using hooks, of course). Let's modify the default greeting email that WordPress sends out to new users. This email is sent out using a so-called "pluggable function." This function is supplied by WordPress, but, contrary to the usual core functions, you are allowed to overwrite it with your own code.

The function in question is called `wp_new_user_notification()`. To modify it, all we need to do is create a function with the same name. Due to the method by which WordPress calls pluggable functions, there will not be any conflict, even though you are creating a function with the same name. Below is the function that I wrote. See the explanation and preview of it further below.

```
function wp_new_user_notification($user_id, $plaintext_pass) {  
    $user = new WP_User($user_id);  
  
    $user_login = stripslashes($user->user_login);  
    $user_email = stripslashes($user->user_email);  
  
    $email_subject = "Welcome to MyAwesomeSite ".$user_login."!";  
  
    ob_start();
```

```
include("email_header.php");

?>

<p>A very special welcome to you, <?php echo $user_login ?>. Thank you
for joining MyAwesomeSite.com!</p>

<p>
    Your password is <strong style="color:orange"><?php echo
$plaintext_pass ?></strong> <br>
    Please keep it secret and keep it safe!
</p>

<p>
    We hope you enjoy your stay at MyAwesomeSite.com. If you have any
problems, questions, opinions, praise,
    comments, suggestions, please feel free to contact us at any time
</p>

<?php
include("email_footer.php");

$message = ob_get_contents();
ob_end_clean();

wp_mail($user_email, $email_subject, $message);
```

As you can see, the function is passed two arguments: the ID of the new user and the generated password. We will be using these to generate the variable parts of our message. On line 2, we've built a user object that will contain the data of the user in question. On line 7, we've created an email subject using the variable `$email_subject`.

Before we move on, let's go back to our `email_header.php` file. Replace “The Subject of My Email” and “The subject of this email” (lines 04 and 22 if you're looking at the code here) with `<?php echo $email_subject ?>`. We don't want all of our subjects to be “The Subject of My Email,” so we need to pull that data from the email that we are building.

From lines 09 to 31, we are using a handy technique called “output buffering.” Because the content `email_header.php` is not stored inside a variable, it is included directly; this would result in it being printed right away, and we would not be able to use it in our function. To get around this problem, we output buffering. When it is turned on (using `ob_start()`), no output is sent from the script; instead, it is stored in an internal buffer.

So, first, we include the header, then we write our message content, then include the footer. Because we are buffering the content, we can simply close our PHP tags and use regular HTML for our message, which I find much cleaner than storing all of it in a variable. On line 30, we pull the contents of the buffer into a variable; and on line 31, we discard the buffer's content, since we don't need it anymore.

With that done, we have all of the information needed to use `wp_mail()`. So, on line 33, we send our email to the user, which should look something like this:

Welcome to MyAwesomeSite daniel!

A very special welcome to you daniel, thank you for joining MyAwesomeSite.com!

Your password is **M1gNUWtSUHWJ**

Please keep it secret, keep it safe!

We hope you enjoy your stay at MyAwesomeSite.com,
if you have any problems, questions, opinions, praise, comments,
suggestions, please feel free not to contact us at any time

Warm Regards,
The MyAwesomeWebsite Editor

You have received this email because you are a member of MyAwesomeSite.com.
If you would like to stop receiving emails from us, feel free to unregister from our
mailing list

PASSWORD RETRIEVAL EMAILS

For some reason, WordPress doesn't use the same pluggable functions to handle all emails. For example, to modify the look and feel of the password retrieval emails, we have to resort to hooks. Let's take a look.

```
add_filter ("retrieve_password_title",
"my_awesome_retrieve_password_title");

function my_awesome_retrieve_password_title() {
    return "Password Reset for MyAwesomeWebsite";
}
```

```
add_filter ("retrieve_password_message",
"my_awesome_retrieve_password_message");
function my_awesome_retrieve_password_message($content, $key) {
    global $wpdb;
    $user_login = $wpdb->get_var("SELECT user_login FROM $wpdb-<users WHERE
user_activation_key = '$key'");
    ob_start();
    $email_subject = imp_retrieve_password_title();
    include("email_header.php");
    ?>
<p>
    It likes like you (hopefully) want to reset your password for your
MyAwesomeWebsite.com account.
</p>
<p>
    To reset your password, visit the following address, otherwise just
ignore this email and nothing will happen.
    <br>
    <?php echo wp_login_url("url") ?>?action=rp&key=<?php echo $key ?
&login=<?php echo $user_login ?>
<p>
    ?>
    include("email_footer.php");
    $message = ob_get_contents();
    ob_end_clean();
```

```
    return $message;  
}
```

First, we've added a filter to `retrieve_password_title`, which will modify the default value of the email's title to our own. Then, we've added a filter to `retrieve_password_message`, which will modify the contents of the message.

On line 10, we've used the `$wpdb` object to query the database for the user's name based on the key that was generated when the retrieval was initiated. We then do the same thing as before: we start the content buffering, pull our email header, add our message content, and pull our email footer.

One fantastic part about using hooks can be seen on line 14. Our password title needs to be "Password Reset for MyAwesomeWebsite." We could well have typed that in, but instead we created a function

(`imp_retrieve_password_title()`) that outputs exactly the same thing. It should be clear by now that all we are doing with these hooks is creating regular ol' functions that can just be plugged into WordPress as actions (which run when initiated by something) or filters (which run and modify data when they are initiated).

This time, instead of using `wp_mail()`, all we need to do is return the message's content. This is because we are creating a filter that modifies the contents of the password-retrieval email, nothing else. WordPress will do whatever it usually does to send that email, but now it will use our content.

PLUGGABLE FUNCTION AND HOOKS

This question is not easily answered, because this is not too well documented yet. Your best bet is looking in the file `pluggable.php` (in your `wp-includes` folder) to see which emails are controlled from there.

Remember not to edit this file; use the plugin we are creating here. You can scan the [list of WordPress filters](#) to find filters that control email content.

Right now, most emails are handled through pluggable functions; only the password-retrieval email and some WordPress MU emails are handled using hooks. This might change, as development is quite active, but I would guess that if any new emails are added, you will be able to use pluggable functions.

Here is a list of emails that you can modify using pluggable functions:

- Notify authors of comments: `wp_notify_postauthor()`
- Notify moderator of comments waiting for approval:
`wp_notify_moderator()`
- Notify administrator of password changes on the website:
`wp_password_change_notification()`
- Notify administrator of new registrations:
`wp_new_user_notification()`

Adding New Emails To The System

So far, we've just been modifying what WordPress has to offer. Now it's time to add some emails of our own! Let's implement an email that will notify an author when you have published their post.

To accomplish this, we need to find the WordPress action hook that publishes a post when we press the "Publish" button. We then have to hook our own function into that, which will perform the task of sending the email.

Looking at the [list of action hooks](#), we can see that the hook we are looking for is called `{$new_status}_{$post->post_type}`. This looks a bit different than what we're used to, but it's really very simple. A post can go

through numerous statuses. It can be a draft, it can be private, published and so on. There are also a lot of potential post types, such as “Post” and “Page,” not to mention that you can create custom post types. This hook simply enables us to put a status and a post type together and then get the hook that runs when that post’s type changes to the indicated status. So, **our hook will be called `publish_post`.**

```
add_action("publish_post", "my_awesome_publication_notification");

function my_awesome_publication_notification($post_id) {
    $post = get_post($post_id);
    $author = get_userdata($post->post_author);

    $author_email = $author->user_email;
    $email_subject = "Your article has been published!";

    ob_start();

    include("email_header.php");

    ?>

    <p>
        Hi, <?php echo $author->display_name ?>. I've just published one of
        your articles
        (<?php echo $post->post_title ?>) on MyAwesomeWebsite!
    </p>

    <p>
        If you'd like to take a look, <a href="<?php echo get_permalink($post-
        >ID) ?>">click here</a>.
    </p>
}
```

I would appreciate it **if** you could come back now **and** again to respond to some comments.

```
</p>
```

```
<?php

include("email_footer.php");

$message = ob_get_contents();

ob_end_clean();

wp_mail($author_email, $email_subject, $message);

}
```

By now, this should be second nature to you. The only real difference here is that we have to retrieve the data of the post, and the author's data on lines 4 and 5, so that we have the necessary data for the email.

One thing you might be wondering is how I know that my function takes the ID of this post as its argument. I cannot freely choose this value, of course; it is dictated by how WordPress is built. Every hook supplies different arguments; some even supply more than one. To find out what arguments are at your disposal, you will have to go into some core files.

I suggest browsing the [hooks database](#), clicking on the hook that you need, and then clicking on “View hook in source” next to your version of WordPress (preferably the latest one). Once there, scroll down, and find the highlighted line, which is where the hook is called. It will be in the form of `do_action($tag, $arg_a, $arg_b, $etc)` or `apply_filters($tag, $arg_a, $arg_b, $etc).`

Extending Our Functions

Interestingly, the `wp_mail()` function itself is a pluggable function, so you can completely override how it works. This may be going a bit over the top, but if you need some serious email-sending power (for example, you want a

system that notifies your 120,000 registered users about new posts), you can completely modify it to use your mass-mailer application.

Because we are using a template for email headers and footers, a lot can be done to extend our emails. We can distinguish between emails to staff and emails to users by using different header files; we can add the latest three posts to the bottom of each email by using footer templates; and so on.

We can add a table to our database that holds information about which users are emailed the most, who responds to emails, and so on. Whenever you plug a function into something, it can contain any sort of code you'd like. You could include code for increasing the email count for user #112 inside the function that sends them the email, for example. This is not a good practice (you should create separate functions and plug them both in), but getting to grips with the vast power that this methodology offers is important.

A Word Of Warning

While the method described here is great, I am not an expert in creating HTML emails. The code for the HTML email above is tested to work in Gmail and some other applications, but each email application handles email differently. Some strip out all CSS, some strip out just background colors, and so on.

Before using the template, please test it with the most common applications (Gmail, Yahoo Mail, Apple Mail, Outlook, Entourage, Thunderbird, etc.) to make sure it works as expected.

Conclusion

Hopefully by now you have learned how to modify the emails that WordPress sends out, how to create your own emails, and how to plug them into different actions.

I also hope that your knowledge of WordPress hooks has expanded, because they are the tool for creating great plugins and add-ons for WordPress, and the thinking behind them is a glimpse into the world of object-oriented programming.

Writing WordPress Guides For The Advanced Beginner

Scott Meaney

Creating WordPress tutorials is a fantastic way to help build the WordPress community and to increase your Web traffic. That's no secret. Just Google "wordpress tutorial" and you'll see hundreds of results. The complete novice will find scores of well-written tutorials clearly demonstrating the basics of the WordPress dashboard and of activating the default template, in simple jargon-free language.



Unfortunately, after the first few “Hello World!” tutorials, they are in for a bit of a learning curve. Suddenly, the guides start to skip a lot of details, assuming that the reader “already knows this stuff.” Others are simply written exclusively for advanced WordPress users.

So, where does a new developer go after square one?

In this chapter, we’ll explore how to create clear easy-to-navigate tutorials, and tailor them to the underserved “advanced beginner” Web developer. The entire goal of this chapter is to make sure we see many more tutorials written for budding new coders who are ready to jump to the next level.

Who Exactly Is An “Advanced Beginner”?

Advanced beginners are people who generally understand how WordPress works but don’t fully understand how to implement its concepts. They are stuck in that awkward stage where a “For Dummies” book has nothing new to offer but raw code is still vaguely confusing. In your tutorials, you should strive to eliminate this common “tough it out” phase.

For our purposes, let’s assume that we are writing for someone who has a reasonably good grasp on the following:

- Can read and write XHTML and CSS, but probably sits with a cheat sheet open to get through those tricky spots;
- Knows little to nothing about PHP;
- Can navigate the WordPress dashboard and has basic knowledge of image resizing and editing;
- Understands the basic idea and principles of WordPress, but not necessarily how to execute them;
- Appreciates the simplicity of WordPress templates but wants to learn how to create or tweak their own.

Admitting That WordPress Can Be Tough

We all need to stop pretending that WordPress is this magical dirt-simple Web development solution. Yes, using it is much easier than designing a custom CMS, but for new users looking to get under the hood, the tool can still be daunting and complicated.

For the average coder who is still just getting a grip on fundamental CSS, even the strange-looking batch of official WordPress folders that come in the installation ZIP file can be intimidating.

Name
► wp-includes
▼ wp-content
► plugins
► themes
index.php
▼ wp-admin
► maint
► js
► includes
► images
► css
install.php
ms-edit.php
admin-ajax.php
edit-attachment-rows.php
edit.php
edit-tags.php
nav-menus.php
menu.php
admin.php
ms-sites.php
user-new.php
press-this.php
import.php
plugins.php
export.php
edit-comments.php

This is way more confusing for a beginner than seeing a simple HTML file, a CSS file and some images.

When you refer to a file such as style.css or an image, be sure to tell readers exactly where to look and where to save these files.

Basic Guide-Writing Principles

Before we delve into WordPress-specific tips, let's go over some basic principles for any tutorial.

KEEP IT TIDY

Readers have sought your advice because they are confused. Don't add to their troubles with a cluttered how-to chapter. Use plenty of bullet points, and keep paragraphs short. If you're tackling a complex idea, split it up into sections.

Take the format of Smashing Magazine's articles. Articles are broken up so that each sub-topic has its own section. This simplifies the navigation, makes the content more visually appealing and clearly guides the reader through the process.

MAKE SURE READERS ARE FULLY PREPARED

Any good tutorial includes all of the resources it recommends. Don't just say "make a blue image" — give it to them. Otherwise you risk over-complicating things for the reader. Provide sample files, and explain that your lesson will deal exclusively with these readily available resources. You wouldn't want them to suddenly have to read a Photoshop tutorial when they're only interested in learning how to customize their header.



Tutorials | Wordpress | Rating: ★★★★

How to Create a WordPress Theme from Scratch

Sam Parkinson on Sep 5th 2008 with 187 comments

Tutorial Details

Program: WordPress

Difficulty: Beginner-Intermediate

Completion Time: 1-2 hours

[Tweet](#)
[Send](#)
[Like](#) 92

[Download](#) [SOURCE FILES](#) [Demo](#) [VIEW IT ONLINE](#)

This tutorial includes everything a reader needs to get started, including a visual demo and easily accessible sample files.

DEFINE YOUR GOAL

The best tutorials focus on a single topic. Plan the article before writing it. You shouldn't explain every aspect of CSS and WordPress on every page of your website. What will readers get from this particular tutorial? A nice neat list at the top of the article should clearly define its parameters.

LIST THE PREREQUISITE SKILLS

A tutorial should always list any skills that the reader will be expected to have. Instead of cluttering an otherwise focused guide with extraneous detail, provide links that direct readers to where they should go to learn

about particular topics. This will help new developers who are nearly clueless, while keeping the article clearly focused for more advanced readers.

Tips Specifically For WordPress Guides

Now that we've discussed some fundamental organizational skills that will make any tutorial clear and easy to follow, let's delve into some WordPress-related areas that many guides seem to miss.

TAMING THE CODEX

The WordPress Codex is a powerful tool that can give your tutorial a much-needed jolt of clarification. Just be aware that to newbie designers, the Codex can seem like a massive labyrinth of articles, with each topic requiring that you read several earlier lessons in order to fully grasp. As the experienced coder, you need to show that, when used properly, the Codex presents the cleanest example of a concept.

Codex

Main Page

Welcome to the **WordPress Codex**, the online manual for WordPress and a living repository for WordPress information and documentation.

What You Most Need to Know About WordPress

[WordPress Features](#)[Download WordPress](#)[Installing WordPress](#)[Current WordPress Version](#)[WordPress News](#)[WordPress Support Forums](#)[Troubleshooting](#)[About WordPress](#)[Glossary](#)

Contents

[\[hide\]](#)

- [1 What You Most Need to Know About WordPress](#)
- [2 Learn How to Use WordPress](#)
- [3 Working With Themes](#)
- [4 Write a Plugin](#)
- [5 Contribute to Development](#)
- [6 Give Back](#)

The Codex is one of the most useful tools available to a WordPress developer.

Don't just say "Check the Codex" and drop in a link. Your readers need context. Your main goal in writing a tutorial that refers to the Codex should be to eliminate the reader's need to plunge into its depths. Tell them what they can expect to read on the page, illustrating exactly how they can use the particular lesson you're linking to.

It might even be to your benefit to point readers to a "beginner's guide" to understanding the codex. [Here is my favorite.](#)

KEEP THEM ON TARGET, VISUALLY

The most important thing to do to keep readers on track is to provide constant updates throughout the article on what they should be seeing in

their own implementation. For example, if your tutorial is multiple pages, always start with an illustration of the finished product. After each milestone, provide a “Here’s what you should be seeing right now” example. Whenever possible, include working samples of the project or its parts for the reader to experiment with. (These functional samples might have to be run from the author’s server or a third-party website.)

A WordPress project could very easily require coding between a few files. If someone isn’t following closely enough, they could miss something simple that wildly alters their results. Your milestone examples will give readers up-to-the-minute feedback on where they are going wrong. It’s the best way to make sure you aren’t losing anyone.

MAKE YOUR CODE SELECTABLE

This is crucial to any WordPress tutorial. If you are explaining a concept in code, allow the reader to copy and paste the examples whenever possible. For curious readers, nothing is worse than wanting to test a sample line of code, only to realize that they have to fully type it out. This principle seems self-evident, but many guides simply explain an idea and say, “Add this code,” alongside a screenshot of the finished style sheet. If the reader misses one semicolon, all their work will be worthless. That’s infuriating.

While there may be some merit to having the reader actually write out the code, most people probably won’t see it that way. They are much more likely to seek out another tutorial, one that doesn’t force them to constantly rewrite code that they don’t yet understand.

BE WARY OF PHP

While it’s a necessary part of WordPress, remember that to someone just getting their footing with something as relatively basic as CSS, PHP code can look like someone fell asleep at the keyboard. Too many tutorial

providers assume that their readers understand even the first thing about PHP. This is often not the case.

In the likely event that you are explaining low-level PHP to readers, be mindful that they might be confused. Give a short description of exactly what is happening in the code. As always, provide a link to a relevant PHP tutorial.

CLARIFYING CUSTOM WIDGETS

Admittedly, this recommendation is pretty specific, but bear with me. When I was getting started, one of the most infuriating things about WordPress tutorials was when they said, “Write a quick widget with this code...”

Now, once a reader has created their first widget, it becomes completely obvious that most of the time all they’ll need to do is drag the “Text Widget” and add some basic HTML code to it. But first they need to get past this initial step. Remember that to someone looking with fresh eyes, they may not understand your shorthand.

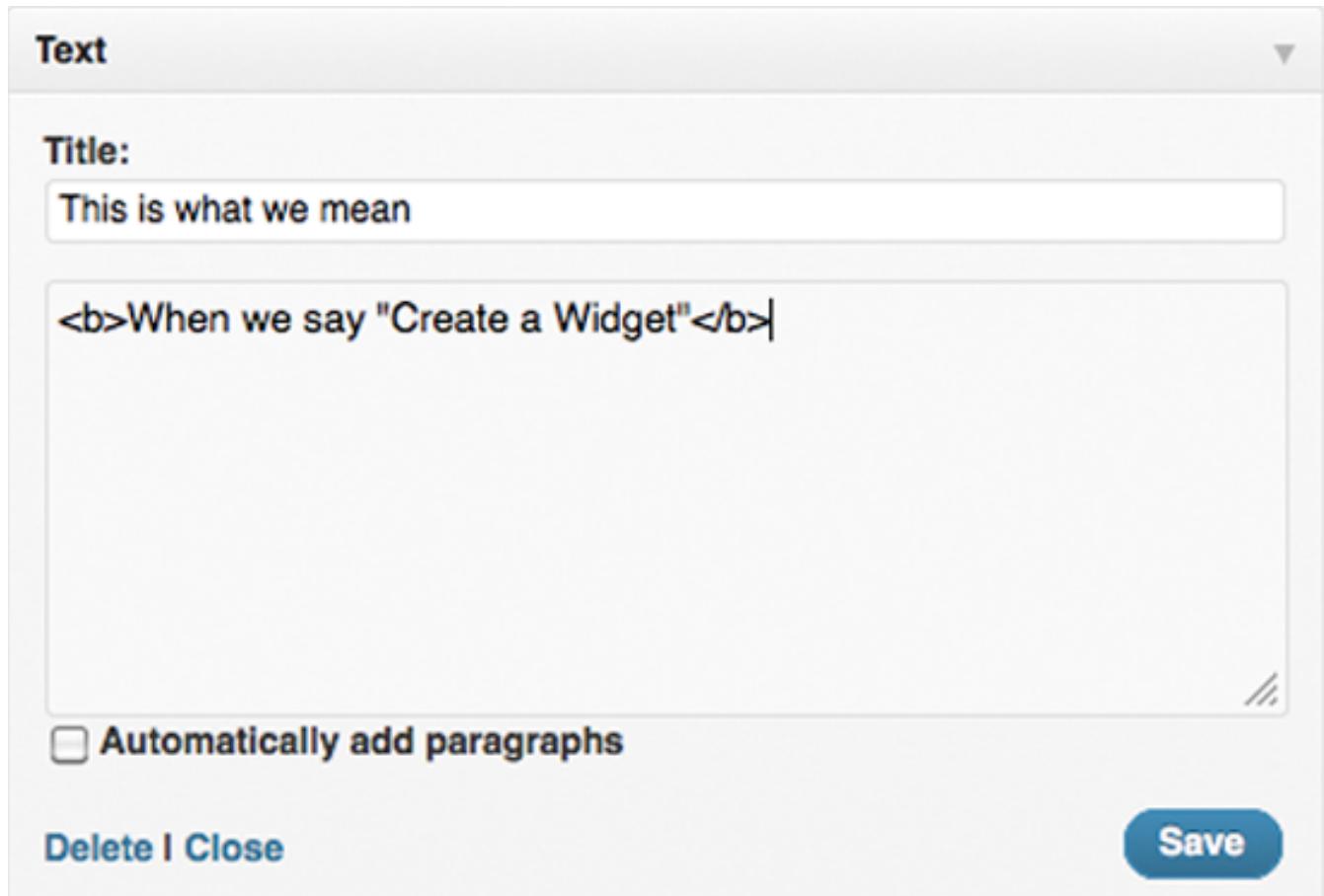
Text

Title:
This is what we mean

When we say "Create a Widget"

Automatically add paragraphs

[Delete](#) | [Close](#) Save



The blank text widget is a simple yet potentially deceptive name for a powerful tool.

So, I always like to see a description such as, “Use the ‘Text’ widget to create this option. You can simply add raw HTML into the blank box and drag it to your sidebars. This will then work just like any other widget.”

ALWAYS PROVIDE DOCUMENTATION FOR VIDEO TUTORIALS

Without a doubt, video is a massive help for confused developers. It provides detail-rich, play-by-play instruction that carefully guides the viewer through the concepts in the tutorial. Just be sure to accompany the video with detailed textual documentation. Otherwise, people will repeatedly have to rewind and squint at the screen just to copy your instructions. That’s an easy way to lose fans.

Treat the video as an aid, not as the main event. [This tutorial on Lifehacker](#), though not specifically for WordPress, illustrates this principle perfectly.

UPDATE YOUR TUTORIAL AS NEEDED

Keep your guides relevant and dynamic. Too often, tutorial writers will clarify major points in the comments section of their page, while the tutorial itself remains static. Or they just ignore the page entirely, leaving now-irrelevant guides to linger on the Internet.

Keeping in touch with your audience is wonderful, but giving new readers the best possible experience is also important. Don't expect people to comb through two years' worth of comments to find your changes.

Make sure your supplemental links remain relevant. Nothing is worse than reading a tutorial from 2007 and seeing the words "With a simple change, it should look like [this!](#)" Surely in 2007 that link was perfect, but if it leads to an unrelated page in 2011, it will undermine your entire article.

TELL THEM WHERE TO CODE

Make sure that newbies are tweaking code in the right place. Point out that, in general, they shouldn't edit files from within the WordPress dashboard. That leaves little room for error, and if the coder isn't careful, they could lose hours of progress.

The screenshot shows the Coda interface. At the top, there's a menu bar with icons for Sites, Edit, Preview, CSS, Terminal, Books, and a search bar labeled 'Search Files'. Below the menu is a toolbar with buttons for creating new files ('+') and opening existing ones ('o'). The main window has two panes: 'Local' on the left and 'Remote' on the right. The 'Local' pane shows a file structure under 'user': 'images', 'new', 'WordpressFIT' (containing '404.php', 'comments.php', 'Fitergize.html', 'footer.php', 'functions.php', 'header.php'), 'images' (containing 'index.php', 'l_sidebar.php', 'LeftColumnBar.png', 'page.php', 'r_sidebar copy.php', 'r_sidebar.php', 'sidebar.php', 'sidebar1.php', 'single.php', 'style copy.css', 'style.css', 'stylecopy.css', 'styleREGULAR.css'), 'old models' (containing '_WordpressFIT', '_WordpressFIT.SUN', '_WordpressFIT.SUN1'). The 'Remote' pane shows the contents of the file 'adv-beg-guide' which contains HTML code for a WordPress tutorial. The code includes sections for the logo, content, post, and sidebar, along with descriptive text for advanced beginners.

A brief glimpse at the SFTP- and FTP-enabled one-stop code editor, Coda.

Instead, teach them to use an SFTP- or FTP-enabled editor, such as Coda or Dreamweaver. It's a safer, fixable way to correct any mistakes that arise.

TEACH THEM HOW TO TEST

This last point is just a personal preference that I wish more people would do. One of the best things about basic HTML and CSS is that you can easily test them locally by simply reloading the browser. When you jump into WordPress, this testing process becomes significantly more complicated.

Advanced beginners will likely be lost once they realize that they can't test by simply dragging their WordPress creations into a browser. This leads many new coders to test their unfinished creations on production websites.

Tutorial writers should stress the importance of not testing WordPress changes on a live website. Explain the myriad benefits of designing on a risk-free local server. Just point the reader to one of the many existing server guides, and briefly mention the pitfalls of testing code on a live website. Michael Doig's article "Installing WordPress Locally Using MAMP" is one of the most useful set-up guides.

Conclusion

Whether you're writing a tutorial about WordPress or anything else, clarity is paramount. Put yourself in the reader's shoes. WordPress is built on the efforts of a wonderfully helpful community that is full of excellent tutorials and experts. But, as in any community, this has resulted in some confusing jargon and common shortcuts.

These can overwhelm new developers. Tutorial writers should avoid unnecessary jargon and always explain any references and functions that they use, no matter how basic they seem. Remember that, as the guide, your knowledge is likely far beyond that of your readers. What is obvious to you could be brand new to them.

By making your tutorials easier to understand, you'll greatly increase your own Web traffic and enrich the greater WordPress community.

OTHER RESOURCES

Here are a few tutorials that are easy to follow and that adhere to many of the points mentioned here.

-
- “[Complete WordPress Theme Guide](#),” Web Designer Wall
A great tutorial to help people get started in coding. Plus, it features an eloquent section on installing WordPress locally.
 - “[How to Make a Website: The Complete Beginner’s Guide](#),” Lifehacker
An excellent blog across the board, Lifehacker has created some absolutely phenomenal video tutorials. The documentation makes this one an expertly designed guide.
 - “[CSS Techniques: Using Sliding Doors with WordPress Navigation](#),” WP Hacks
WP Hacks is a great resource for WordPress designers. This piece is well organized and demonstrates the correct way to present code in a tutorial.
 - “[Installing WordPress Locally Using MAMP](#),” Michael Doig
This is the excellent guide to setting up WordPress using MAMP that I mentioned earlier.
 - “[How to Create a WordPress Theme from Scratch](#),” Nettuts+
Nettuts+ is always a great source of tutorials. In this one, you’ll see how to present all relevant resources in a tutorial.

Advanced Layout Templates In WordPress' Content Editor

David Hansen

As a Web designer, I often find myself building WordPress-based websites that will ultimately be updated and maintained by clients who have little to no experience working with HTML. While the TinyMCE rich-text editor is great for giving Web content managers of any skill level the tools they need to easily style and publish their posts to a degree, creating anything beyond a single column of text with a few floated images generally requires at least a basic understanding of HTML.



This chapter shows you an easy-to-implement trick that enables even the least tech-savvy of clients to manage multi-column content layouts within the comfort of the WYSIWIG editor. And for you advanced users, it's still a great way to standardize and streamline your content entry.

Creating A Custom Layout

All we're really going to do here is inject a few HTML elements into the editing window and style them. WordPress' `default_content` filter allows us to insert set content into any post as soon as it's created so that our clients don't have to. This filter is also great for adding boilerplate text to posts.

THE BACK END

By adding the following to `functions.php`, each new post we create will come pre-stocked with two divs, classed `content-col-main` and `content-col-side`, respectively. I should note now that this code has been tested only in WordPress version 3.0 and up:

```
<?php  
add_filter( 'default_content', 'custom_editor_content' );  
function custom_editor_content( $content ) {  
    $content = '  
        <div class="content-col-main">  
  
            This is your main page content  
  
            &nbsp;  
  
        </div>  
        <div class="content-col-side">
```

```
This is your sidebar content

</div>
';

return $content;
}

?>
```

A couple of things to note:

- The `default_content` filter is fired only when a new post is created; any posts or pages that existed before you added this code will not receive this content.
- The line spacing and additional are not essential, but I've found them to be useful for preventing a few of TinyMCE's little quirks.

Now we just need to give it some style. Add the following to `functions.php`:

```
<?php
add_editor_style( 'editor-style.css' );
?>
```

The `add_editor_style()` function looks for the specified style sheet and applies any CSS it contains to the content in our TinyMCE editing window. If you don't specify the name of a style sheet, it will look for `editor-style.css` by default, but for the purpose of this chapter, I've written it out. Create a style sheet named `editor-style.css`, and place it in the theme folder, with the following styles:

```
body {  
    background: #f5f5f5;  
}  
  
.content-col-main {  
    float:left;  
    width:66%;  
    padding:1%;  
    border: 1px dotted #ccc;  
    background: #fff;  
}  
  
.content-col-side {  
    float:right;  
    width:29%;  
    padding:1%;  
    border: 1px dotted #ccc;  
    background: #fff;  
}  
  
img { /* Makes sure your images stay within their columns */  
    max-width: 100%;  
    width: auto;  
    height: auto;  
}
```

Now, when you create a new post, you will see two columns that you can type or paste your content into:



Add New Page

Enter title here

Permalink:

Edit

Upload/Insert



Visual HTML

This is your main page content

This is your sidebar content

Path:

Word count: 11

Draft saved at 5:09:50 pm.

This basic multi-column template will now appear any time you create a new page or post.

And there you have it: a simple multi-column template in your content editor. You can go back and edit the `default_content` and `editor-styles.css` to adapt the content layout to your needs:

Tactical Bacon

Permalink: [http://example.com/tactical-bacon](#)

Edit

[View Product](#)

Upload/Insert

Visual [HTML](#)



Description:

It's Tactical Bacon in a can. Fully cooked and fully prepared. 10+ year shelf life. Perfect for camping, hunting, emergency preparedness, etc. Don't be caught without Tac Bac! This is seriously good bacon and the can is jam packed FULL. Eat it cold or heat it up in the microwave. There are approximately 54 slices in the can which is the rough equivalent of 4 or 5 packages of store bought bacon or 3 pounds of raw bacon. It really is the ultimate tactical accessory!

Key Features:

- Delicious
- Low-fat
- Tactical
- Bacon

Nutritional Facts

Serving Size	14g (3 slices)
Amount Per Serving	
Calories	60
Calories from Fat	40
Total Fat	5.0 g
Saturated Fat	2.0 g
Cholesterol	15 mg
Sodium	190 mg
Total Carbohydrate	<1 g
Protein	5.0 g
Iron	4 %

* Percent Daily Values are based on a 2,000 calorie diet. Your daily values may be higher or lower depending on your calorie needs.

Path:

Word count: 141

Use this technique to create your own layout templates, customized to your content.

THE FRONT END

When your post is displayed on the front end of the website, the content will still appear in a single column, as before. The styles you wrote out in editor-style.css do not get passed to the front end of the website. However, by viewing the page source, you'll see that the divs we created with our custom_editor_content() function have been passed through and are wrapping the different sections of the content. Simply open style.css (or whatever style sheet you're using for your theme) and style to your heart's desire.

The screenshot shows a product page for "CMMG TACTICAL BACON 9OZ COOKED" priced at \$22.99. The page includes a navigation bar with Home, Edibles, On-the-Go, and the product link. It features a large image of the bacon can, a quantity selector set to 1, and an orange "ADD TO MY CART" button. Below the main image are two smaller thumbnail images of the bacon. A red arrow points from the text below to these thumbnails, indicating they are part of a jQuery Cycle slideshow. The page also includes a "Description" section with a detailed text about the product, a "Nutrition Facts" table, and a sidebar with product highlights.

Home > Edibles > On-the-Go > CMMG Tactical Bacon 9oz Cooked

CMMG TACTICAL BACON 9OZ COOKED -\$22.99

Quantity: 1 [+ ADD TO MY CART](#)

Description:

It's Tactical Bacon in a can. Fully cooked and fully prepared. 10+ year shelf life. Perfect for camping, hunting, emergency preparedness, etc. Don't be caught without Tac Bac! This is seriously good bacon and the can is jam packed FULL. Eat it cold or heat it up in the microwave. There are approximately 54 slices in the can which is the rough equivalent of 4 or 5 packages of store bought bacon or 3 pounds of raw bacon. It really is the ultimate tactical accessory!

Nutrition Facts

Serving Size	14g (3 slices)	
Amount Per Serving		
Calories	60	
Calories from Fat	40	
	% Daily Value*	
Total Fat	5.0 g	8 %
Saturated Fat	2.0 g	10 %
Cholesterol	15 mg	5 %
Sodium	190 mg	8 %
Total Carbohydrate	< 1 g	0 %
Protein	5.0 g	10 %
Iron		4 %

* Percent Daily Values are based on a 2,000 calorie diet. Your daily values may be higher or lower depending on your calorie needs.

Tools like jQuery Cycle can use the photos you place in a specific element to create slideshows

This technique applies not only to the visual layout of content. Use JavaScript to target specific containers to make on-the-fly slideshows and other dynamic effects.

GET MORE FROM YOUR TEMPLATES

Beyond just opening up new styling possibilities, this technique can also be used to create objects to target later with JavaScript.

The screenshot illustrates a WordPress editor interface and its corresponding website output. In the editor, a rich text area contains a paragraph about curriculums and tabs for 'A Letter from the Dean', 'Majors', and 'Convergence Journalism'. A red arrow points from the 'Majors' tab in the editor to the same tab's representation in the rendered website below. The rendered website shows a header with navigation links like 'ACADEMIC PROGRAMS', 'FOR PROSPECTIVE STUDENTS', 'THE SCHOOL', etc. Below the header, there are three tabs: 'Undergraduate Programs', 'Majors', 'Prospective Students', 'Study Abroad', and 'Information for Parents'. The 'Undergraduate Programs' tab is active, displaying a photo of students in a classroom and a paragraph about curriculums. The 'Majors' tab is also visible, showing a photo of a man holding a magnifying glass.

In the example above, we were able to turn a series of content areas into more easily digestible tabbed sections for the user, while still allowing the administrator to update all of the information on one page. These are just a few of the many ways you can take your WordPress templates further.

Templates For Templates

The code above will simply apply the same layout and styling to all of your posts, pages, custom posts... anywhere the TinyMCE editor appears. This is probably not ideal. By adding conditional statements to the `custom_editor_content()` function above, you can serve a different default layout template to each of your post types:

```
<?php  
add_filter( 'default_content', 'custom_editor_content' );  
  
function custom_editor_content( $content ) {  
    global $current_screen;  
    if ( $current_screen->post_type == 'page' ) {  
        $content = '  
  
        // TEMPLATE FOR YOUR PAGES  
  
   ';  
    }  
    elseif ( $current_screen->post_type == 'POSTTYPE' ) {  
        $content = '  
  
        // TEMPLATE FOR YOUR CUSTOM POST TYPE  
  
   ';  
    }  
    else {  
        $content = '  
  
        // TEMPLATE FOR EVERYTHING ELSE  
  
   ';  
    }  
}
```

```
    }

    return $content;

}

?>
```

You can style all of your default content elements in the editor-style.css file, but if you'd like to use a different style sheet entirely for each post type, you can do so with this snippet [from WPStorm](#):

```
<?php

function custom_editor_style() {
    global $current_screen;
    switch ($current_screen->post_type) {
        case 'post':
            add_editor_style('editor-style-post.css');
            break;
        case 'page':
            add_editor_style('editor-style-page.css');
            break;
        case '[POSTTYPE]':
            add_editor_style('editor-style-[POSTTYPE].css');
            break;
    }
}

add_action( 'admin_head', 'custom_editor_style' );
?>
```

Add the above to your functions.php file, and then create the editor-style-[POSTTYPE].css files to use different style sheets for the corresponding post types. Just replace [POSTTYPE] with the name of your custom post type. Extend the code above by adding new cases for each additional post type.

Alternatively, you could use the following code to automatically look for a style sheet named editor-style- followed by the name of the post type that you're currently editing. Again, just make sure that the suffix of the new style sheet you create matches exactly the name of the post type.

```
<?php

function custom_editor_style() {
    global $current_screen;
    add_editor_style(
        array(
            'editor-style.css',
            'editor-style-' . $current_screen->post_type . '.css'
        )
    );
}

add_action( 'admin_head', 'custom_editor_style' );
?>
```

(In this snippet, editor-style.css will also be included on all post-editing pages, in addition to the style sheet that is specific to that post type, which will be named editor-style-[POSTTYPE].css.)

Conclusion

While this method does have its drawbacks—it assumes you already know the layout that your client wants to give their content, and the layout structures cannot be easily edited by the client themselves—it does enable you to create more interesting sandboxes for your client to play in, while encouraging a standardized format for the content.

If the client decides they don't want to use a pre-defined container for a particular post, they can simply click inside the container and hit Backspace until all the content is gone, and then hit Backspace once more, and TinyMCE will delete the div wrapper, leaving a clean slate for them to work with. I hope you've found this little technique useful.

The Authors

Daniel Pataki

Daniel Pataki is a guitar wielding web developer obsessed with web technology, best practices and the awesomeness of WordPress. Take a look at his [personal page](#) or follow him on twitter: [@danielpataki](#)

Dave Donaldson

Dave lives in Columbus, OH and is one of the founders of [Max Foundry](#), a company that makes WordPress plugins for [landing pages](#), [squeeze pages](#), [sales pages](#), and [A/B testing](#). You can [follow Dave on Twitter](#) and [on his blog](#), where he writes about living the bootstrapped startup life.

David Hansen

David Hansen is a designer / front-end developer for [Delta Systems Group](#) in Columbia, Missouri.

Jacob Goldman

Jacob M (Jake) Goldman is the owner of [10up](#) LLC, a web development and strategy agency with a focus on making content management easy and fun. 10up's clients range from small local businesses to major [WordPress.com](#) VIP clients like TechCrunch. You can find his insights and development tips by following him on Twitter [@jakemgold](#)

Jean-Baptiste Jung

Jean-Baptiste Jung is a 29-year-old blogger from Belgium, who blogs about Web Development on [Cats Who Code](#), about WordPress at [WpRecipes](#) and about blogging on [Cats Who Blog](#). You can stay in touch with Jean by following him on [Twitter](#).

Scott Meaney

I'm a web, social media and SEO publicist for several consumer electronics companies. Additionally, I write tech and video game news and reviews for Newegg.com's official blog and several other outlets. Follow me on Twitter @scottmeaney.