



WP

Customizing WordPress

Imprint

© 2016 Smashing Magazine GmbH, Freiburg, Germany

ISBN (PDF): 978-3-945749-41-8

Cover Design: Veerle Pieters

eBook Strategy and Editing: Vitaly Friedman

Technical Editing: Cosima Mielke

Planning and Quality Control: Vitaly Friedman, Iris Lješnjanić

Tools: Elja Friedman

Syntax Highlighting: Prism by Lea Verou

Idea & Concept: Smashing Magazine GmbH

About This Book

For a lot of people, WordPress is the entry into web development. And a lot of them don't stop there. They want more control over their WordPress site, so customizing design and functionality is the next logical step. If WordPress got you hooked, and you want to get more out of it to tailor your site more to your needs and ideas, then this eBook is for you.

To start out, you will learn to build custom page templates and extend WordPress' flexibility with custom post types. Later, our expert authors will provide insights into customizing tree-like data structures to make them fit your particular needs, as well as tips to replace the regular custom field interface with something more powerful and user-friendly. You'll also learn to build an advanced notification system to reach your users, and, last, but not least, we'll dive deep into building, maintaining, and deploying WordPress plugins. WordPress' flexible structure is predestined for customization. So make use of this grand potential to build your projects the way you imagine them to be.

TABLE OF CONTENTS

A Detailed Guide To WordPress Custom Page Templates	5
Extending WordPress With Custom Content Types.....	48
Building A Custom Archive Page For WordPress	71
Customizing Tree-Like Data Structures In WordPress With The Walker Class	94
Extending Advanced Custom Fields With Your Own Controls	133
Building An Advanced Notification System For WordPress	151
How To Use Autoloading And A Plugin Container In WordPress Plugins	181
How To Deploy WordPress Plugins With GitHub Using Transients	221
About The Authors	241

A Detailed Guide To WordPress Custom Page Templates

BY NICK SCHÄFERHOFF 🍷

I like to think of WordPress as the gateway drug of web development. Many people who get started using the platform are initially merely looking for a comfortable (and free) way to create a simple website. Some Googling and consultation of the [WordPress Codex](http://codex.wordpress.org)¹ later, it's done and that should be it. Kind of like "I'm just going to try it once."

However, a good chunk of users don't stop there. Instead, they get hooked. Come up with more ideas. Experiment. Try out new plugins. Discover [Firebug](https://addons.mozilla.org/en-us/firefox/addon/firebug/)². Boom. Soon there is no turning back. Does that sound like your story? As a WordPress user it is only natural to want more and more control over your website. To crave custom design, custom functionality, custom everything.

Luckily, WordPress is built for exactly that. Its flexible structure and compartmentalized architecture allows anyone to change practically anything on their site.

Among the most important tools in the quest for complete website control are page templates. They allow users to dramatically alter their website's design and functionality. Want a customized header for your front

1. <http://codex.wordpress.org>

2. <https://addons.mozilla.org/en-us/firefox/addon/firebug/>

page? Done. An additional sidebar only for your blog page? No problem. A unique 404 error page? Be. My. Guest.

If you want to know how WordPress page templates can help you achieve that, read on. But first, a little background information.

Template Files In WordPress

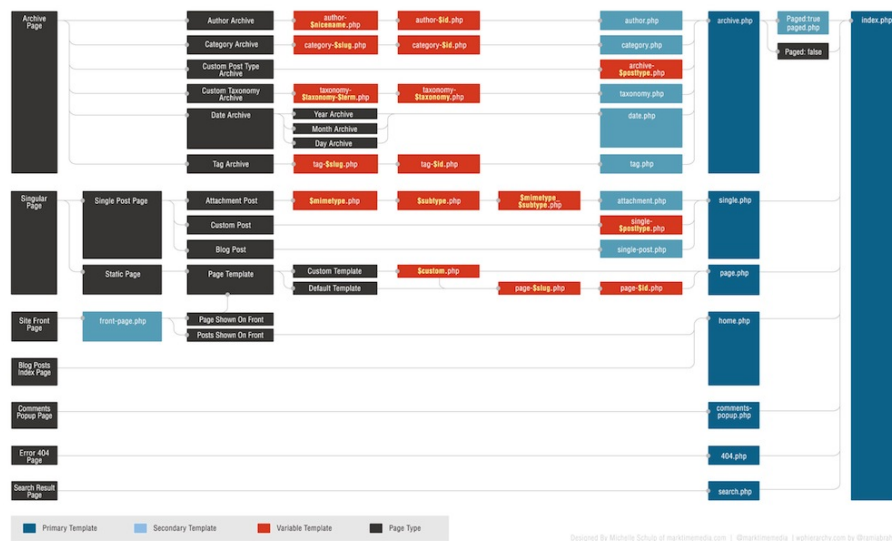
What are we talking about when we speak of templates in the context of WordPress? The short version is that templates are files which tell WordPress how to display different types of content.

The slightly longer version: every time someone sends a request to view part of your website, the WordPress platform will figure out what content they want to see and how that specific part of your website should be rendered.

For the latter, WordPress will attempt to use the most appropriate template file found within your theme. Which one is decided on the basis of a set order, the WordPress template hierarchy³. You can see what this looks like in the screenshot below or in this interactive version⁴.

³. http://codex.wordpress.org/Template_Hierarchy#The_Template_Hierarchy_In_Detail

⁴. <http://wphierarchy.com/>



The WordPress template hierarchy.
(Image credit: [WordPress Codex](http://codex.wordpress.org/)⁵)(View large version⁶)

The template hierarchy is a list of template files WordPress is familiar with that are ranked to determine which file takes precedence over another.

You can think of it as a decision tree. When WordPress tries to decide how to display a given page, it works its way down the template hierarchy until it finds the first template file that fits the requested page. For example, if somebody attempted to access the address <http://yoursite.com/category/news>, WordPress would look for the correct template file in this order:

1. `category-{slug}.php`: in this case `category-news.php`

5. <http://codex.wordpress.org/>

6. <https://media-mediadna.netdna-ssl.com/wp-content/uploads/2015/05/01-wp-template-hierarchy-opt.jpg>

2. *category-{id}.php*: if the category ID were 5, WordPress would try to find a file named *category-5.php*
3. *category.php*
4. *archive.php*
5. *index.php*

At the bottom of the hierarchy is *index.php*. It will be used to display any content which does not have a more specific template file attached to its name. If a template file ranks higher in the hierarchy, WordPress will automatically use that file in order to display the content in question.

PAGE TEMPLATES AND THEIR USE

For pages, the standard template is usually the aptly named *page.php*. Unless there is a more specific template file available (such as *archive.php* for an archive page), WordPress will use *page.php* to render the content of all pages on your website.

However, in many cases it might be necessary to change the design, look, feel or functionality of individual parts of your website. This is where page templates come into play. Customized page templates allow you to individualize any part of your WordPress site without affecting the rest of it.

You might have already seen this at work. For example, many WordPress themes today come with an option to change your page to full width, add a second sidebar or switch the sidebar's location. If that is the case for yours,

it was probably done through template files. There are several ways to accomplish this and we'll go over them later.

First, however, a word of caution: since working with templates involves editing and changing files in your active theme, it's always a good idea to go with a child theme when making these kinds of customizations. That way you don't run the danger of having your changes overwritten when your parent theme gets updated.

How To Customize Any Page In WordPress

There are three basic ways to use custom page templates in WordPress: adding conditional statements to an existing template; creating specific page templates which rank higher in the hierarchy; and directly assigning templates to specific pages. We will take a look at each of these in turn.

USING CONDITIONAL TAGS IN DEFAULT TEMPLATES

An easy way to make page-specific changes is to add WordPress's many conditional tags⁷ to a template already in use. As the name suggests, these tags are used to create functions which are only executed if a certain condition is met. In the context of page templates, this would be

⁷. http://codex.wordpress.org/Conditional_Tags

something along the line of “*Only perform action X on page Y.*”

Typically, you would add conditional tags to your theme’s *page.php* file (unless, of course, you want to customize a different part of your website). They enable you to make changes limited to the homepage, front page, blog page or any other page of your site.

Here are some frequently used conditional tags:

1. **`is_page()`**: to target a specific page. Can be used with the page’s ID, title, or URL/name.
2. **`is_home()`**: applies to the home page.
3. **`is_front_page()`**: targets the front page of your site as set under Settings → Reading.
4. **`is_category()`**: condition for a category page. Can use ID, title or URL/name like **`is_page()`** tag.
5. **`is_single()`**: for single posts or attachments.
6. **`is_archive()`**: conditions for archive pages.
7. **`is_404()`**: applies only to 404 error pages.

For example, when added to your *page.php* in place of the standard **`get_header()`**; tag, the following code will load a custom header file named *header-shop.php* when displaying the page *http://yoursite.com/products*.

```
if ( is_page('products') ) {
    get_header( 'shop' );
} else {
```

```
get_header();  
}
```

A good use case for this would be if you have a shop on your site and you need to display a different header image or customized menu on the shop page. You could then add these customization in *header-shop.php* and it would show up in the appropriate place.

However, conditional tags are not limited to one page. You can make several statements in a row like so:

```
if ( is_page('products') ) {  
    get_header( 'shop' );  
} elseif ( is_page( 42 ) ) {  
    get_header( 'about' );  
} else {  
    get_header();  
}
```

In this second example, two conditions will change the behavior of different pages on your site. Besides loading the aforementioned shop-specific header file, it would now also load a *header-about.php* on a page with the ID of 42. For all other pages the standard header file applies.

To learn more about the use of conditional tags, the following resources are highly recommended:

- [WordPress Codex: Conditional Tags](http://codex.wordpress.org/Conditional_Tags)⁸

⁸. http://codex.wordpress.org/Conditional_Tags

- [ThemeLab: The Ultimate Guide to WordPress Conditional Tags⁹](#)

CREATING PAGE-SPECIFIC FILES IN THE WORDPRESS HIERARCHY

Conditional tags are a great way to introduce smaller changes to your page templates. Of course, you can also create larger customizations by using many conditional statements one after the other. I find this a very cumbersome solution, however, and would opt for designated template files instead.

One way to do this is to exploit the WordPress template hierarchy. As we have seen, the hierarchy will traverse a list of possible template files and choose the first one it can find that fits. For pages, the hierarchy looks like this:

- Custom page template
- *page-{slug}.php*
- *page-{id}.php*
- *page.php*
- *index.php*

In first place are custom page templates which have been directly assigned to a particular page. If one of those exists, WordPress will use it no matter which other tem-

⁹. <http://www.themelab.com/ultimate-guide-wordpress-conditional-tags/>

plate files are present. We will talk more about custom page templates in a bit.

After that, WordPress will look for a page template that includes the slug of the page in question. For example, if you include a file named *page-about.php* in your theme files, WordPress will use this file to display your 'About' page or whichever page can be found under <http://www.yoursite.com/about>.

Alternatively, you can achieve the same by targeting your page's ID. So if that same page has an ID of 5, WordPress will use the template file *page-5.php* before *page.php* if it exists; that is, only if there isn't a higher-ranking page template available.

(By the way, you can find out the ID for every page by hovering over its title under 'All Pages' in your WordPress back-end. The ID will show up in the link displayed by your browser.)

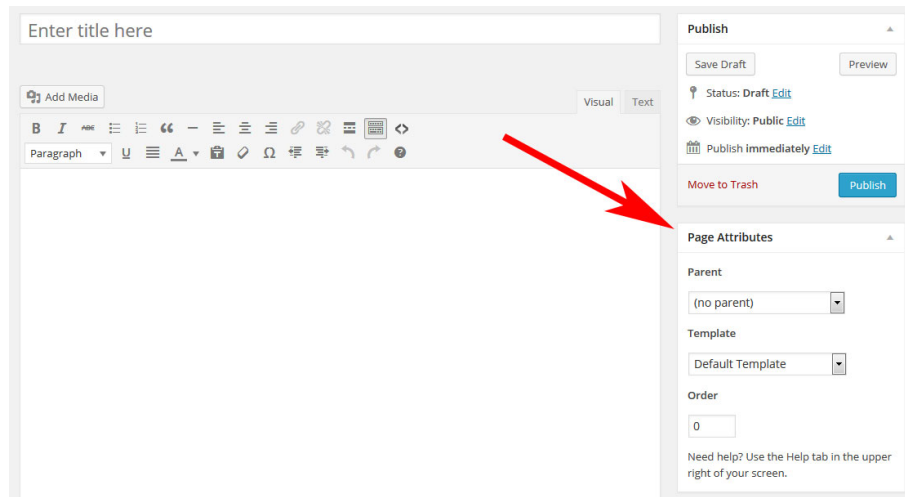
ASSIGNING CUSTOM PAGE TEMPLATES

Besides providing templates in a form that WordPress will use automatically, it is also possible to manually assign custom templates to specific pages. As you can see from the template hierarchy, these will trump any other template file present in the theme folder.

Just like creating page-specific templates for the WordPress hierarchy, this requires you to provide a template file and then link it to whichever page you want to use it for. The latter can be done in two different ways you might already be familiar with. Just in case you aren't, here is how to do it.

1. Assigning Custom Page Templates From The WordPress Editor

In the WordPress editor, you find an option field called ‘Page Attributes’ with a drop-down menu under ‘Template’.



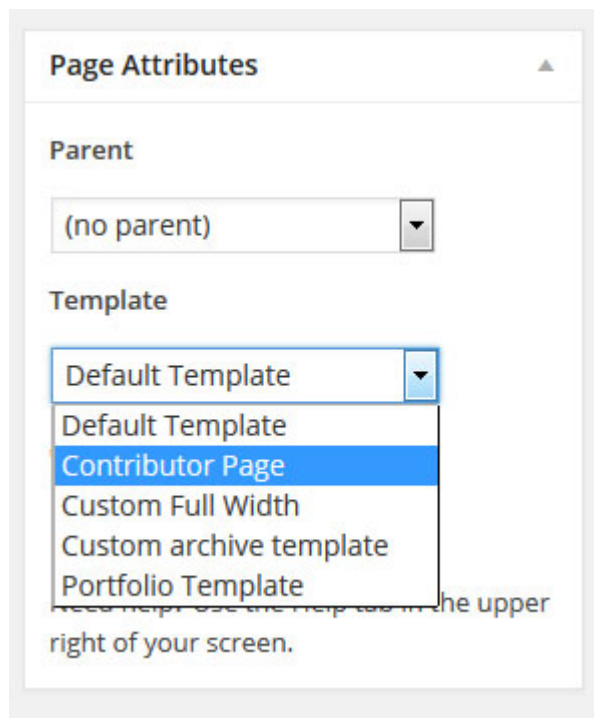
Page Attributes in the WordPress editor.

Clicking on it will give you a list of available page templates on your WordPress website. Choose the one you desire, save or update your page and you are done.

2. Setting A Custom Template Via Quick Edit

The same can also be achieved without entering the WordPress editor. Go to ‘All Pages’ and hover over any item in the list there. A menu will become visible that includes the ‘Quick Edit’ item.

Click on it to edit the page settings directly from the list. You will see the same drop-down menu for choosing a different page template. Pick one, update the page and you are done.



Available templates under Page Attributes.

Not so hard after all, is it? But what if you don't have a custom page template yet? How do you create it so that your website looks exactly the way you want it? Don't worry, that's what the next part is all about.

A Step-By-Step Guide To Creating Custom Page Templates

Putting together customized template files for your pages is not that hard but here are a few details you have to pay attention to. Therefore, let's go over the process bit-by-bit.

1. FIND THE DEFAULT TEMPLATE

A good way is to start by copying the template which is currently used by the page you want to modify. It's easier

to modify existing code than to write an entire page from scratch. In most cases this will be the *page.php* file.

(If you don't know how to find out which template file is being used on the page you want to edit, the plugin What The File¹⁰ will prove useful.)

I will be using the Twenty Twelve theme for demonstration. Here is what its standard page template looks like:

```
<?php
/**
 * The template for displaying all pages
 *
 * This is the template that displays all pages
 * by default.
 * Please note that this is the WordPress construct of
 * pages and that other 'pages' on your WordPress site
 * will use a different template.
 *
 * @package WordPress
 * @subpackage Twenty_Twelve
 * @since Twenty Twelve 1.0
 */
get_header(); ?>

<div id="primary" class="site-content">
    <div id="content" role="main">
```

¹⁰. <https://wordpress.org/plugins/what-the-file/>


```

<?php while ( have_posts() ) : the_post(); ?>
    <?php get_template_part( 'content', 'page' );
    ?>
    <?php comments_template( '', true ); ?>
<?php endwhile; // end of the loop. ?>

```

```

</div><!-- #content -->

```

```

</div><!-- #primary -->

```

```

<?php get_sidebar(); ?>

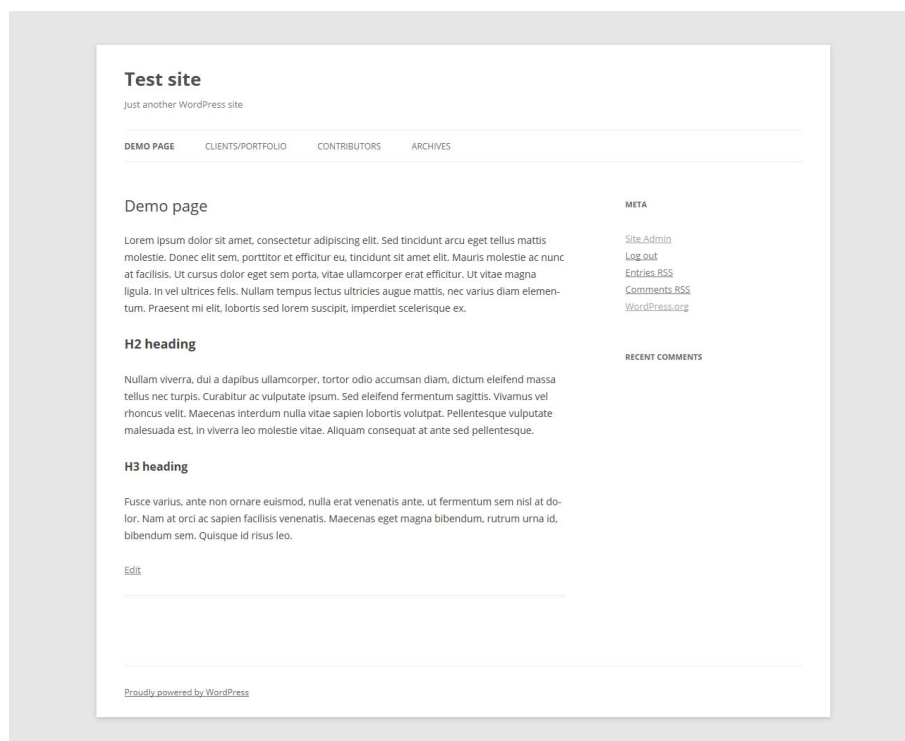
```

```

<?php get_footer(); ?>

```

As you can see, nothing too fancy here: the usual calls for the header and footer, and the loop in the middle. The page in question looks like this:



The default page template in the Twenty Twelve theme.

2. COPY AND RENAME THE TEMPLATE FILE

After identifying the default template file, it's time to make a copy. We will use the duplicated file in order to make the desired changes to our page. For that we will also have to rename it. Can't have two files of the same name, that's just confusing for everyone.

You are free to give the file any name you like as long as it doesn't start with any of the reserved theme file-names¹¹. So don't be naming it *page-something.php* or anything else that would make WordPress think it is a dedicated template file.

It makes sense to use a name which easily identifies what this template file is used for, such as *my-custom-template.php*. In my case I will go with *custom-full-width.php*.

3. CUSTOMIZE THE TEMPLATE FILE HEADER

Next we have to tell WordPress that this new file is a custom page template. For that, we will have to adjust the file header in the following way:

```
<?php
/*
 * Template Name: Custom Full Width
 * Description: Page template without sidebar
 */

// Additional code goes here...
```

¹¹. http://codex.wordpress.org/Page_Templates#Filenames

The name under ‘Template Name’ is what will be displayed under ‘Page Attributes’ in the WordPress editor. Make sure to adjust it to your template name.

4. CUSTOMIZE THE CODE

Now it’s time to get to the meat and potatoes of the page template: the code. In my example, I merely want to remove the sidebar from my demo page.

This is relatively easy, as all I have to do is remove `<?php get_sidebar(); ?>` from my page template since that’s what is calling the sidebar. As a consequence, my custom template ends up looking like this:

```
<?php
/*
 * Template Name: Custom Full Width
 * Description: Page template without sidebar
 */

get_header(); ?>

<div id="primary" class="site-content">
    <div id="content" role="main">

        <?php while ( have_posts() ) : the_post(); ?>
            <?php get_template_part( 'content', 'page' ); ?>
            <?php comments_template( '', true ); ?>
        <?php endwhile; // end of the loop. ?>

    </div><!-- #content -->
```

```
</div><!-- #primary -->
```

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

5. UPLOAD THE PAGE TEMPLATE

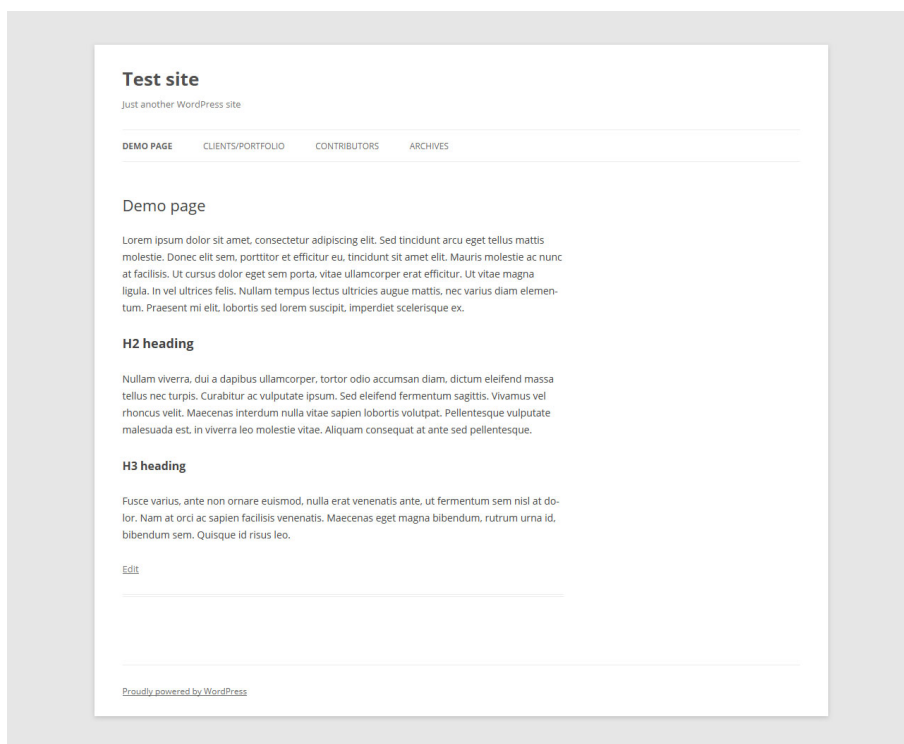
After saving my customized file, it is now time to upload it to my website. Custom page templates can be saved in several places to be recognized by WordPress:

- Your active (child) theme's folder
- The folder of your main parent theme
- A subfolder within either of these

I personally like to create a folder named *page_templates* in my child theme and place any customized templates in there. I find this easiest to retain an overview over my files and customizations.

6. ACTIVATE THE TEMPLATE

As a last step, you need to activate the page template. As mentioned earlier, this is done under Page Attributes → Templates in the WordPress editor. Save, view the page and voilà! Here is my customized page without a sidebar:



Customized page template without the sidebar.

Not so hard, is it? Don't worry, you will quickly get the hang of it. To give you a better impression of what to use these page templates for, I will demonstrate additional use cases (including the code) for the remainder of the article.

Five Different Ways To Use Page Templates

As already mentioned, page templates can be employed for many different purposes. You can customize pretty much anything on any page with their help. Only your imagination (and coding abilities) stand in your way.

1. FULL-WIDTH PAGE TEMPLATE

The first case we will look at is an advanced version of the demo template we created above. Up there, we already removed the sidebar by deleting `<?php get_sidebar(); ?>` from the code. However, as you have seen from the screenshot this does not actually result in a full-width layout since the content section stays on the left.

To address this, we need to deal with the CSS, in particular this part:

```
.site-content {
    float: left;
    width: 65.1042%;
}
```

The `width` attribute limits the element which holds our content to 65.1042% of the available space. We want to increase this.

If we just change it to 100%, however, this will affect all other pages on our site, which is far from what we want. Therefore, the first order here is to change the primary `div`'s class in our custom template to something else, like `class="site-content-fullwidth"`. The result:

```
<?php
/*
 * Template Name: Custom Full Width
 * Description: Page template without sidebar
 */

get_header(); ?>
```

```

<div id="primary" class="site-content-fullwidth">
  <div id="content" role="main">

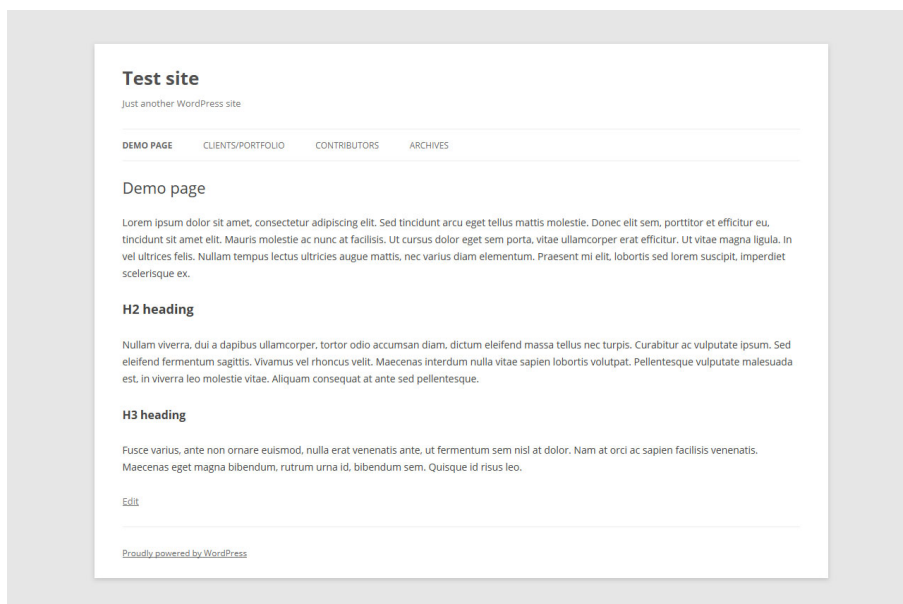
    <?php while ( have_posts() ) : the_post(); ?>
      <?php get_template_part( 'content', 'page' ); ?>
      <?php comments_template( '', true ); ?>
    <?php endwhile; // end of the loop. ?>

  </div><!-- #content -->
</div><!-- #primary -->

<?php get_footer(); ?>

```

Now we can adjust the CSS for our new custom class. As a result, the content now stretches all the way across the screen.



The custom page template at full width.

```
.site-content-fullwidth {
    float: left;
    width: 100%;
}
```

2. DYNAMIC 404 ERROR PAGE WITH WIDGET AREAS

The 404 error page is where every person lands who tries to access a page on your website that doesn't exist, be it through a typo, a faulty link or because the page's permalink has changed.

Despite the fact that getting a 404 is disliked by everyone on the Internet, if you are running a website the 404 error page is of no little importance. Its content can be the decisive factor on whether someone immediately abandons your site or sticks around and checks out your other content.

Coding a customized error page from scratch is cumbersome, especially if you are not confident in your abilities. A better way is to build widget areas into your template so you can flexibly change what is displayed there by drag and drop.

For this we will grab and edit the *404.php* file that ships with Twenty Twelve (template hierarchy, remember?). However, before we change anything on there, we will first create a new widget by inserting the following code into our *functions.php* file:

```
register_sidebar( array(
    'name' => '404 Page',
    'id' => '404',
```



```

    'description' => __( 'Content for your 404 error
page goes here.' ),
    'before_widget' => '<div id="error-box">',
    'after_widget' => '</div>',
    'before_title' => '<h3 class="widget-title">',
    'after_title' => '</h3>'
) );

```

This should display the newly created widget in your WordPress back-end. To make sure that it actually pops up on the site, you need to add the following line of code to your 404 page in the appropriate place:

```
<?php dynamic_sidebar( '404' ); ?>
```

In my case, I want to replace the search form (`<?php get_search_form(); ?>`) inside the template with my new widget, making for the following code:

```

<?php
/**
 * The template for displaying 404 pages (Not Found)
 *
 * @package WordPress
 * @subpackage Twenty_Twelve
 * @since Twenty Twelve 1.0
 */

```

```
get_header(); ?>
```

```

<div id="primary" class="site-content">
    <div id="content" role="main">

```

```

<article id="post-0" class="post error404
no-results not-found">
    <header class="entry-header">
        <h1 class="entry-title"><?php _e( 'This is
        somewhat embarrassing, isn&rsquo;t it?',
        'twentytwelve' ); ?></h1>
    </header>

    <div class="entry-content">
        <?php dynamic_sidebar( '404' ); ?>
    </div><!-- .entry-content -->
</article><!-- #post-0 -->

</div><!-- #content -->
</div><!-- #primary -->

<?php get_footer(); ?>

```

After uploading the template to my site, it's time to populate my new widget area (see image on the next page).

If I now take a look at the 404 error page, my newly created widgets show up there (see image on page 28).

404 Page

Content for your 404 error page goes here.

Search: Would you like to try a s...

Title:
Would you like to try a search?

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

Recent Posts: Or maybe this cou...

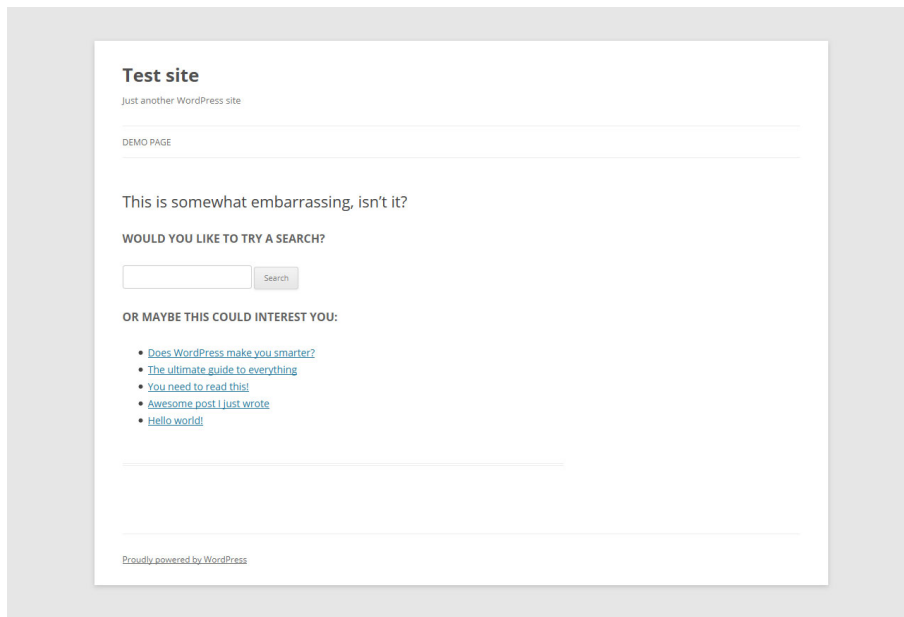
Title:
Or maybe this could interest you:

Number of posts to show: 5

☐ Display post date?

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

404 page template widget.



Customized 404 page.

3. PAGE TEMPLATE FOR DISPLAYING CUSTOM POST TYPES

Custom post types are a great way to introduce content that has its own set of data points, design and other customizations. A favorite use case for these post types are review items such as books and movies. In our case we want to build a page template that shows portfolio items.

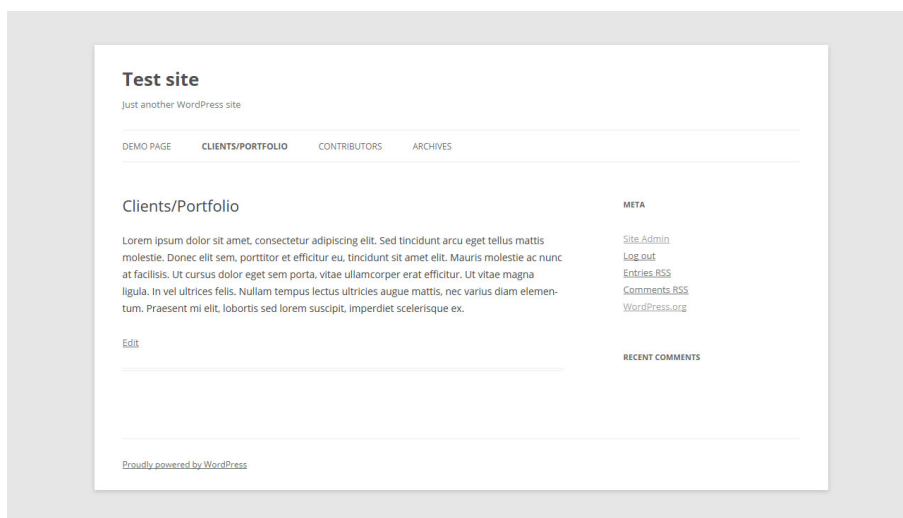
We first need to create our custom post type (CPT). This can be done manually or via plugin. One plugin option I can wholeheartedly recommend is [Types](https://wordpress.org/plugins/types/)¹². It lets you easily create custom post types and custom fields.

Install and activate Types, add a custom post, make sure its slug is 'portfolio', customize any fields you need

¹². <https://wordpress.org/plugins/types/>

(such as adding a featured image), adjust any other options, and save.

Now, that we have our portfolio post type, we want it to show up on our site. The first thing we'll do is create the page in question. Be aware that if you chose 'portfolio' as the slug of your CPT, the page can not have the same slug. I went with my **clients-portfolio** and also added some example text.



Portfolio page without a custom page template.

After adding a few items in the 'portfolio' post type section, we want them to show up on our page right underneath the page content.

To achieve this we will again use a derivative of the *page.php* file. Copy it, call it *portfolio-template.php* and change the header to this:

```
<?php
```

```
/*
```

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

```

* Description: Page template to display portfolio
custom post types
* underneath the page content
*/

```

However, in this case we will have to make a few changes to the original template. When you take a look at the code of *page.php*, you will see that it calls another template file in the middle, named *content-page.php* (where it says `<?php get_template_part('content', 'page'); ?>`). In that file we find the following code:

```

<article id="post-<?php the_ID(); ?>" <?php
post_class(); ?>
  <header class="entry-header">
    <?php if ( ! is_page_template( 'page-templates/
    front-page.php' ) ) : ?>
    <?php the_post_thumbnail(); ?>
    <?php endif; ?>
    <h1 class="entry-title"><?php the_title(); ?></h1>
  </header>

  <div class="entry-content">
    <?php the_content(); ?>
    <?php wp_link_pages( array( 'before' => '<div
    class="page-links">' . __( 'Pages:',
    'twentytwelve' ), 'after' => '</div>' ) ); ?>
  </div><!-- .entry-content -->
  <footer class="entry-meta">
    <?php edit_post_link( __( 'Edit', 'twentytwelve'
    ), '<span class="edit-link">', '</span>' ); ?>

```

```

    </footer><!-- .entry-meta -->
</article><!-- #post -->

```

As you can see, it is here that the page title and content are called. Since we definitely want those on our portfolio site, we will need to copy the necessary parts of this template to our *page.php* file. The result looks like this:

```

get_header(); ?>

<div id="primary" class="site-content">
    <div id="content" role="main">

        <?php while ( have_posts() ) : the_post(); ?>
            <header class="entry-header">
                <?php the_post_thumbnail(); ?>
                <h1 class="entry-title"><?php the_title();
                ?></h1>
            </header>

            <div class="entry-content">
                <?php the_content(); ?>
            </div><!-- .entry-content -->

            <?php comments_template( '', true ); ?>
            <?php endwhile; // end of the loop. ?>

        </div><!-- #content -->
    </div><!-- #primary -->

```

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

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

To get the portfolio items onto our page, we will add the following code right beneath the `the_content()` call.

```
<?php
    $args = array(
        'post_type' => 'portfolio', // enter custom post
        // type
        'orderby' => 'date',
        'order' => 'DESC',
    );

    $loop = new WP_Query( $args );
    if( $loop->have_posts() ):
    while( $loop->have_posts() ): $loop->the_post();
    global $post;
        echo '<div class="portfolio">';
        echo '<h3>' . get_the_title() . '</h3>';
        echo '<div class="portfolio-image">'.
        get_the_post_thumbnail( $id ). '</div>';
        echo '<div class="portfolio-work">
        '. get_the_content(). '</div>';
        echo '</div>';
    endwhile;
    endif;
?>
```

This will make the CPT show up on the page:

Test site

Just another WordPress site

[DEMO PAGE](#) [CLIENTS/PORTFOLIO](#) [CONTRIBUTORS](#) [ARCHIVES](#)

Clients/Portfolio

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed tincidunt arcu eget tellus mattis molestie. Donec elit sem, porttitor et efficitur eu, tincidunt sit amet elit. Mauris molestie ac nunc at facilisis. Ut cursus dolor eget sem porta, vitae ullamcorper erat efficitur. Ut vitae magna ligula. In vel ultrices felis. Nullam tempus lectus ultrices augue mattis, nec varius diam elementum. Praesent mi elit, lobortis sed lorem suscipit, imperdiet scelerisque ex.

META

[Site Admin](#)
[Log out](#)
[Entries RSS](#)
[Comments RSS](#)
[WordPress.org](#)

RECENT COMMENTS

Client 1



Description of client and what they do

- Service provided
- Results obtained
- Other great things that happened

Client 2



Description of client and what they do

- Service provided
- Results obtained
- Other great things that happened

Client 3



Description of client and what they do

- Service provided
- Results obtained
- Other great things that happened

Client 4



Description of client and what they do

- Service provided
- Results obtained
- Other great things that happened

Proudly powered by [WordPress](#)

The custom portfolio template.

I'm sure we all agree that it looks less than stellar, so some styling is in order.

```
/* Portfolio posts */

.portfolio {
  -webkit-box-shadow: 0px 2px 2px 0px rgba(50, 50,
50, 0.75);
  -moz-box-shadow: 0px 2px 2px 0px rgba(50, 50, 50,
0.75);
  box-shadow: 0px 2px 2px 0px rgba(50, 50, 50, 0.75);
  margin: 0 0 20px;
  padding: 30px;
}

.portfolio-image {
  display: block;
  float: left;
  margin: 0 10px 0 0;
  max-width: 20%;
}

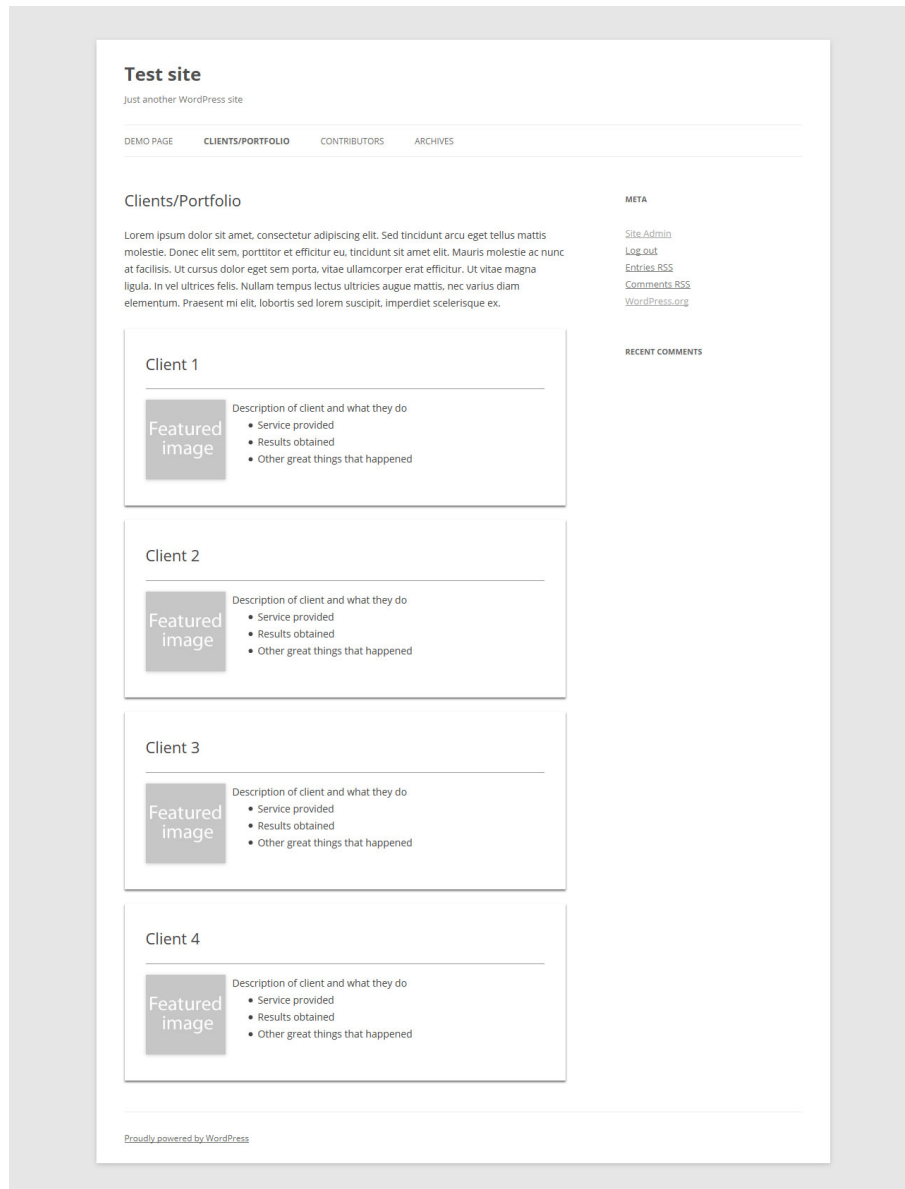
.portfolio-image img {
  border-radius: 0;
}

.portfolio-work {
  display: inline-block;
  max-width: 80%;
}

.portfolio h3 {
  border-bottom: 1px solid #999;
  font-size: 1.57143rem;
  font-weight: normal;
```

```
margin: 0 0 15px;
padding-bottom: 15px;
}
```

Much better, don't you think?



The custom portfolio template with styling.

And here is the entire code for the portfolio page template:

```
<?php
/*
 * Template Name: Portfolio Template
 * Description: Page template to display portfolio
 * custom post types underneath the page content
 */

get_header(); ?>

<div id="primary" class="site-content">
    <div id="content" role="main">

        <?php while ( have_posts() ) : the_post(); ?>

            <header class="entry-header">
                <?php the_post_thumbnail(); ?>
                <h1 class="entry-title"><?php the_title();
                ?></h1>
            </header>

            <div class="entry-content">
                <?php the_content(); ?>
                <?php
                    $args = array(
                        'post_type' => 'portfolio', // enter
                        // custom post type
                        'orderby' => 'date',
                        'order' => 'DESC',
```

```

);

$loop = new WP_Query( $args );
if( $loop->have_posts() ):
while( $loop->have_posts() ):
$loop->the_post(); global $post;
    echo '<div class="portfolio">';
    echo '<h3>' . get_the_title() . '</h3>';
    echo '<div class="portfolio-image">'.
    get_the_post_thumbnail( $id ).'</div>';
    echo '<div class="portfolio-work">'.
    get_the_content().'</div>';
    echo '</div>';
endwhile;
endif;
?>
</div><!-- #entry-content -->
<?php comments_template( '', true );
?>
<?php endwhile; // end of the loop.
?>
</div><!-- #content -->
</div><!-- #primary -->

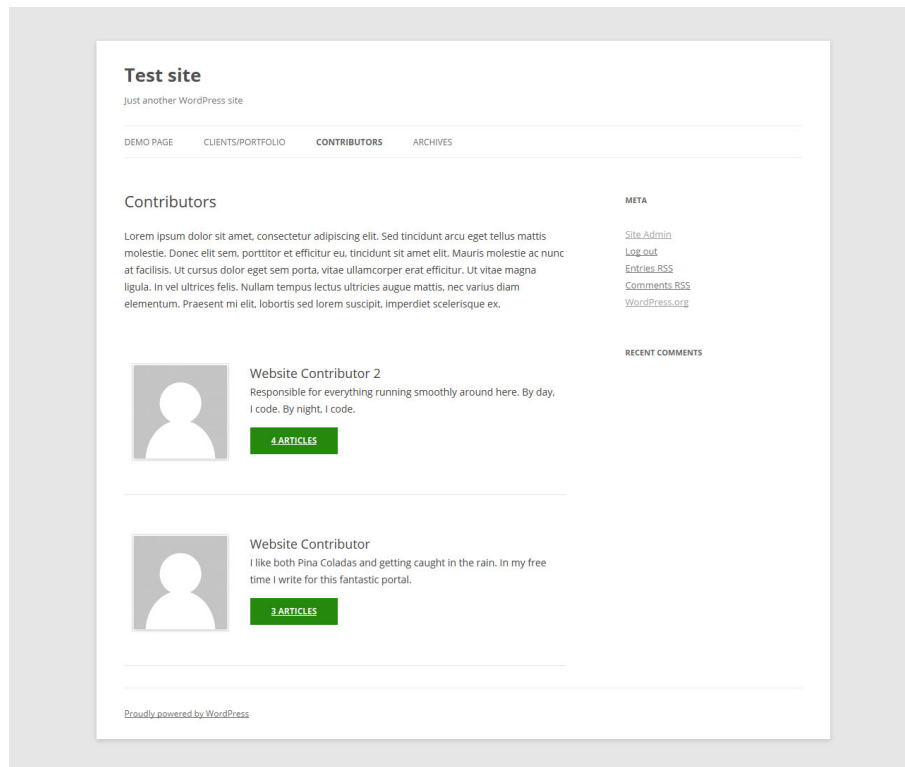
<?php get_sidebar(); ?>
<?php get_footer(); ?>

```

4. CONTRIBUTOR PAGE WITH AVATAR IMAGES

Next up in our page template use cases is a contributor page. We want to set up a list of authors on our website,

including their images and the number of posts they have published under their name. The end result will look like this:



The completed custom contributors page.

We will again start out with our hybrid file from before and add the code for the contributor list to it. But what if you don't know how to create such a thing? No worries, you can get by with intelligent stealing.

You see, the Twenty Fourteen default theme comes with a contributor page by default. You can find its template in the *page-templates* folder with the name *contributors.php*.

When looking into the file, however, you will only find the following call in there:

`twentyfourteen_list_authors()`; . Luckily, as an avid WordPress user you now conclude that this probably refers to a function in Twenty Fourteen's *function.php* file and you would be right.

From what we find in there, the part that interests us is this:

```
<?php
// Output the authors list.
$contributor_ids = get_users( array(
    'fields' => 'ID',
    'orderby' => 'post_count',
    'order' => 'DESC',
    'who' => 'authors',
));

foreach ( $contributor_ids as $contributor_id ) :
    $post_count = count_user_posts( $contributor_id );
    // Move on if user has not published a post (yet).
    if ( ! $post_count ) {
        continue;
    }
?>

<div class="contributor">
    <div class="contributor-info">
        <div class="contributor-avatar"><?php echo
        get_avatar( $contributor_id, 132 ); ?></div>
        <div class="contributor-summary">
            <h2 class="contributor-name"><?php echo
```

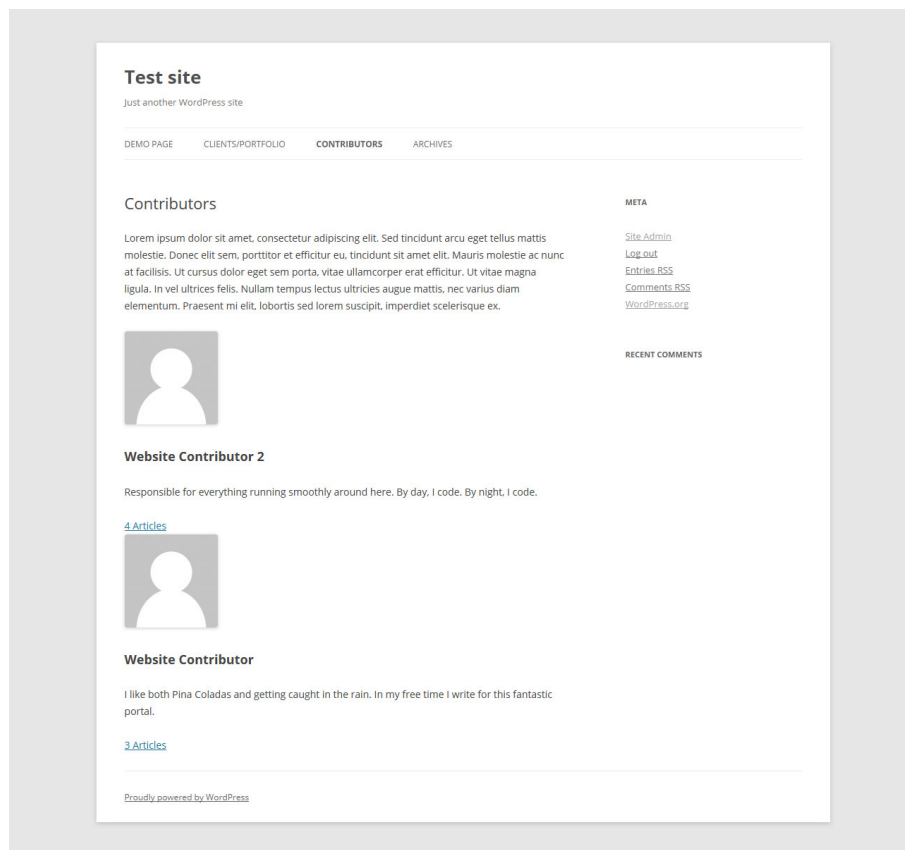
```

get_the_author_meta( 'display_name',
$contributor_id ); ?></h2>
<p class="contributor-bio">
    <?php echo get_the_author_meta(
        'description', $contributor_id ); ?>
</p>
<a class="button contributor-posts-link"
href="<?php echo esc_url( get_author_posts_url(
$contributor_id ) ); ?>">
    <?php printf( _n( '%d Article', '%d Articles',
        $post_count, 'twentyfourteen' ), $post_count
    ); ?>
</a>
</div><!-- .contributor-summary -->
</div><!-- .contributor-info -->
</div><!-- .contributor -->

<?php
endforeach;
?>

```

We will again add it below the call for `the_content()` with the following result:



The unstyled custom contributors page.

Now for a little bit of styling:

```
/* Contributor page */
```

```
.contributor {
    border-bottom: 1px solid rgba(0, 0, 0, 0.1);
    -webkit-box-sizing: border-box;
    -moz-box-sizing: border-box;
    box-sizing: border-box;
    display: inline-block;
    padding: 48px 10px;
}

.contributor p {
```

```
    margin-bottom: 1rem;
}
.contributor-info {
    margin: 0 auto 0 168px;
}
.contributor-avatar {
    border: 1px solid rgba(0, 0, 0, 0.1);
    float: left;
    line-height: 0;
    margin: 0 30px 0 -168px;
    padding: 2px;
}
.contributor-avatar img {
    border-radius: 0;
}
.contributor-summary {
    float: left;
}
.contributor-name {
    font-weight: normal;
    margin: 0 !important;
}
.contributor-posts-link {
    background-color: #24890d;
    border: 0 none;
    border-radius: 0;
    color: #fff;
    display: inline-block;
    font-size: 12px;
    font-weight: 700;
```

```

    line-height: normal;
    padding: 10px 30px 11px;
    text-transform: uppercase;
    vertical-align: bottom;
}

.contributor-posts-link:hover {
    color: #000;
    text-decoration: none;
}

```

And that should be it. Thanks Twenty Fourteen!

5. CUSTOMIZED ARCHIVE PAGE

Twenty Twelve comes with its own template for archive pages. It will jump into action, for example, when you attempt to view all past posts from a certain category.

However, I want something a little more like what [Prologger](http://www.prologger.net/archives/)¹³ has done: a page that lets people discover additional content on my site in several different ways.

That, again, is done with a page template.

Staying with our mixed template from before, we will add the following below the `the_content()` call:

```

<div class="archive-search-form"><?php
get_search_form(); ?></div>

<h2>Archives by Year:</h2>
<ul><?php wp_get_archives('type=yearly'); ?></ul>

```

¹³. <http://www.prologger.net/archives/>

<h2>Archives by Month:</h2>

<?php wp_get_archives('type=monthly'); ?>

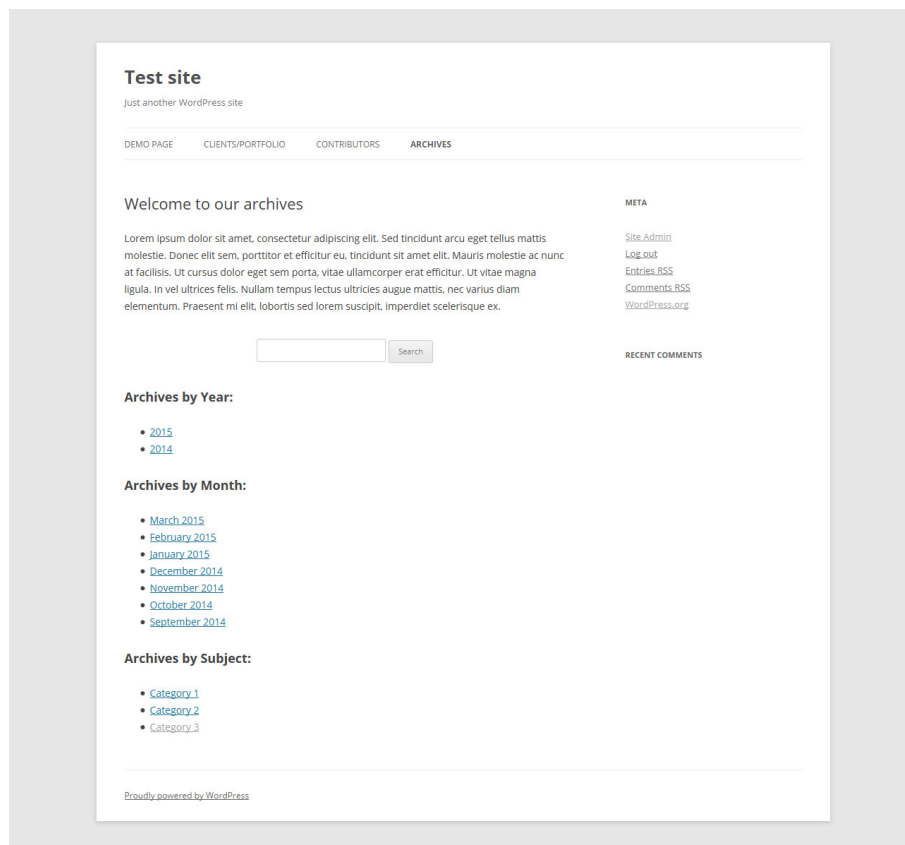
<h2>Archives by Subject:</h2>

 <?php wp_list_categories('title_li='); ?>

Plus, a little bit of styling for the search bar:

```
.archive-search-form {  
    padding: 10px 0;  
    text-align: center;  
}
```

And the result should look a little bit like this:



The custom archive page.

For completion's sake, here is the entire file:

```
<?php
/**
 * Template Name: Custom archive template
 *
 */

get_header(); ?>

<div id="primary" class="site-content">
    <div id="content" role="main">

        <?php while ( have_posts() ) : the_post();
        ?>

        <header class="entry-header">
            <?php the_post_thumbnail(); ?>
            <h1 class="entry-title"><?php the_title();
            ?></h1>
        </header>

        <div class="entry-content">
            <?php the_content(); ?>

            <div class="archive-search-form"><?php
            get_search_form(); ?></div>

            <h2>Archives by Year:</h2>
            <ul><?php wp_get_archives('type=yearly');
            ?></ul>
```

```

<h2>Archives by Month:</h2>
<ul><?php wp_get_archives('type=monthly');
?></ul>

<h2>Archives by Subject:</h2>
<ul><?php wp_list_categories('title_li=');
?></ul>
</div><!-- #entry-content -->

<?php comments_template( '', true );
?>

<?php endwhile; // end of the loop.
?>
</div><!-- #content -->
</div><!-- #primary -->

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

```

Don't forget to assign it to a page!

WordPress Page Templates In A Nutshell

On your way to mastering WordPress, learning to use page templates is an important step. They can make customizing your website very, very easy and allow you to assign unique functionality and design to as many or few pages as you wish. From adding widget areas to showing custom post types to displaying a list of your website's contributors — the possibilities are practically endless.

Whether you use conditional tags, exploit the WordPress template hierarchy, or create page-specific template files is entirely up to you and what you are trying to achieve. Start off small and work your way up to more complicated things. It won't be long before every part of your WordPress website will answer to your every call. 🐼

Extending WordPress With Custom Content Types

BY BRIAN ONORIO 🍷

WordPress does some pretty amazing things out of the box. It handles content management as well as any other open-source solution out there — and better than many commercial solutions. One of the best attributes of WordPress is its ease of use. It's easy because there's not a significant amount of bloat with endless bells and whistles that steepen the learning curve.

On the flip side, some might find WordPress a little... well, light. It does a lot, but not quite enough. If you find yourself hacking WordPress to do the things you wish it would do, then the chances are high that this article is for you.

WordPress can be easily extended to fit the requirements of a custom data architecture. We're going to explore the process of registering new data types in a fully compliant manner.

If you want to follow along at home, we've provided the [full source code](#)¹⁴ (TXT, 5.0 KB).

Custom Post Types

WordPress gives you a very simple and straightforward way to extend the standard two data types (Posts and

¹⁴. <http://provide.smashingmagazine.com/full-source-code.txt>

Pages) into an endless array for your custom needs. A digital agency or freelancer would need a “Project” post type. A mall would need a “Location” post type.

Quick point. Spinning off custom post types is a great idea for content that is intrinsically different than either Posts or Pages. There could be a case where you would want press releases to live in their own type. But more often than not, the press releases would be a Post and categorized as a press release. Or you may want to create a post type for landing pages. It may very well belong as a custom type, but it likely could also exist as a Page.

For the sake of this article, we’re going to follow a real-world scenario of creating a Project post type to store samples of work. We’ll register the post type, add some meta data to it, include additional information in WordPress’ administration screens, and create custom taxonomies to supplement.

Registering The Post Type

To get started, we’ll need some code to register the post type. We’re going to go with an object-oriented approach because we’ll be spinning off this post type later with some added functionality that would be done much more efficiently with an object model. To start, let’s create the function that registers the post type.

```
function create_post_type() {  
    $labels = array(  
        'name'                => 'Projects',  
        'singular_name'       => 'Project',
```

```

'menu_name'           => 'Projects',
'name_admin_bar'      => 'Project',
'add_new'             => 'Add New',
'add_new_item'        => 'Add New Project',
'new_item'            => 'New Project',
'edit_item'           => 'Edit Project',
'view_item'           => 'View Project',
'all_items'           => 'All Projects',
'search_items'        => 'Search Projects',
'parent_item_colon'   => 'Parent Project',
'not_found'           => 'No Projects Found',
'not_found_in_trash'  => 'No Projects Found in
Trash'
);

```

```

$args = array(
    'labels'           => $labels,
    'public'           => true,
    'exclude_from_search' => false,
    'publicly_queryable' => true,
    'show_ui'          => true,
    'show_in_nav_menus' => true,
    'show_in_menu'      => true,
    'show_in_admin_bar' => true,
    'menu_position'     => 5,
    'menu_icon'         => 'dashicons-admin-
appearance',
    'capability_type'   => 'post',
    'hierarchical'     => false,
    'supports'          => array( 'title',

```

```

        'editor', 'author', 'thumbnail', 'excerpt',
        'comments' ),
        'has_archive'          => true,
        'rewrite'              => array( 'slug' =>
        'projects' ),
        'query_var'            => true
    );

    register_post_type( 'sm_project', $args );
}

```

Not much mysterious here. We're calling the **create_post_type** function, which registers our post type. We're giving the type some labels for back-end identification and giving it a list of specifications in what it can do. For a full list of reference for each of these variables, take a look at the [WordPress Codex¹⁵](http://codex.wordpress.org/Function_Reference/register_post_type), but we'll hit on a few key items.

LABELS

For brevity, we've created a **labels** array and simply passed it to the **arguments** array. WordPress enables us to identify a slew of labels for singular, plural and other purposes.

PUBLIC

This setting is a parent of sorts for a few of the other settings that appear later in the list. The default value for the

¹⁵. http://codex.wordpress.org/Function_Reference/register_post_type

public attribute is `false`. The value of `public` is passed to the following other attributes when they are not explicitly defined: `exclude_from_search`, `publicly_queryable`, `show_in_nav_menus` and `show_ui`.

EXCLUDE_FROM_SEARCH

The default setting here is the *opposite* of the `public` attribute. If your post type is `public`, then it will be included in the website's search results. Note that this has no implication for SEO and only restricts or allows searches based on WordPress' native search protocol. There's a chance you would want the post type to be public but not appear in these search results. If that's the case, set this attribute to `true`.

PUBLICLY_QUERYABLE

This attribute is exclusively for front-end queries and has no real back-end implications. The default value is the same as the `public` attribute. Note that when it's set to `false`, you will not be able to view or preview the post type in the front end. For example, if you wanted to create a post type that populates a personnel page with a list of everyone's name, title and bio but didn't want them to have their own URL on the website, then you would set `publicly_queryable` to `false`.

SHOW_UI

Most of the time, you'll want to set `show_ui` to `true`. The default value pulls from the `public` attribute but can be overridden. When it's set to `false`, then a UI element on

WordPress' administration screen won't be available to you. A practical reason why you would want this set to `false` is if you had a post type that merely managed data. For example, you may want an Events post type that has a recurring attribute. When you save an event, new posts of a different type would be created to handle each event occurrence. You would want the UI to show only the primary Events post type and not the event occurrence's meta data.

`SHOW_IN_NAV_MENU`

Pretty simple and straightforward. If you don't want this post type to appear in WordPress' default menu functionality, set this to `false`. It takes the value of `public` as default.

`SHOW_IN_MENU`

You can modify the position of the post type in the back end. When set to `true`, the post type defaults as a top-level menu (on the same hierarchical level as Posts and Pages). If `false`, it won't show at all. You can use a string value here to explicitly nest the post type into a top level's submenu. The type of string you would provide is `tools.php`, which would place the post type as a nested element under "Tools." It derives its default value from `show_ui`. Note that `show_ui` must be set to `true` in order for you to be able to control this attribute.

SHOW_IN_ADMIN_BAR

Pretty self-explanatory. If you want UI elements added to WordPress' administration bar, set this to **true**.

MENU_POSITION

The default value of **null** will place the menu (at the top level and if not overridden using **show_in_menu**) below "Comments" in the back end. You can control this further by specifying an integer value corresponding to WordPress' default menu placements. A value of **5** will place the post type under "Posts," **10** under "Media," **15** under "Links," and **20** under "Pages." For a full list of values, check out the [WordPress Codex](#)¹⁶.

MENU_ICON

You can pass a URL to this attribute, but you could also simply use the name of an icon from Dashicons for a quick solution. Supplying the attribute **dashicons-admin-appearance** would give you a paint brush¹⁷. A [full list of Dashicons is available](#)¹⁸ as a handy resource. The default is the thumbtack icon used for Posts.

CAPABILITY_TYPE

This attribute quickly gets into some advanced user-role segmenting concepts. Essentially, assigning **post** to this attribute generates a capability structure that exactly

¹⁶. http://codex.wordpress.org/Function_Reference/register_post_type

¹⁷. <https://developer.wordpress.org/resource/dashicons/#admin-appearance>

¹⁸. <https://developer.wordpress.org/resource/dashicons/>

mimics how access to Posts works. Using this value, subscribers would not be able to access this post type, whereas Authors, Editors and Administrators would. Using `page` here would limit access to just Editors and Administrators. You can define a more granular structure using `capability_type` and `capabilities` attributes in the arguments list. Note that we did not use the `capabilities` attribute in this example because we're not explicitly defining a custom capability structure to be used with this post type. This is an advanced concept and one for a completely different article.

HIERARCHICAL

This is basically the difference between a Post and a Page. When set to `true`, a parent post can be identified on a per-post basis (basically, Pages). When `false`, it behaves as a Post.

SUPPORTS

A whole bunch of default functionality is attached to each new post type. This array tells WordPress which one of those to include by default. There may be an instance when you don't want the editor on your post type. Removing that from the array will remove the editor box on the post's editing screen. Eligible items for the array include the following:

- `title`
- `editor`

- `author`
- `thumbnail`
- `excerpt`
- `trackbacks`
- `custom-fields`
- `comments`
- `revisions`
- `page-attributes`
- `post-formats`

HAS_ARCHIVE

When this is set to `true`, WordPress creates a hierarchical structure for the post type. So, accessing `/projects/` would give us the standard `archive.php` view of the data. You can template out a variant of `archive.php` for this particular archive by creating a new file in your theme system named `archive-sm_project.php`. You can control the default behavior at a more granular level by spinning it off from your primary `archive.php`.

REWRITE

The `rewrite` option allows you to form a URL structure for the post type. In this instance, our URL would be `http://www.example.com/projects/{slug}/`, where the slug is the portion assigned by each post when it's created (normally, based on the title of the post). A second vari-

able can be assigned inside the `rewrite` array. If you add `with_front => false` (it defaults to `true`), it will not use the identified front half of the URL, which is set in “Settings” → “Permalinks.” For example, if your default WordPress permalink structure is `/blog/%postname%/`, then your custom post type would automatically be `/blog/projects/%postname%/`. That’s not a good outcome, so set `with_front` to `false`.

QUERY_VAR

This attribute controls where you can use a PHP query variable to retrieve the post type. The default is `true` and renders with the permalink structure (when set). You can use a string instead of a variable and control the key portion of the query variable with the string’s value.

Extending The Post Type With A Taxonomy (Or Two)

Out of the box, WordPress Posts have categories and tags attached to them that enable you to appropriately place content in these buckets. By default, new post types don’t have any taxonomies attached to them. You may not want to categorize or tag your post type, but if you do, you’d need to register some new ones. There are two variants of taxonomies, one that behaves like categories (the checklist to the right of the posts) and one like tags, which have no hierarchical structure. They behave in the back end pretty much in the same way (the only discernable difference being that categories can have children,

whereas tags cannot), but how they're presented on the administration screen varies quite wildly. We'll register two taxonomies to give us one of each type.

```
function create_taxonomies() {

    // Add a taxonomy like categories
    $labels = array(
        'name'          => 'Types',
        'singular_name' => 'Type',
        'search_items'  => 'Search Types',
        'all_items'     => 'All Types',
        'parent_item'   => 'Parent Type',
        'parent_item_colon' => 'Parent Type:',
        'edit_item'     => 'Edit Type',
        'update_item'   => 'Update Type',
        'add_new_item'  => 'Add New Type',
        'new_item_name' => 'New Type Name',
        'menu_name'     => 'Types',
    );

    $args = array(
        'hierarchical' => true,
        'labels'       => $labels,
        'show_ui'      => true,
        'show_admin_column' => true,
        'query_var'    => true,
        'rewrite'      => array( 'slug' => 'type' ),
    );
}
```

```
register_taxonomy('sm_project_type',array('sm_project'),$args);
```

```
// Add a taxonomy like tags
```

```
$labels = array(  
    'name'                    => 'Attributes',  
    'singular_name'          => 'Attribute',  
    'search_items'           => 'Attributes',  
    'popular_items'          => 'Popular  
Attributes',  
    'all_items'              => 'All Attributes',  
    'parent_item'            => null,  
    'parent_item_colon'      => null,  
    'edit_item'              => 'Edit Attribute',  
    'update_item'            => 'Update  
Attribute',  
    'add_new_item'           => 'Add New  
Attribute',  
    'new_item_name'          => 'New Attribute  
Name',  
    'separate_items_with_commas' => 'Separate  
Attributes with commas',  
    'add_or_remove_items'    => 'Add or remove  
Attributes',  
    'choose_from_most_used'  => 'Choose from most  
used Attributes',  
    'not_found'              => 'No Attributes  
found',  
    'menu_name'              => 'Attributes',  
);
```

```

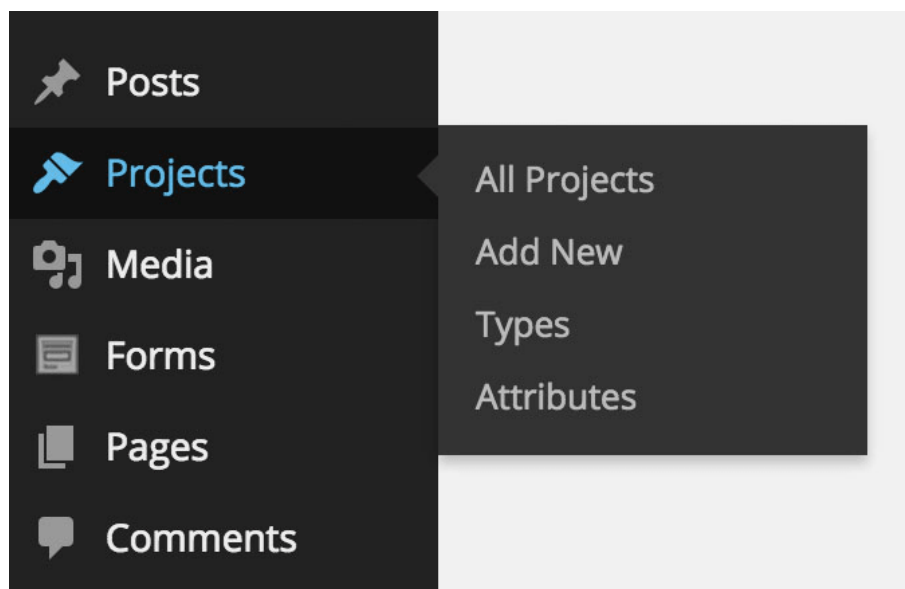
$args = array(
    'hierarchical'      => false,
    'labels'            => $labels,
    'show_ui'          => true,
    'show_admin_column' => true,
    'update_count_callback' =>
        '_update_post_term_count',
    'query_var'         => true,
    'rewrite'           => array( 'slug' =>
        'attribute' ),
);

```

```

register_taxonomy('sm_project_attribute','sm_project',$args);
}

```



After you register the new post type and taxonomies, you'll be greeted with a handy menu in the back end.

All right, now we have two new taxonomies attached to the new post type. The `register_taxonomy` function takes three arguments. The first is the taxonomy's name, the second is an array or string of post types, and the third is the arguments defined above.

A quick note on our prefixing. Our post type and taxonomies are all prefixed with `sm_`. This is by design. We don't want future plugins to interrupt our infrastructure, so we simply prefix. The name of the prefix is completely up to you.

So, we've got a new post type and two new taxonomies attached to it. This essentially replicates the default Posts behavior of WordPress. This is all good stuff, but let's dig a little deeper to make it more integrated.

Enhancing The Experience With Meta Data

Creating additional fields available to the author in WordPress' administration screen can be a bit tricky — but abundantly useful. Where WordPress underperforms its competitors is precisely in this area. There's no user interface where you can define additional pieces of information on a per-post basis. Make no mistake, WordPress fully *supports* this behavior, but it's more of a developer tool than an out-of-the-box tool, which makes sense. One might need an endless number of combinations of additional fields. Even if WordPress provided a slick back-end interface to allow a non-technical user to define these fields, there's no real seamless way to display that infor-

mation in the front end without a developer putting their hands on it and making it so.

This is where Advanced Custom Fields¹⁹ comes in. ACF is a wonderful plugin that gives developers this interface and a full array of templating functions to pull the data in the front end. This article doesn't detail how to do that, but ACF gives ample documentation²⁰ to get you started and working in the ACF environment.

The screenshot shows the 'Add New Field Group' interface in the ACF plugin. At the top, there's a text input for the field group name, which is 'Project Information'. Below this is a table with four columns: 'Order', 'Label', 'Name', and 'Type'. There are two rows in the table. The first row has '1' in the 'Order' column, 'Awards' in the 'Label' column, 'awards' in the 'Name' column, and 'Text' in the 'Type' column. The second row has '2' in the 'Order' column, 'Timeframe' in the 'Label' column, 'timeframe' in the 'Name' column, and 'Text' in the 'Type' column. Below the table is a blue bar with a 'Drag and drop to reorder' icon and a '+ Add Field' button. Below this is a 'Location' section. On the left is a 'Rules' sidebar with the text 'Create a set of rules to determine which edit screens will use these advanced custom fields'. On the right is a 'Show this field group if' section. It has a dropdown menu for 'Post Type', a dropdown for 'is equal to', a dropdown for 'Project', and an 'and' button. Below this is an 'or' section with an 'Add rule group' button.

ACF makes creating meta data and conditionally attaching to custom post types a snap.

Using ACF, you can define new fields and conditionally attach them to content throughout the website. For example, we could create a timeframe meta field that collects how long a particular project has taken. We could add additional fields for awards won or create fields to represent a list of references for any given project.

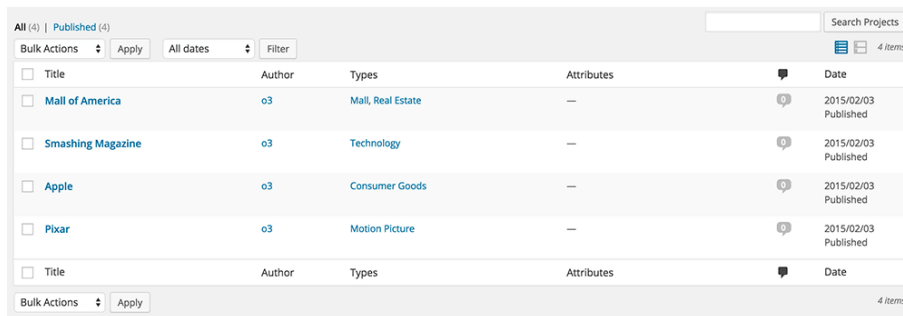
¹⁹. <http://www.advancedcustomfields.com/>

²⁰. <http://www.advancedcustomfields.com/resources/>

Using ACF really opens up the hood for what's possible in WordPress.

Adding Columns To The Administration Screen

Viewing a list of your posts on the administration screen will give you the checkbox, the title and the date published. When registering taxonomies to the post type, you'll get an additional column for each additional taxonomy. For the majority of cases, this is sufficient. But there may be an additional case or two where you need to provide a little more information. For example, referencing pieces of meta data in the administration grid might be useful. Maybe you want a quick reference for the time-frame or the awards field we defined above. We'll need two functions attached to some WordPress hooks.



The screenshot shows the WordPress 'Projects' administration screen. At the top, there are filters for 'All (4)' and 'Published (4)', a search bar, and a 'Search Projects' button. Below the filters, there are tabs for 'Bulk Actions' and 'Apply', and a 'Filter' button. The table has columns for 'Title', 'Author', 'Types', 'Attributes', and 'Date'. There are four rows of data, each with a checkbox in the 'Title' column. The data rows are: 'Mail of America' (Author: o3, Types: Mail, Real Estate, Attributes: —, Date: 2015/02/03 Published), 'Smashing Magazine' (Author: o3, Types: Technology, Attributes: —, Date: 2015/02/03 Published), 'Apple' (Author: o3, Types: Consumer Goods, Attributes: —, Date: 2015/02/03 Published), and 'Pixar' (Author: o3, Types: Motion Picture, Attributes: —, Date: 2015/02/03 Published). At the bottom, there are tabs for 'Bulk Actions' and 'Apply', and a '4 Items' indicator.

<input type="checkbox"/>	Title	Author	Types	Attributes	Date
<input type="checkbox"/>	Mail of America	o3	Mail, Real Estate	—	2015/02/03 Published
<input type="checkbox"/>	Smashing Magazine	o3	Technology	—	2015/02/03 Published
<input type="checkbox"/>	Apple	o3	Consumer Goods	—	2015/02/03 Published
<input type="checkbox"/>	Pixar	o3	Motion Picture	—	2015/02/03 Published

WordPress gives you a standard administration screen out of the box that closely mirrors the built-in Post post type.

Let's look at the code:

```
function columns($columns) {  
    unset($columns['date']);  
}
```

```

unset($columns['taxonomy-sm_project_attribute']);
unset($columns['comments']);
unset($columns['author']);
return array_merge(
    $columns,
    array(
        'sm_awards' => 'Awards',
        'sm_timeframe' => 'Timeframe'
    )
);
}

```

The first line unsets the date column. You can unset any of the default columns that you wish. The second line unsets the custom taxonomy we registered (the tag-like one, not category). This could be useful for keeping the admin screen neat and tidy. As you may have noticed, we also unset the comments and author — information we didn't think was necessary on the screen.

Then, we're simply defining the new columns and merging them with the array that was passed in the function. We created two new columns, one for **awards** and one for **timeline**. The array keys are completely arbitrary. They could be anything, but we'll need to reference them again when it comes time to pull data into those columns... which is what we're going to do next.

```

function column_data($column,$post_id) {
    switch($column) {
        case 'sm_awards' :
            echo get_post_meta($post_id,'awards',1);
            break;
    }
}

```

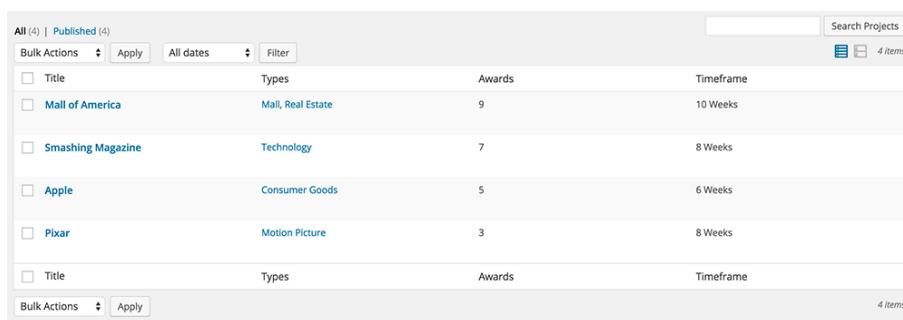


```

case 'sm_timeframe' :
    echo get_post_meta($post_id, 'timeframe', 1);
    break;
}

```

All right, we've fetched the meta data and conditionally outputted it based on what column we're on. This is where we're referencing the array key from above. So, as long as they're both the same, we could use any arbitrary string we want. Note that we're pulling the meta fields over using WordPress' native `get_post_meta` function.



Title	Types	Awards	Timeframe
Mail of America	Mail, Real Estate	9	10 Weeks
Smashing Magazine	Technology	7	8 Weeks
Apple	Consumer Goods	5	6 Weeks
Pixar	Motion Picture	3	8 Weeks

Removing some built-in columns and adding a few of our own enhances the UI and streamlines the information displayed.

Sorting

Ah, sorting. As you probably know, WordPress sorts Pages by menu order and then alphabetically by title and Posts by date published. Let's get fancy and sort our new post type by the number of awards won. The use case here is easy to see. You want your most award-winning work at the top of the list at all times. If we use standard WordPress queries, the order we're about to establish will be honored — universally across the website. We will

need a function to join the `wp_posts` and `wp_postmeta` tables and another to revise how the data is sorted.

```
function join($wp_join) {
    global $wpdb;
    if(get_query_var('post_type') == 'sm_project') {
        $wp_join .= " LEFT JOIN (
            SELECT post_id, meta_value as awards
            FROM $wpdb->postmeta
            WHERE meta_key = 'awards' ) AS meta
            ON $wpdb->posts.ID = meta.post_id ";
    }
    return ($wp_join);
}
```

This function does the joining for us. We won't get into why that `select` statement works (that's for another article altogether). Pay attention to the `if` statement here. We're determining the post type and then conditionally running the `join` if it meets the `sm_project` condition. Absent this `if` statement, you would be doing this `join` regardless of type, which is not likely something you want.

There could also be a case where you just want to sort the administration screens and not the front end. Fortunately, we can use WordPress' built-in conditional statements to do that job. Just wrap your statement with another conditional and check against `is_admin`.

```
function set_default_sort($orderby,&$query) {
    global $wpdb;
    if(get_query_var('post_type') == 'sm_project') {
```

```

        return "meta.awards DESC";
    }
    return $orderby;
}

```

Once again, we're verifying our post type and then returning an amended **order** statement. Now we're telling WordPress to order by the value from the **wp_postmeta** table descending. So, we'll get a list of our awards from the most won per project to the least won per project.

Putting It All Together

None of these functions will do anything until they're called and attached to WordPress hooks. We'll do this and keep it neat by creating an object around the post type and using the constructor to attach each function to the appropriate hook. For brevity, we're not going to repeat the code already referenced above.

```

class sm_project {

    function __construct() {
        add_action('init',array($this,
            'create_post_type'));
        add_action('init',array($this,
            'create_taxonomies'));
        add_action('manage_sm_project_posts_columns',
            array($this, 'columns'),10,2);
        add_action(
            'manage_sm_project_posts_custom_column',

```

```

    array($this, 'column_data'), 11, 2);
    add_filter('posts_join', array($this, 'join'),
    10, 1);
    add_filter('posts_orderby', array($this,
    'set_default_sort'), 20, 2);
}

function create_post_type() {
    ...
}

function create_taxonomies() {
    ...
}

function columns($columns) {
    ...
}

function column_data($column, $post_id) {
    ...
}

function join($wp_join) {
    ...
}

function set_default_sort($orderby, &$query) {
    ...
}

```

```
}
```

```
new sm_project();
```

Voilà! Everything has come together quite nicely. In our constructor, we referenced the appropriate actions and filters. We’re performing these functions in a particular order — and this must be followed. The post type has to be created first, the taxonomies attached second, then any sort of custom sorting. Keep that in mind as you’re creating your data type.

Summing Up

Once you get the hang of it and you create a few of these, it’ll start to come naturally. I’ve got a bunch of these clips saved up in my toolbox. I rarely create these from scratch anymore. Although this article is long and in-depth, it really is about a 10-minute process from concept to conclusion once you fully understand what’s going on.

REFERENCES

- “[register_post_type](#)²¹,” WordPress Codex
- “[register_taxonomy](#)²²,” WordPress Codex
- “[Metadata API](#)²³,” WordPress Codex

²¹. http://codex.wordpress.org/Function_Reference/register_post_type

²². http://codex.wordpress.org/Function_Reference/register_taxonomy

²³. https://codex.wordpress.org/Metadata_API

- “[get_post_meta](#)²⁴,” WordPress Codex
- “[get_fields](#)²⁵,” ACF Documentation
- “[Template Hierarchy](#)²⁶,” WordPress Codex
- “[Action Reference](#)²⁷,” WordPress Codex
- “[Filter Reference](#)²⁸,” WordPress Codex 🐘

²⁴. http://codex.wordpress.org/Function_Reference/get_post_meta

²⁵. http://www.advancedcustomfields.com/resources/get_fields/

²⁶. http://codex.wordpress.org/Template_Hierarchy

²⁷. http://codex.wordpress.org/Plugin_API/Action_Reference

²⁸. http://codex.wordpress.org/Plugin_API/Filter_Reference

Building A Custom Archive Page For WordPress

BY KAROL K 🍷

If I were to ask you what the least used default page type in WordPress is, chances are you'd say the archive template. Or, more likely, you'd probably not even think of the archive template at all — that's how unpopular it is. The reason is simple. As great as WordPress is, the standard way in which it approaches the archive is far from user-friendly.

Let's fix that today! Let's build an archive page for WordPress that's actually useful. The best part is that you will be able to use it with any modern WordPress theme installed on your website at the moment.

But first, what do we mean by “archive page” exactly?

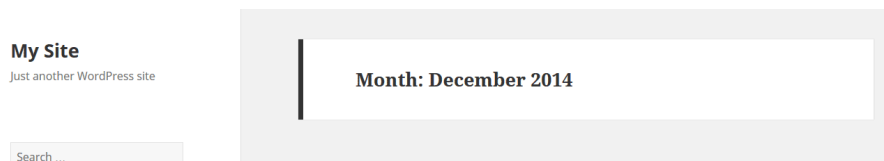
The Story Of WordPress Archives

In WordPress, you get to work with a range of different page templates and structures in the standard configuration. Looking at the directory listing of the default theme at the time of writing, Twenty Fifteen, we find the following:

- 404 error page,
- archive page (our hero today),
- image attachments page,

- index page (the main page),
- default page template (for pages),
- search results page,
- single post and attachment pages.

Despite their different purposes, all of these pages are really similar in structure, and they usually only differ in a couple of places and several lines of code. In fact, the only visible difference between the index page and the archive page is the additional header at the top, which changes according to the particular page being viewed.



Standard archive page in Twenty Fifteen.

The idea behind such an archive structure is to provide the blog administrator with a way to showcase the archive based on various criteria, but to do so in a simplified form. At the end of the day, these various archive pages are just versions of the index page that filter content published during a specific time period or by a particular author or with particular categories or tags.

While this sounds like a good idea from a programmer's perspective, it doesn't make much sense from the user's point of view. Or, more accurately, one layer is missing here — a layer that would come between the

user's intent to find content and the individual items in the archive themselves.

Here's what I mean. Right now, the only built-in way to showcase the archive links on a WordPress website is with a widget. So, if you want to allow visitors to dig into the archive in any clear way, you'd probably have to devote a whole sidebar just to the archive (just to be able to capture different types of organization, such as a date-based archive, a category archive, a tag archive, an author archive and so on).

So, what we really need here is a middleman, a page that welcomes the visitor, explains that they're in the archive and then points them to the exact piece of content they are interested in or suggests some popular content.

That is why we're going to create a custom archive page.

How To Build A Custom Archives Page In WordPress

Here's what we're going to do in a nutshell. Our custom archive page will be based on a custom page template²⁹. This template will allow us to do the following:

- include a custom welcome message (may contain text, images, an opt-in form, etc. — standard WordPress stuff);
- list the 15 latest posts (configurable);

²⁹. http://codex.wordpress.org/Page_Templates

- display links to the author archive;
- display links to the monthly archive;
- add additional widget areas (to display things like the most popular content, categories, tags).

Lastly, the page will be responsive and will not depend on the current theme of the website it's being used on.

That being said, we do have to start by using *some* theme as the base of our work here. I'll use Zerif Lite³⁰. I admit, I may be a bit biased here because it is one of our own themes (at ThemeIsle). Nonetheless, it was one of the 10 most popular themes released last year in WordPress' theme directory, so I hope you'll let this one slide.

And, hey, if you don't like the theme, no hard feelings. You can use the approach presented here with any other theme.

Getting Started With The Main File

The best model on which to build your archive page is the `page.php` file of your current theme, for a couple of reasons:

- Its structure is already optimized to display custom content within the main content block.
- It's probably one of the simplest page templates in your theme's structure.

³⁰. <https://wordpress.org/themes/zerif-lite>

Therefore, starting with the `page.php` file of the Zerif Lite theme, I'm going to make a copy and call it `tmpl_archives.php`.

(Make sure not to call your page something like `page-archives.php`. All file names starting with `page-` will be treated as new page templates within the main file hierarchy of WordPress themes³¹. That's why we're using the prefix `tmpl_` here.)

Next, all I'm going to do is change one single line in that file:

```
<?php get_template_part( 'content', 'page' ); ?>
```

We'll change that to this:

```
<?php get_template_part( 'content', 'tmpl_archives' ); ?>
```

All this does is fetch the right content file for our archive page.

If you want, you could remove other elements that seem inessential to your archive page (like comments), but make sure to leave in all of the elements that make up the HTML structure. And in general, don't be afraid to experiment. After all, if something stops working, you can easily bring back the previous code and debug from there.

Also, don't forget about the standard custom template declaration comment, which you need to place at the very beginning of your new file (in this case, `tmpl_archives.php`):

³¹. <http://www.codeinwp.com/blog/wordpress-theme-heirarchy/>

```
<?php
/* Template Name: Archive Page Custom */
?>
```

After that, what we're left with is the following file structure (with some elements removed for readability):

```
<?php
/* Template Name: Archive Page Custom */
get_header(); ?>

<div class="clear"></div>
</header> <!-- / END HOME SECTION -->

<div id="content" class="site-content">

<div class="container">

    <div class="content-left-wrap col-md-9">
        <div id="primary" class="content-area">
            <main id="main" class="site-main" role="main">

                <?php while ( have_posts() ) : the_post(); //
                standard WordPress loop. ?>

                <?php get_template_part( 'content',
                'tmpl_archives' ); // loading our
                custom file. ?>

                <?php endwhile; // end of the loop. ?>
```

```

        </main><!-- #main -->
    </div><!-- #primary -->
</div>
<div class="sidebar-wrap col-md-3
content-left-wrap">
    <?php get_sidebar(); ?>
</div>

</div><!-- .container -->

<?php get_footer(); ?>

```

Next, let's create the other piece of the puzzle — a custom content file. We'll start with the `content-page.php` file by making a copy and renaming it to `content-templ_archives.php`.

In this file, we're going to remove anything that's not essential, keeping only the structural elements, plus the basic WordPress function calls:

```

<?php
/**
 * The template used to display archive content
 */
?>

<article id="post-<?php the_ID(); ?>" <?php
post_class(); ?>>

    <header class="entry-header">
        <h1 class="entry-title"><?php the_title(); ?></h1>

```

```
</header><!-- .entry-header -->
```

```
<div class="entry-content">
```

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

```
<!-- THIS IS WHERE THE FUN PART GOES -->
```

```
</div><!-- .entry-content -->
```

```
</article><!-- #post-## -->
```

The placeholder comment visible in the middle is where we're going to start including our custom elements.

ADDING A CUSTOM WELCOME MESSAGE

This one's actually already taken care of by WordPress. The following line does the magic:

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

ADDING NEW WIDGET AREAS

Let's start this part by setting up new widget areas in WordPress using the standard process. However, let's do it through an additional functions file, just to keep things reusable from theme to theme.

So, we begin by creating a new file, **archives-page-functions.php**, placing it in the theme's main directory, and registering the two new widget areas in it:

```
if(!function_exists('archives_page_widgets_init')) :
function archives_page_widgets_init() {
```

```

/* First archive page widget, displayed to the
LEFT. */
register_sidebar(array(
    'name' => __('Archives page widget LEFT',
    'zerif-lite'),
    'description' => __('This widget will be shown on
the left side of your archive page.',
    'zerif-lite'),
    'id' => 'archives-left',
    'before_widget' => '<div class="archives-widget-
left">',
    'after_widget' => '</div>',
    'before_title' => '<h1 class="widget-title">',
    'after_title' => '</h1>',
));

```

```

/* Second archive page widget, displayed to the
RIGHT. */
register_sidebar(array(
    'name' => __('Archives page widget RIGHT',
    'zerif-lite'),
    'description' => __('This widget will be shown on
the right side of your archive page.',
    'zerif-lite'),
    'id' => 'archives-right',
    'before_widget' => '<div class="archives-widget-
right">',
    'after_widget' => '</div>',
    'before_title' => '<h1 class="widget-title">',
    'after_title' => '</h1>',

```

```

    ));
}
endif;
add_action('widgets_init',
'archives_page_widgets_init');
```

Next, we'll need some custom styling for the archive page, so let's also "enqueue" a new CSS file:

```

if(!function_exists('archives_page_styles')) :
function archives_page_styles() {
    if(is_page_template('tpl_archives.php')) {
        wp_enqueue_style('archives-page-style',
            get_template_directory_uri() . '/archives-page-
            style.css'); // standard way of adding
            style sheets in WP.
    }
}
endif;
add_action('wp_enqueue_scripts',
'archives_page_styles');
```

This is a conditional enqueue operation. It will run only if the visitor is browsing the archive page.

Also, let's not forget to enable this new **archives-page-functions.php** file by adding this line at the very end of the current theme's **functions.php** file:

```

require get_template_directory() .
'/archives-page-functions.php';
```


Finally, the new block that we'll use in our main `content-tmpl_archives.php` file is quite simple. Just place the following right below the call to `the_content()`:

```
<?php /* Enabling the widget areas for the archive
page. */ ?>
<?php if(is_active_sidebar('archives-left'))
dynamic_sidebar('archives-left'); ?>
<?php if(is_active_sidebar('archives-right'))
dynamic_sidebar('archives-right'); ?>
<div style="clear: both; margin-bottom:
30px;"></div><!-- clears the floating -->
```

All that's left now is to take care of the only missing file, `archives-page-style.css`. But let's leave it for later because we'll be using it as a place to store all of the styles of our custom archive page, not just those for widgets.

LISTING THE 15 LATEST POSTS

For this, we'll do some manual PHP coding. Even though displaying this could be achieved through various widgets, let's keep things diverse and get our hands a bit dirty just to show more possibilities.

You're probably asking why the arbitrary number of 15 posts? Well, I don't have a good reason, so let's actually make this configurable through custom fields.

Here's how we're going to do it:

- Setting the number of posts will be possible through the custom field `archived-posts-no`.

- If the number given is not correct, the template will default to displaying the 15 latest posts.

Below is the code that does this. Place it right below the previous section in the `content-tmpl_archives.php` file, the one that handles the new widget areas.

```
<?php
$show_many_last_posts =
intval(get_post_meta($post->ID, 'archived-posts-no',
true));

/* Here, we're making sure that the number fetched is
reasonable. In case it's higher than 200 or lower
than 2, we're just resetting it to the default value
of 15. */
if($show_many_last_posts > 200 || $show_many_last_posts
< 2) $show_many_last_posts = 15;

$my_query = new WP_Query('post_type=post&nopaging=1');
if($my_query->have_posts()) {
    echo '<h1 class="widget-title">Last
    '.$show_many_last_posts.' Posts <i class="fa
    fa-bullhorn" style="vertical-align:
    baseline;"></i></h1>&nbsp;';
    echo '<div class="archives-latest-section"><ol>';
    $counter = 1;
    while($my_query->have_posts() && $counter <=
    $show_many_last_posts) {
        $my_query->the_post();
        ?>
```

```

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

<?php
$counter++;
}
echo '</ol></div>';
wp_reset_postdata();
}
?>

```

Basically, all this does is look at the custom field's value, set the number of posts to display and then fetch those posts from the database using `WP_Query()`. I'm also using some Font Awesome icons to add some flare to this block.

DISPLAYING LINKS TO THE AUTHOR ARCHIVES

(This section is only useful if you're dealing with a multi-author blog. Skip it if you are the sole author.)

This functionality can be achieved with a really simple block of code placed right in our main `content-tmpl_archives.php` file (below the previous block):

```

<h1 class="widget-title">Our Authors <i class="fa
fa-user" style="vertical-align: baseline;"></i></h1>
<div class="archives-authors-section">
  <ul>
    <?php wp_list_authors(
      'exclude_admin=0&optioncount=1'); ?>
  </ul>
</div>

```

```
</ul>
</div>
```

We'll discuss the styles in just a minute. Right now, please note that everything is done through a `wp_list_authors()` function call.

DISPLAYING LINKS TO THE MONTHLY ARCHIVES

I'm including this element at the end because it's not the most useful one from a reader's perspective. Still, having it on your archive page is nice just so that you don't have to use widgets for the monthly archive elsewhere.

Here's what it looks like in the `content-tmpl_archives.php` file:

```
<h1 class="widget-title">By Month <i class="fa
fa-calendar" style="vertical-align:
baseline;"></i></h1>
<div class="archives-by-month-section">
  <p><?php wp_get_archives(
    'type=monthly&format=custom&after= | '); ?></p>
</div>
```

This time, we're displaying this as a single paragraph, with entries separated by a pipe (|).

(Smashing Magazine already has a really good tutorial on how to customize individual archive pages³² for categories, tags and other taxonomies in WordPress.)

THE COMPLETE ARCHIVE PAGE TEMPLATE

OK, just for clarity, let's look at our complete `content-tmpl_archives.php` file, which is the main file that takes care of displaying our custom archive:

```
<?php
/**
 * The template used to display archive content
 */
?>

<article id="post-<?php the_ID(); ?>" <?php
post_class(); ?>>

<header class="entry-header">
    <h1 class="entry-title"><?php the_title(); ?></h1>
</header><!-- .entry-header -->

<div class="entry-content">
    <?php the_content(); ?>

    <?php if(is_active_sidebar('archives-left'))
```

³². <http://www.smashingmagazine.com/2014/08/27/customizing-wordpress-archives-categories-terms-taxonomies/>

```

dynamic_sidebar('archives-left'); ?>
<?php if(is_active_sidebar('archives-right'))
dynamic_sidebar('archives-right'); ?>
<div style="clear: both; margin-bottom: 30px;">
</div><!-- clears the floating -->

<?php
$show_many_last_posts = intval(get_post_meta($post->
ID,] 'archived-posts-no', true));
if($show_many_last_posts > 200 ||
$show_many_last_posts < 2) $show_many_last_posts = 15;

$my_query = new WP_Query('post_type=post&nopaging=
1');
if($my_query->have_posts()) {
    echo '<h1 class="widget-title">Last
    '. $show_many_last_posts . ' Posts <i class="fa
    fa-bullhorn" style="vertical-align:
    baseline;"></i></h1>&nbsp;';
    echo '<div class="archives-latest-section"><ol>';
    $counter = 1;
    while($my_query->have_posts() && $counter <=
    $show_many_last_posts) {
        $my_query->the_post();
        ?>
        <li><a href="<?php the_permalink() ?>"
        rel="bookmark" title="Permanent Link to <?php
        the_title_attribute(); ?>"><?php the_title();
        ?></a></li>
        <?php

```

```

        $counter++;
    }
    echo '</ol></div>';
    wp_reset_postdata();
}
?>

```

```

<h1 class="widget-title">Our Authors <i class="fa
fa-user" style="vertical-align: baseline;"></i></h1>
&nbsp;
<div class="archives-authors-section">
    <ul>
        <?php
            wp_list_authors('exclude_admin=0&optioncount=
                1'); ?>
        </ul>
    </div>

```

```

<h1 class="widget-title">By Month <i class="fa
fa-calendar" style="vertical-align: baseline;">
</i></h1>&nbsp;
<div class="archives-by-month-section">
    <p><?php
        wp_get_archives('type=monthly&format=custom&after=
            |'); ?></p>
    </div>

```

```

</div><!-- .entry-content -->

```

```

</article><!-- #post-## -->

```

THE STYLE SHEET

Lastly, let's look at the style sheet and, most importantly, the effect it gives us.

Here's the `archives-page-style.css` file:

```
.archives-widget-left {
    float: left;
    width: 50%;
}

.archives-widget-right {
    float: left;
    padding-left: 4%;
    width: 46%;
}

.archives-latest-section { }
.archives-latest-section ol {
    font-style: italic;
    font-size: 20px;
    padding: 10px 0;
}
.archives-latest-section ol li {
    padding-left: 8px;
}

.archives-authors-section { }
.archives-authors-section ul {
    list-style: none;
    text-align: center;
    border-top: 1px dotted #888;
```



```

border-bottom: 1px dotted #888;
padding: 10px 0;
font-size: 20px;
margin: 0 0 20px 0;
}

.archives-authors-section ul li {
display: inline;
padding: 0 10px;
}

.archives-authors-section ul li a {
text-decoration: none;
}

.archives-by-month-section {
text-align: center;
word-spacing: 5px;
}

.archives-by-month-section p {
border-top: 1px dotted #888;
border-bottom: 1px dotted #888;
padding: 15px 0;
}

.archives-by-month-section p a {
text-decoration: none;
}

@media only screen and (max-width: 600px) {
.archives-widget-left {
width: 100%;

```

```

}

.archives-widget-right {
    width: 100%;
}
}

```

This is mostly typography and not a lot of structural elements, except for the couple of `float` alignments, plus the responsive design block at the end.

OK, let's see the result! The image on the following page shows what it looks like on a website that already has quite a bit of content in the archive.

How To Integrate This Template With Any Theme

The custom archive page we are building here is for the Zerif Lite theme, in the official WordPress directory. However, like I said, it can be used with any theme. Here's how to do that:

1. Take the `archives-page-style.css` file and the `archives-page-functions.php` file that we built here and put them in your theme's main directory.
2. Edit the `functions.php` file of your theme and add this line at the very end: `require get_template_directory() . '/archives-page-functions.php';`.

Archives

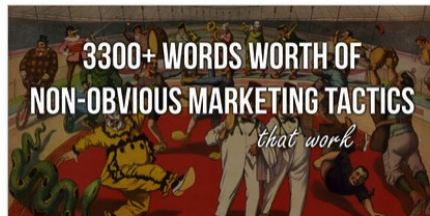


Welcome to the archives. Here, you can find all of my posts, including some of the most popular resources I published over the years.

The popular stuff:



1 Simplistically Simple Way to Simplify Your To-Do List (Hint: It Involves the -P-)



3300+ Words Worth of Non-Obvious Marketing Tactics That Work

Categories

- [Blogging & Writing](#)
- [Late Night Posting](#)
- [Marketing & Selling](#)
- [Productivity](#)
- [Reviews](#)
- [Running a Business](#)
- [Site Building](#)
- [Social Media](#)

Tags

AdSense advertising adwords affiliate marketing Backlinks [business](#) buy blogging branding conversions copywriting domain names facebook fail flickr freelancing Google Analytics Google Webmaster Tools gtd Headlines keyword research [marketing](#) mindset networking Online Business Photography Poland productivity tips self development SEO social media [success](#) traffic [twitter](#) uncalled videos web design [WordPress](#) YouTube

Last 5 Posts

1. [13 Successful Entrepreneurs Share How to Gain Confidence When Starting an Online Business](#)
2. [3300+ Words Worth of Non-Obvious Marketing Tactics That Work](#)
3. [1 Simplistically Simple Way to Simplify Your To-Do List \(Hint: It Involves the -P-\)](#)
4. [\[Downloadable\] The Words to Avoid if You Don't Want Your Emails Flagged as Spam](#)
5. [\[Real-Talk\] 19 Things You Need to Do if You Want to Be Successful](#)

Our Authors

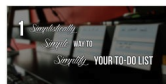
Editor E (5) [Karol K \(261\)](#)

By Month

3. Take the `page.php` file of your theme, make a copy, rename it, change the `get_template_part()` function call to `get_template_part('content', 'tmpl_archives');`, and then add the main declaration comment at the very beginning: `/* Template Name: Archive Page Custom */`.
4. Take the `content-page.php` file of your theme, make a copy, rename it to `content-tmpl_archives.php`, and include all of the custom blocks that we created in this guide right below the `the_content();` function call.
5. Test and enjoy.

My Site
Just another WordPress site

Search ...



1 Simplistically Simple Way to Simplify Your To-Do List (Hint: It Involves the -P-)



3300+ Words Worth of Non-Obvious Marketing Tactics That Work

Archives



Welcome to the archives. Here, you can find all of my posts, including some of the most popular resources I published over the years.

CATEGORIES

- [Blogging & Writing](#)
- [Late Night Posting](#)
- [Marketing & Selling](#)
- [Productivity](#)
- [Reviews](#)
- [Running a Business](#)
- [Site Building](#)
- [Social Media](#)

TAGS

[Affiliate](#) [advertising](#) [adwords](#) [affiliate](#)
[marketing](#) [Backlinks](#) [backups](#) [banners](#) [blog-](#)
[ging](#) [branding](#) [business](#) [buy button](#)
[conversions](#) [copywriting](#) [domain names](#)
[facebook](#) [fail](#) [flickr](#) [freelancing](#) [Google](#)
[Analytics](#) [Google](#) [Webmaster Tools](#) [gtd](#) [Headlines](#) [key-](#)
[word research](#) [keywords](#) [landing pages](#) [launching a](#)
[website](#) [marketing](#) [mindset](#) [network-](#)
[ing](#) [Online Business](#) [Photog-](#)
[raphy](#) [Poland](#) [productivity tips](#) [self devel-](#)
[opment](#) [SEO](#) [social media](#) [split testing](#)
[success](#) [traffic](#) [twitter](#) [upselling](#)
[videos](#) [web design](#) [WordPress](#)
[YouTube](#)

LAST 5 POSTS ↗

1. [13 Successful Entrepreneurs Share How to Gain Confidence When Starting an Online Business](#)

Our custom archive page in the Twenty Fifteen theme.

What's Next?

We've covered a lot of ground in this guide, but a lot can still be done with our archive page. We could widgetize the whole thing and erase all of the custom code elements. We could add more visual blocks for things like the latest content, and so on. 🐼

Customizing Tree-Like Data Structures In WordPress With The Walker Class

BY CARLO DANIELE 🐼

In WordPress, a navigation menu, a list of categories or pages, and a list of comments all share one common characteristic: They are the visual representation of tree-like data structures. This means that a relationship of superordination and subordination exists among the elements of each data tree.

There will be elements that are parents of other elements and, conversely, elements that are children of other elements. A reply to a comment depends logically on its parent, in the same way that a submenu item depends logically on the root element of the tree (or subtree).

Starting with version 2.1, WordPress provides the **Walker** abstract class³³, with the specific function of traversing these tree-like data structures. But an abstract class does not produce any output by itself. It has to be extended with a concrete child class that builds the HTML bricks for specific lists of items. With this precise function, WordPress provides the **Walker_Category** class to produce a nested list of categories, the **Walker_Page**

³³. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/class-wp-walker.php>

`class`³⁴, which builds a list of pages, and several other **Walker** concrete child classes.

But when we build a WordPress theme, we might need to customize the HTML list's default structure to fit our particular needs. We may want to add a description to a menu item, add custom class names or perhaps redefine the HTML structure of a list of categories or comments.

Before we begin, let's look at the most important concept in this article.

Tree-Like Data Structures

Wikipedia defines a tree³⁵ as a hierarchical organization of data:

“A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the ‘children’), with the constraints that no reference is duplicated, and none points to the root.”

So, the structure is characterized by a root element (at the first level), parent elements (which are directly referenced to subordered elements, named children), siblings (which are elements placed at the same hierarchical level) and descendant and ancestor elements (which are connected by more than one parent-child relation).

³⁴. https://codex.wordpress.org/Class_Reference/Walker_Page

³⁵. [https://en.wikipedia.org/wiki/Tree_\(data_structure\)](https://en.wikipedia.org/wiki/Tree_(data_structure))

#	Name	Type	Collation	Attributes	Null	Default	Extra
1	comment_ID 📄	bigint(20)		UNSIGNED	No	None	AUTO_INCREMENT
2	comment_post_ID	bigint(20)		UNSIGNED	No	0	
3	comment_author	tinytext	utf8mb4_unicode_ci		No	None	
4	comment_author_email	varchar(100)	utf8mb4_unicode_ci		No		
5	comment_author_url	varchar(200)	utf8mb4_unicode_ci		No		
6	comment_author_IP	varchar(100)	utf8mb4_unicode_ci		No		
7	comment_date	datetime			No	0000-00-00 00:00:00	
8	comment_date_gmt	datetime			No	0000-00-00 00:00:00	
9	comment_content	text	utf8mb4_unicode_ci		No	None	
10	comment_karma	int(11)			No	0	
11	comment_approved	varchar(20)	utf8mb4_unicode_ci		No	1	
12	comment_agent	varchar(255)	utf8mb4_unicode_ci		No		
13	comment_type	varchar(20)	utf8mb4_unicode_ci		No		
14	comment_parent	bigint(20)		UNSIGNED	No	0	
15	user_id	bigint(20)		UNSIGNED	No	0	

Each element in the **wp_comments** is referenced to its parent by the value of the **comment_parent** field.

As an example of this kind of structure, let's take the **wp_comments** table from the WordPress database. The **comment_parent** field stores the parent ID of each element in the structure, making it possible to create the reference from the child node to its parent.

What We'll Be Doing

Before moving forward, I'll show you a simple example of a concrete child class producing the HTML markup of a category list.

The category list in WordPress is printed out by the **wp_list_categories** template tag. When we call this function, it executes the **Walker_Category** class, which actually builds the HTML structure, producing something like the following code:

```
<li class="categories">Categories
    <ul>
        <li class="cat-item cat-item-10">
```



```

<a href="http://example.com/wordpress/
category/coding/">Coding</a>
<ul class="children">
    <li class="cat-item cat-item-39">
        <a href="http://example.com/wordpress/
category/coding/php/">PHP</a>
    </li>
</ul>
</li>
<li class="cat-item cat-item-11">
    <a href="http://example.com/wordpress/
category/design/">Design</a>
</li>
</ul>
</li>

```

`wp_list_categories` will print the output produced by the `Walker_Category` concrete class. It will be a wrapper list item holding a nested list of categories.

Now, suppose you don't like this kind of list, and you want to print custom HTML code. You can do this trick by defining your own `Walker` extension. In your theme's `functions.php` file, add the following code:

```

class My_Custom_Walker extends Walker
{
    public $tree_type = 'category';

    public $db_fields = array ('parent' => 'parent',
        'id' => 'term_id');
}

```

```

public function start_lvl( &$output, $depth = 0,
$args = array() ) {
    $output .= "<ul class='children'>\n";
}

public function end_lvl( &$output, $depth = 0,
$args = array() ) {
    $output .= "</ul>\n";
}

public function start_el( &$output, $category,
$depth = 0, $args = array(), $current_object_id =
0 ) {
    $output .= "<li>" . $category->name . "\n";
}

public function end_el( &$output, $category,
$depth = 0, $args = array() ) {
    $output .= "</li>\n";
}
}

```

Even if you don't know a lot about PHP classes³⁶, the code is quite descriptive. The `start_lvl()` method prints the `start` tag for each level of the tree (usually a `` tag), while `end_lvl()` prints the end of each level. In the same way, the `start_el` and `end_el` methods open and close each item in the list.

36. <http://php.net/manual/en/language.oop5.basic.php>

This is just a basic example, and we won't dive deep into the **Walker** properties and methods for now. I'll just say that the concrete child class **My_Custom_Walker** extends the abstract class **Walker**, redefining some of its properties and methods.

As I wrote above, the category list is printed out by the **wp_list_categories** template tag, but the HTML structure is built by the **Walker_Category** class. WordPress allows us to pass a custom **walker** class to the **template** tag. The new **walker** will build a custom HTML structure that will be printed on the screen by **wp_list_categories**.

As a first example, let's create a shortcode that will print new markup for the lists of categories. In your **functions.php** file, add the following code:

```
function my_init() {
    add_shortcode( 'list', 'my_list' );
}
add_action('init', 'my_init');

function my_list( $atts ){
    $list = wp_list_categories( array( 'echo' => 0,
    'walker' => new My_Custom_Walker() ) );
    return $list;
}
```

We are passing the function an array of two arguments: **echo** keeps the result in a variable, and **walker** sets a custom **Walker** (or **Walker_Category**) child class.

Finally, the shortcode `[list]` will print the resulting HTML code:

```
<ul>
  <li class="categories">Categories
    <ul>
      <li>Coding
        <ul class="children">
          <li>PHP</li>
        </ul>
      </li>
      <li>Design</li>
    </ul>
  </li>
</ul>
```

As you can see, the example is very basic but should give you an idea of our goals.

The Walker Class And Its Extensions

The `Walker` class³⁷ is defined in `wp-includes/class-wp-walker.php`³⁸ as “a class for displaying various tree-like structures.”

As I mentioned before, it’s an abstract class³⁹ and does not produce any output by itself; rather, it has to be extended by defining one or more concrete child classes.

³⁷. https://codex.wordpress.org/Class_Reference/Walker

³⁸. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/class-wp-walker.php>

³⁹. <http://php.net/manual/en/language.oop5.abstract.php>

Only these concrete child classes will produce the HTML markup. WordPress provides several extensions of the **Walker** class, each one producing a hierarchical HTML structure.

Look at the table below:

Class name	Defined in	Extends
Walker_Page	post-template.php ⁴⁰	Walker
Walker_PageDropdown	post-template.php ⁴¹	Walker
Walker_Category	category-template.php ⁴²	Walker
Walker_CategoryDropdown	category-template.php ⁴³	Walker
Walker_Category_Checklist	template.php ⁴⁴	Walker
Walker_Comment	comment-template.php ⁴⁵	Walker

⁴⁰. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/post-template.php#L1325>

⁴¹. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/post-template.php#L1471>

⁴². <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/category-template.php#L962>

⁴³. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/category-template.php#L1163>

⁴⁴. <https://core.trac.wordpress.org/browser/tags/4.2.2/src/wp-admin/includes/template.php#L24>

⁴⁵. <http://core.trac.wordpress.org/browser/tags/4.2.2/src/wp-includes/comment-template.php#L1662>

Walker_Nav_Menu	nav-menu-template.php ⁴⁶	Walker
Walker_Nav_Menu_Checklist	nav-menu.php ⁴⁷	Walker_Nav_Menu
Walker_Nav_Menu_Edit	nav-menu.php ⁴⁸	Walker_Nav_Menu

This table shows the built-in **Walker** child classes, the template files they are defined in, and the corresponding parent class.

The goal of this article is to put the **Walker** class to work, so we need to dive into its structure.

The Walker Class' Structure

Walker is defined in [wp-includes/class-wp-walker.php](#)⁴⁹. It declares four properties and six main methods, most of which are left empty and should be redefined by the concrete child classes.

The properties are:

- **\$tree_type**
- **\$db_fields**

⁴⁶. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/nav-menu-template.php#L16>

⁴⁷. <https://core.trac.wordpress.org/browser/tags/4.2.2/src/wp-admin/includes/nav-menu.php#L237>

⁴⁸. <https://core.trac.wordpress.org/browser/tags/4.2.2/src/wp-admin/includes/nav-menu.php#L10>

⁴⁹. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/class-wp-walker.php>

- `$max_pages`
- `$has_children`

`$TREE_TYPE`

This is a string or an array of the data handled by the `Walker` class and its extensions.

```
public $tree_type;
```

The `Walker` class declares the property but does not set its value. To be used, it has to be redeclared by the concrete child class. For example, the `Walker_Page` class declares the following `$tree_type` property:

```
public $tree_type = 'page';
```

Whereas `Walker_Nav_menu` declares the following `$tree_type`:

```
public $tree_type = array( 'post_type', 'taxonomy',  
    'custom' );
```

`$DB_FIELDS`

Just like `$tree_type`, `$db_fields` is declared with no value assigned. In the `Walker` class' documentation, it just says that its value is an array:

```
public $db_fields;
```

The elements of the array should set the database fields providing the ID and the parent ID for each item of the traversed data set.

The `Walker_Nav_Menu` class redeclares `$db_fields` as follows:

```
public $db_fields = array( 'parent' =>
'menu_item_parent', 'id' => 'db_id' );
```

`$db_fields['parent']` and `$db_fields['id']` will be used by the `display_element` method.

`$MAX_PAGES`

This keeps in memory the maximum number of pages traversed by the `paged_walker` method. Its initial value is set to `1`.

```
public $max_pages = 1;
```

`$HAS_CHILDREN`

This boolean is set to `true` if the current element has children.

```
public $has_children;
```

After the properties, the methods are:

- `start_lvl`
- `end_lvl`
- `start_el`
- `end_el`
- `display_element`

- walk

START_LVL

This method is executed when the **Walker** class reaches the root level of a new subtree. Usually, it is the container for subtree elements.

```
public function start_lvl( &$output, $depth = 0,
$args = array() ) {}
```

start_lvl() takes three arguments:

Argument	Type	Description
\$output	string	passed by reference; used to append additional content
\$depth	int	depth of the item
\$args	array	an array of additional arguments

Because it produces HTML output, **start_lvl** should be redefined when extending the **Walker** class. For instance, the **start_lvl** method of the **Walker_Nav_Menu** class will output a **ul** element and is defined as follows:

```
public function start_lvl( &$output, $depth = 0,
$args = array() ) {
    $indent = str_repeat("\t", $depth);
    $output .= "\n$indent<ul class=\"sub-menu\">\n";
}
```

END_LVL

The second method of the **Walker** class closes the tag previously opened by **start_lvl**.

```
public function end_lvl( &$output, $depth = 0, $args
= array() ) {}
```

The **end_lvl** method of **Walker_Nav_Menu** closes the unordered list:

```
public function end_lvl( &$output, $depth = 0, $args
= array() ) {
    $indent = str_repeat("\t", $depth);
    $output .= "$indent</ul>\n";
}
```

START_EL

This method opens the tag corresponding to each element of the tree. Obviously, if **start_lvl** opens a **ul** element, then **start_el** must open an **li** element. The **Walker** class defines **start_el** as follows:

```
public function start_el( &$output, $object, $depth =
0, $args = array(), $id = 0 ) {}
```

Argument	Type	Description
\$output	string	passed by reference; used to append additional content
\$object	object	the data object
\$depth	int	depth of the item
\$args	array	an array of additional arguments

<code>\$id</code>	int	current item ID
-------------------	-----	-----------------

END_EL

It closes the tag opened by `start_el`.

```
public function end_el( &$output, $object, $depth =
0, $args = array() ) {}
```

Finally, we've reached the core of the **Walker** class. The following two methods are used to iterate over the elements of the arrays of objects retrieved from the database.

DISPLAY_ELEMENT

I won't show the full code here, because you can find it in the [WordPress Trac](https://core.trac.wordpress.org/browser/trunk/src/wp-includes/class-walker.php#L112)⁵⁰. I'll just say that this method displays the elements of the tree.

```
public function display_element( $element,
&$children_elements, $max_depth, $depth, $args,
&$output ) {}
```

`display_element` takes the following arguments:

Argument	Type	Description
<code>\$element</code>	object	the data object
<code>\$children_elements</code>	array	list of elements to continue traversing

⁵⁰. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/class-walker.php#L112>

<code>\$max_depth</code>	int	maximum depth to traverse
<code>\$depth</code>	int	depth of current element
<code>\$args</code>	array	an array of arguments
<code>\$output</code>	string	passed by reference; used to append additional content

This method does not output HTML on its own. The markup will be built by a call to each of the previously described methods: `start_lvl`, `end_lvl`, `start_el` and `end_el`.

WALK

`walk` is the core of the `Walker` class. It iterates over the elements of the tree depending on the value of `$max_depth` argument (see the Trac for the [full code⁵¹](https://core.trac.wordpress.org/browser/trunk/src/wp-includes/class-wp-walker.php#L175)).

```
public function walk( $elements, $max_depth ) {}
```

`walk` takes two arguments.

Argument	Type	Description
<code>\$elements</code>	array	an array of elements
<code>\$max_depth</code>	int	maximum depth to traverse

Other methods are used for more specific purposes.

⁵¹. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/class-wp-walker.php#L175>

PAGED_WALK

This builds a page of nested elements. The method establishes which elements of the data tree should belong to a page, and then it builds the markup by calling `display_element` and outputs the result.

```
public function paged_walk( $elements, $max_depth,
$page_num, $per_page ) {}
```

GET_NUMBER_OF_ROOT_ELEMENTS

This gets the number of first-level elements.

```
public function get_number_of_root_elements(
$elements ){} 
```

UNSET_CHILDREN

This last method unsets the array of child elements for a given root element.

```
public function unset_children( $e,
&$children_elements ){} 
```

Now that we've introduced the `Walker` properties and methods, we can explore one of its possible applications: changing the HTML structure of the navigation menu. The default markup for navigation menus is produced by the `Walker_Nav_Menu` concrete class. For this reason, we'll create a new concrete `Walker` child class based on `Walker_Nav_Menu`, and we'll pass it to `wp_nav_menu` to replace the output.

But is extending the `Walker_Nav_Menu` class always necessary when rebuilding a menu? Of course not!

Most of the time, a call to the `wp_nav_menu` template tag will suffice.

Menu Structure

Drag each item into the order you prefer. Click the arrow on the right of the item to reveal additional configuration options.

Final Conference	Event ▼
Final result	Page ▼
Events	Custom Link ▼
Partners	Page ▼
Contact	Page ▼

Menu Settings

Auto add pages	<input type="checkbox"/> Automatically add new top-level pages to this menu
Theme locations	<input checked="" type="checkbox"/> Primary Menu

In the “Menus” editing page, setting theme locations for each custom menu is possible.

Basic Customization Of The Navigation Menu: The `wp_nav_menu` Template Tag

The navigation menu can be included in the theme’s template files with a call to the `wp_nav_menu()` template tag.

This function takes just one argument, an array of parameters that is well described in the Codex⁵².

`wp_nav_menu()` can be included in your templates as follows:

```
$defaults = array(
    'theme_location' => '',
    'menu'           => '',
    'container'      => 'div',
    'container_class' => '',
    'container_id'   => '',
    'menu_class'     => 'menu',
    'menu_id'        => '',
    'echo'           => true,
    'fallback_cb'    => 'wp_page_menu',
    'before'         => '',
    'after'          => '',
    'link_before'    => '',
    'link_after'     => '',
    'items_wrap'     => '<ul id="%1$s" class="%2$s">
                        %3$s</ul>',
    'depth'          => 0,
    'walker'         => ''
);

wp_nav_menu( $defaults );
```

WordPress provides many parameters to configure the navigation menu. We can change the menu's container (it

⁵². https://codex.wordpress.org/Function_Reference/wp_nav_menu

defaults to a `div`), the container's CSS class and ID, as well as the text strings and markup to be included before and after the anchor element and before and after the item's title. Furthermore, we can change the root element's structure (`items_wrap`) and the depth of the tree.

So, we don't need to extend the `Walker` (or the `Walker_Nav_Menu`) class each time we want to make changes to the menu's structure — only when we have to produce more advanced structural customizations. This happens when we have to assign CSS classes to the menu elements when a specific condition occurs, or when we have to add data or HTML code to the menu items.

This can be done by setting a value for the `walker` parameter, which will be nothing but an instance of a custom concrete class. So, from now on, we'll dive more and more deeply into WordPress menus, from changing the menu structure to adding custom fields to the menu items' editing boxes. As I said, we'll do this job by extending the `Walker_Nav_Menu`, so it's time to get acquainted with it.

The Walker_Nav_Menu Class

The `Walker_Nav_Menu` class is defined in `wp-includes/nav-menu-template.php`⁵³. This is the class used by WordPress to build the navigation menu's structure. Each time you want to make relevant changes to the menu's default structure, you can extend `Walker_Nav_Menu`.

⁵³. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/nav-menu-template.php>

The concrete class redeclares the `$tree_type` and `$db_fields` properties and the `start_lvl`, `end_lvl`, `start_el` and `end_el` methods.

```
class Walker_Nav_Menu extends Walker {

    public $tree_type = array( 'post_type',
                              'taxonomy', 'custom' );

    public $db_fields = array( 'parent' =>
                              'menu_item_parent', 'id' => 'db_id' );

    public function start_lvl( &$output, $depth = 0,
                              $args = array() ) {
        $indent = str_repeat("\t", $depth);
        $output .= "\n$indent<ul class=\"sub-menu\">\n";
    }

    public function end_lvl( &$output, $depth = 0,
                              $args = array() ) {
        $indent = str_repeat("\t", $depth);
        $output .= "$indent</ul>\n";
    }

    public function start_el( &$output, $item, $depth
                              = 0, $args = array(), $id = 0 ) {
        // code below
    }

    public function end_el( &$output, $item, $depth
                              = 0, $args = array() ) {
```

```

        $output .= "</li>\n";
    }

} // Walker_Nav_Menu

```

The `start_el` public method builds the HTML markup of the opening `li` tag for each menu item. It is defined as follows:

```

public function start_el( &$output, $item, $depth =
0, $args = array(), $id = 0 ) {
    $indent = ( $depth ) ? str_repeat( "\t", $depth )
    : '';

    $classes = empty( $item->classes ) ? array() :
(array) $item->classes;
    $classes[] = 'menu-item-' . $item->ID;

    $class_names = join( ' ', apply_filters(
'nav_menu_css_class', array_filter( $classes ),
$item, $args, $depth ) );
    $class_names = $class_names ? ' class="' .
esc_attr( $class_names ) . '"' : '';

    $id = apply_filters( 'nav_menu_item_id',
'menu-item-' . $item->ID, $item, $args, $depth );
    $id = $id ? ' id="' . esc_attr( $id ) . '"' : '';

    $output .= $indent . '<li' . $id . $class_names
    . '>';
}

```

```

$atts = array();
$atts['title'] = ! empty( $item->attr_title ) ?
$item->attr_title : '';
$atts['target'] = ! empty(
$item->target ) ? $item->target : '';
$atts['rel'] = ! empty(
$item->xfn ) ? $item->xfn : '';
$atts['href'] = ! empty( $item->url ) ?
$item->url : '';

$atts = apply_filters( 'nav_menu_link_attributes',
$atts, $item, $args, $depth );

$attributes = '';
foreach ( $atts as $attr => $value ) {
    if ( ! empty( $value ) ) {
        $value = ( 'href' === $attr ) ? esc_url(
$value ) : esc_attr( $value );
        $attributes .= ' ' . $attr . '="' . $value .
        '"';
    }
}

$item_output = $args->before;
$item_output .= '<a'. $attributes .'>';

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

```

```

$item_output .= $args->after;

$output .= apply_filters(
    apply_filters('walker_nav_menu_start_el',
    apply_filters($item_output, $item, $depth, $args ));
}

```

The code is quite self-explanatory:

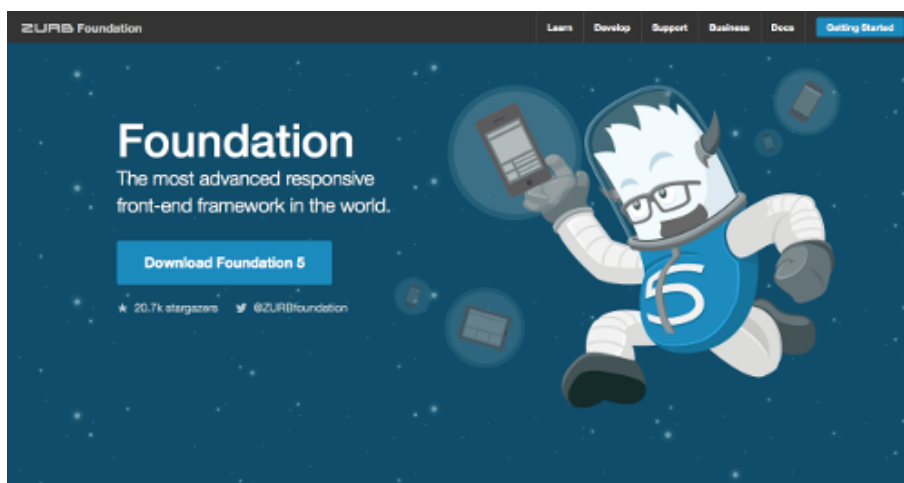
- **\$indent** stores a number of `'/t'` strings corresponding to **\$depth**'s value;
- **\$classes** is an array of the CSS classes assigned to the list item;
- **\$id** records the element's ID, composed by the prefix `'menu-item-'` and the item's ID retrieved from the database;
- **\$atts** is an array of the element's attributes;
- **\$item_output** keeps track of the HTML code before it is assigned to **\$output**;
- **\$output** stores the HTML markup.

Extending The Walker_Nav_Menu Class

When building your custom navigation menu markup, you might decide to extend the **Walker** class itself or its concrete child class **Walker_Nav_Menu**. If you opt to extend the **Walker** class, you'll need to define all necessary properties and methods. Otherwise, if you choose to ex-

tend the concrete child class, you'll just need to define those methods whose output has to be changed.

The following example describes a real-life situation in which the default menu's structure has to be changed in order to integrate a WordPress theme with the Foundation 5 framework.



Foundation 5's home page.

A Working Example: The Foundation Top Bar As WordPress Navigation Menu

Foundation 5⁵⁴ comes with a flexible grid system and with plugins and components that make it easy to develop solid and responsive websites. So, chances are that you'll decide to enrich your theme with all of this great stuff.

First, you need to include all necessary scripts and style sheets in your theme. This isn't our topic, so we

⁵⁴. <http://foundation.zurb.com/>

won't dive deep into the configuration, but you can pull out all necessary information directly from [Foundation's documentation](#)⁵⁵ and the [WordPress Codex](#)⁵⁶.

Here, we'll see how to force WordPress to display [Foundation Top Bar](#)⁵⁷ as a navigation menu.

WordPress should print something like the following HTML:

```
<nav class="top-bar" data-topbar role="navigation">
  <ul class="title-area">
    <li class="name">
      <h1><a href="#">My Site</a></h1>
    </li>
    <!-- Remove the class "menu-icon" to get rid of
    menu icon. Take out "Menu" to just have icon alone
    -->
    <li class="toggle-topbar menu-icon"><a
href="#"><span>Menu</span></a></li>
  </ul>

  <div class="top-bar-section">
    <!-- Left Nav Section -->
    <ul class="left">
      <li class="active"><a href="#">Right Button
Active</a></li>
      <li class="has-dropdown">
        <a href="#">Left Button Dropdown</a>
```

⁵⁵. <http://foundation.zurb.com/docs/javascript.html>

⁵⁶. https://codex.wordpress.org/Function_Reference/wp_enqueue_script

⁵⁷. <http://foundation.zurb.com/docs/components/topbar.html>

```

        <ul class="dropdown">
            <li><a href="#">First link in
            dropdown</a></li>
            <li class="active"><a href="#">Active
            link in dropdown</a></li>
        </ul>
    </li>
</ul>
</div>
</nav>

```

To achieve our goal, add the following code to your `header.php` file or to whichever template file contains the navigation menu:

```

<nav class="top-bar" data-topbar role="navigation">
    <ul class="title-area">
        <li class="name">
            <h1><a href="#"><?php echo get_bloginfo();
            ?></a></h1>
        </li>
        <li class="toggle-topbar menu-icon"><a
        href="#"><span><!--<?php echo get_bloginfo();
        ?>--></span></a></li>
    </ul>
    <?php wp_nav_menu( array(
        'theme_location'    => 'primary',
        'container_class'   => 'top-bar-section',
        'menu_class'        => 'left',
        'walker'            => new

```

```
Custom_Foundation_Nav_Menu()) ); ?>
</nav>
```

We have set the Foundation CSS class `top-bar-section`, along with a custom `Walker` class, on the navigation menu's container, location and alignment. Now, we can define `Custom_Foundation_Nav_Menu`:

```
class Custom_Foundation_Nav_Menu extends
Walker_Nav_Menu {
    public function start_lvl( &$output, $depth = 0,
    $args = array() ) {
        $indent = str_repeat("\t", $depth);

        // add the dropdown CSS class
        $output .= "\n$indent<ul class=\"sub-menu
        dropdown\">\n";
    }
    public function display_element( $element,
    &$children_elements, $max_depth, $depth = 0,
    $args, &$output ) {

        // add 'not-click' class to the list item
        $element->classes[] = 'not-click';

        // if element is current or is an ancestor of
        // the current element, add 'active' class to
        // the list item
        $element->classes[] = ( $element->current ||
        $element->current_item_ancestor ) ? 'active' :
        '';
    }
}
```



```

        // if it is a root element and the menu is not
        // flat, add 'has-dropdown' class from
        // https://core.trac.wordpress.org/browser
        // /trunk/src/wp-includes/
        // class-wp-walker.php#L140
        $element->has_children = ! empty(
            $children_elements[ $element->ID ] );
        $element->classes[] = ( $element->has_children
            && 1 !== $max_depth ) ? 'has-dropdown' : '';

        // call parent method
        parent::display_element( $element,
            $children_elements, $max_depth, $depth, $args,
            $output );
    }
}

```

As you can see, we’ve redefined the `start_lvl` and `display_element` methods. The first one generates the markup of the opening `ul` tag and assigns the `dropdown` CSS class.

The second method, `display_element`, is described in the Trac⁵⁸: It’s a method to “traverse elements to create a list from elements,” so it is a good place to make changes to the menu items.

Here we’ve accessed the `has_children`, `classes`, `current` and `current_item_ancestor` properties and

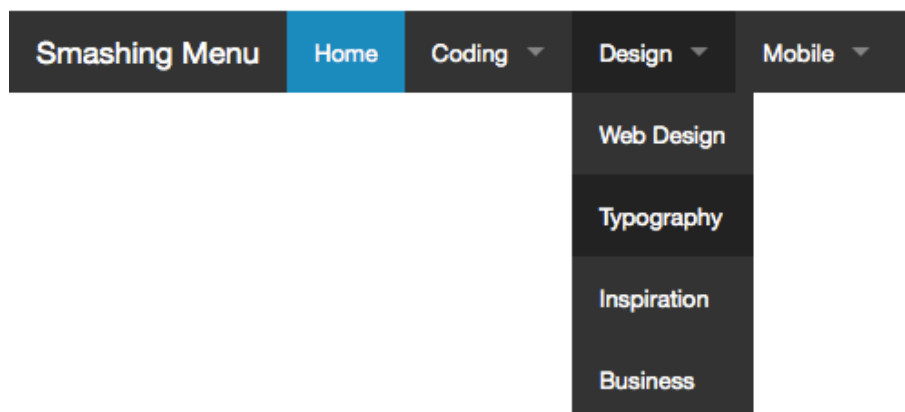
⁵⁸. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/class-wp-walker.php#L112>

passed the updated `$element object` to the parent `display_element` method. That's enough to achieve our goal, but we could do much more on the menu items. If you call `var_dump` on the `$element` object, you'll see all of the available properties at your disposal to build more advanced navigation menus.

And, as we'll see in a moment, we can add new properties to the object.

Smashing Menu

Just another WordPress site

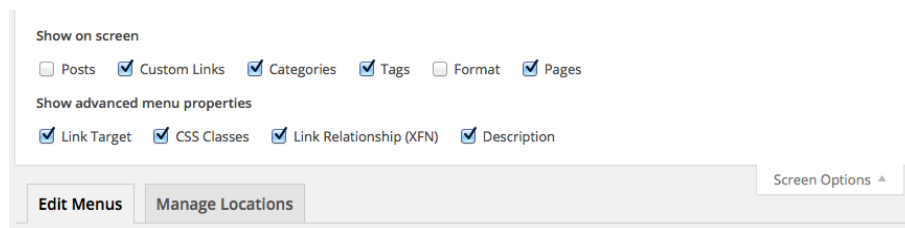


The Foundation Top Bar on a WordPress website.

The menu is up and running, but we may want deeper customization. In the next example, we'll get more from our navigation menu, allowing the website administrator to prepend icons to the items's titles the easy way: directly from the administration panel. In our example, we'll use Foundation Icon Fonts 3⁵⁹.

Adding Fields To The WordPress Menu Items' Editing Box

Before we start coding, let's open the menu editing page and make sure that all of the advanced menu properties in the "Screen Options" tab are checked.



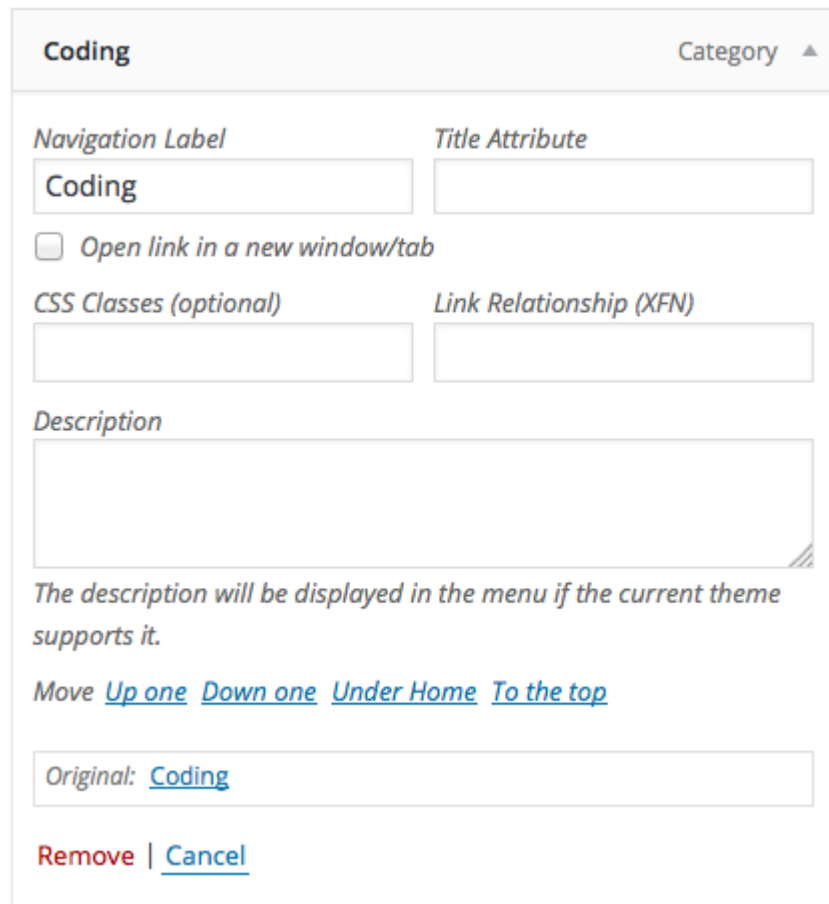
WordPress' "Screen Options" tab on the administration page for menus.

Each checkbox enables or disables certain fields in the menu item's editing box. (See image on the next page.)

But we can do more than add or change field values. The menu items are considered specific post types, and the values of the menu items' fields are stored in the database as hidden custom fields. So we can add a new menu item's field exactly as we do with regular posts' custom fields.

Any kind of form field is allowed: inputs, checkboxes, textboxes and so on.

59. <http://zurb.com/playground/foundation-icon-fonts-3>



The screenshot shows the 'Coding' menu item editing box. It has a title bar with 'Coding' and a 'Category' dropdown. The form contains several fields: 'Navigation Label' (containing 'Coding'), 'Title Attribute' (empty), a checkbox for 'Open link in a new window/tab', 'CSS Classes (optional)' (empty), 'Link Relationship (XFN)' (empty), and a 'Description' text area. Below the text area is a note: 'The description will be displayed in the menu if the current theme supports it.' and a set of links: 'Move Up one Down one Under Home To the top'. At the bottom, there is a field for 'Original: Coding' and two buttons: 'Remove' and 'Cancel'.

The menu items' editing box.

In the following example, I will show you how to add a simple text field that enables the website administrator to add a new property to the `$item` object. It will be stored as a custom field and will be used to show data in the website's front end.

To do that, we will:

1. register a custom field for the navigation menu item,
2. save the new custom field's value,
3. set up a new `Walker` class for the edit menu tree.

STEP 1: REGISTER A CUSTOM FIELD FOR THE NAV MENU ITEM

First, we'll have to register a new custom field for the navigation menu item in the `functions.php` file:

```
/**
 * Add a property to a menu item
 *
 * @param object $item The menu item object.
 */
function custom_nav_menu_item( $item ) {
    $item->icon = get_post_meta( $item->ID,
        '_menu_item_icon', true );
    return $item;
}
add_filter( 'wp_setup_nav_menu_item',
    'custom_nav_menu_item' );
```

`wp_setup_nav_menu_item` filters the navigation menu's `$item` object, allowing us to add the `icon` property.

STEP 2: SAVE THE USER'S INPUT

When the user submits the form from the menu's administration page, the following callback will store the fields' values in the database:

```
/**
 * Save menu item custom fields' values
 *
 * @link https://codex.wordpress.org/
 * Function_Reference/sanitize_html_class
```

```

*/
function custom_update_nav_menu_item( $menu_id,
$menu_item_db_id, $menu_item_args ){
    if ( is_array( $_POST['menu-item-icon'] ) ) {
        $menu_item_args['menu-item-icon'] =
        $_POST['menu-item-icon'][$menu_item_db_id];
        update_post_meta( $menu_item_db_id,
            '_menu_item_icon', sanitize_html_class(
                $menu_item_args['menu-item-icon'] ) );
    }
}
add_action( 'wp_update_nav_menu_item',
'custom_update_nav_menu_item', 10, 3 );

```

When updating the menu items, this action calls `custom_update_nav_menu_item()`, which will sanitize and update the value of the `_menu_item_icon` meta field.

Now we have to print the custom field's markup.

STEP 3: SET UP A NEW WALKER FOR THE EDIT MENU TREE

The structure of the menu's administration page is built by the `Walker_Nav_Menu_Edit` class, which is an extension of `Walker_Nav_Menu`. To customize the menu items' editing boxes, we'll need a new custom `Walker_Nav_Menu` child class based on the `Walker_Nav_Menu_Edit` class.

To set a custom `Walker`, this time we'll need the following filter:

```

add_filter( 'wp_edit_nav_menu_walker', function(
$class ){ return 'Custom_Walker_Nav_Menu_Edit'; } );

```

When fired, this filter executes an anonymous function⁶⁰ that sets a custom class that will build the list of menu items.

Finally, the new **Walker** can be declared. We won't reproduce the full code here. Just copy and paste the full **Walker_Nav_Menu_Edit** code from the Trac⁶¹ into your custom class and add the custom field markup as shown below:

```
class Custom_Walker_Nav_Menu_Edit extends
Walker_Nav_Menu {
    public function start_lvl( &$output, $depth = 0,
    $args = array() ) {}
    public function end_lvl( &$output, $depth = 0,
    $args = array() ) {}
    public function start_el( &$output, $item, $depth
    = 0, $args = array(), $id = 0 ) {
        ...

        <p class="field-xfn description
description-thin">
            <label for="edit-menu-item-xfn-<?php echo
$item_id; ?>">
                <?php _e( 'Link Relationship (XFN)' );
                ?><br />
                <input type="text" id="edit-menu-item-xfn-
                <?php echo $item_id; ?>" class="widefat
```

⁶⁰. <http://php.net/manual/en/functions.anonymous.php>

⁶¹. <https://core.trac.wordpress.org/browser/tags/4.2.4/src/wp-admin/includes/nav-menu.php>

```

        code edit-menu-item-xfn"
        name="menu-item-xfn[<?php echo $item_id;
        ?>]"
        value="<?php echo esc_attr( $item->xfn );
        ?>" />
    </label>
</p>
<p class="field-custom description
description-thin">
<label for="edit-menu-item-icon-<?php echo
$item_id; ?>">
    <?php _e( 'Foundation Icon' ); ?><br />
    <input type="text"
    id="edit-menu-item-icon-<?php echo
    $item_id; ?>" class="widefat code
    edit-menu-item-icon"
    name="menu-item-icon[<?php echo $item_id;
    ?>]" value="<?php echo esc_attr(
    $item->icon ); ?>" />
</label>
</p>
...
}
}

```

Now, with the new input field in place, the website administrator will be able to add the new icon property to the menu `$item` object.

Coding
Category ▲

Navigation Label
Title Attribute

Coding

☐ Open link in a new window/tab

CSS Classes (optional)
Link Relationship (XFN)

Foundation Icon

html5

Description

The description will be displayed in the menu if the current theme supports it.

Move [Up one](#) [Down one](#) [To the top](#)

Original: [Coding](#)

Remove | [Cancel](#)

A customized version of the menu items' editing box.

At this time, the `Walker_Nav_Menu` class won't be able to access the new property value; so, the value of `$item->icon` would not be available to build the menu structure. To make it accessible, in the `Custom_Foundation_Nav_Menu` class of our previous example, we will redefine the `start_el` method. The easiest way to proceed is to copy

the code from the `Walker_Nav_Menu` class⁶² and paste it in our custom class, editing it where necessary.

So, paste the code and jump to the bottom of the new `start_el` method and make the following edits:

```
public function start_el( &$output, $item, $depth =
0, $args = array(), $id = 0 ) {

    ...

    $item_output = $args->before;
    $item_output .= '<a'. $attributes .'>';

    if( !empty( $item->icon ) )
        $item_output .= '<i class="fi-' . $item->icon .
        '" style="margin-right: .4em"></i>';

    $item_output .= $args->link_before .
    apply_filters( 'the_title', $item->title,
    $item->ID ) . $args->link_after;
    $item_output .= '</a>';
    $item_output .= $args->after;

    $output .= apply_filters(
    'walker_nav_menu_start_el', $item_output, $item,
    $depth, $args );
}
```

62. <https://core.trac.wordpress.org/browser/trunk/src/wp-includes/nav-menu-template.php>

No other changes are being done here except the condition that checks the value of the `$item->icon` property. If a value has been set, a new `i` element is attached to `$item_output` and assigned to the proper CSS class.

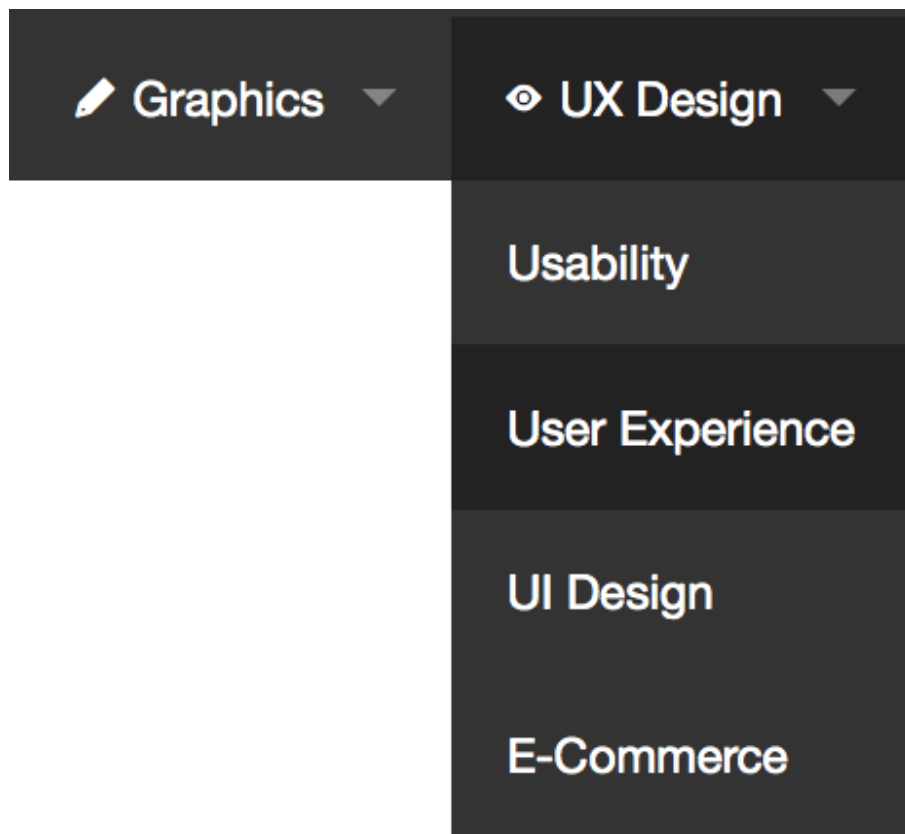
Finally, the following markup shows the new HTML menu structure.

```
<div class="top-bar-section">
  <ul id="menu-nav-menu" class="left">
    <li id="menu-item-46" class="menu-item
    menu-item-type-custom menu-item-object-custom
    current-menu-item current_page_item
    menu-item-home not-click active menu-item-46">
      <a href="http://localhost:8888/wordpress/">
        <i class="fi-social-smashing-mag"></i>
        <span>Home</span>
      </a>
    </li>
  </ul>
</div>
```

The image on the next page shows the final result on the desktop.

Final Notes

In this article, we've explored some of the most common uses of the `Walker` class. Note, however, that our examples do not cover all possible applications and alternative ways to take advantage of the class. But you'll discover more just by making use of your imagination and your skills as a programmer.



The custom Foundation 5 Top Bar integrated in a WordPress theme, enriched with icon fonts from Foundation Icon Font 3.

And never lose sight of the official documentation:

- [Class Reference/Walker](#)⁶³
- [Function Reference/wp_nav_menu](#)⁶⁴ 🐼

⁶³. https://codex.wordpress.org/Class_Reference/Walker

⁶⁴. https://codex.wordpress.org/Function_Reference/wp_nav_menu

Extending Advanced Custom Fields With Your Own Controls

BY DANIEL PATAKI 🍷

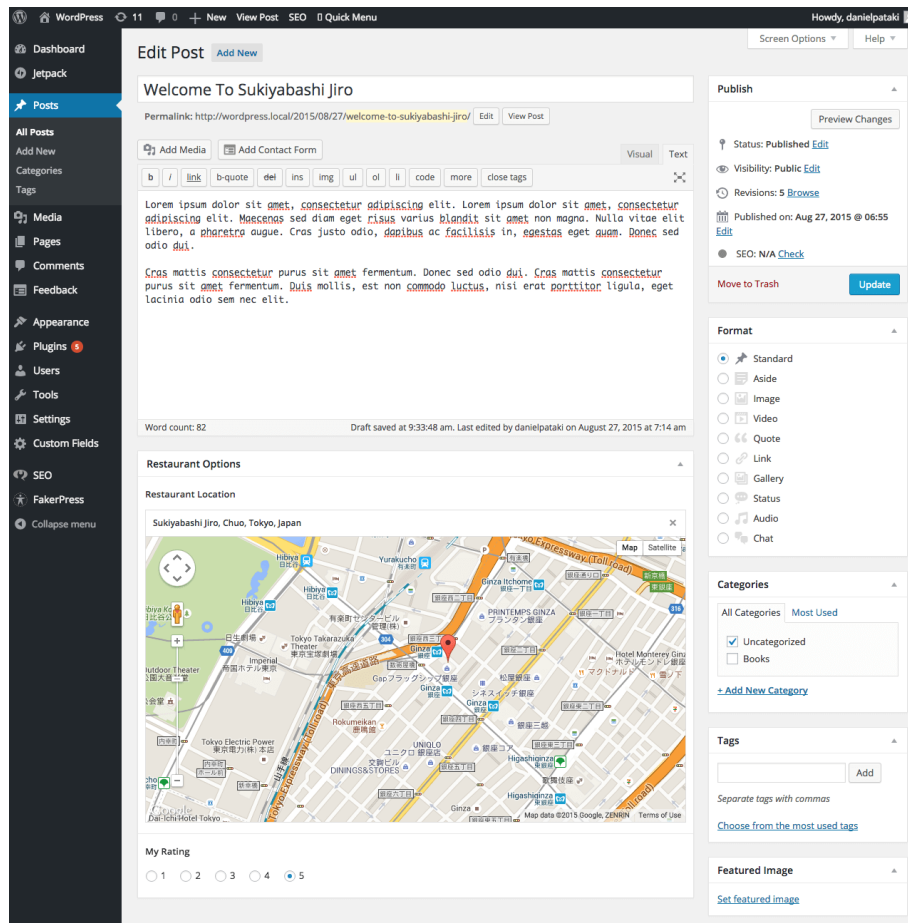
Advanced Custom Fields⁶⁵ (ACF) is a free WordPress plugin that replaces the regular custom fields interface in WordPress with something far more powerful, offering a user-friendly interface for complex fields like location maps, date pickers and more.

In this article I'll show you how you can extend ACF by adding your own controls to tailor the experience to your needs.

How ACF Works

ACF is a collection of fields which can be added to a number of locations in WordPress, such as posts, taxonomies, users and so on. They share a common interface and a WordPress-compatible saving mechanism (using meta fields and options).

⁶⁵. <http://www.advancedcustomfields.com/>



A map and a radio field created with ACF.

Each ACF field has field settings which you can think of as the back-end options for a field. Field settings allow you to control how the field behaves when displayed to the user. You may be able to control default values, how data is saved, and so on.

There is a default set shared by all fields, like field label, field name, field type, field instructions, required setting and conditional logic, but there are some field-specific settings as well.

A good example is the image field which allows users to select an image and save it. The field settings allow you to define the following:

- Return value (image object, image URL or image ID)
- Preview size (any defined image size)
- Library (all or uploaded to post)

3 Restaurant Front restaurant_front Image	
Field Label * This is the name which will appear on the EDIT page	Restaurant Front
Field Name * Single word, no spaces. Underscores and dashes allowed	restaurant_front
Field Type *	Image
Field Instructions Instructions for authors. Shown when submitting data	Select an image which shows the front facade of the restaurant
Required?	<input checked="" type="radio"/> Yes <input type="radio"/> No
Return Value Specify the returned value on front end	<input checked="" type="radio"/> Image Object <input type="radio"/> Image URL <input type="radio"/> Image ID
Preview Size Shown when entering data	<input type="radio"/> Thumbnail <input checked="" type="radio"/> Medium <input type="radio"/> Large <input type="radio"/> Full <input type="radio"/> Post Thumbnail
Library Limit the media library choice	<input checked="" type="radio"/> All <input type="radio"/> Uploaded to post
Conditional Logic	<input type="radio"/> Yes <input checked="" type="radio"/> No
Close Field	

Drag and drop to reorder + Add Field

Settings for the image field in ACF.

Fields are created within field groups which can be placed in a number of locations using powerful rule sets. Not only can you add your fields to edit profile pages, you can also restrict the visibility of a field group based on role, allowing access to admins only, for example.

Location

Rules
Create a set of rules to determine which edit screens will use these advanced custom fields

Show this field group if

User is equal to All and

Logged in User Type is equal to Administrator and

or

Add rule group

Location rules for an ACF field group.

What's Available Out Of The Box

Right now ACF has 22 fields available, which includes basic fields like number, email, multi-select and radio, but more advanced ones as well, such as taxonomy selection and the user selector. You also get a few fields that use JavaScript to create a great interface, like the map field or the date and color-picker fields.

There are also quite a few fields available in the plugin repository. I've created [six ACF plugins myself](#)⁶⁶, like the Google Font Selector and Sidebar Selector, but a quick search in the repository⁶⁷ will come up with star rating fields, recent posts, link pickers, widget area fields and so on.

If you are a developer I can also heartily recommend the [pro version of ACF](#)⁶⁸. It costs \$100 — which is quite a bit — but you can use it in unlimited projects. It contains a repeater field, a gallery field, a flexible content field and the ability to add options to options pages very easily.

⁶⁶. <https://profiles.wordpress.org/danielpataki#content-plugins>

⁶⁷. <https://wordpress.org/plugins/search.php?type=term&q=advanced+custom+fields>

⁶⁸. <http://www.advancedcustomfields.com/pro/>

The great thing about ACF is that the pro version is nice, but you don't need it in order to build something awesome. You can use the built-in fields or write your own if you need something different. Let's look at how that can be done.

Extending Advanced Custom Fields

ACF can be extended by creating a separate plugin which integrates with the main ACF plugin. The heavy lifting is done by the ACF Field Type Template⁶⁹ which you can grab from GitHub.

This includes all the files you need and has great in-line commenting that walks you through the process. It also contains all the possible functions you can use. You won't use all of them for each field, but you can leave the unwanted ones empty or delete them altogether.

In this tutorial I'll show you the steps I follow when creating an ACF plugin. I'll be creating a country selector which lets you select any country from a drop-down list. By the end you should be able to reproduce it and create your own, so let's get cracking!

STEP 1: A LOCAL ENVIRONMENT

I like to do all my WordPress work locally. I won't go into too much detail here — you can take a look at Rachel Andrew's "A Simple Workflow From Development To Deployment"⁷⁰ if you need some help. In a nutshell, I use a

⁶⁹. <https://github.com/elliotcondon/acf-field-type-template>

simple Vagrant box to create a local server and I use virtual hosts to create multiple projects.

The one additional aspect of my workflow is using symlinks to manage my plugins. I do this for three reasons:

- I can keep my plugins separate from my WordPress installs.
- A plugin can be symlinked to multiple WordPress installs which means I update the plugin in a central location and all installations use that code.
- I can use any folder structure I like which comes in handy when working with Git packages.

The process isn't too difficult. Take a look at Tom McFarlin's "[Symbolic Links with WordPress⁷¹](https://tommcfarlin.com/symbolic-links-with-wordpress/)" article for more information. Since the template is designed to be a plugin, this step is not strictly necessary but I think symlinks are worth looking into for development purposes.

STEP 2: ADDING THE PLUGIN AND RENAMING

Once you've grabbed the template from GitHub, copy and paste the whole directory into your plugins folder and rename it to `acf-country_selector`. You'll see that some of the files have **FIELD_NAME** in them; this is a placeholder for your actual field name which should be the same as the

⁷⁰. <http://www.smashingmagazine.com/2015/07/development-to-deployment-workflow/>

⁷¹. <https://tommcfarlin.com/symbolic-links-with-wordpress/>

name of the folder (after “acf-”). In our case the field name is `country_selector`.

Some of you may be wondering why I’ve used an underscore instead of a dash: it’s common practice to use dashes in file names. The use of an underscore relates to consistency and a rule for translatable strings.

We’ll need to replace `FIELD_NAME` inside files as well. In some cases the placeholder is part of a function name where we can’t use dashes. I could decide to use dashes in file names and underscores within files, but there would then be a problem with the text domain.

The text domain is set to `acf-FIELD_NAME`, which actually needs to be the same as the folder name. This excludes the use of a dash outside and an underscore inside files. Because of this I’ve decided to use underscores. There are simply fewer downsides to it.

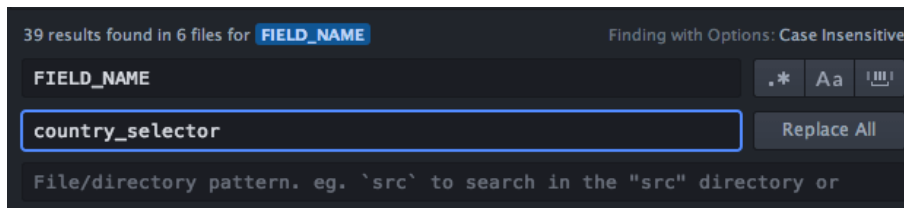
Back to creating our plugin! Replace `FIELD_NAME` in all file names with `country_selector`. Once done, I recommend dropping the whole folder in a good editor such as Sublime⁷² or Atom⁷³. Any editor that allows you to search and replace within multiple files at once will do.

Within our files there should be another set of placeholders: `FIELD_NAME` and `FIELD_LABEL`. There are a few more, but the others are used only once or twice and only in the readmes so you can replace those manually.

For this plugin I mass-replaced `FIELD_LABEL` with `Country Selector` and `FIELD_NAME` with `country_selector`.

⁷². <http://www.sublimetext.com/>

⁷³. <https://atom.io/>

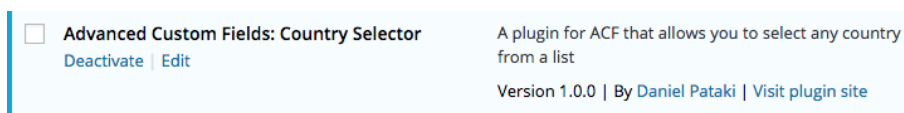


Search and replace in a project in Atom.

Finally, open `acf-country_selector.php` and fill out the meta information in the header. This will ensure that the plugin shows up in the admin with the data you provide. In this plugin's case I've filled it out like this:

```
/*
Plugin Name: Advanced Custom Fields: Country Selector
Plugin URI: http://danielpataki.com
Description: A plugin for ACF that allows you to
select any country from a list
Version: 1.0.0
Author: Daniel Pataki
Author URI: http://danielpataki.com
License: GPLv2 or later
License URI: http://www.gnu.org/licenses/gpl-2.0.html
*/
```

At this stage you can go to the plugins section and activate it. You won't see any new fields just yet, but we can start chipping away at the code.



Our country selector plugin displayed in the admin.

STEP 3: FIELD BASICS

You may have noticed that there are two similar files: *acf-country_selector-v4.php* and *acf-country_selector-v5.php*. One is for version 4 of ACF, the other for version 5. Right now, version 5 is actually the pro version, but soon the free version will also be updated to 5.

This doesn't mean that the free version will have the premium fields, but it will run on a new and improved system. Since the free version is currently version 4, I will be looking at *acf-country_selector-v4.php* only. The methods for version 5 are very nearly the same, so it shouldn't be too difficult to create both.

If you're planning on releasing your plugin, I recommend using both files. Aside from some minor changes it's really a matter of copying and pasting than anything else.

Our first stop, then, is *acf-country_selector-v4.php*, and the `__construct()` function within. Most of this is filled out for us. We'll need to modify the value of `$this->category`. This determines which group our field type will be listed under. Since there's a "Choice" group and we'll be building a select field, let's use `Choice` as the value here.

I also want to allow the person who creates the field to set an initial value for the country field. This will be really useful if the field is used on a website which predominantly uses a specific country for the field. Instead of having to go down and select "Hungary", users could make that the initial value.

The `$this->defaults` array contains the defaults for our fields, like the `initial_value` field. I'll make the de-

fault “United States”, which may be the most popular choice for the initial value. Here’s the full contents of the `__construct()` function at the end of all that:

```
// vars
$this->name = 'country_selector';
$this->label = __( 'Country Selector' );
$this->category = __( 'Choice', 'acf' ); // Basic,
Content, Choice, etc
$this->defaults = array(
    'initial_value' => 'United States'
);

// do not delete!
parent::__construct();

// settings
$this->settings = array(
    'path' => apply_filters('acf/helpers/
get_path', __FILE__),
    'dir' => apply_filters('acf/helpers/get_dir',
__FILE__),
    'version' => '1.0.0'
);
```

STEP 4: FIELD SETTINGS

The next step is to configure the field settings we’ll have. In our case this will be a single field for selecting the initial value of the front-end selector. This will be a selector with all countries selectable.

Before we create the selector, we need a list of countries. I created a little [Gist](#)⁷⁴ which lists all of them in an array. I decided to create a method that returns all countries to make sure I can create country lists anywhere, without needing to add the large array more than once.

```
function get_countries() {  
    $countries = array( 'Afghanistan' => 'Afghanistan',  
        'Albania' => 'Albania', '...' );  
    return $countries;  
}
```

I placed this function at the very end of the class, to separate it from the built-in ones. I can now use `$this->get_countries` within this class to return the array.

To add our initial value field we need to modify the contents of the `create_options()` method. Here's the full code for that function:

```
function create_options( $field )  
{  
    $field = array_merge($this->defaults, $field);  
    $key = $field['name'];  
  
    // Create Field Options HTML  
    ?>  
  
    <tr class="field_option field_option_<?php echo  
    $this->name; ?>">  
    <td class="label">
```

⁷⁴. <https://gist.github.com/danielpataki/2652bb42697b1a248761>

```

        <label><?php _e("Initial Value", 'acf');
        ?></label>
        <p class="description"><?php _e("The initial
        value of the country field", 'acf'); ?></p>
    </td>
    <td>
        <?php
        do_action('acf/create_field', array(
            'type'    =>    'select',
            'name'    =>    'fields['.$key.']['[
                        initial_value]',
            'value'   =>    $field['initial_value'],
            'choices' =>    $this->get_countries()
        ));

        ?>
    </td>
</tr>

<?php
}

```

The main thing to keep in mind is that the name of the field must be `fields[field_key][field_name]`. My particular case translates to `fields[field_55e584fa90223][initial_value]`, but the key is generated for you, so you'll need a variable to reference it. Hence the use of `fields['.$key.']['initial_value]`.

If you need to add more field settings simply create some more defaults and add more fields following the pattern above. In version 5 the process is somewhat simpler as you don't have to wrap the whole thing in a bunch of HTML — that is done for you.

3	Country	country	Country Selector
Field Label * This is the name which will appear on the EDIT page	<input type="text" value="Country"/>		
Field Name * Single word, no spaces. Underscores and dashes allowed	<input type="text" value="country"/>		
Field Type *	<div>Country Selector</div>		
Field Instructions Instructions for authors. Shown when submitting data	<div>Set a country for the address</div>		
Required?	<div><input type="radio"/> Yes <input checked="" type="radio"/> No</div>		
Initial Value The initial value of the country field	<div>United States</div>		
Conditional Logic	<div><input type="radio"/> Yes <input checked="" type="radio"/> No</div>		
	<div>Close Field</div>		

Drag and drop to reorder

+ Add Field

Our custom setting for the country field.

STEP 5: FIELD FRONT-END

All that's left is to create the control the user will actually see. This is done in the `create_field()` method. You'll need to create the HTML yourself here but it shouldn't be too difficult to create a standard `<select>` field, right?

```

function create_field( $field )
{
    $field = array_merge($this->defaults, $field);
    ?>
    <div>
        <select name='<?php echo
            $field['name'] ?>'>
            <?php
                foreach( $this->
                    get_countries() as
                        $country ) :
                    ?>
                        <option <?php
selected( $field['value'], $country ) ?> value='<?php
echo $country ?>'><?php echo $country ?></option>
                    <?php endforeach; ?>
            </select>
        </div>
    <?php
}

```

At this stage, your field is ready to go. If you add your field to a field group and view the field in action, you should see the country selector with the initial value selected. If you save the object you've added the field to, it will retain its value.

Country
Select a Country

Haiti

The completed country selector in action.

STEP 6: ADDING SCRIPTS AND STYLES

This plugin doesn't require any scripts and styles at the moment, but if we were to create a drop-down with the help of Chosen⁷⁵, for example, we would need to leverage the `input_admin_enqueue_scripts()` function.

In many cases you don't actually need to modify the code of this function. The enqueued script and style points to the already created files in the `js` and `css` folders — simply use those to add your code. If you enqueue third-party scripts like Chosen you should drop that in the `js` folder and enqueue it the same way.

Once downloaded, I placed the files required by Chosen in the correct directories. I put *chosen.min.css* in the `css` folder, *chosen.jquery.min.js* in the `js` directory, and the two images in the `images` directory. To make sure images are referenced correctly I replaced `url()` in the CSS file with `url(../img.`

With all the files in place, I used the `input_admin_enqueue_scripts()` to enqueue them, resulting in the following code.

```
function input_admin_enqueue_scripts() {  
  
    // register ACF scripts  
    wp_register_script(  
        'acf-input-country_selector',  
        $this->settings['dir'] . 'js/input.js',  
        array('acf-input'),  
        $this->settings['version'] );  
}
```

⁷⁵. <https://harvesthq.github.io/chosen/>

```

// Chosen
wp_register_script( 'chosen',
$this->settings['dir'] . 'js/
chosen.jquery.min.js', array('acf-input',
'jquery'), $this->settings['version'] );
wp_register_style( 'chosen',
$this->settings['dir'] . 'css/
chosen.min.css', array('acf-input'),
$this->settings['version'] );

// scripts
wp_enqueue_script(array(
    'acf-input-country_selector',
    'chosen',
));

// styles
wp_enqueue_style(array(
    'chosen',
));

}

```

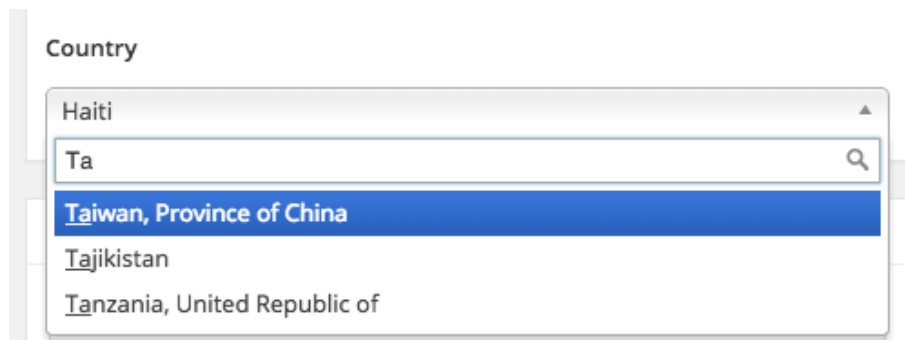
Note that I removed the original CSS file that was added (*input.css*), but I kept the *input.js* file enqueued. We will need a little bit of JavaScript to apply Chosen, but we won't need any additional CSS, apart from Chosen's own.

The last step is to apply Chosen to our field, which is where *input.js* comes in handy. Opening up the file, you can see that it has a dedicated section for version 4 and

for version 5. We'll be using the version 4 section, adding the Chosen initiation code. It's only one additional line which you need to add below `initialize_field($(this));`

```
$(this).find('select').chosen()
```

Once in place, you should be able to search within the select field's values, making the UI that much better.



Chosen applied to the select field.

STEP 7: MODIFYING VALUES

There are a number of methods which serve to modify values after or before they are sent/received. These can be helpful for manipulating data between the user interface and the back-end logic. Here are the four most important methods, along with how they can be used:

- `load_value()` is used to modify the value of the field after it is loaded from the database. This could be useful if you're saving complex things like geolocation. The saved value may be an array containing the latitude and longitude, but you actually want to display a string.

- `update_value()` modifies the value before it is saved in the database. If the user enters a comma-separated value of IDs, you may want to save that as an array. Using the `update_value()` function you can modify it easily.
- `load_field()` modifies the whole field after it is returned from the database.
- `update_field()` modifies the whole field before it is saved to the database.

Further Possibilities

I hope you can see that creating your own field is actually a pretty simple matter. If you want to add elaborate JavaScript to make things as user-friendly as possible, that's all up to you — ACF supports it nicely. You can use a bunch of methods to play around with values and fields and much more. Browse through the template file for more information.

If that wasn't enough, ACF also has powerful actions and filters⁷⁶. Check out the documentation for more info.

If you'd like to check out a more complex field which contains JavaScript, styles and interaction with a third-party API, I invite you to check out the GitHub repository for my Google Font Selector Field⁷⁷, which allows users to select fonts available from Google. 🐼

⁷⁶. <http://www.advancedcustomfields.com/resources/>

⁷⁷. <https://github.com/danielpataki/ACF-Google-Font-Selector>

Building An Advanced Notification System For WordPress

BY CARLO DANIELE 🐼

A lot of tools enable us to distribute a website's content, but when we need to promptly reach a target group, an email notification system might be the best option. If your website is not frequently updated, you could notify all subscribers each time a post is published. However, if it's updated frequently or it covers several topics, you could filter subscribers before mailing them.

If you opt for the latter, you could set up a user meta field that stores a bit of information to identify the subscribers to be notified. The same bit of information would label the posts you're publishing. Depending on the website's architecture, you could store the meta data in a category, a tag, a custom taxonomy or a custom field. In this article we'll show you how to let your website's subscribers decide when they want notifications, and linked to a particular location.

Can I Use A Plugin?

If WordPress is your CMS, you can choose from a number of plugins, such as the comprehensive JetPack⁷⁸ or the more specialized Subscribe 2⁷⁹.

⁷⁸. <http://jetpack.me/support/subscriptions/>

Notification Settings

Receive email as: ☐ HTML - Full ☐ HTML - Excerpt ☐ Plain Text - Full ☒ Plain Text - Excerpt

Automatically subscribe me to newly created categories: ☐ Yes ☐ No

Subscribed Categories

☐ Select / Unselect All

☐ Events ☐ WebDesign

☐ Uncategorized ☐ WordPress

Do not send notifications for post made by these authors

☐ Select / Unselect All

☐ marialuisa

☐ chiara

☐ luca

☐ anna

[Update Preferences »](#)

Subscribe 2's settings page.

Jetpack is easy to use, whereas Subscribe 2 is specialized and full-featured. Both plugins enable you to send email notifications to subscribers whenever a post is published. Unfortunately, neither allows you to notify specific users about specific content. And we want to select posts based on custom fields, mailing them to specific groups of users. Unfortunately, no plugin seems able to help us with this.

Things To Do

We are going to add several functionalities to WordPress' core, and the CMS allows us to declare our own custom functions in the main file of a plugin. We're not going to

⁷⁹. <https://wordpress.org/plugins/subscribe2/>

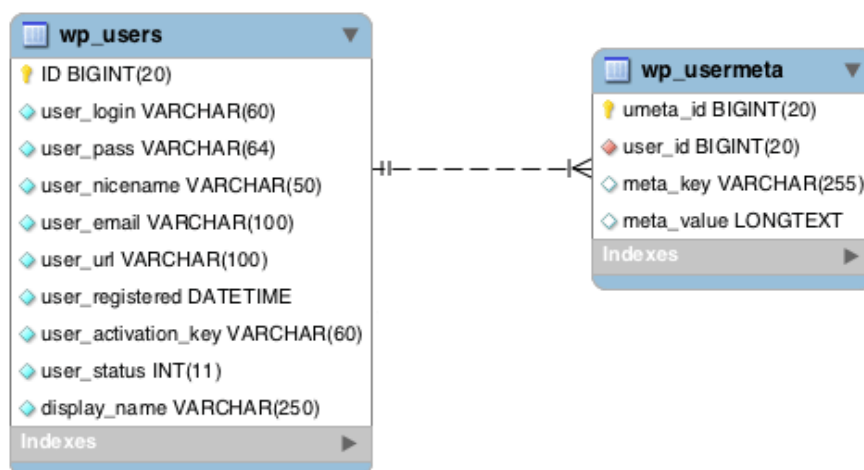
dive deep into plugin development, but you can get the information you need directly from the Codex⁸⁰.

We have to accomplish the following tasks:

1. add two meta fields to the user's profile, the first of which stores the name of a location and the second of which determines whether the user will receive emails;
2. add a custom meta box to the post-editing page containing the location-related custom field;
3. select the users to be notified and send an email to them.

Add Meta Fields To User Profiles

WordPress stores user data in the `wp_users` and `wp_usermeta` tables.



WordPress' database description. (Image: Codex⁸¹)

⁸⁰. http://codex.wordpress.org/Writing_a_Plugin

⁸¹. http://codex.wordpress.org/Database_Description

Here, `wp_users` holds the list of all website users, while `wp_usermeta` contains all meta data associated with each user's profile. Meta data is registered as key-value pairs in the `meta_key` and `meta_value` fields.

WordPress generates a bunch of meta data, such as `nickname`, `first_name`, `last_name`, `description` and `wp_capabilities`. Much of this data is automatically assigned to each user's profile, and a user is able to edit it later from their profile page.

To perform our first task, we'll add two meta fields to the profile pages of users. These fields will store the name of a geographic location and will allow the user to activate (or deactivate) the notification feature.

In the main file of the plugin, let's define a global associative array whose elements consist of the names of US states:

```
$smashing_notification_states = array( 'AL' =>
'Alabama', 'AK' => 'Alaska', 'AZ' => 'Arizona', 'AR'
=> 'Arkansas', 'CA' => 'California', 'CO' =>
'Colorado', 'CT' => 'Connecticut', 'DE' =>
'Delaware', 'FL' => 'Florida', 'GA' => 'Georgia',
'HI' => 'Hawaii', 'ID' => 'Idaho', 'IL' =>
'Illinois', 'IN' => 'Indiana', 'IA' => 'Iowa', 'KS'
=> 'Kansas', 'KY' => 'Kentucky', 'LA' => 'Louisiana',
'ME' => 'Maine', 'MD' => 'Maryland', 'MA' =>
'Massachusetts', 'MI' => 'Michigan', 'MN' =>
'Minnesota', 'MS' => 'Mississippi', 'MO' =>
'Missouri', 'MT' => 'Montana', 'NE' => 'Nebraska',
'NV' => 'Nevada', 'NH' => 'New Hampshire', 'NJ' =>
'New Jersey', 'NM' => 'New Mexico', 'NY' => 'New
```

```

York', 'NC' => 'North Carolina', 'ND' => 'North
Dakota', 'OH' => 'Ohio', 'OK' => 'Oklahoma', 'OR' =>
'Oregon', 'PA' => 'Pennsylvania', 'RI' => 'Rhode
Island', 'SC' => 'South Carolina', 'SD' => 'South
Dakota', 'TN' => 'Tennessee', 'TX' => 'Texas', 'UT'
=> 'Utah', 'VT' => 'Vermont', 'VA' => 'Virginia',
'WA' => 'Washington', 'WV' => 'West Virginia', 'WI'
=> 'Wisconsin', 'WY' => 'Wyoming' );

```

Thanks to this array, we will generate a select menu to avoid input errors by users. Now, we need to add two form fields to the user's profile page. To do this, we will use two action hooks:

```

add_action( 'show_user_profile',
'smashing_show_user_meta_fields' );
add_action( 'edit_user_profile',
'smashing_show_user_meta_fields' );

```

Here, show_user_profile⁸² is triggered when a user is viewing their own profile, while edit_user_profile⁸³ is triggered when a user is viewing another user's profile.

The callback function prints the markup.

```

/**
 * Show custom user profile fields.
 *
 * @param obj $user The user object.
 */

```

⁸². http://codex.wordpress.org/Plugin_API/Action_Reference/show_user_profile

⁸³. http://codex.wordpress.org/Plugin_API/Action_Reference/edit_user_profile

```

function smashing_show_user_meta_fields( $user ) {
    global $smashing_notification_states;
    ?>
    <h3><?php _e( 'Smashing profile information',
    'smashing' ); ?></h3>
    <table class="form-table">
        <tr>
            <th scope="row"><?php _e( 'State',
            'smashing' ); ?></th>
            <td>
                <label for="state">
                    <select name="state">
                        <option value="" <?php
                        selected( get_user_meta(
                            $user->ID, 'state', true ),
                            "" ); ?>>Select</option>
                        <?php foreach
                        ($smashing_notification_states
                        as $key => $value) { ?>
                            <option value="<?php echo
                            $key; ?>" <?php selected(
                            esc_attr( get_user_meta(
                                $user->ID, 'state', true )
                                ), $key ); ?>><?php echo
                                $value; ?></option>
                            <?php } ?>
                        </select>
                        <?php _e( 'Select state',
                        'smashing' ); ?>
                    </label>

```

```

        </td>
    </tr>
    <tr>
        <th scope="row"><?php _e(
            'Notifications', 'smashing' ); ?></th>
        <td>
            <label for="notification">
                <input id="notification"
                    type="checkbox" name=
                        "notification" value="true" <?php
                            checked( esc_attr( get_user_meta(
                                $user->ID, 'notification', true )
                                    ), 'true' ); ?> />
                <?php _e( 'Subscribe to email
                    notifications', 'smashing' ); ?>
            </label>
        </td>
    </tr>
</table>
<?php }

```

This table contains two custom meta fields. The first is a select menu whose options are generated by a **foreach** loop that iterates over the **\$smashing_notification_states** global array. This way, the user doesn't have to type the name of their state, but instead chooses it from a dropdown list.

As you can see, we're calling the **selected()** function⁸⁴ twice from inside two **<option>** tags; **selected()** is a WordPress function for comparing two strings. When the strings have the same value, the function

prints `selected='selected'`; otherwise, it echoes an empty string.

The first time we call `selected()`, we're comparing the current option's value (`'state'`) with an empty string (which means no state was selected). When iterating over the `$smashing_notification_states` array, we're comparing the value of each element to the current value of the `'state'` meta field. This way, we can automatically select the option corresponding to the existing `'state'` value.

The second meta field to be added to users' profiles is a checkbox. Its value will be `'true'` or `'false'` depending on whether the user chooses to receive notifications. Similar to `selected()`, `checked()`⁸⁵ prints out the string `checked='checked'` when its two arguments have the same value. Of course, `checked()` applies to checkboxes and radio buttons.

Now that we've got the fields, we can save the user's input. We need two action hooks to store the user data:

```
add_action( 'personal_options_update',
'smashing_save_user_meta_fields' );
add_action( 'edit_user_profile_update',
'smashing_save_user_meta_fields' );
```

Here, `personal_options_update` is triggered when the user is viewing their own profile page, while `edit_user_profile_update` is triggered when a user with sufficient

⁸⁴. http://codex.wordpress.org/Function_Reference/selected

⁸⁵. http://codex.wordpress.org/Function_Reference/checked

privileges is viewing another user's profile page. We have two hooks but just one callback:

```
/**
 * Store data in wp_usermeta table.
 *
 * @param int $user_id the user unique ID.
 */
function smashing_save_user_meta_fields( $user_id ) {

    if ( !current_user_can( 'edit_user', $user_id ) )
        return false;

    if( isset($_POST['state']) )
        update_user_meta( $user_id, 'state',
            sanitize_text_field( $_POST['state'] ) );

    if( !isset($_POST['notification']) )
        $_POST['notification'] = 'false';

    update_user_meta( $user_id, 'notification',
        sanitize_text_field( $_POST['notification'] ) );

}
```

This function verifies whether the user is allowed to `edit_user`, and if `current_user_can` is true, it checks the data and saves it in the `wp_usermeta` table.

The custom meta fields added to the user's profile page.

Custom Meta Box And Custom Fields

We have to decide what kind of content should be included in the notification to subscribers. This decision will depend on your website's architecture. In this example, we'll go for regular posts, but you could choose a custom post type instead. The choice depends on your needs.

That being said, we are going to build a custom meta box containing a set of custom fields. These fields will be used to store an address, city, state and some other data related to location. Two other custom fields will enable and disable notifications on a per-post basis, and they will register the number of emails sent to users whenever a new post has been published. Let's put another action hook to work:

```
add_action( 'add_meta_boxes', 'smashing_add_meta_box'
);
function smashing_add_meta_box(){
```



```

$screens = array( 'post' ); // possible values:
// 'post', 'page', 'dashboard', 'link',
// 'attachment', 'custom_post_type'

foreach ( $screens as $screen ) {
    add_meta_box(
        'smashing_metabox', // $id - meta_box ID
        __( 'Venue information', 'smashing' ),
        // $title - a title for the meta_box
        // container
        'smashing_meta_box_callback',
        // $callback - the callback that outputs
        // the html for the meta_box
        $screen, // $post_type - where to show
        // the meta_box. Possible values: 'post',
        // 'page', 'dashboard', 'link',
        // 'attachment', 'custom_post_type'
        'normal', // $context - possible values:
        // 'normal', 'advanced', 'side'
        'high' // $priority - possible values:
        // 'high', 'core', 'default', 'low'
    );
}
}

```

Here, `add_meta_box`⁸⁶ accepts seven arguments: a unique ID for the meta box, a title, a callback function, a value for `screen`, the context (i.e. the part of the page where to

⁸⁶. http://codex.wordpress.org/Function_Reference/add_meta_box

show the meta box), and priority and callback arguments. Because we are not setting a value for the callback argument parameter, the `$post` object will be the only argument passed to `smashing_meta_box_callback`. Finally, let's define the callback function to print out the meta box:

```
/*
 * Print the meta_box
 *
 * @param obj $post The object for the current post
 */
function smashing_meta_box_callback( $post ){
    global $smashing_notification_states;

    // Add a nonce field
    wp_nonce_field( 'smashing_meta_box',
        'smashing_meta_box_nonce' );

    $address = esc_attr( get_post_meta( get_the_ID(),
        'address', true ) );
    $city = esc_attr( get_post_meta( get_the_ID(),
        'city', true ) );
    $state = esc_attr( get_post_meta( get_the_ID(),
        'state', true ) );
    $zip = esc_attr( get_post_meta( get_the_ID(),
        'zip', true ) );
    $phone = esc_attr( get_post_meta( get_the_ID(),
        'phone', true ) );
    $website = esc_attr( get_post_meta( get_the_ID(),
        'website', true ) );
```

```

$disable = esc_attr( get_post_meta( get_the_ID(),
'disable', true ) );
?>
<table id="venue">
  <tbody>
    <tr>
      <td class="label"><?php _e( 'Address',
'smashing' ); ?></td>
      <td><input type="text" id="address"
name="venue[address]" value="<?php echo
$address; ?>" size="30" /></td>
    </tr>
    <tr>
      <td><?php _e( 'City', 'smashing' );
?></td>
      <td><input type="text" id="city"
name="venue[city]" value="<?php echo
$city; ?>" size="30" /></td>
    </tr>
    <tr>
      <td><?php _e( 'State', 'smashing' );
?></td>
      <td>
        <select name="venue[state]">
          <option value="" <?php selected(
$state, "" ); ?>>Select</option>
          <?php foreach
($smashing_notification_states as
$key => $value) { ?>
            <option value="<?php echo

```

```

        $key; ?>" <?php selected(
        $state, $key ); ?><?php echo
        $value; ?></option>
    <?php } ?>
</select>
</td>
</tr>
<tr>
    <td><?php _e( 'Disable notification',
    'smashing' ); ?></td>
    <td><input id="disable" type="checkbox"
    name="venue[disable]" value="true" <?php
    checked( $disable, 'true' ); ?> /></td>
</tr>
</tbody>
</table>
<?php
}

```

First, we're initializing the **global** array and registering a nonce field⁸⁷. We then add two simple text fields. The **name** attribute is set in the form of an array element, while the value is set to the corresponding custom field's value. Finally, the main custom fields are added.

Just like with the user's meta data, we add a select menu whose options are echoed, iterating over the elements in the **\$smashing_notification_states** global array. Once we have built the select menu, let's continue

⁸⁷. http://codex.wordpress.org/WordPress_Nonces

with a checkbox to enable and disable the single post notification.

Now we have to save the data: Our action hook is `save_post`. We'll perform a number of tasks with the callback function. Take a look at the inline documentation for more information.

```
add_action( 'save_post',
'smashing_save_custom_fields' );

/*
 * Save the custom field values
 *
 * @param int $post_id the current post ID
 */
function smashing_save_custom_fields( $post_id ){

    // Check WP nonce
    if ( !isset( $_POST['smashing_meta_box_nonce'] )
    || ! wp_verify_nonce(
        $_POST['smashing_meta_box_nonce'],
        'smashing_meta_box' ) )
        return;

    // Return if this is an autosave
    if ( defined( 'DOING_AUTOSAVE' ) &&
        DOING_AUTOSAVE )
        return;

    // check the post_type and set the corresponding
    // capability value
```

```

$capability = ( isset( $_POST['post_type'] ) &&
'page' == $_POST['post_type'] ) ? 'edit_page' :
'edit_post';

// Return if the user lacks the required
// capability
if ( !current_user_can( $capability, $post_id ) )
    return;

if( !isset($_POST['venue']['disable']) )
    $_POST['venue']['disable'] = 'false';

// validate custom field values
$fields = ( isset( $_POST['venue'] ) ) ? (array)
$_POST['venue'] : array();
$fields = array_map( 'sanitize_text_field',
$fields );

foreach ( $fields as $key => $value ) {
    // store data
    update_post_meta( $post_id, $key, $value );
}
}

```

Our custom meta box is up and running, and it looks like this:

Venue information	
Address	1600 Amphitheatre Parkway
City	Mountain View
State	California
zip	94043
Phone	+1 650-253-0000
Website	http://www.google.com/
Disable notification	<input type="checkbox"/>

The custom meta box showing the location details.

Building The Notification System

If you were working with custom post types, you would need the `publish_{$post_type}` hook (i.e. `publish_recipes`, `publish_events`, etc.). But since we are working with posts, `publish_post` is the hook for us:

```
add_action( 'publish_post',
'smashing_notify_new_post' );

/*
 * Notify users sending them an email
 *
 * @param int $post_ID the current post ID
 */
function smashing_notify_new_post( $post_ID ){
    global $smashing_notification_states;

    $url = get_permalink( $post_ID );
    $state = get_post_meta( $post_ID, 'state', true );

    if( 'true' == get_post_meta( $post_ID, 'disable',
```

```

true ) )
    return;

// build the meta query to retrieve subscribers
$args = array(
    'meta_query' => array(
        array( 'key' => 'state', 'value'
            => $state, 'compare' => '=' ),
        array( 'key' => 'notification',
            'value' => 'true', 'compare' =>
            '=' )
    ),
    'fields' => array( 'display_name',
        'user_email' )
);

// retrieve users to notify about the new post
$users = get_users( $args );
$num = 0;
foreach ( $users as $user ) {

    $to = $user->display_name . ' <' .
        $user->user_email . '>';

    $subject = sprintf( __( 'Hei! We have news
        for you from %s', 'smashing' ),
        $smashing_notification_states[$state] );

    $message = sprintf( __( 'Hi %s!', 'smashing'
        ), $user->display_name ) . "\r\n" .
        sprintf( __( 'We have a new post from %s',

```



```

        'smashing' ),
        $smashing_notification_states[$state] ) . "\r\n" .
        sprintf( __( 'Read more on %s', 'smashing' ),
        $url ) . '.' . "\r\n";

        $headers[] = 'From: Yourname
        <you@yourdomain.com>';
        $headers[] = 'Reply-To: you@yourdomain.com';

        if( wp_mail( $to, $subject, $message,
        $headers ) )
            $num++;
    }
    // a hidden custom field
    update_post_meta( $post_ID, '_notified_users',
    $num );
    return $post_ID;
}

```

Once again, we declare the global array `$smashing_notification_states`. The two variables `$url` and `$state` will store the post's permalink and state. The succeeding condition checks the value of the `disable` custom field: If it's `'true'`, we exit the function. We have to retrieve from the database all users whose `state` meta field has the same value as the `state` custom field of the current post, and we use the `get_users()` function to accomplish this.

The `wp_mail`⁸⁸ function accepts five arguments: recipient(s), subject, message, headers, attachments. The recip-

ients could be passed as an array or as a comma-separated string of addresses. So, we could have passed to the function all of the addresses together, but doing so would have made them publicly visible (this is the way `wp_mail()` works).

So, we'll iterate over the `$users` array and call `wp_mail` repeatedly (which shouldn't be done with a huge number of emails, as we'll see in a moment). In case of success, `wp_mail` returns `true`. The counter is incremented by 1, and the loop continues with the next user.

When the `foreach` cycle ends, the current value of `$num` is registered in the hidden `_notified_users` custom field (notice the underscore preceding the name of the custom field).

Unfortunately, a loop iterating over and over hundreds of times could considerably slow down the script, as pointed out in the reference on the [PHP mail\(\) function](#)⁸⁹:

"It is worth noting that the `mail()` function is not suitable for larger volumes of email in a loop. This function opens and closes an SMTP socket for each email, which is not very efficient.

For the sending of large amounts of email, see the » [PEAR::Mail](#)⁹⁰, and » [PEAR::Mail_Queue](#)⁹¹ packages."

⁸⁸. http://codex.wordpress.org/Function_Reference/wp_mail

⁸⁹. <http://php.net/manual/it/function.mail.php>

⁹⁰. <http://pear.php.net/package/Mail>

⁹¹. http://pear.php.net/package/Mail_Queue

We could work around this, passing to the function the email addresses as BCCs, setting them in the headers⁹², as shown here:

```
function smashing_notify_new_post( $post_ID ){
    global $smashing_notification_states;

    $url = get_permalink( $post_ID );
    $state = get_post_meta( $post_ID, 'state', true );

    if( 'true' == get_post_meta( $post_ID, 'disable',
    true ) )
        return;

    // build the meta query to retrieve subscribers
    $args = array(
        'meta_query' => array(
            array( 'key' => 'state', 'value'
            => $state, 'compare' => '=' ),
            array( 'key' => 'notification',
            'value' => 'true', 'compare' =>
            '=' )
        ),
        'fields' => array( 'display_name',
        'user_email' )
    );

    // retrieve users to notify about the new post
    $users = get_users( $args );
```

⁹². http://codex.wordpress.org/Function_Reference/wp_mail#Using_.24headers_To_Set_.22From:.22.2C_.22Cc:.22_and_.22Bcc:.22_Parameters

```

$num = 0;

$to = 'Yourname <you@yourdomain.com>';

$subject = sprintf( __( 'Hei! We have news for
you from %s', 'smashing' ),
$smashing_notification_states[$state] );

$message = __( 'Hi ', 'smashing' ) . "\r\n" .
    sprintf( __( 'We have a new post from %s',
'smashing' ),
$smashing_notification_states[$state] ) . "\r\n" .
    sprintf( __( 'Read more on %s', 'smashing' ),
$url ) . '.' . "\r\n";

$headers[] = 'From: Yourname
<you@yourdomain.com>';
$headers[] = 'Reply-To: you@yourdomain.com';

foreach ( $users as $user ) {
    $headers[] = 'Bcc: ' . $user->user_email;
    $num++;
}

if( wp_mail( $to, $subject, $message, $headers ) )
    update_post_meta( $post_ID, '_notified_users',
$num );

```

```
        return $post_ID;
    }
```

As you can see, in case of `wp_mail()`'s success, we update the `_notified_user` custom field with `$num`'s value. However, in the code above, `$num` stores the number of retrieved users, not the number of times we call `wp_mail()`.

Finally, if none of the solutions presented fit your needs, you could consider a third-party email notification system, such as [MailChimp](#)⁹³ or [FeedBurner](#)⁹⁴, which enable you to deliver notifications from a website's feed.

A Note About Status Transitions

We hooked the `smashing_notify_new_post` callback to the `publish_post` action. This hook is triggered each time the status of an existing post is changed to `publish`. Unfortunately, `publish_post` is not fired when a new post is published. So, to send notifications, first save the post as “draft” (or “pending”). If you prefer to email subscribers each time a post is published, consider calling the `save_post` action instead:

```
add_action( 'save_post', 'smashing_notify_new_post' );
/*
 * Save the custom field values
 */
```

⁹³. <http://kb.mailchimp.com/campaigns/rss-in-campaigns/create-an-rss-driven-campaign>

⁹⁴. <https://support.google.com/feedburner/answer/78982?hl=en>

```

* @param int $post_id the current post ID
*/
function smashing_notify_new_post( $post_ID ){
    global $smashing_notification_states;

    if( 'publish' != get_post_status( $post_ID ) )
        return;

    ...
}

```

Check the Codex for further information about [status transitions](#)⁹⁵ and the [save_post](#) action hook⁹⁶.

A Confirmation Message

When you work with the `publish_post` action hook, you will soon realize that testing your scripts can get a little tricky. When a new post is published, WordPress loads a script that saves data and, when it is done, redirects the user to the post-editing page. This double redirection does not allow variable values to be printed on the screen.

A confirmation message could be a good workaround. This solution allows us to check a variable's values and to give the publisher useful information: specifically, the number of times `wp_mail` has been called (or the number of users to be notified).

Remember the `$num` variable? Its value was stored in a hidden custom field, `_notified_users`. Now we have to

⁹⁵. http://codex.wordpress.org/Post_Status_Transitions

⁹⁶. https://codex.wordpress.org/Plugin_API/Action_Reference/save_post

retrieve that value and print it out in a message using a filter hook.

Thanks to the `post_updated_messages` filter, we can customize WordPress confirmation messages and output them to the screen whenever a new post is saved or published (the Codex does not provide a reference for this filter hook, only an [example of usage](#)⁹⁷). Here is the callback function we can use to customize the message when a post is published:

```
add_filter( 'post_updated_messages',
'smashing_updated_messages' );

/**
 * Post update messages.
 *
 * See /wp-admin/edit-form-advanced.php
 *
 * @param array $messages Existing post update
messages.
 *
 * @return array Amended post update messages with
new update messages.
 */
function smashing_updated_messages( $msgs ){

    $post = get_post();
    $post_type = get_post_type( $post );
    $post_type_object = get_post_type_object(
```

⁹⁷. http://codex.wordpress.org/Function_Reference/register_post_type#Example

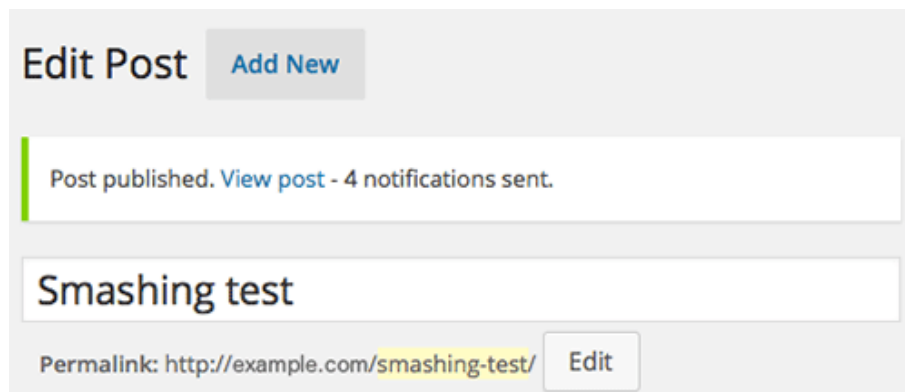
```

$post_type );

$num = get_post_meta( $post->ID,
    '_notified_users', true );

if ( $post_type_object->publicly_queryable ) {
    $msgs[$post_type][6] .= ' - ' . $num . __(
        ' notifications sent.', 'smashing' );
}
return $msgs;
}

```



When a post is published, a custom message is printed informing the author about the number of emails sent to users.

wp_mail Function And SMTP

WordPress' `wp_mail()` function works the same way as PHP's `mail()` function. Whether an email has been successfully sent will depend on `php.ini`'s settings, but most hosts include SMTP in their services. If you aren't able to set that up, you could choose an external SMTP service

and use it in tandem with the WP Mail SMTP⁹⁸ plugin, which routes your emails through an SMTP service.

SMTP Options

These options only apply if you have chosen to send mail by SMTP above.

SMTP Host:

SMTP Port:

Encryption:

- ☐ No encryption.
- ☒ Use SSL encryption.
- ☐ Use TLS encryption. This is not the same as STARTTLS. For most servers SSL is the recommended option.

Authentication:

- ☐ No: Do not use SMTP authentication.
- ☒ Yes: Use SMTP authentication.

If this is set to no, the values below are ignored.

Username:

Password:

[Save Changes](#)

WP Mail SMTP's settings page.

Be careful when you save data: the “from” field should have the same value as your account’s email address; otherwise, the server might respond with an error message.

Be aware that a plugin is not necessary: WordPress allows for the possibility of overwriting `php.ini`’s settings from within a script, with the `phpmailer_init` action hook. This hook allows us to pass our own parameters to the PHPMailer⁹⁹ object. See the Codex for more information¹⁰⁰ on this.

⁹⁸. <https://wordpress.org/plugins/wp-mail-smtp/>

⁹⁹. <http://phpmailer.worxware.com/>

¹⁰⁰. https://codex.wordpress.org/Plugin_API/Action_Reference/phpmailer_init

Designing Better Emails

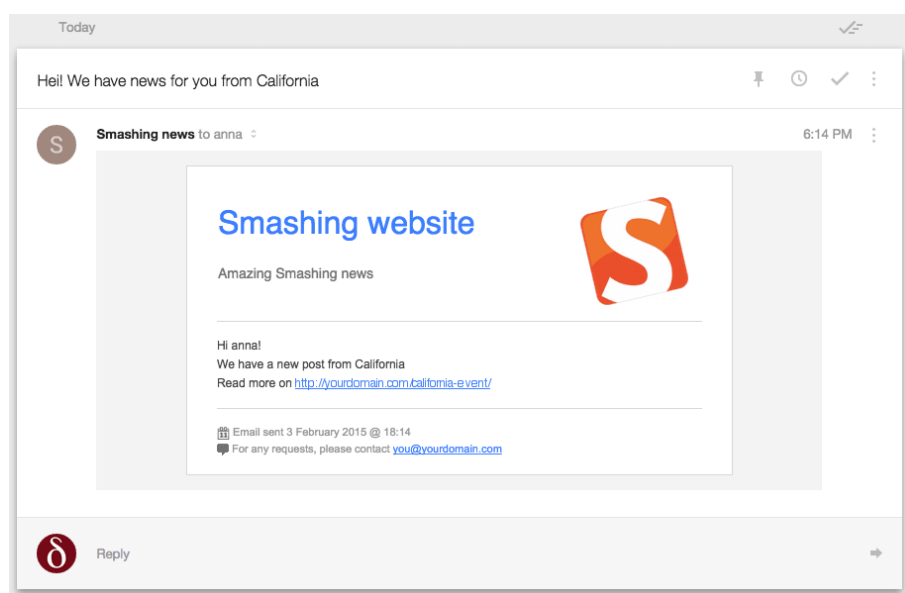
Just like the PHP `mail()` function, `wp_mail()`'s default **Content Type** is `text/plain`. And just like the `mail()` function, `wp_mail()` allows us to set the **Content Type** to `text/html`. You can specify a different **Content Type** using the `wp_mail_content_type` filter or by setting the following headers:

```
$headers[] = "MIME-Version: 1.0";
$headers[] = "Content-Type: text/html;
charset=ISO-8859-1";
```

Of course, a number of plugins allow you to manage your email's **Content Type** from the administration panel. WP Better Emails¹⁰¹ is just one, but it's one of the most appreciated. This plugin forces WordPress to send HTML emails, but it's not its only feature. It also allows administrators to build their own email templates and to send arbitrary emails for testing purposes.

Finally, the following image shows what will be delivered to a Gmail user's inbox.

¹⁰¹. <https://wordpress.org/plugins/wp-better-emails/>



The result in Gmail: The content has been generated by our code; SMTP parameters have been saved on WP Mail SMTP's settings page; and the template is provided by WP Better Emails.

Conclusion

Our notification system is ready to be used. In building it, we've toured many of WordPress' core features: user meta fields, custom meta boxes, custom queries, the mailing from script and more. If you're interested, download the full code¹⁰² (ZIP file), and remember to switch each occurrence of the `you@yourdomain.com` string with your own email address.

You can expand on this a lot more. You could integrate this system with a third-party email management application such as MailChimp¹⁰³ or Mad Mimi¹⁰⁴. You could

¹⁰². <http://provide.smashingmagazine.com/sm-email-notification.zip>

¹⁰³. <http://mailchimp.com/>

¹⁰⁴. <https://madmimi.com/>

design flashy email templates. Or you could create even more personalized notifications.

FURTHER READING

- “[Create Perfect Emails For Your WordPress Website¹⁰⁵](#),” Daniel Pataki, Smashing Magazine
- “[Post Status Transitions¹⁰⁶](#),” WordPress Codex
- “[Plugin API/Action Reference¹⁰⁷](#),” WordPress Codex
- “[Plugin API/Filter Reference¹⁰⁸](#),” WordPress Codex
- “[Custom Fields¹⁰⁹](#),” WordPress Codex
- “[Creating Custom Meta Boxes¹¹⁰](#),” Plugin Handbook, WordPress
- “[Validating Sanitizing and Escaping User Data¹¹¹](#),” WordPress Codex
- “[Data Validation¹¹²](#),” WordPress Codex 🐘

¹⁰⁵. <http://www.smashingmagazine.com/2011/10/25/create-perfect-emails-wordpress-website/>

¹⁰⁶. http://codex.wordpress.org/Post_Status_Transitions

¹⁰⁷. http://codex.wordpress.org/Plugin_API/Action_Reference

¹⁰⁸. http://codex.wordpress.org/Plugin_API/Filter_Reference

¹⁰⁹. http://codex.wordpress.org/Custom_Fields

¹¹⁰. <https://developer.wordpress.org/plugins/metadata/creating-custom-meta-boxes/>

¹¹¹. http://codex.wordpress.org/Validating_Sanitizing_and_Escaping_User_Data

¹¹². https://codex.wordpress.org/Data_Validation

How To Use Autoloading And A Plugin Container In WordPress Plugins

BY NICO AMARILLA 🍷

Building and maintaining¹¹³ a WordPress plugin can be a daunting task. The bigger the codebase, the harder it is to keep track of all the working parts and their relationship to one another. And you can add to that the limitations imposed by working in an antiquated version of PHP, 5.2.

In this article we will explore an alternative way of developing WordPress plugins, using the lessons learned from the greater PHP community, the world outside WordPress. We will walk through the steps of creating a plugin and investigate the use of autoloading and a plugin container.

Let's Begin

The first thing you need to do when creating a plugin is to give it a unique name. The name is important as it will be the basis for all our unique identifiers (function prefix, class prefix, textdomain, option prefix, etc.). The name should also be unique across the wordpress.org space. It won't hurt if we make the name catchy. For our sample

¹¹³. <https://shop.smashingmagazine.com/products/wordpress-maintenance-keeping-your-website-safe-and-efficient>

plugin I chose the name *Simplarity*, a play on the words “simple” and “clarity”.

We’ll assume you have a working WordPress installation already.

FOLDER STRUCTURE

First, create a directory named *simplarity* inside *wp-content/plugins*. Inside it create the following structure:

- *simplarity.php*: our main plugin file
- *css/*: directory containing our styles
- *js/*: directory containing JavaScript files
- *languages/*: directory that will contain translation files
- *src/*: directory containing our classes
- *views/*: directory that will contain our plugin view files

THE MAIN PLUGIN FILE

Open the main plugin file, *simplarity.php*, and add the plugin information header:

```
<?php
/*
Plugin Name: Simplarity
Description: A plugin for smashingmagazine.com
Version: 1.0.0
License: GPL-2.0+
*/
```

This information is enough for now. The plugin name, description, and version will show up in the plugins area of WordPress admin. The license details are important to let your users know that this is an open source plugin. A full list of header information can found in the [WordPress codex](#)¹¹⁴.

Autoloading

Autoloading allows you to automatically load classes using an autoloader so you don't have to manually include the files containing the class definitions. For example, whenever you need to use a class, you need to do the following:

```
require_once '/path/to/classes/class-container.php';
require_once '/path/to/classes/class-view.php';
require_once '/path/to/classes/
class-settings-page.php';
```

```
$plugin = new Container();
$view = new View();
$settings_page = new SettingsPage();
```

With autoloading, you can use an autoloader instead of multiple `require_once`¹¹⁵ statements. It also eliminates the need to update these require statements whenever you add, rename, or change the location of your classes. That's a big plus for maintainability.

¹¹⁴. http://codex.wordpress.org/Writing_a_Plugin

¹¹⁵. <http://php.net/manual/en/function.require-once.php>

ADOPTING THE PEAR NAMING CONVENTION FOR CLASS NAMES

Before we create our autoloader we need to create a convention for our class names and their location in the file system. This will aid the autoloader in mapping out the class to its source file.

For our class names we will adopt the PEAR naming convention¹¹⁶. The gist is that class names are alphabetic characters in StudlyCaps. Each level of the hierarchy is separated with a single underscore. Class names will directly map to the directories in which they are stored.

It's easier to illustrate it using examples:

- A class named `Simplarity_Plugin` would be defined in the file `src/Simplarity/Plugin.php`.
- A class named `Simplarity_SettingsPage` would be defined in `src/Simplarity/SettingsPage.php`.

As you can see with this convention, the autoloader will just replace the underscores with directory separators to locate the class definition.

WHAT ABOUT THE WORDPRESS CODING STANDARDS FOR CLASS NAMES?

As you might be aware, WordPress has its own naming convention¹¹⁷ for class names. It states:

¹¹⁶. <http://pear.php.net/manual/en/standards.naming.php>

¹¹⁷. <https://make.wordpress.org/core/handbook/coding-standards/php/#naming-conventions>

*“Class names should use capitalized words separated by underscores. Any acronyms should be all upper case. [...] Class file names should be based on the class name with **class-** prepended and the underscores in the class name replaced with hyphens, for example **WP_Error** becomes **class-wp-error.php**”*

I know that we should follow the standards of the platform that we are developing on. However, we suggest using the PEAR naming convention because:

- WP coding standards do not cover autoloading.
- WP does not follow its own coding standards. Examples: *class.wp-scripts.php* and SimplePie. This is understandable since WordPress grew organically.
- Interoperability allows you to easily use third-party libraries that follow the PEAR naming convention, like Twig. And conversely, you can easily port your code to other libraries sharing the same convention.
- It's important your autoloader is future-ready. When WordPress decides to up the ante and finally move to PHP 5.3 as its minimum requirement, you can easily update the code to be PSR-0 or PSR-4-compatible and take advantage of the built-in namespaces instead of using prefixes. This is a big plus for interoperability.

Note that we are only using this naming convention for classes. The rest of our code will still follow the WordPress coding standards. It's important to follow and re-

spect the standards of the platform that we are developing on.

Now that we have fully covered the naming convention, we can finally build our autoloader.

Building Our Autoloader

Open our main plugin file and add the following code below the plugin information header:

```
spl_autoload_register( 'simplarity_autoloader' );
function simplarity_autoloader( $class_name ) {
    if ( false !== strpos( $class_name, 'Simplarity' )
) {
        $classes_dir = realpath( plugin_dir_path(
            __FILE__ ) ) . DIRECTORY_SEPARATOR . 'src' .
            DIRECTORY_SEPARATOR;
        $class_file = str_replace( '_',
            DIRECTORY_SEPARATOR, $class_name ) . '.php';
        require_once $classes_dir . $class_file;
    }
}
```

At the heart of our autoloading mechanism is PHP's built in `spl_autoload_register`¹¹⁸ function. All it does is register a function to be called automatically when your code references a class that hasn't been loaded yet.

The first line tells `spl_autoload_register` to register our function named `simplarity_autoloader`:

¹¹⁸. <http://php.net/manual/en/function.spl-autoload-register.php>

```
spl_autoload_register( 'simplarity_autoloader' );
```

Next we define the *simplarity_autoloader* function:

```
function simplarity_autoloader( $class_name ) {  
    ...  
}
```

Notice that it accepts a `$class_name` parameter. This parameter holds the class name. For example when you instantiate a class using `$plugin = new Simplarity_Plugin()`, `$class_name` will contain the string “Simplarity_Plugin”. Since we are adding this function in the global space, it’s important that we have it prefixed with `simplarity_`.

The next line checks if `$classname` contains the string “Simplarity” which is our top level namespace:

```
if ( false !== strpos( $class_name, 'Simplarity' ) ) {
```

This will ensure that the autoloader will only run on our classes. Without this check, our autoloader will run every time an unloaded class is referenced, even if the class is not ours, which is not ideal.

The next line constructs the path to the directory where our classes reside:

```
$classes_dir = realpath( plugin_dir_path( __FILE__ )  
) . DIRECTORY_SEPARATOR . 'src' . DIRECTORY_SEPARATOR;
```

It uses WP's `plugin_dir_path`¹¹⁹ to get the plugin root directory. `__FILE__` is a magic constant¹²⁰ that contains the full path and filename of the current file. `DIRECTORY_SEPARATOR` is a predefined constant that contains either a forward slash or backslash depending on the OS your web server is on. We also use `realpath`¹²¹ to normalize the file path.

This line resolves the path to the class definition file:

```
$class_file = str_replace( '_', DIRECTORY_SEPARATOR,  
$class_name ) . '.php';
```

It replaces the underscore (`_`) in `$class_name` with the directory separator and appends `.php`.

Finally, this line builds the file path to the definition and includes the file using `require_once`:

```
require_once $classes_dir . $class_file;
```

That's it! You now have an autoloader. Say goodbye to long lines of `require_once` statements.

Plugin Container

A plugin container is a special class that holds together our plugin code. It simplifies the interaction between the many working parts of your code by providing a centralized location to manage the configuration and objects.

¹¹⁹. http://codex.wordpress.org/Function_Reference/plugin_dir_path

¹²⁰. <http://php.net/manual/en/language.constants.predefined.php>

¹²¹. <http://php.net/manual/en/function.realpath.php>

USES OF OUR PLUGIN CONTAINER

Here are the things we can expect from the plugin container:

Store global parameters in a single location

Often you'll find this code in plugins:

```
define( 'SIMPLARITY_VERSION', '1.0.0' );
define( 'SIMPLARITY_PATH', realpath( plugin_dir_path(
__FILE__ ) ) . DIRECTORY_SEPARATOR );
define( 'SIMPLARITY_URL', plugin_dir_url( __FILE__ )
);
```

Instead of doing that, we could do this instead:

```
$plugin = new Simplarity_Plugin();
$plugin['version'] = '1.0.0';
$plugin['path'] = realpath( plugin_dir_path( __FILE__
) ) . DIRECTORY_SEPARATOR;
$plugin['url'] = plugin_dir_url( __FILE__ );
```

This has the added benefit of not polluting the global namespace with our plugin's constants, which in most cases aren't needed by other plugins.

Store objects in a single location

Instead of scattering our class instantiations everywhere in our codebase we can just do this in a single location:

```
$plugin = new Simplarity_Plugin();
/.../
```

```
$plugin['scripts'] = new Simplarity_Scripts(); // A
class that loads JavaScript files
```

Service definitions

This is the most powerful feature of the container. A service is an object that does something as part of a larger system. Services are defined by functions that return an instance of an object. Almost any global object can be a service.

```
$plugin['settings_page'] = function ( $plugin ) {
    return new SettingsPage(
        $plugin['settings_page_properties'] );
};
```

Services result in lazy initialization whereby objects are only instantiated and initialized when needed.

It also allows us to easily implement a self-resolving dependency injection design. An example:

```
$plugin = new Plugin();
$plugin['door_width'] = 100;
$plugin['door_height'] = 500;
$plugin['door_size'] = function ( $plugin ) {
    return new DoorSize( $plugin['door_width'],
        $plugin['door_height'] );
};
$plugin['door'] = function ( $plugin ) {
    return new Door( $plugin['door_size'] );
};
$plugin['window'] = function ( $plugin ) {
```

```

        return new Window();
    };
    $plugin['house'] = function ( $plugin ) {
        return new House( $plugin['door'],
            $plugin['window'] );
    };
    $house = $plugin['house'];

```

This is roughly equivalent to:

```

$door_width = 100;
$door_height = 500;
$door_size = new DoorSize( $door_width, $door_height
);
$door = new Door( $door_size );
$window = new Window();
$house = new House( $door, $window );

```

Whenever we get an object, as in `$house = $plugin['house'];`, the object is created (lazy initialization) and dependencies are resolved automatically.

Building The Plugin Container

Let's start by creating the plugin container class. We will name it "Simplarity_Plugin". As our naming convention dictates, we should create a corresponding file: `src/Simplarity/Plugin.php`.

Open `Plugin.php` and add the following code:

```

<?php
class Simplarity_Plugin implements ArrayAccess {

```

```
protected $contents;

public function __construct() {
    $this->contents = array();
}

public function offsetSet( $offset, $value ) {
    $this->contents[$offset] = $value;
}

public function offsetExists($offset) {
    return isset( $this->contents[$offset] );
}

public function offsetUnset($offset) {
    unset( $this->contents[$offset] );
}

public function offsetGet($offset) {
    if( is_callable($this->contents[$offset]) ){
        return call_user_func(
            $this->contents[$offset], $this );
    }
    return isset( $this->contents[$offset] ) ?

    $this->contents[$offset] : null;
}

public function run(){
    foreach( $this->contents as $key => $content ){
```



```

// Loop on contents
    if( is_callable($content) ){
        $content = $this[$key];
    }
    if( is_object( $content ) ){
        $reflection = new ReflectionClass( $content );
        if( $reflection->hasMethod( 'run' ) ){
            $content->run(); // Call run method on
            // object
        }
    }
}
}
}
}
}

```

The class implements the **ArrayAccess** interface:

```
class Simplarity_Plugin implements ArrayAccess {
```

This allows us to use it like PHP's array:

```

$plugin = new Simplarity_Plugin();
$plugin['version'] = '1.0.0'; // Simplicity is beauty

```

The functions **offsetSet**, **offsetExists**, **offsetUnset** and **offsetGet** are required by **ArrayAccess** to be implemented. The **run** function will loop through the contents of the container and run the runnable objects.

To better illustrate our plugin container, let's start by building a sample plugin.

Example Plugin: A Settings Page

This plugin will add a settings page named “Simplarity” under WordPress Admin → Settings.

Let’s go back to the main plugin file. Open up *simplarity.php* and add the following code. Add this below the autoloader code:

```
add_action( 'plugins_loaded', 'simplarity_init' );
// Hook initialization function
function simplarity_init() {
    $plugin = new Simplarity_Plugin(); // Create
    // container
    $plugin['path'] = realpath( plugin_dir_path(
        __FILE__ ) ) . DIRECTORY_SEPARATOR;
    $plugin['url'] = plugin_dir_url( __FILE__ );
    $plugin['version'] = '1.0.0';
    $plugin['settings_page_properties'] = array(
        'parent_slug' => 'options-general.php',
        'page_title' => 'Simplarity',
        'menu_title' => 'Simplarity',
        'capability' => 'manage_options',
        'menu_slug' => 'simplarity-settings',
        'option_group' => 'simplarity_option_group',
        'option_name' => 'simplarity_option_name'
    );
    $plugin['settings_page'] = new
    Simplarity_SettingsPage(
    $plugin['settings_page_properties'] );
    $plugin->run();
}
```

Here we use WP's `add_action` to hook our function `simplarity_init` into `plugins_loaded`:

```
add_action( 'plugins_loaded', 'simplarity_init' );
```

This is important as this will make our plugin overridable by using `remove_action`. An example use case would be a premium plugin overriding the free version.

Function `simplarity_init` contains our plugin's initialization code. At the start, we simply instantiate our plugin container:

```
$plugin = new Simplarity_Plugin();
```

These lines assign global configuration data:

```
$plugin['path'] = realpath( plugin_dir_path( __FILE__ ) ) . DIRECTORY_SEPARATOR;  
$plugin['url'] = plugin_dir_url( __FILE__ );  
$plugin['version'] = '1.0.0';
```

The plugin path contains the full path to our plugin, the `url` contains the URL to our plugin directory. They will come in handy whenever we need to include files and assets. `version` contains the current version of the plugin that should match the one in the header info. Useful whenever you need to use the version in code.

This next code assigns various configuration data to `settings_page_properties`:

```
$plugin['settings_page_properties'] = array(  
    'parent_slug' => 'options-general.php',  
    'page_title' => 'Simplarity',
```

```
'menu_title' => 'Simplarity',  
'capability' => 'manage_options',  
'menu_slug' => 'simplarity-settings',  
'option_group' => 'simplarity_option_group',  
'option_name' => 'simplarity_option_name'  
);
```

These configuration data are related to WP settings API¹²².

This next code instantiates the settings page, passing along `settings_page_properties`:

```
$plugin['settings_page'] = new  
Simplarity_SettingsPage(  
$plugin['settings_page_properties'] );
```

The `run` method is where the fun starts:

```
$plugin->run();
```

It will call `Simplarity_SettingsPage`'s own `run` method.

THE SIMPLARITY_SETTINGSPAGE CLASS

Now we need to create the `Simplarity_SettingsPage` class. It's a class that groups together the settings API functions.

Create a file named `SettingsPage.php` in `src/Simplarity/`. Open it and add the following code:

¹²². http://codex.wordpress.org/Settings_API

```

<?php
class Simlarity_SettingsPage {
    protected $settings_page_properties;

    public function __construct(
        $settings_page_properties ){
        $this->settings_page_properties =
            $settings_page_properties;
    }

    public function run() {
        add_action( 'admin_menu', array( $this,
            'add_menu_and_page' ) );
        add_action( 'admin_init', array( $this,
            'register_settings' ) );
    }

    public function add_menu_and_page() {

        add_submenu_page(
            $this->settings_page_properties['parent_slug'],
            $this->settings_page_properties['page_title'],
            $this->settings_page_properties['menu_title'],
            $this->settings_page_properties['capability'],
            $this->settings_page_properties['menu_slug'],
            array( $this, 'render_settings_page' )
        );
    }

    public function register_settings() {

```

```

register_setting(
    $this->settings_page_properties['option_group'],
    $this->settings_page_properties['option_name']
);
}

```

```

public function get_settings_data(){
    return get_option(
        $this->settings_page_properties['option_name'],
        $this->get_default_settings_data() );
}

```

```

public function render_settings_page() {
    $option_name =
    $this->settings_page_properties['option_name'];
    $option_group =
    $this->settings_page_properties['option_group'];
    $settings_data = $this->get_settings_data();
    ?>
    <div class="wrap">
        <h2>Simplarity</h2>
        <p>This plugin is using the settings API.</p>
        <form method="post" action="options.php">
            <?php
                settings_fields(
                    $this->plugin['settings_page_properties']
                    ['option_group']);
            ?>
            <table class="form-table">

```

```

        <tr>
            <th><label for="textbox">Textbox:
            </label></th>
            <td>
                <input type="text" id="textbox"
                    name="<?php echo esc_attr(
                        $option_name."<[textbox]" ); ?>"
                    value="<?php echo esc_attr(
                        $settings_data['textbox'] ); ?>" />
            </td>
        </tr>
    </table>
    <input type="submit" name="submit"
        id="submit" class="button button-primary"
        value="Save Options">
</form>
</div>
<?php
}

public function get_default_settings_data() {
    $defaults = array();
    $defaults['textbox'] = '';

    return $defaults;
}
}

```

The class property `$settings_page_properties` stores the settings related to WP settings API:

```
<?php
class Simlarity_SettingsPage {
    protected $settings_page_properties;
```

The constructor function accepts the `settings_page_properties` and stores it:

```
public function __construct(
    $settings_page_properties ){
    $this->settings_page_properties =
    $settings_page_properties;
}
```

The values are passed from this line in the main plugin file:

```
$plugin['settings_page'] = new
Simlarity_SettingsPage(
    $plugin['settings_page_properties'] );
```

The `run` function is use to run startup code:

```
public function run() {
    add_action( 'admin_menu', array( $this,
        'add_menu_and_page' ) );
    add_action( 'admin_init', array( $this,
        'register_settings' ) );
}
```

The most likely candidate for startup code are filters¹²³ and action hooks¹²⁴. Here we add the action hooks related

¹²³. http://codex.wordpress.org/Plugin_API/Filter_Reference

to our settings page. Do not confuse this run method with the run method of the plugin container. This run method belongs to the settings page class.

This line hooks the `add_menu_and_page` function on to the `admin_menu` action:

```
add_action( 'admin_menu', array( $this,
    'add_menu_and_page' ) );
```

Function `add_submenu_page` in turn calls WP's `add_submenu_page`¹²⁵ function to add a link under the WP Admin → Settings:

```
public function add_menu_and_page() {

    add_submenu_page(
        $this->settings_page_properties['parent_slug'],
        $this->settings_page_properties['page_title'],
        $this->settings_page_properties['menu_title'],
        $this->settings_page_properties['capability'],
        $this->settings_page_properties['menu_slug'],
        array( $this, 'render_settings_page' )
    );

}
```

As you can see, we are pulling the info from our class property `$settings_page_properties` which we specified in the main plugin file.

¹²⁴. http://codex.wordpress.org/Plugin_API/Hooks

¹²⁵. http://codex.wordpress.org/add_submenu_page

The parameters for `add_submenu_page` are:

- `parent_slug`: slug name for the parent menu
- `page_title`: text to be displayed in the `<title>` element of the page when the menu is selected
- `menu_title`: text to be used for the menu
- `capability`: the capability required for this menu to be displayed to the user
- `menu_slug`: slug name to refer to this menu by (should be unique for this menu)
- `function`: function to be called to output the content for this page

This line hooks the `register_settings` function on to the `admin_init` action:

```
add_action( 'admin_init', array( $this,
    'register_settings' ) );
```

`array($this, 'register_settings')` means to call `register_settings` on `$this`, which points to our `SettingsPage` instance.

The `register_settings` then calls WP's `register_setting` to register a setting:

```
public function register_settings() {

    register_setting(
        $this->settings_page_properties['option_group'],
```

```

        $this->settings_page_properties['option_name']
    );

}

```

Function `render_settings_page` is responsible for rendering the page:

```

public function render_settings_page() {
    $option_name =
    $this->settings_page_properties['option_name'];
    $option_group =
    $this->settings_page_properties['option_group'];
    $settings_data = $this->get_settings_data();
    ?>
    <div class="wrap">
        <h2>Simplarity</h2>
        <p>This plugin is using the settings API.</p>
        <form method="post" action="options.php">
            <?php
            settings_fields( $option_group );
            ?>
            <table class="form-table">
                <tr>
                    <th><label for="textbox">Textbox:
                    </label></th>
                    <td>
                        <input type="text" id="textbox"
                        name="<?php echo esc_attr(
                        $option_name."<[textbox]<" ); ?>"
                        value="<?php echo esc_attr(

```

```

        $settings_data['textbox'] ); ?>" />
    </td>
</tr>
</table>
<input type="submit" name="submit" id="submit"
class="button button-primary" value="Save
Options">
</form>
</div>
<?php
}

```

We hooked `render_settings_page` earlier using `add_submenu_page`.

Function `get_settings_data` is a wrapper function for `get_option`:

```

public function get_settings_data(){
    return get_option(
        $this->plugin['settings_page_properties']
        ['option_name'] );
}

```

This is to easily get the settings data with a single function call.

Function `get_default_settings_data` is used to supply us with our own default values:

```

public function get_default_settings_data() {
    $defaults = array();
    $defaults['textbox'] = '';
}

```

```
    return $defaults;
}
```

Abstracting Our Settings Page Class

Right now our settings page class cannot be reused if you want to create another subpage. Let's move the reusable code for the settings page to another class.

Let's call this class `Simplarity_WpSubPage`. Go ahead and create the file `src/Simplarity/WpSubPage.php`.

Now add the code below:

```
<?php
abstract class Simplarity_WpSubPage {
    protected $settings_page_properties;

    public function __construct(
        $settings_page_properties ){
        $this->settings_page_properties =
            $settings_page_properties;
    }

    public function run() {
        add_action( 'admin_menu', array( $this,
            'add_menu_and_page' ) );
        add_action( 'admin_init', array( $this,
            'register_settings' ) );
    }

    public function add_menu_and_page() {
```

```
add_submenu_page(  
    $this->settings_page_properties['parent_slug'],  
    $this->settings_page_properties['page_title'],  
    $this->settings_page_properties['menu_title'],  
    $this->settings_page_properties['capability'],  
    $this->settings_page_properties['menu_slug'],  
    array( $this, 'render_settings_page' )  
);  
  
}  
  
public function register_settings() {  
  
    register_setting(  
        $this->settings_page_properties['option_group'],  
        $this->settings_page_properties['option_name']  
    );  
  
}  
  
public function get_settings_data(){  
    return get_option(  
        $this->settings_page_properties['option_name'],  
        $this->get_default_settings_data() );  
}  
  
public function render_settings_page(){  
  
}
```

```

    public function get_default_settings_data() {
        $defaults = array();

        return $defaults;
    }
}

```

Notice that it is an abstract class. This will prevent instantiating this class directly. To use it you need to extend it first with another class, which in our case is **Simplarity_SettingsPage**:

```

<?php
class Simplarity_SettingsPage extends
Simplarity_WpSubPage {

    public function render_settings_page() {
        $option_name =
        $this->settings_page_properties['option_name'];
        $option_group =
        $this->settings_page_properties['option_group'];
        $settings_data = $this->get_settings_data();
        ?>
        <div class="wrap">
            <h2>Simplarity</h2>
            <p>This plugin is using the settings API.</p>
            <form method="post" action="options.php">
                <?php
                settings_fields( $option_group );
                ?>

```

```

<table class="form-table">
    <tr>
        <th><label for="textbox">Textbox:
        </label></th>
        <td>
            <input type="text" id="textbox"
                name="<?php echo esc_attr(
                    $option_name."<?php echo esc_attr(
                        $settings_data['textbox'] );?>"
                value="<?php echo esc_attr(
                    $settings_data['textbox'] );?>"
                />
            </td>
        </tr>
    </table>
    <input type="submit" name="submit" id="submit"
        class="button button-primary" value="Save
        Options">
    </form>
</div>
<?php
}

public function get_default_settings_data() {
    $defaults = array();
    defaults['textbox'] = '';

    return $defaults;
}
}

```


The only functions we have implemented are `render_settings_page` and `get_default_settings_data`, which are customized to this settings page.

To create another WP settings page you'll just need to create a class and extend the `Simplarity_WpSubPage`. And implement your own `render_settings_page` and `get_default_settings_data`.

Defining A Service

The power of the plugin container is in defining services. A service is a function that contains instantiation and initialization code that will return an object. Whenever we pull a service from our container, the service function is called and will create the object for you. The object is only created when needed. This is called lazy initialization.

To better illustrate this, let's define a service for our settings page.

Open `simplarity.php` and add this function below the `Simplarity` code:

```
function simplarity_service_settings( $plugin ){  
  
    $object = new Simplarity_SettingsPage(  
        $plugin['settings_page_properties'] );  
    return $object;  
}
```

Notice that our service function has a `$plugin` parameter which contains our plugin container. This allows us to access all configuration, objects, and services that have been stored in our plugin container. We can see that the

`Simplarity_SettingsPage` has a dependency on `$plugin['settings_page_properties']`. We inject this dependency to `Simplarity_SettingsPage` here. This is an example of dependency injection. Dependency injection is a practice where objects are designed in a manner where they receive instances of the objects from other pieces of code, instead of constructing them internally. This improves decoupling of code.

Now let's replace this line in `simplarity_init`:

```
$plugin['settings_page'] = new
Simplarity_SettingsPage(
    $plugin['settings_page_properties'] );
```

with a service definition assignment:

```
$plugin['settings_page'] =
'simplarity_service_settings'
```

So instead of assigning our object instance directly, we assign the name of our function as string. Our container handles the rest.

Defining A Shared Service

Right now, every time we get `$plugin['settings_page']`, a new instance of `Simplarity_SettingsPage` is returned. Ideally, `Simplarity_SettingsPage` should only be instantiated once as we are using WP hooks, which in turn should only be registered once.

To solve this we use a shared service. A shared service will return a new instance of an object on first call, on succeeding calls it will return the same instance.

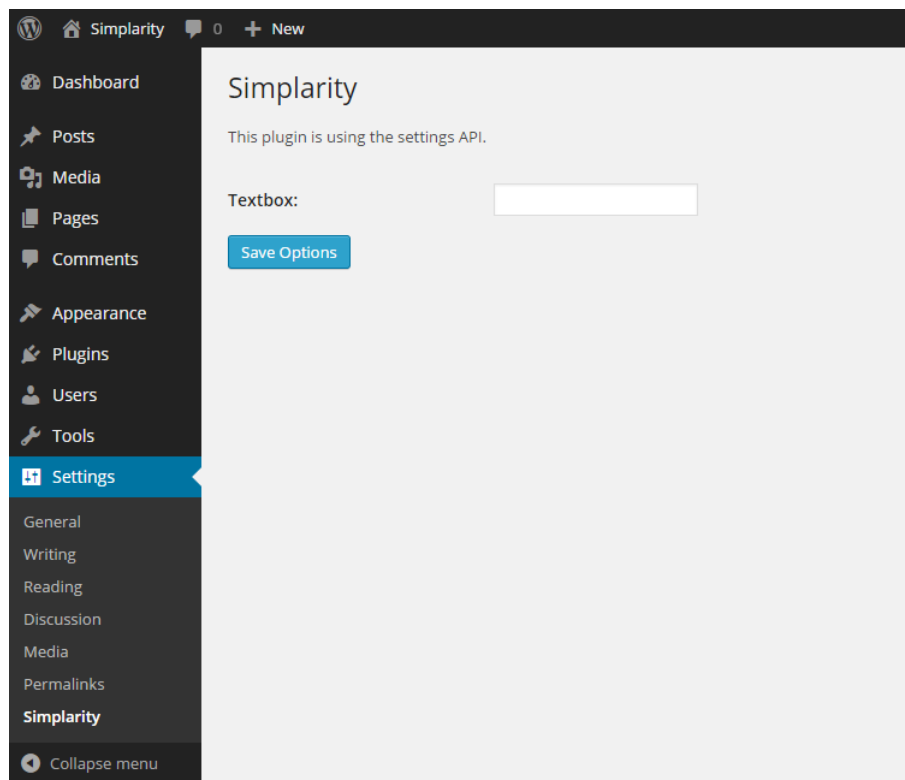
Let's create a shared service using a static variable:

```
function simplarity_service_settings( $plugin ){  
    static $object;  
  
    if (null !== $object) {  
        return $object;  
    }  
  
    $object = new Simplarity_SettingsPage(  
        $plugin['settings_page_properties'] );  
    return $object;  
}
```

On first call, `$object` is null, and on succeeding calls it will contain the instance of the object created on first call. Notice that we are using a static variable. A static variable exists only in a local function scope, but it does not lose its value when program execution leaves this scope.

That's it.

Now if you activate the plugin, an admin menu will appear in Admin → Settings named “Simplarity”. Click on it and you will be taken to the settings page we have created.



Settings Page In Action

The Future: PHP 5.3+

Earlier we mentioned that our class naming convention was future-ready. In this section we will discuss how our codebase will work in PHP version 5.3 and up. Two of the best features that have graced the PHP world are namespaces and anonymous functions.

NAMESPACES

PHP does not allow two classes or functions to share the same name. When this happens, a name collision occurs and causes a nasty error.

With namespaces you can have the same class names as long as they live in their own namespace. A good analo-

gy for namespaces are the folders you have in your OS. You cannot have files with the same name in one folder. However, you can have the same filenames in different folders.

With namespaces, class and function names won't need unique prefixes anymore.

ANONYMOUS FUNCTIONS

Anonymous functions, also known as closures, allow the creation of functions which have no specified name. They are most useful as the value of callback parameters, but they have many other uses. You can also store closures in variables.

Here's an example of closure:

```
<?php
$greet = function($name) {
    printf("Hello %s\r\n", $name);
};

$greet('World');
$greet('PHP');
```

Using Namespaces In Classes

Let's go ahead and use namespaces in our class definitions. Open up the following files in *src/Simplarity*:

- *Plugin.php*
- *SettingsPage.php*

- *WpSubPage.php*

In each of these files, add a namespace declaration on top and remove the “Simplarity_” prefix on class names:

```
// Plugin.php
namespace Simplarity;

class Plugin {
    ...

// SettingsPage.php
namespace Simplarity;

class SettingsPage extends WpSubPage {
    ...

// WpSubPage.php
namespace Simplarity;

abstract class WpSubPage {
    ...
```

Since we have updated our class names we also need to update our class instantiations in *simplarity.php*. We do this by deleting the prefixes:

```
function simplarity_init() {
    $plugin = new Plugin();
    ...
}
...
```

```
function simplarity_service_settings( $plugin ){

    ...

    $object = new SettingsPage(
        $plugin['settings_page_properties'] );
    return $object;
}
```

By default, PHP will try to load the class from the root namespace so we need to tell it about our namespaced classes. We add this to the top of *simplarity.php* just above the autoloader code:

```
use Simplarity\Plugin;
use Simplarity\SettingsPage;
```

This is called importing/aliasing with the use operator¹²⁶.

Updating The Autoloader

Open up *simplarity.php* and change this line in the autoloader from:

```
$class_file = str_replace( '_', DIRECTORY_SEPARATOR,
    $class_name ) . '.php';
```

to:

```
$class_file = str_replace( '\\', DIRECTORY_SEPARATOR,
    $class_name ) . '.php';
```

¹²⁶. <http://php.net/manual/en/language.namespaces.importing.php>

Remember that in 5.2 code we are using underscores as hierarchy separators. For 5.3+ we are using namespaces which use backslash “\” as hierarchy separators. Thus we simply swap “_” for “\”. We use another backslash to escape the original one: “\\”.

Updating Our Service Definitions To Use Anonymous Functions

We can now replace the global functions we created for our service definitions with anonymous functions. So instead of doing this:

```
function simplarity_init() {
    ...
    $plugin['settings_page'] =
        'simplarity_service_settings';
    ...
}
...
function simplarity_service_settings( $plugin ){
    static $object;

    if (null !== $object) {
        return $object;
    }

    $object = new Simplarity_SettingsPage(
        $plugin['settings_page_properties'] );
    return $object;
}
```


we can just replace this with an inline anonymous function:

```
function simplarity_init() {  
    $plugin = new Plugin();  
    ...  
    $plugin['settings_page'] = function ( $plugin ) {  
        static $object;  
  
        if (null !== $object) {  
            return $object;  
        }  
        return new SettingsPage(  
            $plugin['settings_page_properties'] );  
    };  
    ...  
}
```

Using Pimple As A Plugin Container

Pimple is a small dependency injection (DI) container for PHP 5.3+. Pimple has the same syntax as our simple plugin container. In fact our plugin container was inspired by Pimple. In this part, we will extend Pimple and use it.

Download Pimple container from GitHub¹²⁷ and save it in `src/Simplarity/Pimple.php`.

Open up `Pimple.php` and replace the namespace and the classname to:

¹²⁷. <https://raw.githubusercontent.com/silexphp/Pimple/master/src/Pimple/Container.php>

```
...
namespace Simplarity;

/**
 * Container main class.
 *
 * @author Fabien Potencier
 */
class Pimple implements \ArrayAccess
...

```

Open up *Plugin.php* and replace all the code with:

```
<?php
namespace Simplarity;

class Plugin extends Pimple {

    public function run(){
        foreach( $this->values as $key => $content ){
            // Loop on contents
            $content = $this[$key];

            if( is_object( $content ) ){
                $reflection = new \ReflectionClass( $content
                );
                if( $reflection->hasMethod( 'run' ) ){
                    $content->run(); // Call run method on
                    // object
                }
            }
        }
    }
}

```

```

    }
}
}

```

Now let's change the service definition in *simplarity.php* to:

```

$plugin['settings_page'] = function ( $plugin ) {
    return new SettingsPage(
$plugin['settings_page_properties'] );
};

```

By default, each time you get a service, Pimple returns the same instance of it. If you want a different instance to be returned for all calls, wrap your anonymous function with the **factory()** method:

```

$plugin['image_resizer'] = $plugin->factory(function
( $plugin ) {
    return new ImageResizer( $plugin['image_dir'] );
});

```

Conclusion

The PHP community is big. A lot of best practices have been learned over the years. It's good to always look beyond the walled garden of WordPress to look for answers. With autoloading and a plugin container we are one step closer to better code.

CODE SAMPLES

- [Simplarity: settings page](#)¹²⁸
- [Simplarity: settings page \(PHP5.3+\)](#)¹²⁹

RESOURCES

- [PEAR naming standards](#)¹³⁰
- [Namespaces explanation](#)¹³¹
- [Pimple \(a small DI container\)](#)¹³² 🐘

¹²⁸. <https://github.com/kosinix/simplarity>

¹²⁹. <https://github.com/kosinix/simplarity-php53>

¹³⁰. <http://pear.php.net/manual/en/standards.naming.php>

¹³¹. <http://daylerees.com/php-namespaces-explained>

¹³². <https://github.com/silexphp/Pimple>

How To Deploy WordPress Plugins With GitHub Using Transients

BY MATTHEW RAY 🐘

If you've worked with WordPress for a while, you may have tried your hand at writing a plugin. Many developers will start creating plugins to enhance a custom theme or to modularize their code. Eventually, though, you may want to distribute your plugin to a wider audience.

While you always have the option to use the WordPress Subversion repository, there may be instances where you prefer to host a plugin yourself. Perhaps you are offering your users a premium plugin. Maybe you need a way to keep your client's code in sync across multiple sites. It could simply be that you want to use a Git workflow instead of Subversion. Whatever the reason, this tutorial will show you how to set up a GitHub repository to push updates to your plugin, wherever it resides.

The Plan

Before we get too far into the code we should start with an outline of what we will be doing:

1. First, we will learn a little bit about *transients* and how they work within WordPress.
2. Then, we will build a PHP class to house all of our code.

3. Next, we are going to connect our plugin to GitHub.
4. Lastly, we will create the interface elements that allow users to interact with our plugin.

When you finish this article you should have a fully functioning plugin that will update directly using GitHub releases. Ready to get started?

WordPress Transients

First of all, what are WordPress transients¹³³? WordPress transients are a short-lived entry of data that is stored for a defined duration and will be automatically removed when it has expired — think of them as server-side cookies. WordPress uses transients to store information about all of the plugins that have been installed, including their version number. Every once in a while WordPress will refresh the data that is stored in the transient. It is this event that WordPress uses to check the Subversion repository for updated plugins. It is also this event that we will use to check for updates on our own plugin on GitHub.

Getting Started

Let's begin by setting up our updater class:

```
class Smashing_Updater {  
    protected $file;
```

¹³³ https://codex.wordpress.org/Transients_API

```

public function __construct( $file ) {
    $this->file = $file;
    return $this;
}
}

```

The first thing we must do is create a class name and a constructor (the class name can be anything you want). We are going to need to figure out some basic information about the WordPress plugin we are updating, including the version number. We'll do this by passing the main plugin file's path into our updater and then we'll assign that value to a property.

You might be wondering why we need to pass the plugin's path into our class. You see, the way WordPress stores plugin information is by using the main plugin file's path as a unique identifier (i.e. the *basename*). Let's say our plugin is in the directory */wp-content/plugins/smashing-plugin/smashing-plugin.php*, the *basename* for our plugin would be *smashing-plugin/smashing-plugin.php*. We will use this *basename* to check if the plugin we are updating is activated, among other things.

Next we need to get the plugin data and set it to a property in our class:

```

class Smashing_Updater {
    protected $file;
    protected $plugin;
    protected $basename;
    protected $active;
}

```

```

public function __construct( $file ) {
    $this->file = $file;
    add_action( 'admin_init', array( $this,
        'set_plugin_properties' ) );
    return $this;
}

public function set_plugin_properties() {
    $this->plugin    = get_plugin_data( $this->file );
    $this->basename = plugin_basename( $this->file );
    $this->active    = is_plugin_active(
        $this->basename );
}

}

```

You may have noticed that I am using the action `admin_init`¹³⁴ to set the plugin properties. This is because the function `get_plugin_data()` may not have been defined at the point in which this code was called. By hooking it to `admin_init` we are ensuring that we have that function available to get our plugin data. We are also checking if the plugin is activated, and assigning that and the plugin object to properties in our class.

To learn more about WordPress actions and filters you should take a look at the plugin API¹³⁵.

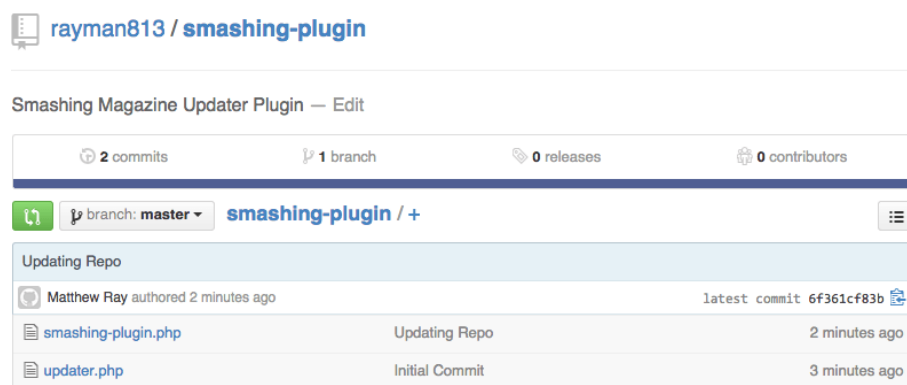
¹³⁴. https://codex.wordpress.org/Plugin_API/Action_Reference

¹³⁵. https://codex.wordpress.org/Plugin_API

Now that we have our class set up and grabbed some of the basic plugin data we'll need, it's time we start talking about GitHub.

Setting Up GitHub

We can start by creating a repository for our plugin. It is important to remember that we will pull the files down from here, so the directory structure is important. You'll need the root of your repository to be the contents of your plugin folder, not the plugin folder itself. Here is an example of what you should see:



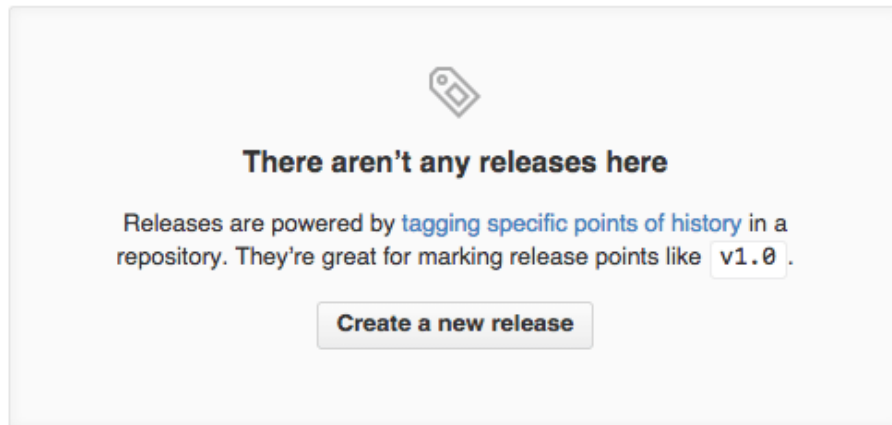
The root directory of our GitHub repository.

In my repository I have two files. The main plugin file *smashing-plugin.php* and the updater script *updater.php*. Your repo may look a little different depending on your plugin files.

Now that we have a repository set up, let's talk about how we are going to check the version number on GitHub. We have a couple of options here. We could pull down the main file and parse the contents to try to find

the version number. However, I prefer using the built-in release functionality of GitHub. Let's add a new release.

Start by going to the “releases” tab and hitting “Create a new release”:



Creating a new release in the GitHub release section.

Here you will find a few fields we need to fill out. The most important one is the “Tag version” field. This is where we will put the current release version of our plugin. By now, you are probably familiar with the semantic versioning format¹³⁶. This is the format we are going to use for our field. We should also use this format for our main plugin file, in the PHP comments. Continue filling out the title and description fields in the release until your release looks like something like this:

¹³⁶. <http://semver.org/>

1.0.0

@

Target: master

Excellent! This tag will be created from the target when you publish this release.

This is our first release!

Write

Preview

Markdown supported

Edit in fullscreen

Initial release notes about our plugin!

Attach images by dragging & dropping, [selecting them](#), or pasting from the clipboard.

Attach binaries by dropping them here or [selecting them](#).

☐ **This is a pre-release**
We'll point out that this release is identified as non-production ready.

Publish release

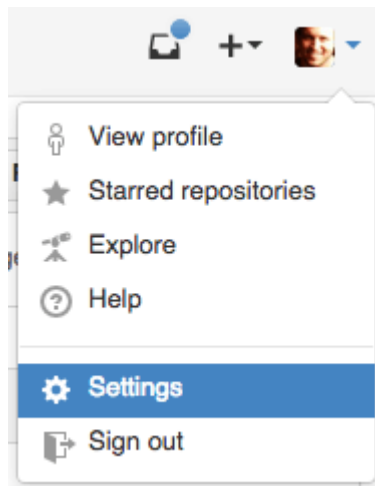
Save draft

A new release of our plugin.

Hit the “Publish release” button to create the release. This will create a ZIP file in our repo that we can use in our up-dater script. Then, you simply repeat this process whenever you want to deploy a new plugin release. We can now use the GitHub API to check the version number and we have a ZIP file ready to go.

BUT I HAVE A PRIVATE PLUGIN!

Don’t worry, just a couple of extra steps. Private plugins require you to pass an authorization token when you make requests to the API. First go to your account settings:



GitHub settings panel.

Then go to “Personal access tokens” on the left menu and click “Generate a new access token” button on the top right. You will be brought to this page:

A screenshot of the GitHub 'New personal access token' page. The page title is 'Applications / New personal access token'. It has a 'Token description' field. Below it, a question 'What's this token for?' is followed by a 'Select scopes' section. The 'Select scopes' section lists various scopes with checkboxes: 'repo' (checked), 'public_repo', 'user:email', 'write:org', 'write:public_key', 'write:repo_hook', 'gist', 'repo:status', 'delete_repo', 'user:follow', 'read:org', 'read:public_key', 'read:repo_hook', 'notifications', 'repo_deployment', 'user' (checked), 'admin:org', 'admin:public_key', 'admin:repo_hook', and 'admin:org_hook'. A green 'Generate token' button is at the bottom. A footnote explains that personal access tokens function like ordinary OAuth access tokens and can be used instead of a password for Git over HTTPS, or to authenticate to the API over Basic Authentication.

GitHub generate token.

You can give the token any description you like, then select the repo scope and hit “Generate token”. Once you’ve generated your token, copy it down somewhere — we will use it later.

Connecting To GitHub

We can now start adding some code to connect our updater to GitHub. First we’ll add a few more properties to hold our repo information, and one to hold our response from GitHub:

```
private $username;  
private $repository;  
private $authorize_token;  
private $github_response;
```

Then we’ll add some setters for our new properties:

```
public function set_username( $username ) {  
    $this->username = $username;  
}  
public function set_repository( $repository ) {  
    $this->repository = $repository;  
}  
public function authorize( $token ) {  
    $this->authorize_token = $token;  
}
```

These setters allow us to modify usernames and repositories without reinstantiating our updater class. Say you wanted to have an unlockable feature or require the users

to register, you could change the repo to a pro version of your plugin by simply adding a conditional statement. Let's take a look at how we should include this code in our main plugin file:

```
// Include our updater file
include_once( plugin_dir_path( __FILE__ ) .
'update.php' );

$updater = new Smashing_Updater( __FILE__ ); //
// instantiate our class
$updater->set_username( 'rayman813' ); // set username
$updater->set_repository( 'smashing-plugin' ); // set
// repo
```

Now let's write the code we will need to get the version tag from GitHub.

```
private function get_repository_info() {
    if ( is_null( $this->github_response ) ) { // Do we
        // have a response?
        $request_uri = sprintf( 'https://api.github.com/
        repos/%s/%s/releases', $this->username,
        $this->repository ); // Build URI
        if( $this->authorize_token ) { // Is there an
            // access token?
            $request_uri = add_query_arg( 'access_token',
            $this->authorize_token, $request_uri );
            // Append it
        }
        $response = json_decode( wp_remote_retrieve_body(
```

```

wp_remote_get( $request_uri ) ), true ); // Get
// JSON and parse it
if( is_array( $response ) ) { // If it is an array
    $response = current( $response ); // Get the
    // first item
}
if( $this->authorize_token ) { // Is there an
// access token?
    $response['zipball_url'] = add_query_arg(
        'access_token', $this->authorize_token,
        $response['zipball_url'] ); // Update our zip
    // url with token
}
$this->github_response = $response; // Set it to
// our property
}
}

```

In this method we are checking if we've already gotten a response; if we haven't, then make a request to the GitHub API endpoint using the username and repo that we've supplied. We've also added some code to check for an access token for private repos. If there is one, append it to the URL and update the zipball URL (the file we will download when we update). We are then getting the JSON response, parsing it, and grabbing the latest release. Once we have the latest release, we're setting it to the property we created earlier.

Modifying WordPress

Alright, so far we've collected information about our plugin and our repo, and we've connected to the repo using the API. Now it's time to start modifying WordPress to find our new data and put it in the right spots. This is going to involve three steps:

1. Modifying the output of the WordPress update transient.
2. Updating the WordPress interface to show our plugin's info properly.
3. Making sure our plugin is working properly after the update.

Before we get too deep into how we are going to accomplish each step, we should talk about how we are going to hook into the transient.

HOOKING INTO THE UPDATE TRANSIENT

There are a few ways to hook into this transient event. There are two filters and two actions we can use to inject our code; they are:

- Filter: `pre_set_transient_update_plugins`
- Filter: `pre_set_site_transient_update_plugins`
- Action: `set_transient_update_plugins`
- Action: `set_site_transient_update_plugins`

You will notice that the tags are fairly similar. The difference here is the addition of the word `site`. This distinction is important because it changes the scope of our injected code. The tags without the word `site` will only work on single-site installations of WordPress; the other will work on both single-site and multi-site installations. Depending on your plugin you may choose to use one over the other.

I simply want to modify the default transient, so I am going to use one of the filters. I've decided to use `pre_set_site_transient_update_plugins` so that our code will work on both single and multi-site installations.

Update: If you are using this script to update a theme you can also use the filter `pre_set_site_transient_update_themes`. Source: Andy Fragen¹³⁷

Let's take a look at our initialize function:

```
public function initialize() {
    add_filter(
        'pre_set_site_transient_update_plugins', array(
            $this, 'modify_transient' ), 10, 1 );
    add_filter( 'plugins_api', array( $this,
        'plugin_popup' ), 10, 3);
    add_filter( 'upgrader_post_install', array( $this,
        'after_install' ), 10, 3 );
}
```

In this example I have added three filters. Each one corresponds to the steps we talked about earlier. Our first filter

¹³⁷. <https://github.com/afragen>

is going to modify the transient; the second will ensure that our plugin data gets passed into the WordPress interface; and the last will make sure that our plugin is activated after the update.

Now we need to call this new method in our main plugin file to initialize the updater script. Here is what the updated code should look like:

```
// Include our updater file
include_once( plugin_dir_path( __FILE__ ) .
'update.php' );

$updater = new Smashing_Updater( __FILE__ ); //
instantiate our class
$updater->set_username( 'rayman813' ); // set username
$updater->set_repository( 'smashing-plugin' ); // set
// repo
$updater->initialize(); // initialize the updater
```

Now our initialize method will run and the filters within the method will be active. If you are planning on having your plugin's automatic updates be a premium feature, the initialize method would be a good spot to put your checks and conditionals.

WRITING THE CODE FOR OUR FILTERS

I have written out all of the methods we will need for each filter we will be using. We can start with the `modify_transient` method:

```

public function modify_transient( $transient ) {

    if( property_exists( $transient, 'checked' ) ) {
        // Check if transient has a checked property
        if( $checked = $transient->checked ) { // Did
            // WordPress check for updates?

            $this->get_repository_info(); // Get the repo
            // info
            $out_of_date = version_compare(
                $this->github_response['tag_name'],
                $checked[$this->basename], 'gt' ); // Check if
            // we're out of date
            if( $out_of_date ) {
                $new_files =
                    $this->github_response['zipball_url']; // Get
                    // the ZIP
                $slug = current( explode('/', $this->basename
                ) ); // Create valid slug
                $plugin = array( // setup our plugin info
                    'url' => $this->plugin["PluginURI"],
                    'slug' => $slug,
                    'package' => $new_files,
                    'new_version' =>
                        $this->github_response['tag_name']
                );
                $transient->response[ $this->basename ] =
                    (object) $plugin; // Return it in response
            }
        }
    }
}

```

```

    return $transient; // Return filtered transient
}

```

This snippet simply takes the version number from the comments in our main plugin file, and compares them with the tag name we gave our release on GitHub. If the one on GitHub is higher, our code tells WordPress that there is a newer version available. If you change the version number in the main plugin file to a version that is lower than our GitHub tag, you should start seeing the update notification in the WordPress admin:



WordPress plugin interface showing updates available.

You might notice, however, that if you click on the “View version 1.0.0 details” link, you receive an error from WordPress. This is because WordPress is trying to find the details of this plugin from its repositories. Instead, we need to load our own data about the plugin. That is where our next method, `plugin_popup`, comes in:

```

public function plugin_popup( $result, $action, $args
) {
    if( ! empty( $args->slug ) ) { // If there is a slug
        if( $args->slug == $this->basename ) { // And
            // it's our slug
            $this->get_repository_info(); // Get our repo
            // info
        }
    }
}

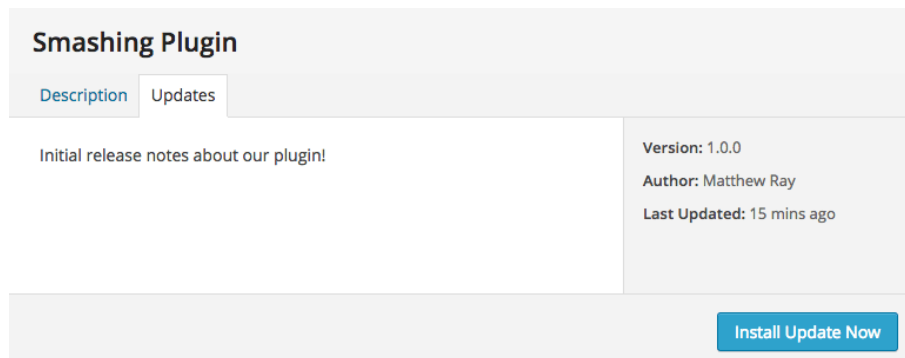
```

```

// Set it to an array
$plugin = array(
    'name'            => $this->plugin["Name"],
    'slug'            => $this->basename,
    'version'         =>
        $this->github_response['tag_name'],
    'author'          =>
        $this->plugin["AuthorName"],
    'author_profile'  =>
        $this->plugin["AuthorURI"],
    'last_updated'    =>
        $this->github_response['published_at'],
    'homepage'        =>
        $this->plugin["PluginURI"],
    'short_description' =>
        $this->plugin["Description"],
    'sections'        => array(
        'Description'  =>
            $this->plugin["Description"],
        'Updates'      =>
            $this->github_response['body'],
    ),
    'download_link'   =>
        $this->github_response['zipball_url']
);
return (object) $plugin; // Return the data
}
}
return $result; // Otherwise return default
}

```

This snippet is actually pretty simple. We are just checking if WordPress is looking for data about our plugin, and if it is, returning our own array of data. We get the data from a combination of our GitHub response and our plugin's comment data. You might have also noticed the **sections** key in this array. Each item in that array is output as a tab in the plugin popup meta box. You could add pretty much anything you'd like in those sections, including HTML. Right now, we are just showing the plugin description and the release notes we wrote earlier.



Plugin update popup.

Let's take a look at our final method, **after_install**:

```
public function after_install( $response,
$hook_extra, $result ) {
    global $wp_filesystem; // Get global FS object

    $install_directory = plugin_dir_path( $this->file
    ); // Our plugin directory
    $wp_filesystem->move( $result['destination'],
    $install_directory ); // Move files to the plugin
    // dir
```

```

$result['destination'] = $install_directory; // Set
// the destination for the rest of the stack

if ( $this->active ) { // If it was active
    activate_plugin( $this->basename ); // Reactivate
}
return $result;
}

```

This snippet does two things. First, it sets an argument named **destination**, which is simply the directory where WordPress is going to install our new code. We are passing the main plugin file’s directory into this argument since it *should* always be the root of the plugin. Second, it checks if the plugin was activated using the property we set earlier. If it was active before updating, our method will reactivate it.

Conclusion

Congratulations! You should now be able to update your plugin by clicking the “Update now” button on the plugins page. You should keep in mind, though, that your plugin’s version in the main plugin file needs to be updated to match the current release of your plugin — they should always match. If you didn’t update your version in the comments, you may find yourself in an endless “There is an update available” loop.

If you would like to see the completed script from this tutorial, you can view it in the sample [GitHub repository](https://github.com/rayman813/smashing-plugin)¹³⁸ that we made. 🐼

¹³⁸. <https://github.com/rayman813/smashing-plugin>

About The Authors

Brian Onorio

Brian Onorio is the Founder and President of O3 Creative¹³⁹. He earned his Bachelor of Science in Computer Science from North Carolina State University and brings 10 years of industry experience to the table. He plays an active role in day-to-day operations, supporting his team throughout the project journey, from strategy, design, implementation, and release. Brian is recognized in the industry for bringing creative approaches to digital marketing and was instrumental in growing and tripling the size of O3 Creative over the course of 2015.

Twitter: @brianonorio¹⁴⁰.

Carlo Daniele

Carlo is a freelance front-end designer and developer based in Romagna, Italy. In the last years he's been writing for both printed and digital computer magazines, dealing with web services and web standards, but his main interest is WordPress. Currently, he's writing for Smashing Magazine¹⁴¹ and HTML.it¹⁴². You can follow him on Twitter @carlodaniele¹⁴³.

¹³⁹. <http://www.weareo3.com/>

¹⁴⁰. <https://twitter.com/brianonorio>

¹⁴¹. <http://www.smashingmagazine.com/author/carlodaniele/>

¹⁴². <http://www.html.it/autore/c-daniele/>

¹⁴³. <https://twitter.com/carlodaniele>

Daniel Pataki

Daniel Pataki builds plugins, themes and apps - then proceeds to write or talk about them. He's the editor for the WordPress section on Smashing Magazine and contributes to various other online sites. When not coding or writing you'll find him playing board games or running with his dog. Drop him a line on [Twitter](#)¹⁴⁴ or visit his [personal website](#)¹⁴⁵.

Karol K

Karol K is a blogger and writer for hire. His work has been published all over the web, on sites like [NewInternetOrder.com](#)¹⁴⁶, [MarketingProfs.com](#), [Adobe.com](#), [ProBlogger](#), [ThemeIsle](#)¹⁴⁷, and others. Feel free to contact him to find out how he can help your business grow by writing unique and engaging content for your blog or website. Twitter: [@carlosinho](#)¹⁴⁸.

Matthew Ray

[Matthew Ray](#)¹⁴⁹ is a full-stack web developer working for the worldwide public relations firm, Weber Shandwick. He currently focusses his efforts on product development, plugin architecture, and service integration. In his

¹⁴⁴. <http://twitter.com/danielpataki/>

¹⁴⁵. <http://danielpataki.com/>

¹⁴⁶. <http://newinternetorder.com/>

¹⁴⁷. <https://themeisle.com/>

¹⁴⁸. <https://twitter.com/carlosinho>

¹⁴⁹. <http://www.matthewray.com>

free time, Matthew is an avid photographer, musician and bulldog enthusiast. Twitter: [@matthewray1](https://twitter.com/matthewray1)¹⁵⁰.

Nick Schäferhoff

Nick is an entrepreneur, online marketer and professional blogger. A longtime WordPress enthusiast, he helps clients across the world build successful online businesses through a mix of content marketing, blogging and web design. You can get in touch with him via [Twitter](#)¹⁵¹ or through his [website](#)¹⁵².

Nico Amarilla

Nico Amarilla is a web developer working remotely on a small island in the Philippines where he makes plugins and themes for WordPress. He is fond of finding simple solutions to complex problems. When not coding, he is either backpacking or riding his mountain bike. He has a [blog](#)¹⁵³ where he occasionally posts about web related stuff.

¹⁵⁰. <https://twitter.com/matthewray1>

¹⁵¹. <https://twitter.com/nschaeferhoff>

¹⁵². <http://www.nickschaeferhoff.de/en>

¹⁵³. <http://www.kosinix.com/>

About Smashing Magazine

Smashing Magazine¹⁵⁴ is an online magazine dedicated to Web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy¹⁵⁵.

Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is — and always has been — a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised. Since its emergence back in 2006 Smashing Magazine has proven to be a trustworthy online source.

¹⁵⁴. <http://www.smashingmagazine.com>

¹⁵⁵. <http://www.smashingmagazine.com/publishing-policy/>