

WordPress Essentials

Techniques for Beginners

A large, bold, blue letter 'W' is positioned in the bottom right corner of the cover. It is set against a white circular background that overlaps with the blue background of the cover. The 'W' is a simple, sans-serif font.

Imprint

Published in November 2011

Smashing Media GmbH, Freiburg, Germany

Cover Design: Ricardo Gimenes

Editing: Andrew Rogerson, Talita Telma

Proofreading: Andrew Lobo, Iris Ljesnjanin

Idea and Concept: Sven Lennartz, Vitaly Friedman

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

ISBN: 9783943075168

Version: December 16, 2011

Table of Contents

[Preface](#)

[Building WordPress Themes You Can Sell](#)

[Developing WordPress Locally With MAMP](#)

[The Developer's Guide To Conflict-Free JavaScript And CSS In WordPress](#)

[Interacting With The WordPress Database](#)

[How To Create A WordPress Plugin](#)

[How To Integrate Facebook With WordPress](#)

[How To Use AJAX In WordPress](#)

[Better Image Management With WordPress](#)

[Using HTML5 To Transform WordPress' TwentyTen Theme](#)

[The Authors](#)

Preface

The advanced flexibility of WordPress is one of the main reasons for its popularity among online publishers as it is considered as the number one blogging tool in its category. With its latest releases, it has extended its potential well beyond blogging, moving toward an even more advanced, robust and very powerful content management solution, and so much more. However, where it falls short, there are a wealth of plugins, widgets and themes that extend its limitations.

This Smashing eBook #10: WordPress Essentials is created to help Web developers as well as designers how to extend the functionality of WordPress with plugins and introduce techniques and methods for customizing themes. Several new features were added which make WordPress manage media even more practical, and this eBook is going to show you just how.

The eBook contains 9 articles that will guide you on how to add and organize media, to avoid problems with JavaScript and CSS, build themes, interact with your database, and learn how to setup AJAX – just to mention a few techniques. Social networks are a wonderful marketing tool that should not be ignored. Find out how to present your WordPress blog on Facebook, as well!

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

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

— Andrew Rogerson, Smashing eBook Editor

Building WordPress Themes You Can Sell

Sawyer Hollenshead

When I took my first steps into the WordPress theme arena, I didn't know much about it. I wandered blindly into the business, not knowing whether I was doing things correctly. Over time, through trial and error and making rookie mistakes, I learned some valuable lessons and gained important insights. To save you from going down the same winding path, I'll share some of the important takeaways that I've learned so far, like how to gain a solid user base, what to include in your themes and, most importantly, what to leave out.



Gaining a Solid User Base

You could build the best WordPress theme in the world, but it won't matter unless people know about it and use it. One of the smartest things I did when starting my theme business was to release a free theme. It took a while for it to gain traction, but things took off once it got some attention from being featured on other websites. Consumers are willing to download a free theme from the new kid on the block and try it out because hardly any financial risk is involved.

The free theme was unique and easy to use, and people liked it so much that they began requesting a premium (i.e. commercial) version, with more features (the free version had the bare essentials). To this day, the premium version is still one of my best sellers. Consumers like to download the free version to try it out, and then they typically purchase the commercial version. Value is added to the commercial version with support, updates, easier customization and a bundle of exclusive features. Releasing a free theme enabled me to gain momentum and build on a solid user base as I began creating commercial themes, and I'll return to that strategy in the future to increase sales.

MY FIRST MISTAKE

The first mistake I made when getting started, and one that I still kick myself over, is that I didn't implement a newsletter opt-in method for users who downloaded my themes. This would have given me a long list of consumers to whom I could market my premium themes, and it would have been extremely valuable when I launched the commercial version of my theme a year later. I've now gotten my act together and have a booming mailing list that I email every time I release a new theme, thus generating sales that would otherwise have been lost.

YOU'VE GOT 'EM, NOW KEEP 'EM

Once I had a solid user base, I found that in order to keep them as returning customers, I had to add value not only to my themes but to my services. When you start a theme business, you're not just selling themes; you're also providing support and updates. Some of the top brands make great products and provide excellent support. Think of Apple, MediaTemple and Zappos. Say what you want about these companies, but there's no denying that their user base is loyal because of both their products and their support and services.

One way to provide great support is simply to be timely with your responses. A customer loves nothing more than being responded to the same day. If you don't know the answer to their question, at least let them know that you received it and are looking into it. You would think this is common practice, but you'd be amazed at how long some companies take to respond. If you can provide killer support, you're already one step ahead of a lot of the competition.

Another way to add value is to provide educational resources that teach customers how to get the most out of your products. Some users will be more advanced than others, and they are usually the ones who purchase themes regularly. If you can provide a resource that enables those users to derive extra value from your products, then they will be more likely to stay with you and purchase more of your themes.

Streamline Your Process

Streamline and standardize your development process as much as possible. One way to do this is to use a theme framework, whether your own or a third party's. Using a framework to quickly develop a theme is important when an eager audience is waiting on you. Most importantly, when you use the same framework, updating all of your themes after they've been released is easier. For example, all of my themes display a notification in the administration panel when an update becomes available.

The code that enables this notification is in a file named `framework-init.php`. In this file is a bunch of other important blocks of code that add features, such as the theme options panel and custom post fields, as well as common functions used throughout all of my themes. When I need to update that code, I simply make the change to my framework's file and then that file gets replaced in all of my themes. By knowing that the file is the same throughout all of my themes, I don't have to bother going through each theme to find that block of code to update. You can see how this becomes valuable when your inventory starts to accumulate.

Hybrid Theme

A WordPress theme framework

[Home](#)[About](#)[Showcase](#)[Test](#)

Lists help keep you organized

by [Justin Tadlock](#) on November 2, 2008

No WordPress theme worth its salt would be complete without some good default styles for lists. Lists really draw the reader's attention. They're usually short and to the point. And since most people have short attention spans these days, you need to wow them with your lists. [Continue reading "Lists help keep you organized"](#)

Posted in [Elements](#) | Tagged [Elements](#), [Lists](#) | [47 Responses](#)

Styling all the elements

by [Justin Tadlock](#) on November 2, 2008

One of the great things about using a theme framework is that just about everything has been thought of, especially when it comes to handling all the elements that you might think to use. Solid element styling is the foundation of any good theme though. [Continue reading "Styling all the elements"](#)

Posted in [Elements](#) | Tagged [Elements](#), [Tags](#) | [11 Responses](#)

Pages

- [About](#)
- [Showcase](#)
 - [Archives](#)
 - [Authors](#)
 - [Biography](#)
 - [Blog](#)
 - [Bookmarks](#)
 - [Categories](#)
 - [Logged In](#)
 - [No Widgets](#)
 - [Tags](#)
 - [Widgets](#)
- [Test](#)

Text

This is an example of a WordPress widget, you could edit this to put information about yourself or your site so readers know where you are coming from. You can create as many widgets like this one as you like and manage all of your content inside of WordPress.

Hybrid is one of the more popular theme frameworks, thanks to its extensive list of features, including translations into 20+ languages and theme hooks.

CUSTOM VS. THIRD-PARTY FRAMEWORKS

From the beginning, I decided to build my own framework, mainly because I would know it back to front, making it easier to maintain and build on (being a control freak might have contributed to the decision as well). A custom framework also meant that I wouldn't have to rely on someone else, and the framework would have exactly what I needed and nothing else.

This is, of course, just personal preference, and many people prefer to use a third-party framework. By using a third party's, you save the time it takes to develop a solid framework. It also means that you're not solely responsible for maintaining the framework, and you will usually have a support system to turn to if you run into development issues. A lot of impressive frameworks offer useful functionality, such as theme hooks, extensible layout options, styling for popular plugins and much more. Lastly, there is a growing market for child themes of such frameworks as [Genesis](#), [StartBox](#) and [Hybrid](#).

What To Include In Your Theme

Depending on the type of theme you're creating, the expectations of consumers will vary. But you should consider certain features and functionality for the majority of your themes. You needn't implement all of these, but at least consider whether they would add value to your theme.

INTERNATIONALIZE THE THEME FOR OTHER LANGUAGES

Internationalizing your theme enables users to translate the text displayed by your theme, and implementing it is fairly straightforward. This one is a must-have. I was amazed at how many non-English-speaking users downloaded my themes. Looking back, I should have internationalized my themes from the beginning, knowing that millions of people all over the world use WordPress. You would be silly not to internationalize your theme. Look at the "[Translating WordPress](#)" section of the Codex and [this helpful tutorial](#) by AppThemes for more information.

SUPPORT WORDPRESS' CODING STANDARDS AND PRACTICES

Develop your themes in a way that supports WordPress' latest coding standards and practices. In doing so, you ensure that the theme is compatible with future versions of WordPress, and you'll avoid a flood of emails from customers who have run into conflicts. Also, avoid deprecated functions, which are functions that are “no longer supported and may be removed in future versions of WordPress.”

An easy way to check all of this is to install the [Theme-Check plugin](#). This great little plugin runs the same tests as those that WordPress.org runs on submitted themes.

```
REQUIRED: Could not find comment_form. See: comment\_form
<?php comment_form(); ?>

REQUIRED: get_settings found in the file functions.php. Deprecated since version 2.1. Use get_option
Line 225: <input name='<?php echo $value['id']; ?>' id='<?php echo $value['id']; ?>' value='<?php echo $value['id']; ?>' />
$valu
Line 237: <textarea name='<?php echo $value['id']; ?>' type='<?php echo $value['id']; ?>' />
Line 253: <option <?php if ( get_settings( $value['id'] ) == $option) { echo '
REQUIRED: functions.php. Themes should use add_theme_page() for adding admin pages.
Line 171: add_menu_page($themename, $themename, 'administrator', basename(__FILE__), 'index.php');

REQUIRED: License: is missing from your style.css header.
REQUIRED: License URI: is missing from your style.css header.
REQUIRED: .wp-caption css class is needed in your theme css.
REQUIRED: .wp-caption-text css class is needed in your theme css.
REQUIRED: .sticky css class is needed in your theme css.
REQUIRED: .gallery-caption css class is needed in your theme css.
REQUIRED: .bypostauthor css class is needed in your theme css.
RECOMMENDED: could not find the file readme.txt in the theme. Please see Theme Documentation
RECOMMENDED: Text domain problems in sidebar.php. You have not included a text domain!
Line 9: <h3 class='widget-title'><?php e( 'Archives' ); ?></h3>
RECOMMENDED: Text domain problems in searchform.php. You have not included a text domain!
```

The [Theme-Check plugin](#) has saved me many times from leaving out important details and using deprecated functions.

DOCUMENTATION AND READABLE CODE

Write thorough and helpful documentation for your themes. This will not only help users, but also cut down on the number of support requests you get from aggravated users. And trust me: the less support requests you get, the happier you will be. Document everything that's unique about your theme that WordPress users might be unfamiliar with, as well as any built-in features such as custom backgrounds and headers, menus, and post formats. Also provide instructions on how to update the theme and on the proper way to customize the code (in case a user wants to create a child theme).

```
// The height and width of your custom header.
// Add a filter to twentyeleven_header_image_width and twentyeleven_header_image_height
define( 'HEADER_IMAGE_WIDTH', apply_filters( 'twentyeleven_header_image_width', 940 ) );
define( 'HEADER_IMAGE_HEIGHT', apply_filters( 'twentyeleven_header_image_height', 125 ) );

// We'll be using post thumbnails for custom header images on posts and pages.
// We want them to be the size of the header image that we just defined.
// Larger images will be auto-cropped to fit, smaller ones will be ignored. See http://codex.wordpress.org/Post_Thumbnails
set_post_thumbnail_size( HEADER_IMAGE_WIDTH, HEADER_IMAGE_HEIGHT, true );

// Add Twenty Eleven's custom image sizes
add_image_size( 'large-feature', HEADER_IMAGE_WIDTH, HEADER_IMAGE_HEIGHT, true );
add_image_size( 'small-feature', 500, 300 ); // Used for featured posts if a post has a featured image.

// Turn on random header image rotation by default.
add_theme_support( 'custom-header', array( 'random-default' => true ) );

// Add a way for the custom header to be styled in the admin panel that controls
// custom headers. See twentyeleven_admin_header_style(), below.
add_custom_image_header( 'twentyeleven_header_style', 'twentyeleven_admin_header_style' );

// ... and thus ends the changeable header business.

// Default custom headers packaged with the theme. %s is a placeholder for the theme name.
register_default_headers( array(
    'wheel' => array(
        'url' => get_template_directory_uri() . '/images/headers/wheel.jpg',
        'thumbnail_url' => get_template_directory_uri() . '/images/headers/wheel_thumb.jpg',
        'thumbnail_width' => 500,
        'thumbnail_height' => 300,
        'description' => 'A black and white photograph of a wheel, with a dark background and a light, glowing rim. The wheel is centered and takes up most of the frame. The background is dark and textured, possibly a wall or a floor. The lighting is dramatic, highlighting the spokes and the rim of the wheel. The overall mood is mysterious and artistic.'
    )
)
```

The Twenty Eleven theme is a good example of a theme with well-documented code.

Another important aspect of documentation is to make the code easy to read and understand. Some advanced users will want to customize the code, so it should be commented in a way that helps them understand what you've done under the hood. For a good example of well-documented code, check out the `functions.php` file in the default Twenty Eleven theme.

CHILD THEMEABLE

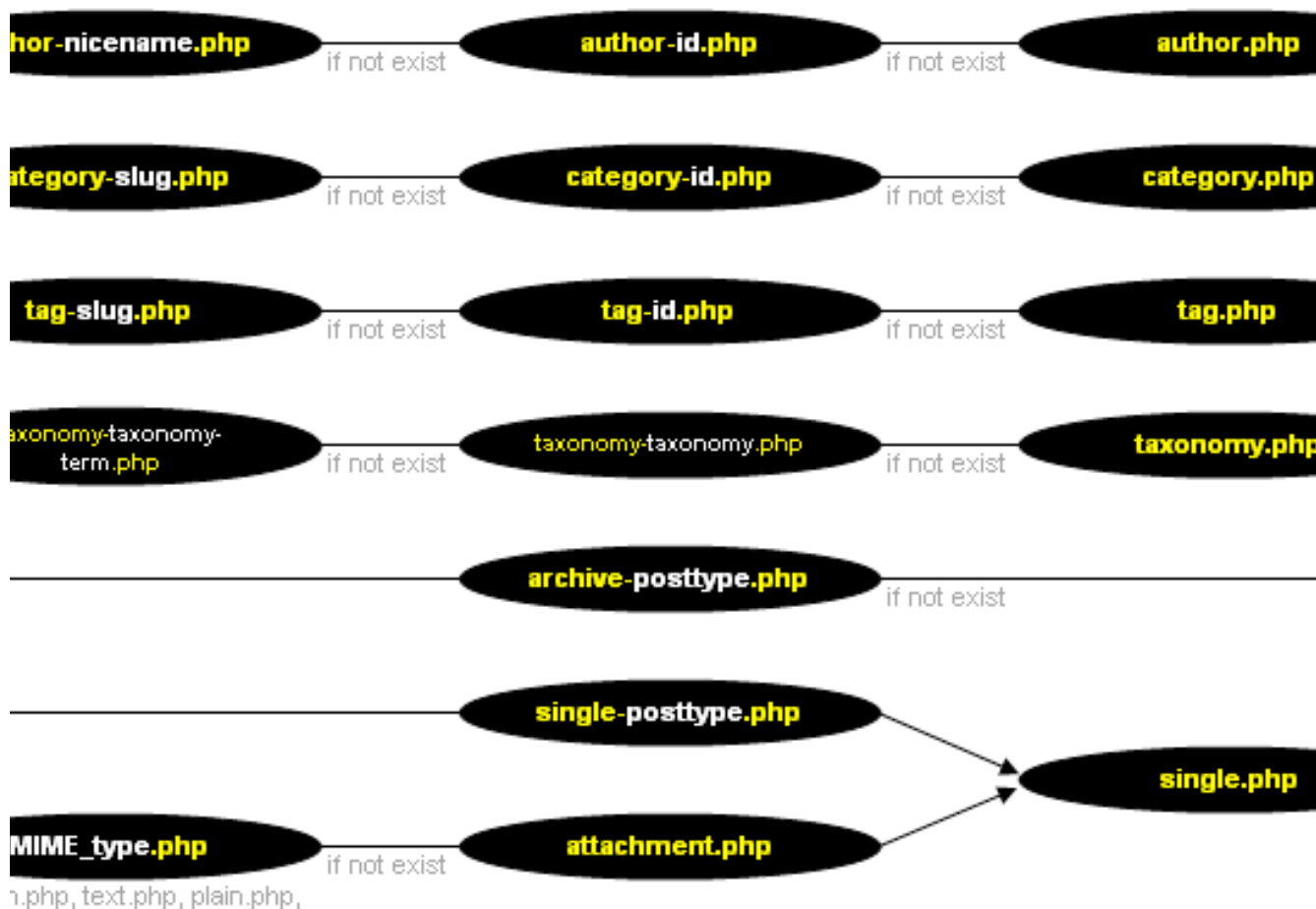
As noted, many users will want to customize the code. The trick is that, when you release an update, the developer has to avoid overwriting the files that they've customized. The solution is for them to make their customizations in a child theme. So, make sure to support this functionality by allowing child themes to be easily created.

If you don't want users to have to worry about including a particular script when creating a child theme, then use the

`get_template_directory_uri()`; function to reference the parent theme's folder. To allow the developer to overwrite this file, use `get_stylesheet_directory_uri()`; instead, which references the folder in the child theme, if one is being used.

PAGE TEMPLATES

Your theme should support the various page templates that a WordPress website can have. Because you don't know how each developer will use the theme, you have to prepare for all possibilities. This is where testing comes in. For a typical WordPress theme, you should **at the very least** support these templates: `page.php`, `archive.php`, `404.php`, `search.php`, `single.php`, `attachment.php` and, of course, `index.php`, which is the ultimate fallback. For a full list of templates, check out the "[Template Hierarchy](#)" section of the WordPress Codex.



WordPress’ “[Template Hierarchy](#)” is a great reference to have on hand.

You can also provide users with custom page templates. The two most common that I include with my themes is one with a widgetized sidebar (the default `page.php`) and one with a full-width page. You’ll likely be able to come up with other templates that users would benefit from once you’ve designed the theme.

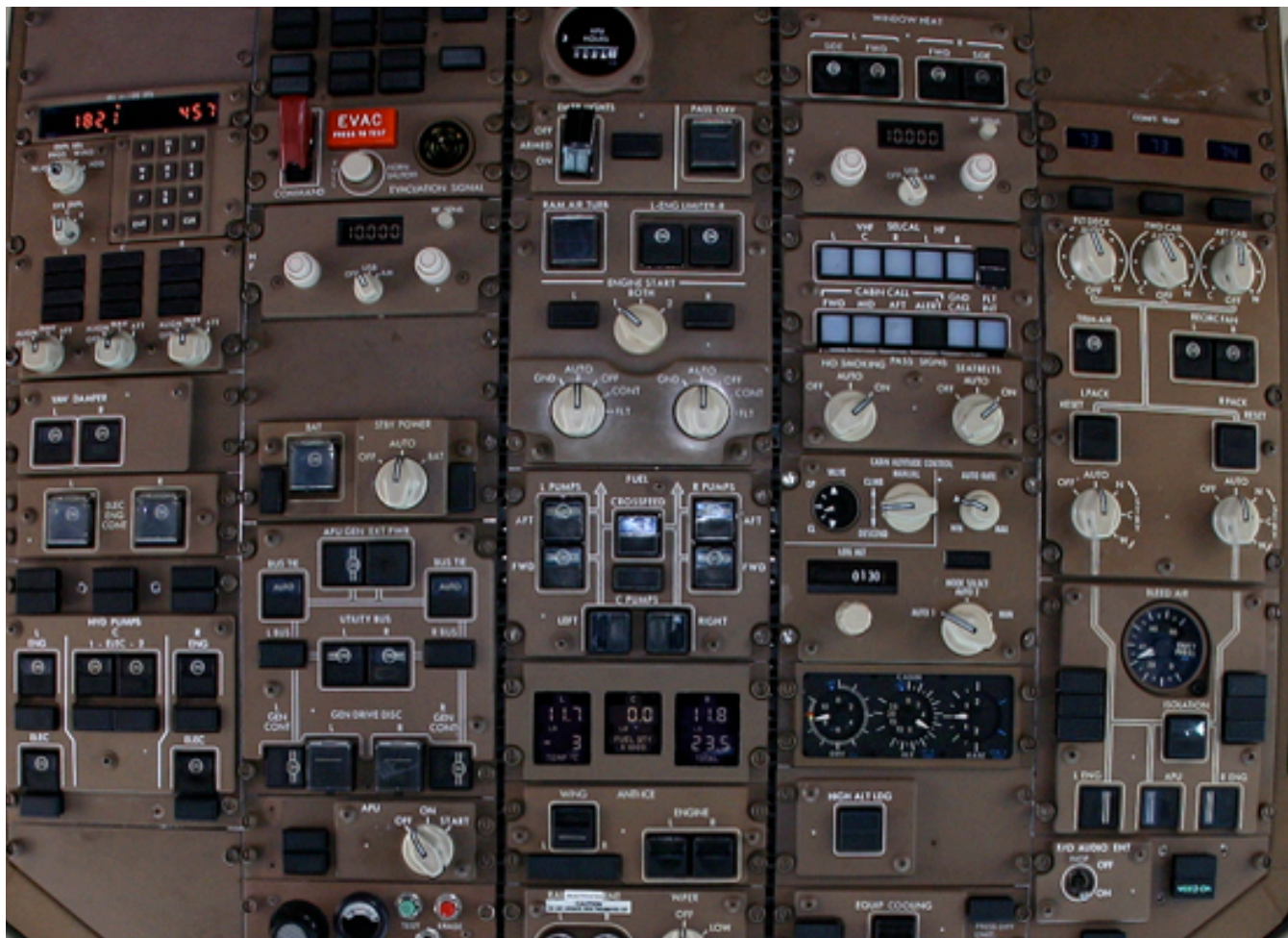
Some theme developers use custom fields for this functionality, instead of page templates. This seems counterintuitive because the functionality is built into WordPress and is so simple to use. Creating your own page template is as easy as creating a new PHP file in the theme's folder and adding the following PHP comment at the top (replacing "Full Width" with the template name of your choice):

```
<?php
/*
Template Name: Full Width
*/
?>
```

Of course, the code that follows the line above is up to you and will determine what the template does.

A NOTE ON THEME OPTIONS

There seems to be a misunderstanding about what users of premium themes expect. The common belief is that they expect an options panel that looks like the control panel of a Boeing 747, where they can tweak the smallest detail of the theme. Sure, users want to be able to control certain aspects of their website, but simplicity and ease of use trump bloat and complexity.



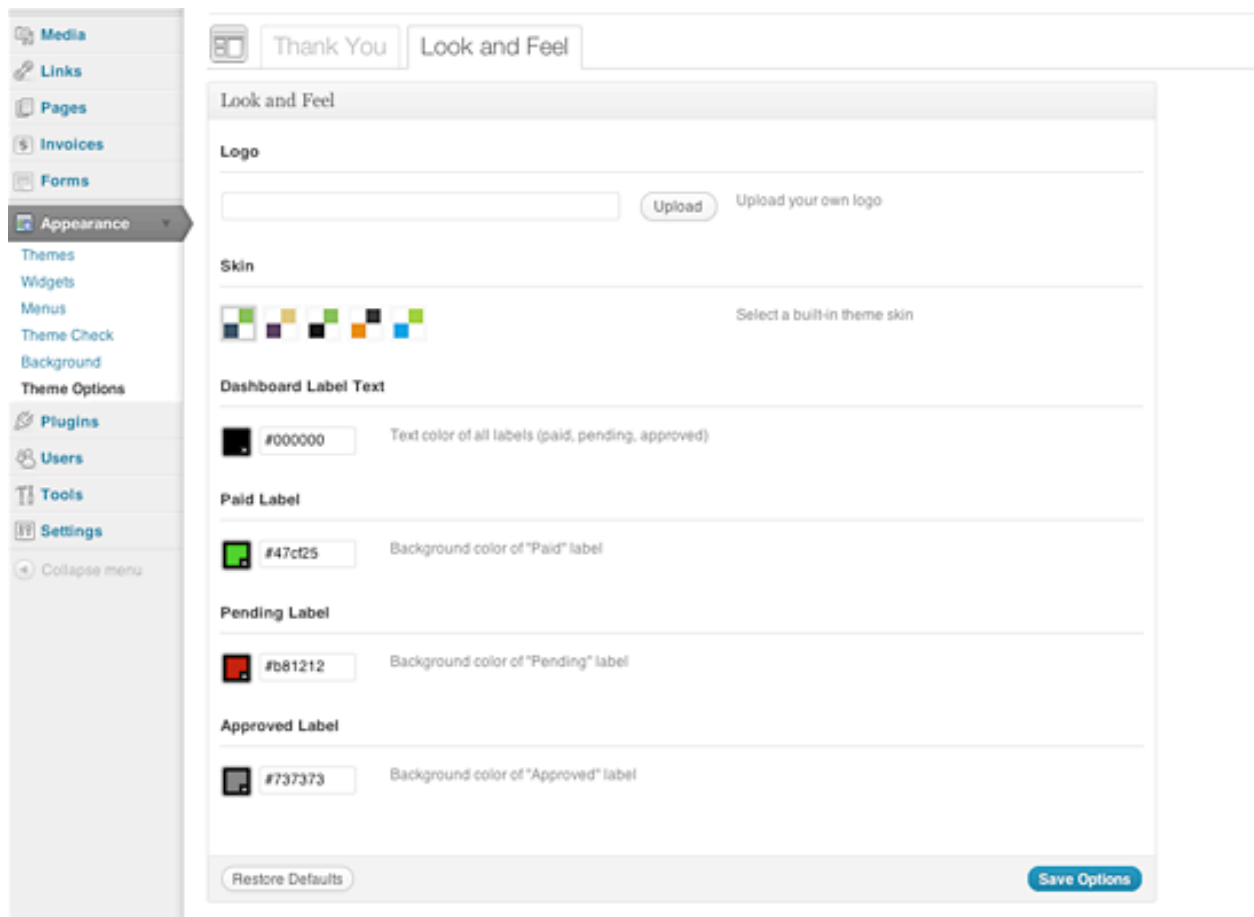
Your options panel shouldn't be this complicated. (Image: [Fly For Fun](#))

When deciding whether to include an option in your theme, consider whether it's really necessary and whether that functionality is already built into WordPress. The more options you add, the more complicated the code becomes and the steeper the learning curve for users. I keep the options for my themes to the bare essentials, and a goal of mine is to **create a theme for which an options panel isn't even necessary**. I challenge you to do the same.

You can build your theme's options on top of any one of the several great options frameworks. These are the ones I've come across: [Options Framework](#), [UpThemes-Framework](#), [OptionTree](#).

APPEARANCE OPTIONS

One reason to include an options panel is to enable the user to tweak the appearance of the theme without having to mess with the code. The option demanded by most users is surely to be able to upload a logo. Adding a logo is the easiest way for a user to personalize their theme. I enable it in all of my themes.



A snapshot of the options page in my latest commercial theme (based on the [Options Framework](#)).

Most theme buyers aren't designers. They might not have an eye for color or be able to make informed design decisions. So, in addition to providing options to customize the theme's main elements (like the color of text, the color of the call-to-action button, etc.), I include a selection of "skins," which are basically just pre-defined palettes that a user can select from. This way, if the user doesn't have an eye for color, they at least have options and aren't restricted to one scheme. I usually provide several styles that cater to a variety of audiences.

SOCIAL NETWORK OPTIONS

Most individuals and businesses have some type of presence on social networks, whether on Twitter, Facebook, YouTube or whatever the next big thing is. Because the design and placement of these social-network links vary from theme to theme, you can provide an option that allows users to customize the links.

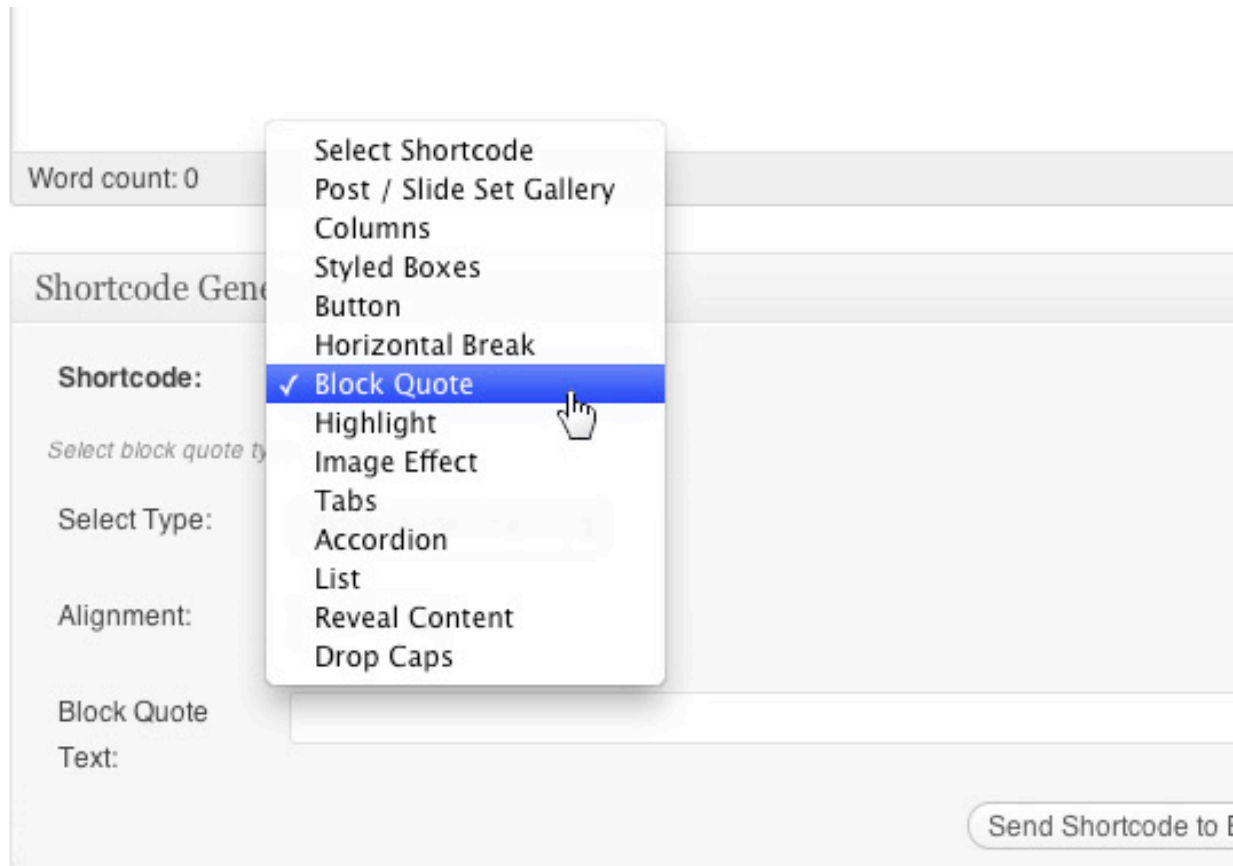
Aside (and a little plug): I used to recommend including social-network options in the theme's panel, but having given it more thought, I now feel it's better suited to a plugin. New social networks pop up every day, and anticipating which ones your theme's users will be on is hard. You will never be able to cover all bases, which is why I recently built a plugin that I'll soon be supporting in all of my themes, and I suggest you do the same if you plan on including this feature. The plugin adds a new settings page where the user can create a list of social-network links. Users can select from the range of icons built into the plugin or built into the theme (if present) or upload their own. If this interests you, the plugin is called [Social Bartender](#) and is in the WordPress repository.

ADVERTISING OPTIONS

You could also enable users to add advertisements, either through a widget or through an option that positions the ads in certain spots (like following the top blog post). Many people want to monetize their website and so advertising options would be important to them. Being able to select the locations of ads to suit the design is a selling point.

What To Leave Out

Almost as important as what to include in the theme is what to leave out. Many themes have options and functionality that are better done as plugins or that are already built into WordPress. Use the functionality that WordPress already supports, such as [custom backgrounds](#), [headers](#), [post thumbnails](#) and [post formats](#). This is easier to implement because WordPress does all of the heavy lifting, and many users are already familiar with it. That being said, if your theme doesn't need this functionality, then don't include it in the first place!



Shortcodes should not replace standard HTML tags. Many of the shortcodes shown above are unnecessary.

SHORTCODES

Shortcodes are great for executing a set of functions, but they're unnecessary simply to embed a link or add a class to an element. Use standard HTML tags for this. For example, don't create a `[quote]` shortcode when the HTML `<blockquote>` tag does a perfectly good job.

I've seen themes that have shortcodes for quotes, citations and headers but no support for the same styling with HTML tags. This is a big no-no. Many users will switch from theme to theme and will already have content on their website when they activate yours. HTML tags will stay the same, but shortcodes vary from theme to theme. Don't force the user to go back through all of their content just to add your custom shortcodes. Use shortcodes only to execute functions, not to apply styling. There may be a few exceptions, such as to wrap a message in complicated HTML, but if you're simply adding a class, then adding it to the "Format" menu in the post editor's kitchen sink makes more sense.

A great [tutorial was recently published](#) by Luke McDonald that details how to add your own styles to the drop-down menu in the visual editor, giving you one more reason not to use shortcodes to style elements.

PLUGIN TERRITORY

Don't include options for things that should really be added with existing plugins; for example, Google Analytics and favicons. I hear someone in the back asking, "Why not include such things?" Well, person in the back, what if the user decides to switch themes, even to another of yours? They would lose all of that information and have to figure out how to get it back. The option is unnecessary, would make the code overly complicated, and would cause trouble when the user switches themes. Include only options that alter functionality that is unique to your theme; otherwise it's better suited to a plugin.

Developing WordPress Locally With MAMP

Ryan Olson

Local development refers to the process of building a website or Web application from the comfort of a virtual server, and not needing to be connected to the Internet in order to run PHP and MySQL or even to test a contact form. One of the most annoying parts of development, at least for me, is the constant cycle of edit, save, upload and refresh, which, depending on bandwidth and traffic, can turn a menial task into a nightmare.

With application platforms such as WordPress, which require a server back end to work, you would normally be constrained to develop on a live server, with the headaches that go along with that. [MAMP](#) and its Windows counterpart, [WAMP](#), are tools that allow you to locally develop applications that require a server on the back end.

The Local Server

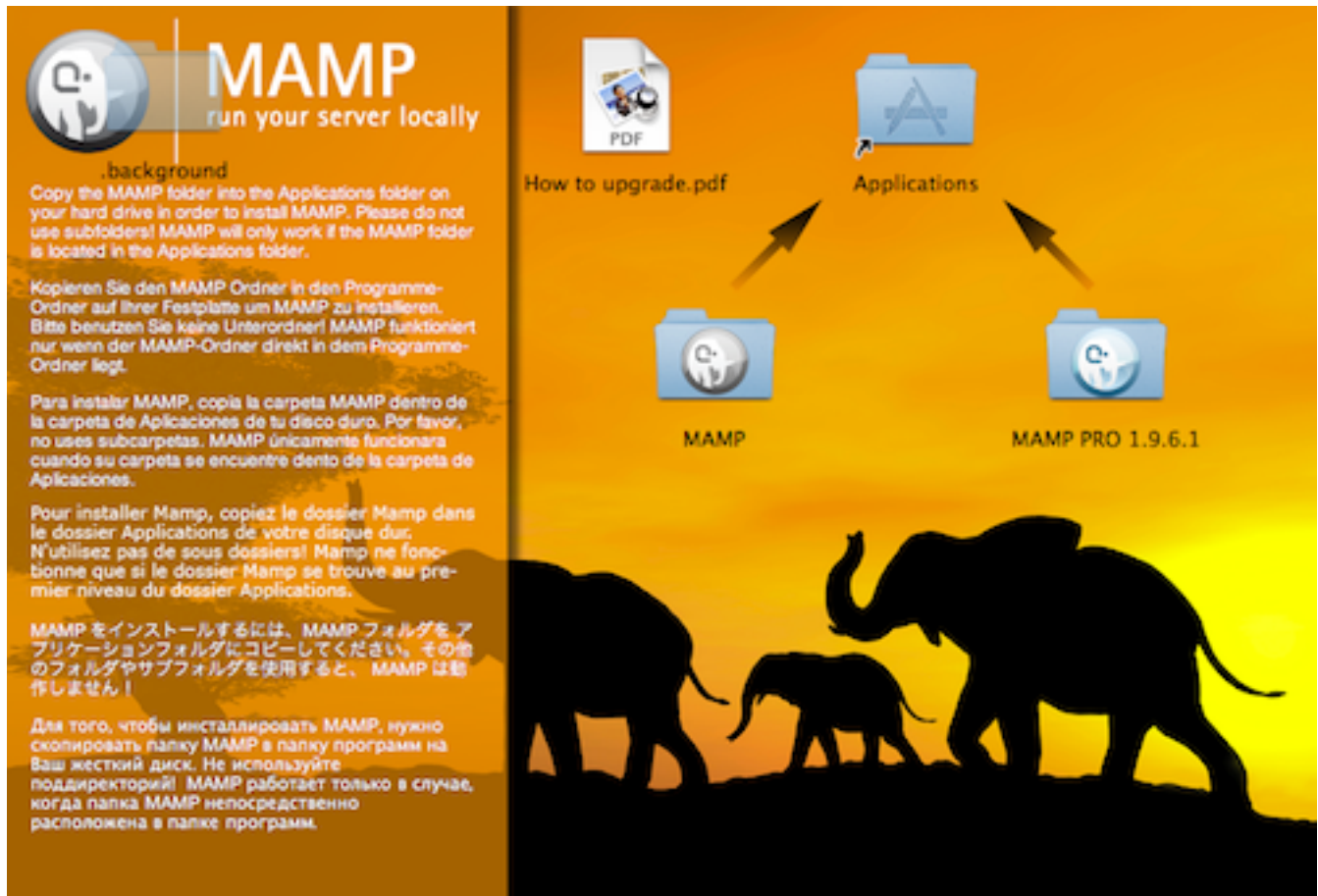
MAMP, which stands for Macintosh, Apache, MySQL and PHP, is an application that allows you to install a local server-type environment in order to construct websites that would normally require you be on a live server somewhere.

Ever opened a contact form in a browser from your desktop and wondered why it doesn't work? The server-side components cannot operate without (in this case) the PHP back end, and this is where MAMP comes in. By installing this application, we can have a virtual server locally as our development sandbox. It is worth noting, from a portability standpoint, that this component can be run only from your desktop environment and [cannot](#) be installed on a USB drive. With that all settled, let's get to it.

In order to be able to work with MAMP, we must first obtain it. So, head over to the [project page](#) and download the disc image. Double-click to begin the installation, and you will be presented with a choice:

Both MAMP and MAMP Pro come in the same download. You need to install only one, and for most scenarios, MAMP is more than adequate. The pro version costs \$59.00 USD and offers more options, and you can compare the two versions for yourself.

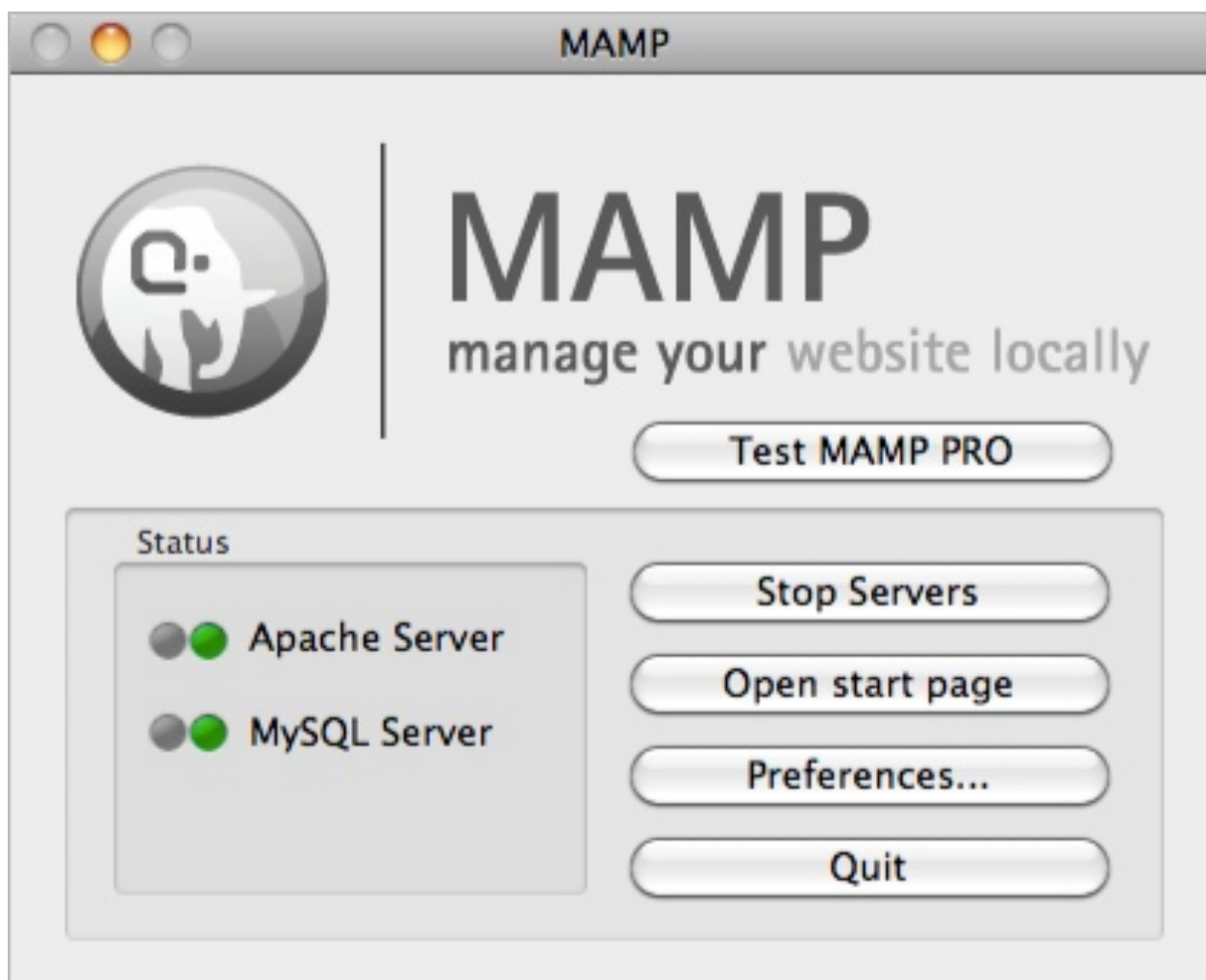
Drag the MAMP folder onto the "Application" shortcut, and the installation will be underway. Once it's completed, feel free to eject the disc image. Open up your "Applications" folder, and locate the new MAMP directory. Inside you'll find *MAMP.app*, so —you guessed it— open it up. The program should start right away and open up your default browser, pointing to the start page. Congratulations, you now have a local server!



MAMP and MAMP Pro are on the installation disc image.

Preferences

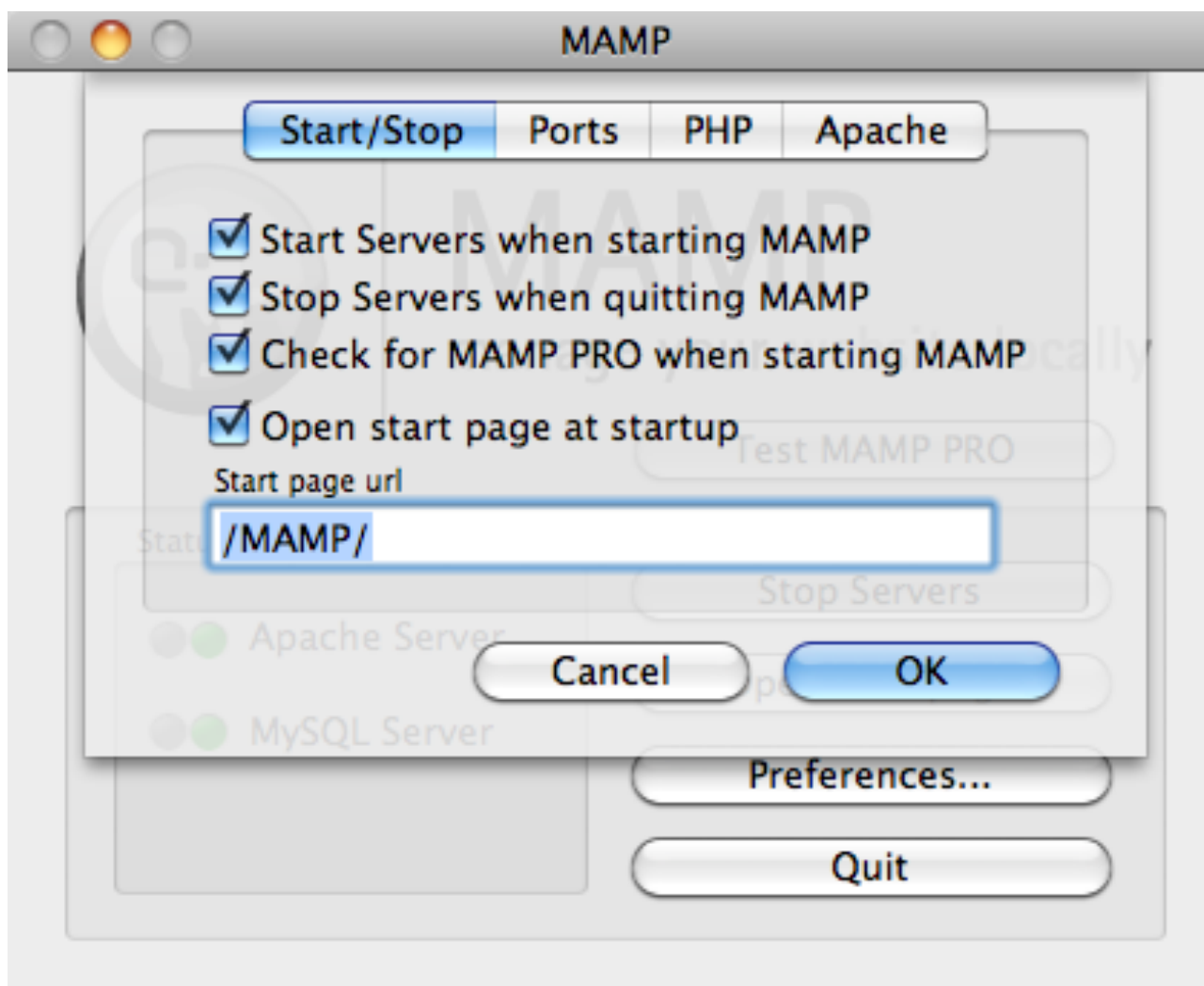
From the main MAMP app screen, you will notice a “Preferences” button. Feel free to click on it to view the few options available.



The MAMP app.

1. START/STOP

From here, you have the option to tell MAMP when to start and stop the servers. If you choose to not start the servers automatically, then you will need to explicitly tell them to run each time you open the app. You may also set your home page, which defaults to the MAMP start screen, giving you quick access to phpMyAdmin; but you may set it to something like a WordPress directory.

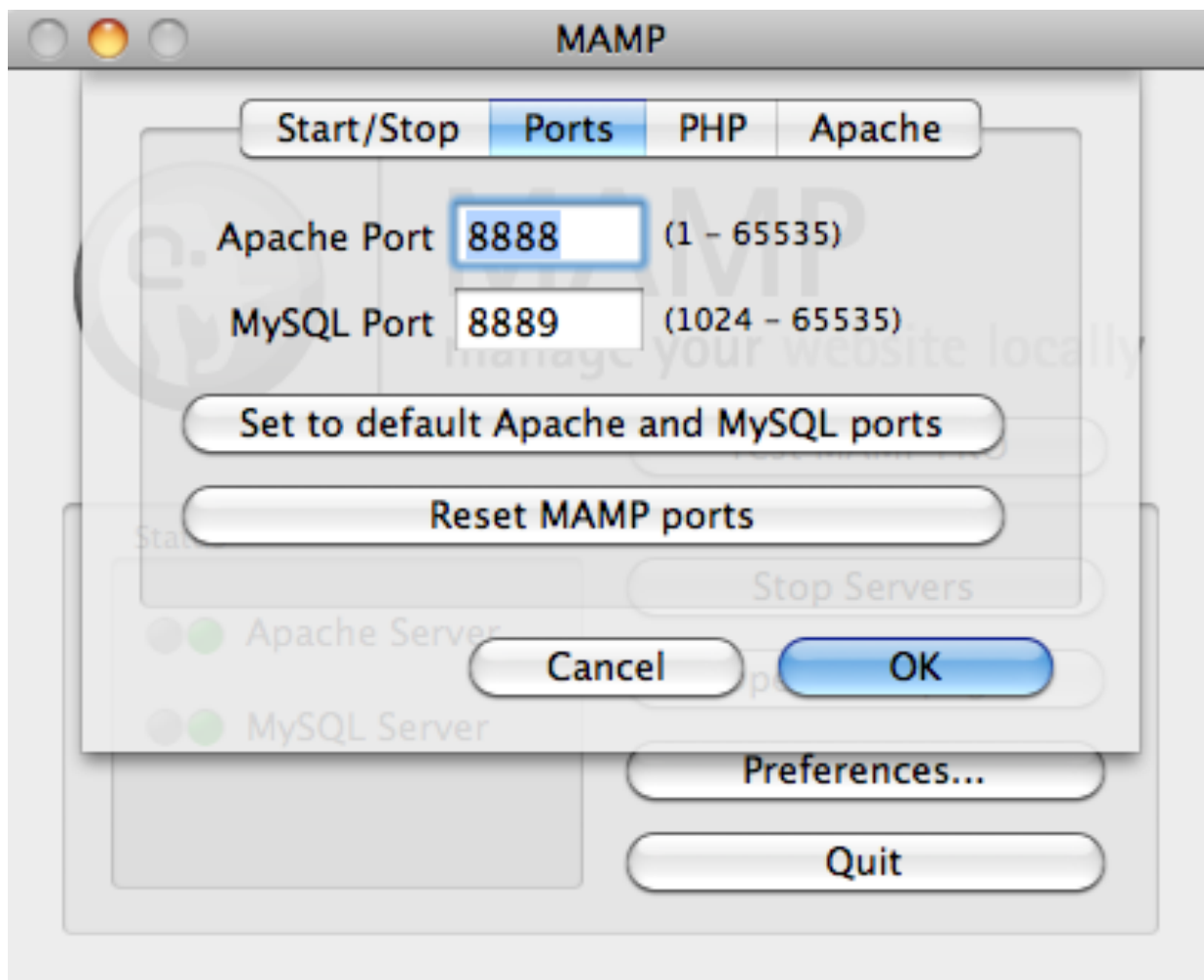


Configuring the server.

2. PORTS

In the “Ports” tab, the default Apache port will usually be 8888, and the default MySQL port will be 8889. I, for one, do not change these because they do not interfere with any of my other settings and do not require me to enter my password every time I start and stop the servers.

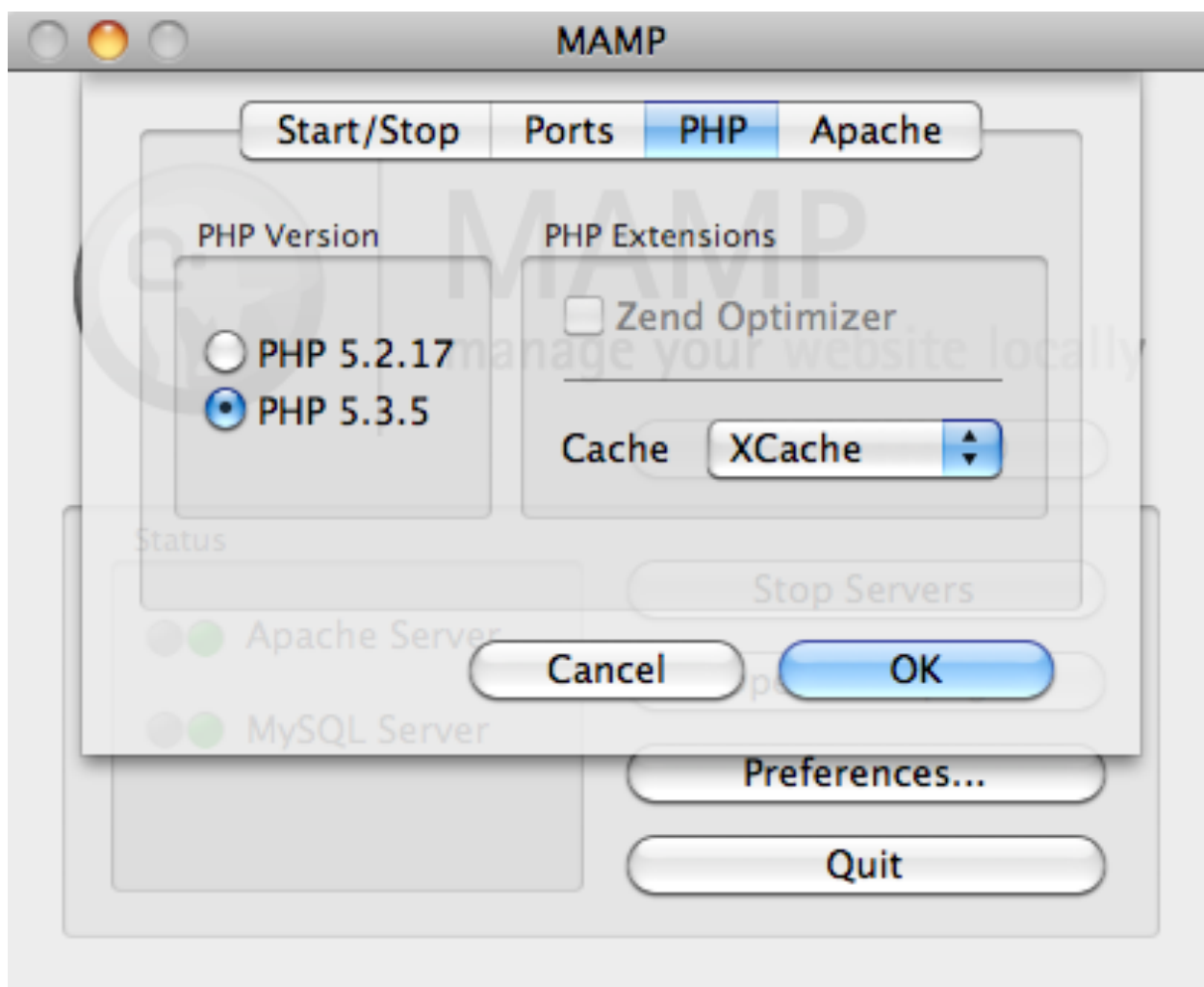
You must include the port number in your URL this way; so, it would be `localhost:8888/`. To avoid this, you could change the ports to what general Web servers operate on: ports 80 and 3306. This will allow your URL to simply be `localhost/`; but you will most likely need to enter your password when switching the servers on and off. Another factor to consider is whether you are installing WordPress “multisite”; if you are, then you are required to set the ports to the default Apache and SQL ports of 80 and 3306, respectively.



Setting up MAMP ports.

3. PHP

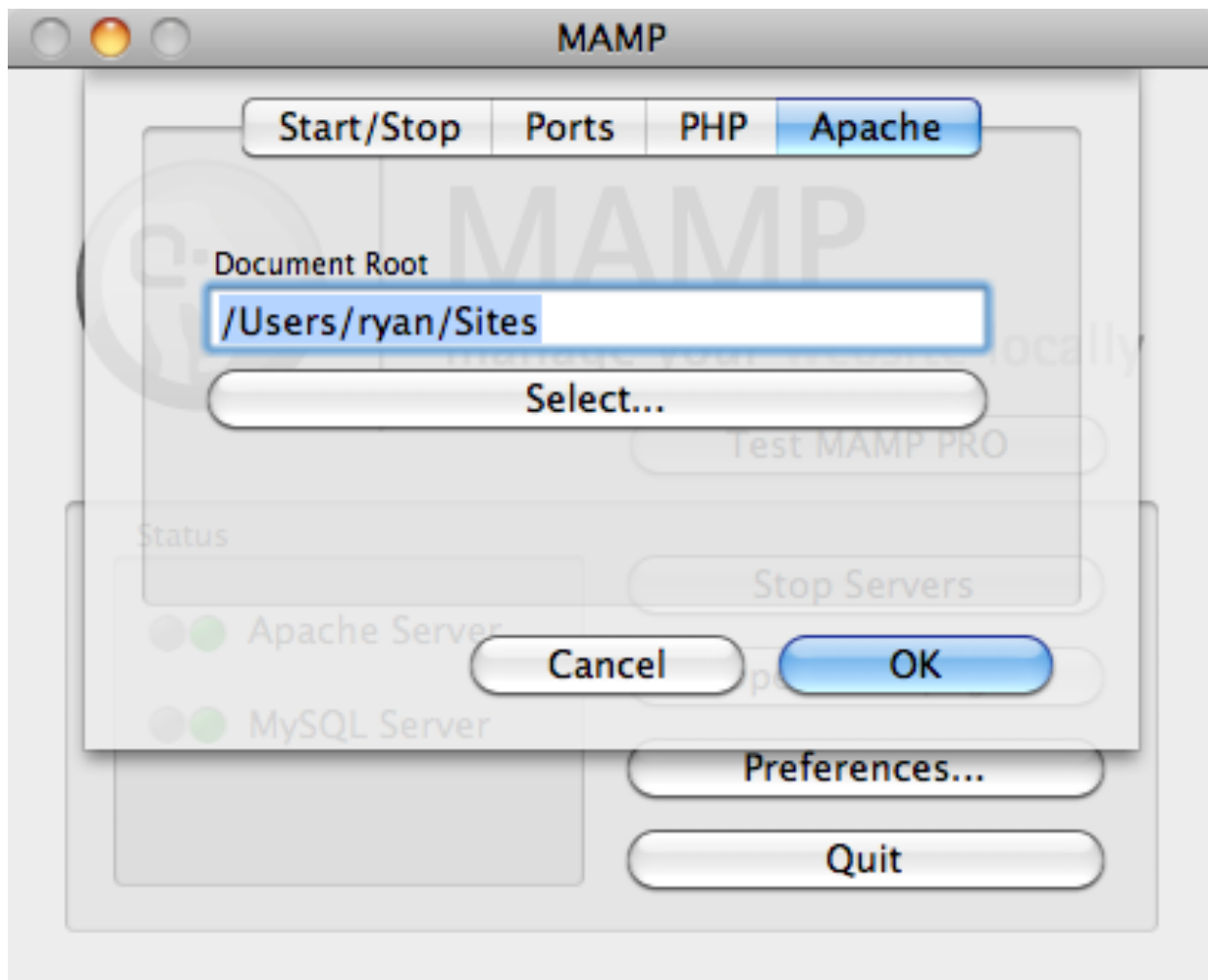
The “PHP” tab allows you to choose which version of PHP to run in the set-up. It will default to 5.3, and I do not change this because most applications I run either require PHP 5.3 or do not care. Just know that this option is available if you need it to run something such as legacy software.



Setting up the MAMP PHP version.

4. APACHE

The “Apache” tab is one that I like to mess with, to change the document root directory. The root is where all of your websites and directories will be stored and accessed by MAMP, and it defaults to `/Applications/MAMP/htdocs`, which I find annoying to get to. So, I change mine to my `sites` folder. From the MAMP app window, click on “Preferences,” then on “Apache.” You can click “Select” and then set the installation to use the location of your choice for your websites. Again, I set mine to the `sites` folder for easier access.

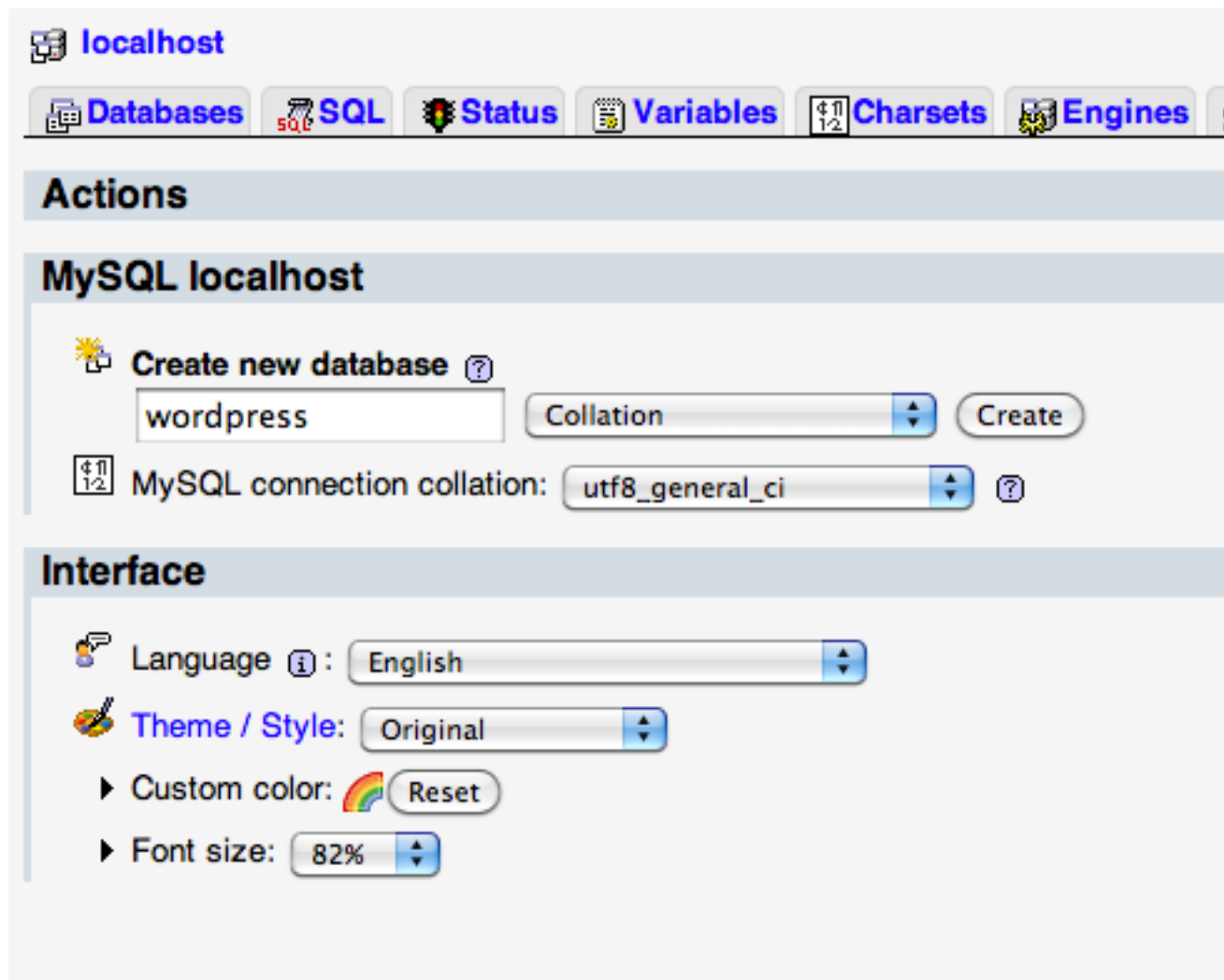


Setting up MAMP Apache.

Installing WordPress

Now it is time to install WordPress. Head to the WordPress website and download the latest version, 3.2 as of this writing. Unzip the folder, and then simply drag it to your `sites` folder, (or wherever you chose to set the document root for MAMP). WordPress requires PHP and MySQL to operate, which is why we needed MAMP to develop locally; so, we now need need to make a database. Fear not: it is simple!

Open the MAMP start page—you can access it via the button in the main app—and click on “phpMyAdmin” in the top menu. Creating a new database is as simple as typing a name in the field and hitting the “Create” button. You can see below that I am creating a new database aptly named “wordpress.” Once that’s done, feel free to close phpMyAdmin, and navigate to the WordPress directory in your document root.



Simply type a name for the database, and hit “Create.”

BASIC CONFIGURATION

Find the file named `wp-config-sample.php`, and open it in your favorite text editor. We have to configure a few settings. The default values for MAMP installations make this really easy to fill out, so follow the table below to see what to type where:

Variable	Value
DB_NAME	wordpress
DB_USER	root
DB_PASSWORD	root

```
16
17 // ** MySQL settings - You can get this info from your web host ** //
18 /** The name of the database for WordPress */
19 define('DB_NAME', 'database_name_here');
20
21 /** MySQL database username */
22 define('DB_USER', 'username_here');
23
24 /** MySQL database password */
25 define('DB_PASSWORD', 'password_here');
26
27 /** MySQL hostname */
28 define('DB_HOST', 'localhost');
```

Change the values of the variables to match the table above.

You should not need to alter anything else in this file, at least for now. You could add in the unique keys and salts, but I recommend doing that once you move the website into production.

Save and close wp-config-sample.php. We're nearly done. Rename this file to wp-config.php—removing the -sample—and we are ready to complete the installation. You should now be able to point your browser to `http://localhost:8888/wordpress` and see the WordPress installation screen. Enter in your basic data and install the app. You are now ready to log into the admin section and get going!



Welcome

Welcome to the famous five minute WordPress installation process! You may want to browse the [ReadMe documentation](#) at your leisure. Otherwise, just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.

Information needed

Please provide the following information. Don't worry, you can always change these settings later.

Site Title

Local Dev

Username

admin

Username can have only alphanumeric characters, spaces, underscores, hyphens, periods and the @ symbol.

Password, twice

A password will be automatically generated for you if you leave this blank.

•••••

•••••

Weak

Hint: The password should be at least seven characters long. To make it stronger, use upper and lower case letters, numbers and symbols like ! " ? \$ % ^ &).

Your E-mail

ryan@thatryan.com

Double-check your email address before continuing.

☐ Allow my site to appear in search engines like Google and Technorati.

Install WordPress

Enter your information... but choose a stronger password.

Permalinks

Always follow WordPress' permalink structure. In order for you to get these “[pretty URLs](#),” Apache will need `mod_rewrite` to update your `.htaccess` file, so let's make sure that is set up.

The file we have to edit is `httpd.conf`, and you can find it in Applications → MAMP → conf → apache → /. Open this file, and search for a line like this:

```
LoadModule rewrite_module modules/mod_rewrite.so
```

Note that a hash (`#`) may or may not be in front of it. The hash indicates a comment, and if you see it, you must remove it to allow the `mod_rewrite` module to load. If the line is not commented out, then congratulations: you are already done! Close the file, and permalinks should now work in your local installation.

THE FINAL COUNTDOWN

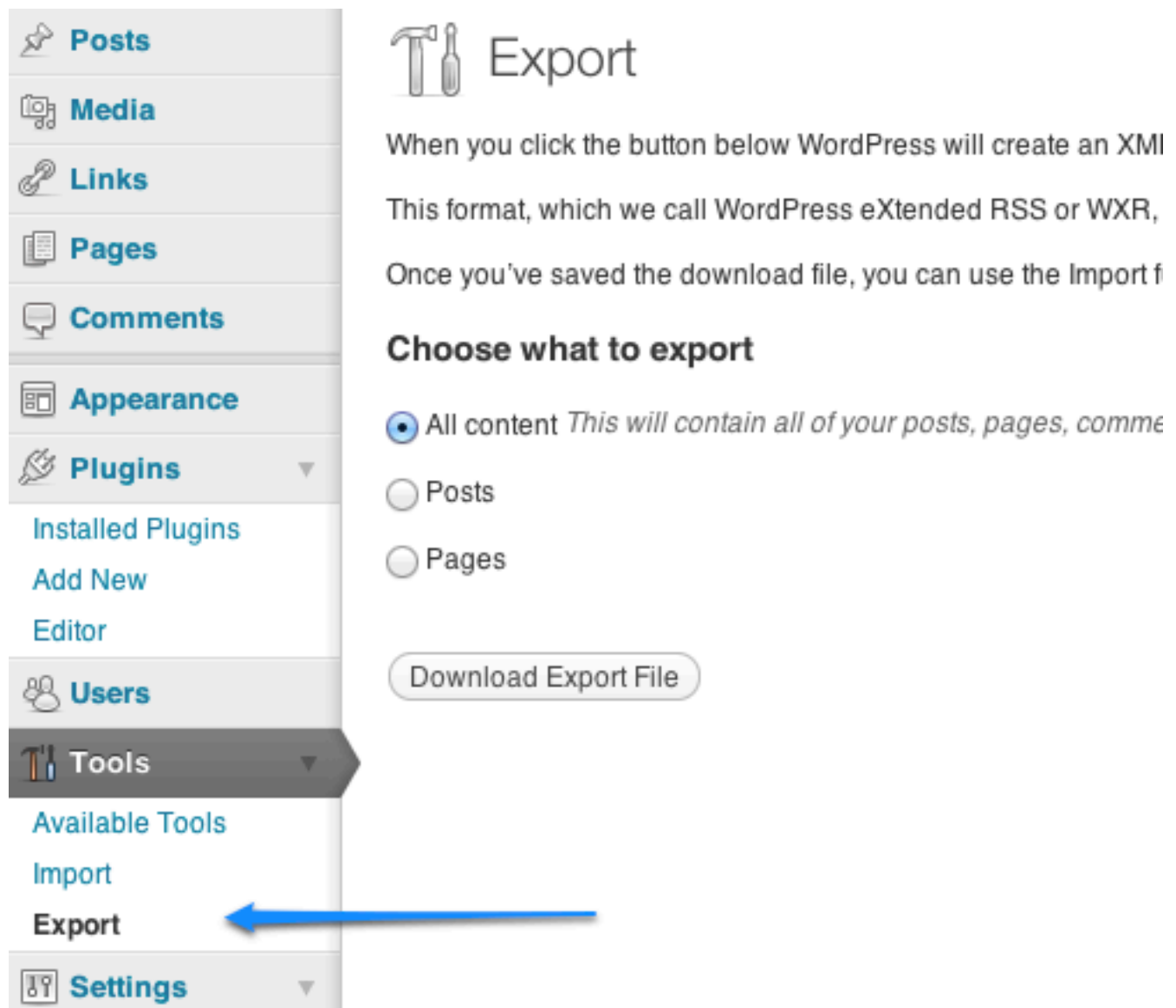
By now, a local server set up with WordPress should be installed and running. The remaining steps are both short and crucial to sharing your creation with the Internet. All that remains is to transfer your local accomplishments to a global environment by moving both our WordPress files and our content. So, let's finish this up!

Going Live

The time has finally arrived. So, how do you bring your WordPress creation to the live server? Well, we have two options.

JUST GRAB THE CONTENT

A sometimes simpler way, with only a few steps, is to just grab all of your content. This is easiest if WordPress is already installed and you just need to import your theme and content. To do this, head to the admin dashboard, to the “Tools” section in the sidebar. Click on “Export,” and choose “All content.” This will export a file that you can then import into your new installation.



The screenshot shows the WordPress admin dashboard. On the left is a sidebar menu with the following items: Posts, Media, Links, Pages, Comments, Appearance, Plugins (with a dropdown arrow), Users, Tools (highlighted with a blue arrow pointing to its 'Export' sub-item), and Settings. The main content area is titled 'Export' with a hammer and wrench icon. Below the title, it says: 'When you click the button below WordPress will create an XML file. This format, which we call WordPress eXtended RSS or WXR, is the standard format for importing content into WordPress. Once you've saved the download file, you can use the Import tool to import it into your new installation.' Under the heading 'Choose what to export', there are three radio button options: 'All content' (selected), 'Posts', and 'Pages'. The 'All content' option has a descriptive text: 'This will contain all of your posts, pages, comments, and media files (uploads)'. At the bottom of the main content area is a button labeled 'Download Export File'.

Exporting WordPress content.

You can now upload your WordPress theme files to the live location. Head to the “Tools” section of the dashboard again, and choose “Import.” Simply point to the file that you just exported, and bring in your content.

BRINGING IN EVERYTHING

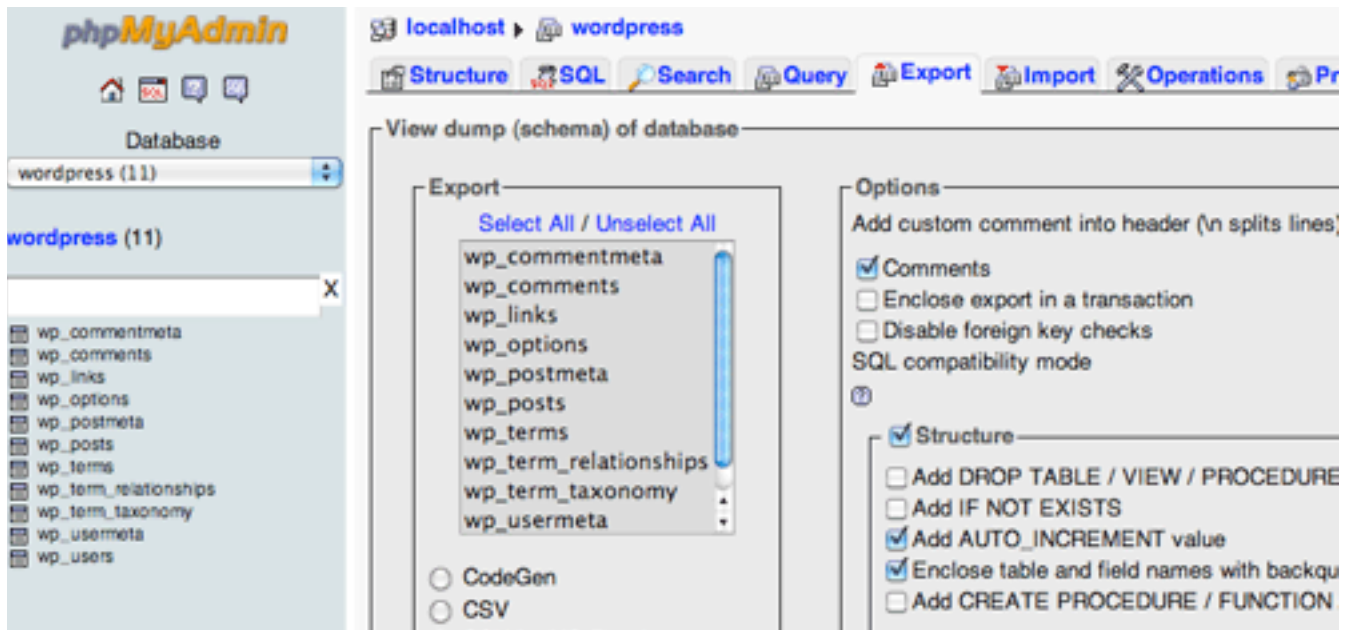
I use this method if I have done everything locally from the ground up. I’ll upload my entire local WordPress directory (in this case, `http://localhost:8888/wordpress`) to the live server and then grab the database file and transfer that from local to live as well.

Because you could certainly build nearly the entire website in your development environment, bear in mind that WordPress uses absolute paths for URLs. So, every image and link will be prepended with `http://localhost:8888/` (depending on your set-up). We need a way to alter this to fit the live website. We have a few options.

1. EXPORT, SEARCH AND REPLACE

Using this method, we export our local database as a text file and run a “Find and replace” on the text to replace all occurrences of the localhost URL with the production URL.

Begin by opening phpMyAdmin and clicking on your WordPress database on the left. Click on the “Export” tab in the top menu, and be sure to choose “Select all” when choosing which tables to export. At the bottom, check the box to “Save as file,” and then hit “Go.” Open the resulting file in your favorite text editor, and simply run a “Find and replace” to replace all instances of `http://localhost:8888/wordpress` with `http://www.YOUR_SITE_URL.com`.



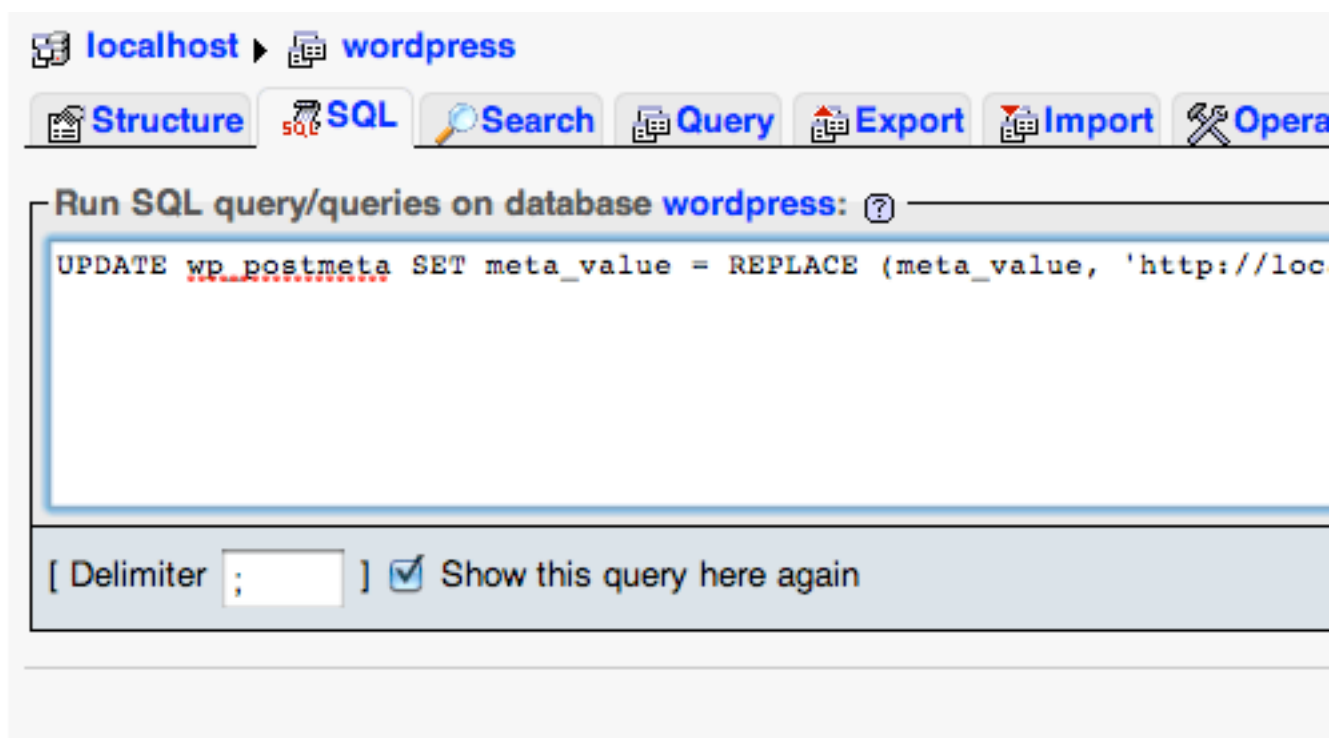
Exporting the WordPress database (click image for full-size view).

Save the edited file, and visit phpMyAdmin on your live server. Again, click on your WordPress database, and this time choose the “Import” option from the top menu, and browse for your newly edited file.

Once it successfully imports, upload your WordPress directory to the live server. If WordPress is already installed, simply upload your theme, any plugins you have installed locally and the contents of your wp-content/uploads folder; or else, upload the entire local directory to your live website’s root. Once that’s uploaded, be sure you can log into wp-admin, and browse around to make sure everything made it in. Update your permalink’s structure to something friendlier, and you are off!

2. USING SQL QUERIES

A second way to alter URL paths is to first bring everything into the live server version, and then use a few SQL queries to find and replace the necessary strings. Open phpMyAdmin on your local server, and export the database, again making sure to select all tables and to save it as a file. Go to your live server, and import the .sql file that you just saved. In the top menu, click on the tab for “SQL,” whereupon you will see a text area. You will need to enter some query syntax; be sure to replace the URLs in these code fragments with the ones that pertain to your set-up — namely, the localhost’s path and the URL of your new live website.



Running SQL queries to update the URL paths (click image for full-size view).

Replacing WordPress' base URL path:

```
UPDATE wp_options SET option_value = replace(option_value, 'http://localhost:8888/wordpress_', '') WHERE option_name = 'home' OR option_name = 'siteurl';
```

Update the GUID that controls WordPress' translating paths and post locations:

```
UPDATE wp_posts SET guid = REPLACE (guid, 'http://localhost:8888/wordpress_', '');
```

Update the URL paths in the content:

```
UPDATE wp_posts SET post_content = REPLACE (post_content, 'http://localhost:8888/wordpress_', '');
```

Update the URLs in the meta data of posts, such as attachments:

```
UPDATE wp_postmeta SET meta_value = REPLACE (meta_value, 'http://localhost:8888/wordpress_', '');
```

Final Thoughts

We have managed to install MAMP to set up a local server sandbox to develop in, and we've configured and installed a WordPress platform to develop in, saving the need for purely online development tactics.

I hope this has given you some insight into setting up a local environment to work with WordPress. Keep in mind that this is just scratching the surface; WordPress is versatile. Now that we have this faster new way to develop, the next time we'll get into some custom WordPress configurations.

HELPFUL LINKS

You may be interested in these related resources:

- [“MAMP vs. MAMP Pro”](#)
A chart comparing the two versions of MAMP.
- [MAMP Documentation](#)
- [“Installing WordPress”](#)
The walkthrough to install WordPress.
- [“13 Useful WordPress SQL Queries You Wish You Knew Earlier”](#)
A few SQL queries to aid with your WordPress development.
- [“WordPress MultiSite with Subdomains on MAMP”](#)
3-step tutorial on setting up subdomains with WordPress on MAMP.

The Developer's Guide To Conflict-Free JavaScript And CSS In WordPress

Peter Wilson

Imagine you're playing the latest hash-tag game on Twitter when you see this friendly tweet:

You might want to check your #WP site. It includes two copies of jQuery. Nothing's broken, but loading time will be slower.

You check your source code, and sure enough you see this:

```
<script src="/wp-includes/js/jquery/jquery.js?ver=1.6.1" type="text/
javascript"></script>
<script src="/wp-content/plugins/some-plugin/jquery.js"></script>
```

WHAT WENT WRONG?

The first copy of jQuery is included the WordPress way, while some-plugin includes jQuery as you would on a static HTML page.

A number of JavaScript frameworks are included in WordPress by default, including:

- Scriptaculous,
- jQuery (running in [noConflict mode](#)),
- the jQuery UI core and selected widgets,
- Prototype.

A [complete list can be found in the Codex](#). On the same page are instructions for using [jQuery in noConflict mode](#).

AVOIDING THE PROBLEM

Registering a file alone doesn't do anything to the output of your HTML; it only adds the file to WordPress's list of known scripts. As you'll see in the next section, we register files early on in a theme or plugin where we can keep track of versioning information.

Registering a file requires more complex code than enqueueing the files; so, quickly parsing the file is harder when you're reviewing your code. Enqueueing the file is far simpler, and you can easily parse how the HTML is being affected.

For these techniques to work, the theme's `header.php` file must include the line `<?php wp_head(); ?>` just before the `</head>` tag, and the `footer.php` file must include the line `<?php wp_footer(); ?>` just before the `</body>` tag.

Registering JavaScript

Before registering your JavaScript, you'll need to decide on a few additional items:

- the file's handle (i.e. the name by which WordPress will know the file);
- other scripts that the file depends on (jQuery, for example);
- the version number (optional);
- where the file will appear in the HTML (the header or footer).

This article refers to building a theme, but the tips apply equally to building a plugin.

EXAMPLES

We'll use two JavaScript files to illustrate the power of the functions:

The first is *base.js*, which is a toolkit of functions used in our example website.

```
function makeRed(selector){
  var $ = jQuery; //enable $ alias within this scope
  $(function(){
    $(selector).css('color','red');
  });
}
```

The `base.js` file relies on jQuery, so jQuery can be considered a dependency.

This is the first version of the file, version 1.0.0, and there is no reason to run this file in the HTML header.

The second file, `custom.js`, is used to add the JavaScript goodness to our website.

```
makeRed( '*' );
```

This `custom.js` file calls a function in `base.js`, so `base.js` is a dependency.

Like *base.js*, *custom.js* is version 1.0.0 and can be run in the HTML footer.

The `custom.js` file also indirectly relies on jQuery. But in this case, `base.js` could be edited to be self-contained or to rely on another framework. There is no need for jQuery to be listed as a dependency of `custom.js`.

It's now simply a matter of registering your JavaScript using the function `wp_register_script`. This takes the following arguments:

- `$handle`
A string
- `$source`
A string
- `$dependencies`
An array (optional)

-
- `$version`
A string (optional)
 - `$in_footer`
True/false (optional, default is false)

When registering scripts, it is best to use the `init` hook and to register them all at once.

To register the scripts in our example, add the following to the theme's *functions.php* file:

```
function mytheme_register_scripts() {
    //base.js - dependent on jQuery
    wp_register_script(
        'theme-base', //handle
        '/wp-content/themes/my-theme/base.js', //source
        array('jquery'), //dependencies
        '1.0.0', //version
        true //run in footer
    );

    //custom.js - dependent on base.js
    wp_register_script(
        'theme-custom',
        '/wp-content/themes/my-theme/custom.js',
        array('theme-base'),
        '1.0.0',
        TRUE
    );
}
add_action('init', 'mytheme_register_scripts');
```

There is no need to register jQuery, because WordPress already has. Re-registering it could lead to problems.

YOU HAVE ACHIEVED NOTHING!

All of this registering JavaScript files the WordPress way has, so far, achieved nothing. Nothing will be outputted to your HTML files.

To get WordPress to output the relevant HTML, we need to enqueue our files. Unlike the relatively long-winded commands required to register the functions, this is a very simple process.

Outputting the JavaScript HTML

Adding the `<script>` tags to your HTML is done with the `wp_enqueue_script` function. Once a script is registered, it takes one argument, the file's handle.

Adding JavaScript to the HTML is done in the `wp_print_scripts` hook with the following code:

```
function mytheme_enqueue_scripts(){
    if (!is_admin()):
        wp_enqueue_script('theme-custom'); //custom.js
    endif; //!is_admin
}
add_action('wp_print_scripts', 'mytheme_enqueue_scripts');
```

Of our two registered JavaScript files (base.js and custom.js), only the second adds JavaScript functionality to the website. Without the second file, there is no need to add the first.

Having enqueued custom.js for output to the HTML, WordPress will figure out that it depends on base.js being present and that base.js, in turn, requires jQuery. The resulting HTML is:

```
<script src="/wp-includes/js/jquery/jquery.js?ver=1.6.1" type="text/
javascript"></script>
<script src="/wp-content/themes/my-theme/base.js?ver=1.0.0" type="text/
javascript"></script>
<script src="/wp-content/themes/my-theme/custom.js?ver=1.0.0" type="text/
javascript"></script>
```

Registering Style Sheets

Both of the functions for adding JavaScript to our HTML have sister PHP functions for adding style sheets to the HTML: `wp_register_style` and `wp_enqueue_style`.

As with the JavaScript example, we'll use a couple of CSS files throughout this article, employing the mobile-first methodology for responsive Web design.

The `mobile.css` file is the CSS for building the mobile version of the website. It has no dependencies. The `desktop.css` file is the CSS that is loaded for desktop devices only. The desktop version builds on the mobile version, so `mobile.css` is a dependency.

Once you've decided on version numbers, dependencies and media types, it's time to register your style sheets using the `wp_register_style` function. This function takes the following arguments:

- `$handle`
A string
- `$source`
A string
- `$dependencies`
An array (optional, default is none)

-
- `$version`
A string (optional, the default is the current WordPress version number)
 - `$media_type`
A string (optional, the default is all)

Again, registering your style sheets using the `init` action is best. To your theme's `functions.php`, you would add this:

```
function mytheme_register_styles(){
    //mobile.css for all devices
    wp_register_style(
        'theme-mobile', //handle
        '/wp-content/themes/my-theme/mobile.css', //source
        null, //no dependencies
        '1.0.0' //version
    );

    //desktop.css for big-screen browsers
    wp_register_style(
        'theme-desktop',
        '/wp-content/themes/my-theme/desktop.css',
        array('theme-mobile'),
        '1.0.0',
        'only screen and (min-width : 960px)' //media type
    );

    /* *keep reading* */
}
add_action('init', 'mytheme_register_styles');
```

We have used CSS3 media queries to prevent mobile browsers from parsing our desktop style sheet. But Internet Explorer versions 8 and below do not understand CSS3 media queries and so will not parse the desktop CSS either.

IE8 is only two years old, so we should support its users with conditional comments.

CONDITIONAL COMMENTS

When registering CSS using the register and enqueue functions, conditional comments are a little more complex. WordPress uses the object `$wp_styles` to store registered style sheets.

To wrap your file in conditional comments, add extra information to this object.

For Internet Explorer 8 and below, excluding mobile IE, we need to register another copy of our desktop style sheet (using the media type `all`) and wrap it in conditional comments.

In the code sample above, `/* *keep reading* */` would be replaced with the following:

```
global $wp_styles;
wp_register_style(
    'theme-desktop-ie',
    '/wp-content/themes/my-theme/desktop.css',
    array('theme-mobile'),
    '1.0.0'
);

$wp_styles->add_data(
    'theme-desktop-ie', //handle
    'conditional', //is a conditional comment
    '!(IEMobile)&(lte IE 8)' //the conditional comment
);
```

Unfortunately, there is no equivalent for wrapping JavaScript files in conditional comments, presumably due to the concatenation of JavaScript in the admin section.

If you need to wrap a JavaScript file in conditional comments, you will need to add it to `header.php` or `footer.php` in the theme. Alternatively, you could use the `wp_head` or `wp_footer` hooks.

Outputting The Style Sheet HTML

Outputting the style sheet HTML is very similar to outputting the JavaScript HTML. We use the `enqueue` function and run it on the `wp_print_styles` hook.

In our example, we could get away with telling WordPress to queue only the style sheets that have the handles `theme-desktop` and `theme-desktop-ie`. WordPress would then output the `mobile/all` media version.

However, both style sheets apply styles to the website beyond a basic reset: `mobile.css` builds the website for mobile phones, and `desktop.css` builds on top of that. If it does something and I've registered it, then I should enqueue it. It helps to keep track of what's going on.

Here is the code to output the CSS to the HTML:

```
function mytheme_enqueue_styles(){
    if (!is_admin()):
        wp_enqueue_style('theme-mobile'); //mobile.css
        wp_enqueue_style('theme-desktop'); //desktop.css
        wp_enqueue_style('theme-desktop-ie'); //desktop.css lte ie8
    endif; //!is_admin
}
add_action('wp_print_styles', 'mytheme_enqueue_styles');
```

What's The Point?

You may be wondering what the point is of going through all of this extra effort when we could just output our JavaScript and style sheets in the theme's `header.php` or using the `wp_head` hook.

In the case of CSS in a standalone theme, it's a valid point. It's extra work without much of a payoff.

But with JavaScript, it helps to prevent clashes between plugins and themes when each uses a different version of a JavaScript framework. It also makes page-loading times as fast as possible by avoiding file duplication.

WORDPRESS FRAMEWORKS

This group of functions can be most helpful when using a framework for theming. We've built a framework to speed up our production of websites in our agency.

As with most agencies, we have internal conventions for naming JavaScript and CSS files.

When we create a bespoke WordPress theme for a client, we develop it as a child theme of our framework. In the framework itself, we register a number of JavaScript and CSS files in accordance with our naming convention.

In the child theme, we then simply enqueue files to output the HTML.

```
function clienttheme_enqueue_css() {
    if (!is_admin()):
        wp_enqueue_style('theme-mobile');
        wp_enqueue_style('theme-desktop');
        wp_enqueue_style('theme-desktop-ie');
    endif; //!is_admin
}
add_action('wp_print_styles', 'clienttheme_enqueue_css');

function clienttheme_enqueue_js() {
    if (!is_admin()):
        wp_enqueue_script('theme-custom');
    endif; //!is_admin
}
add_action('wp_print_scripts', 'clienttheme_enqueue_js');
```

Adding CSS and JavaScript to our themes the WordPress way enables us to keep track of exactly what's going on at a glance.

A Slight Limitation

If you use a JavaScript framework in your theme or plugin, then you're stuck with the version that ships with the current version of WordPress, which sometimes falls a version or two behind the latest official release of the framework. (Upgrading to a newer version of the framework is technically possible, but this could cause problems with other themes or plugins that expect the version that ships with WordPress, so I've omitted this information from this chapter.)

While this prevents you from using any new features of the framework that were added after the version included in WordPress, the advantage is that all theme and plugin authors know which version of the framework to expect.

A Single Point Of Registration

Register your styles and scripts in a single block of code, so that when you update a file, you will be able to go back and update the version number easily.

If you use different code in different parts of the website, you can wrap the logic around the enqueue scripts.

If, say, your archive pages use different JavaScript than the rest of the website, then you might register three files:

- base JavaScript (registered as `theme-base`),
- archive JavaScript (registered as `theme-archive`),
- general JavaScript (registered as `theme-general`).

Again, the base JavaScript adds nothing to the website. Rather, it is a group of default functions that the other two files rely on. You could then enqueue the files using the following code:

```
function mytheme_enqueue_js(){  
    if (is_archive()) {  
        wp_enqueue_script('theme-archive');  
    }  
    elseif (!is_admin()) {  
        wp_enqueue_script('theme-general');  
    }  
}  
add_action('wp_print_scripts', 'mytheme_enqueue_js');
```

Using The Google AJAX CDN

While using JavaScript the WordPress way will save you the problem of common libraries conflicting with each other, you might prefer to serve these libraries from Google's server rather than your own.

Using Jason Penny's [Use Google Libraries plugin](#) is the easiest way to do this. The plugin automatically keeps jQuery in noConflict mode.

Putting It All Together

Once you've started registering and outputting your scripts and styles the WordPress way, you will find that managing these files becomes a series of logical steps:

1. Registration to manage:
 - version numbers,
 - file dependencies,
 - media types for CSS,
 - code placement for JavaScript (header or footer);
2. Enqueue/output files to HTML:
 - logic targeting output to specific WordPress pages,
 - WordPress automating dependencies.

Eliminating potential JavaScript conflicts from your WordPress theme or plugin frees you to get on with more important things, such as following up on sales leads or getting back to that hash-tag game that was so rudely interrupted.

Interacting With The WordPress Database

Daniel Pataki

While you already use many functions in WordPress to communicate with the database, there is an easy and safe way to do this directly, using the `$wpdb` class. Built on the great `ezSQL` class by Justin Vincent, `$wpdb` enables you to address queries to any table in your database, and it also helps you handle the returned data. Because this functionality is built into WordPress, there is no need to open a separate database connection (in which case, you would be duplicating code), and there is no need to perform hacks such as modifying a result set after it has been queried.



The `$wpdb` class modularizes and automates a lot of database-related tasks.

In this chapter, I will show you how to get started with the `$wpdb` class, how to retrieve data from your WordPress database and how to run more advanced queries that update or delete something in the database. The techniques here will remove some of the constraints that you run into with functions such as `get_posts()` and `wp_list_categories()`, allowing you to tailor queries to your particular needs. This method can also make your website more efficient by getting only the data that you need—nothing more, nothing less.

Getting Started

If you know how MySQL or similar languages work, then you will be right at home with this class, and you will need to keep only a small number of function names in mind. The basic usage of this class can be best understood through an example. So let's query our database for the IDs and titles of the four most recent posts, ordered by comment count (in descending order).

```
<?php
    $posts = $wpdb->get_results("SELECT ID, post_title FROM $wpdb->posts
WHERE post_status = 'publish'
    AND post_type='post' ORDER BY comment_count DESC LIMIT 0,4")
?>
```

As you can see, this is a basic SQL query, with some PHP wrapped around it. The `$wpdb` class contains a method (a method is a special name for functions that are inside classes), named `get_results()`, which will not only fetch your results but put them in a convenient object.

You might have noticed that, instead of using `wp_posts` for the table's name, I have used `$wpdb->posts`, which is a helper to access your core WordPress tables. More on why to use these later.

The `$results` object now contains your data in the following format:

```
Array
(
    [0] => stdClass Object
        (
            [ID] => 6
            [post_title] => The Male Angler Fish Gets Completely Screwed
        )

    [1] => stdClass Object
        (
            [ID] => 25
            [post_title] => 10 Truly Amazing Icon Sets From Germany
        )

    [2] => stdClass Object
        (
            [ID] => 37
            [post_title] => Elderberry Is Awesome
        )

    [3] => stdClass Object
        (
            [ID] => 60
            [post_title] => Gathering Resources and Inspiration With Evernote
        )

)
```

Retrieving Results From The Database

If you want to retrieve some information from the database, you can use one of four helper functions to structure the data.

GET_RESULTS()

This is the function that we looked at earlier. It is best for when you need two-dimensional data (multiple rows and columns). It converts the data into an array that contains separate objects for each row.

```
<?php
    $posts = $wpdb->get_results("SELECT ID, post_title FROM wp_posts
WHERE post_status = 'future' AND post_type='post' ORDER BY post_date
ASC LIMIT 0,4")

    // Echo the title of the first scheduled post
    echo $posts[0]->post_title;
?>
```

GET_ROW

When you need to find only one particular row in the database (for example, the post with the most comments), you can use `get_row()`. It pulls the data into a one-dimensional object.

```
<?php
    $posts = $wpdb->get_row("SELECT ID, post_title FROM wp_posts WHERE
post_status = 'publish'
AND post_type='post' ORDER BY comment_count DESC LIMIT 0,1")

    // Echo the title of the most commented post
    echo $posts->post_title;
?>
```

GET_COL

This method is much the same as `get_row()`, but instead of grabbing a single row of results, it gets a single column. This is helpful if you would like to retrieve the IDs of only the top 10 most commented posts. Like `get_row()`, it stores your results in a one-dimensional object.

```
<?php
    $posts = $wpdb->get_col("SELECT ID FROM wp_posts WHERE post_status
= 'publish'
    AND post_type='post' ORDER BY comment_count DESC LIMIT 0,10")

    // Echo the ID of the 4th most commented post
    echo $posts[3]->ID;
?>
```

GET_VAR

In many cases, you will need only one value from the database; for example, the email address of one of your users. In this case, you can use `get_var` to retrieve it as a simple value. The value's data type will be the same as its type in the database (i.e. integers will be integers, strings will be strings).

```
<?php
    $email = $wpdb->get_var("SELECT user_email FROM wp_users WHERE
user_login = 'danielpataki' ")

    // Echo the user's email address
    echo $email;
?>
```

Inserting Into The Database

To perform an insert, we can use the `insert` method:

```
$wpdb->insert( $table, $data, $format);
```

This method takes three arguments. The first specifies the name of the table into which you are inserting the data. The second argument is an array that contains the columns and their respective values, as key-value pairs. The third parameter specifies the data type of your values, in the order you have given them. Here's an example:

```
<?php
    $wpdb->insert($wpdb->usermeta, array("user_id" => 1, "meta_key" =>
    "awesome_factor", "meta_value" => 10), array("%d", %s", "%d"));

    // Equivalent to:
    // INSERT INTO wp_usermeta (user_id, meta_key, meta_value) VALUES
    (1, "awesome_factor", 10);
?>
```

If you're used to writing out your inserts, this may seem unwieldy at first, but it actually gives you a lot of flexibility because it uses arrays as inputs.

Specifying the format is optional; all values are treated as strings by default, but including this in the method is a good practice. The three values you can use are %s for strings, %d for decimal numbers and %f for floats.

Updating Your Data

By now, you won't be surprised to hear that we also have a helper method to update our data—shockingly, called `update()`.

Its use resembles what we saw above; but to handle the where clause of our update, it needs two extra parameters.

```
$wpdb->update( $table, $data, $where, $format = null, $where_format
= null );
```

The `$table`, `$data` and `$format` parameters should be familiar to you; they are the same as before. Using the `$where` parameter, we can specify the conditions of the update. It should be an array in the form of column-value pairs. If you specify multiple parameters, then they will be joined with AND logic. The `$where_format` is just like `$format`: it specifies the format of the values in the `$where` parameter.

```
$wpdb->update( $wpdb->posts, array("post_title" => "Modified Post Title"), array("ID" => 5), array("%s"), array("%d") );
```

Other Queries

While the helpers above are great, sometimes performing different or more complex queries than the helpers allow is necessary. If you need to perform an update with a complex `where` clause containing multiple AND/OR logic, then you won't be able to use the `update()` method. If you wanted to do something like delete a row or set a connection character set, then you would need to use the “general” `query()` method, which let's you perform any sort of query.

```
$wpdb->query("DELETE FROM wp_usermeta WHERE meta_key = 'first_login' OR meta_key = 'security_key' ");
```

Protection And Validation

I hope I don't have to tell you how important it is to make sure that your data is safe and that your database can't be tampered with! Data validation is a bit beyond the scope of this article, but do take a look at what the WordPress Codex has to say about “Data Validation” at some point.

In addition to validating, you will need to escape all queries. Even if you are not familiar with SQL injection attacks, still use this method and then read up on it later, because it is critical. The good news is that if you use any of the helper functions, then you don't need to do anything: the query is escaped for you. If you use the `query()` method, however, you will need to escape manually, using the `prepare()` method.

```
$sql = $wpdb->prepare( 'query' [, value_parameter,  
value_parameter ... ] );
```

To make this a bit more digestible, let's rewrite this basic format a bit.

```
$sql = $wpdb->prepare( "INSERT INTO $wpdb->postmeta (post_id,  
meta_key, meta_value ) VALUES ( %d, %s, %d )", 3342, 'post_views',  
2290 )  
$wpdb->query($sql);
```

As you can see, this is not that scary. Instead of adding the actual values where you usually would, you enter the type of data, and then you add the actual data as subsequent parameters.

Class Variables And Other Methods

Apart from these excellent methods, there are quite a few other functions and variables to make your life easier.

I'll show you some of the most common ones, but please do look at the WordPress Codex page linked to above for a full list of everything that `$wpdb` has to offer.

INSERT_ID()

Whenever you insert something into a table, you will very likely have an auto-incrementing ID in there. To find the value of the most recent insert performed by your script, you can use `$wpdb->insert_id`.

```
$sql = $wpdb->prepare( "INSERT INTO $wpdb->postmeta (post_id,
meta_key, meta_value ) VALUES ( %d, %s, %d )", 3342, 'post_views',
2290 )

$wpdb->query($sql);

$meta_id = $wpdb->insert_id;
```

NUM_ROWS()

If you've performed a query in your script, then this variable will return the number of results of your last query. This is great for post counts, comment counts and so on.

TABLE NAMES

All the core table names are stored in variables whose names are exactly the same as their core table equivalent. The name of your posts table (probably `wp_posts`) would be stored in the `$posts` variable, so you could output it by using `$wpdb->posts`.

We use this because we are allowed to choose a prefix for our WordPress tables. While most people use the default `wp`, some users want or need a custom one. For the sake of flexibility, this prefix is not hardcoded, so if you are writing a plugin and use `wp_postmeta` in a query instead of `$wpdb->postmeta`, your code will not work on some websites.

If you want to get data from a non-core WordPress table, no special variable will be available for it. In this case, you can just write the table's name as usual.

ERROR HANDLING

By calling the `show_errors()` or `hide_errors()` methods, you can turn error-reporting on or off (it's off by default) to get some more info about what's going on. Either way, you can also use the `print_error()` method to print the errors for the latest query.

```
$wpdb->show_errors();  
$wpdb->query("DELETE FROM wp_posts WHERE post_id = 554 ");  
  
// When run, because show_errors() is set, the error message will  
tell you that the "post_id" field is an unknown  
// field in this table (since the correct field is ID)
```

Building Some Basic Tracking With Our New \$wpdb Knowledge

If you're new to all of this, you probably get what I'm going on about but may be finding it hard to implement. So, let's take the example of a simple WordPress tracking plugin that I made for a website.

For simplicity's sake, I won't describe every detail of the plugin. I'll just show the database's structure and some queries.

OUR TABLE'S STRUCTURE

To keep track of ad clicks and impressions, I created a table; let's call it "tracking." This table records user actions in real time. Each impression and click is recorded in its own row in the following structure:

- ID
The auto-incremented ID.

-
- `time`
The date and time that the action occurred.
 - `deal_id`
The ID of the deal that is connected to the action (i.e. the ad that was clicked or viewed).
 - `action`
The type of action (i.e. click or impression).
 - `action_url`
The page on which the action was initiated.
 - `user_id`
If the user is logged in, their ID.
 - `user_ip`
The IP of the user, used to weed out any naughty business.

This table will get pretty big pretty fast, so it is aggregated into daily statistics and flushed periodically. But let's just work with this one table for now.

Inserting Data Into Our Tables

When a user clicks an ad, it is detected, and the information that we need is sent to our script in the form of a `$_POST` array, with the following data:

```
Array
(
    [deal_id] => 643
    [action] => click
    [action_url] => http://thisiswhereitwasclicked.com/about/
    [user_id] => 223
    [user_ip] = 123.234.223.12
)
```

We can then insert this data into the database using our helper method, like so:

```
$wpdb->insert('tracking', array("deal_id" => 643, "action" =>
"click", "action_url" => "http://thisiswhereitwasclicked.com/about/
",
"user_id" => 223, "user_ip" => "123.234.223.12"), array("%d", %s",
"%s", "%d", "%s"));
```

At the risk of going on a tangent, I'll address some questions you might have about this particular example. You may be thinking, what about data validation? The click could have come from a website administrator, or a user could have clicked twice by mistake, or a bunch of other things might have happened.

We decided that because we don't need real-time stats (daily stats is enough), there is no point to check the data at every insert.

Data is aggregated into a new table every day around midnight, a low traffic time. Before aggregating the data, we take care to clean it up, taking out duplicates and so on. The data is, of course, escaped before being inserted into the table, because we are using a helper function; so, we are safe there.

Just deleting in bulk all at once the ones that are made by administrators is easier than checking at every insert. This takes a considerable amount of processing off our server's shoulders.

DELETING ACTIONS FROM A BLACKLISTED IP

If we find that the IP address 168.211.23.43 is being naughty-naughty, we could blacklist it. In this case, when we aggregate the daily data, we would need to delete all of the entries by this IP.

```
$sql = $wpdb->prepare("DELETE FROM tracking WHERE user_ip = %s ",  
'168.211.23.43');  
$wpdb->query($sql);
```

You have probably noticed that I am still escaping the data, even though the IP was received from a secure source. I would suggest escaping your data no matter what. First of all, proper hackers are good at what they do, because they are excellent programmers and can outsmart you in ways that you wouldn't think of. Also, I personally have done more to hurt my own websites than hackers have, so I do these things as a safety precaution against myself as well.

UPDATING TOTALS

We store our ads as custom post types; and to make statistical reporting easier, we store the total amount of clicks that an ad receives separately as well. We could just add up all of the clicks in our tracking database for the given deal as well, so let's look at that first.

```
$total = $wpdb->get_var("SELECT COUNT(ID) WHERE deal_id = 125 ");
```

Because getting a single variable is easier than always burdening ourselves with a more complex query, whenever we aggregate our data, we would store the current total separately. Our ads are stored as posts with a custom post type, so a logical place to store this total is in the `postmeta` table. Let's use the `total_clicks` meta key to store this data.

```
$wpdb->update( $wpdb->postmeta, array("meta_value" => $total),  
array("ID" => 125), array("%d"), array("%d") );  
  
// note that this should be done with update_post_meta(), I just  
did it the way I did for example's sake
```

Final Thoughts And Tips

I hope you have gained a better understanding of the WordPress `$wpdb` class and that you will be able to use it to make your projects better. To wrap up, here are some final tips and tricks for using this class effectively.

- I urge you to be cautious: with great power comes great responsibility. Make sure to escape your data and to validate it, because improper use of this class is probably a leading cause of hacked websites!
- Ask only for the data that you need. If you will only be displaying an article's title, there is no need to retrieve all of the data from each row. In this case, just ask for the title and the ID: `SELECT title, ID FROM wp_posts ORDER BY post_date DESC LIMIT 0,5.`
- While you can use the `query()` method for any query, using the helper methods (`insert`, `update`, `get_row`, etc.) is better if possible. They are more modular and safer, because they escape your data automatically.
- Take care when deleting records from a WordPress (or any other) database. When WordPress deletes a comment, a bunch of other actions also take place: the comment count in the `wp_posts` table needs to be reduced by one, all of the data in the `comment_meta` table needs to be deleted as well, and so on. Make sure to clean up properly after yourself, especially when deleting things.

-
- Look at all of the class variables and other bits of information in the official documentation. These will help you use the class to its full potential. I also recommend looking at the ezSQL class for general use in your non-WordPress projects; I use it almost exclusively for everything I do.

How To Create A WordPress Plugin

Daniel Pataki

WordPress plugins are PHP scripts that alter your website. The changes could be anything from the simplest tweak in the header to a more drastic makeover (such as changing how log-ins work, triggering emails to be sent, and much more).

Whereas themes modify the look of your website, plugins change how it functions. With plugins, you can create custom post types, add new tables to your database to track popular articles, automatically link your contents folder to a “CDN” server such as Amazon S3... you get the picture.



Theme Or Plugin?

If you've ever played around with a theme, you'll know it has a `functions.php` file, which gives you a lot of power and enables you to build plugin-like functionality into your theme. So, if we have this `functions.php` file, what's the point of a plugin? When should we use one, and when should we create our own?

The line here is blurrier than you might think, and the answer will often depend on your needs. If you just want to modify the default length of your posts' excerpts, you can safely do it in `functions.php`. If you want something that lets users message each other and become friends on your website, then a plugin would better suit your needs.

The main difference is that a plugin's functionality persists regardless of what theme you have enabled, whereas any changes you have made in `functions.php` will stop working once you switch themes. Also, grouping related functionality into a plugin is often more convenient than leaving a mass of code in `functions.php`.

Creating Our First Plugin

To create a plugin, all you need to do is create a folder and then create a single file with one line of content. Navigate to the `wp-content/plugins` folder, and create a new folder named `awesomeplugin`. Inside this new folder, create a file named `awesomeplugin.php`. Open the file in a text editor, and paste the following information in it:

```
<?php
/*
Plugin Name: Awesomeness Creator
Plugin URI: http://my-awesomeness-emporium.com
Description: a plugin to create awesomeness and spread joy
Version: 1.2
Author: Mr. Awesome
Author URI: http://mrtotallyawesome.com
License: GPL2
*/
?>
```

Of all this information, only the plugin's name is required. But if you intend to distribute your plugin, you should add as much data as possible.

With that out of the way, you can go into the back end to activate your plugin. That's all there is to it! Of course, this plugin doesn't do anything; but strictly speaking, it is an active, functioning plugin.

Structuring Plugins

When creating complex functionality, splitting your plugin into multiple files and folders might be easier. The choice is yours, but following a few good tips will make your life easier.

If your plugin focuses on one main class, put that class in the main plugin file, and add one or more separate files for other functionality. If your plugin enhances WordPress' back end with custom controls, you can create the usual CSS and JavaScript folders to store the appropriate files.

Generally, aim for a balance between layout structure, usability and minimalism. Split your plugin into multiple files as necessary, but don't go overboard. I find it useful to look at the structure of popular plugins such as [WP-PageNavi](#) and [Akismet](#).

Naming Your Plugin And Its Functions

When creating a plugin, exercise caution in naming the functions, classes and plugin itself. If your plugin is for generating awesome excerpts, then calling it “excerpts” and calling its main function “the_excerpt” might seem logical. But these names are far too generic and might clash with other plugins that have similar functionality with similar names.

The most common solution is to use unique prefixes. You could use “acme_excerpt,” for example, or anything else that has a low likelihood of matching someone else's naming scheme.

Plugin Safety

If you plan to distribute your plugin, then security is of utmost importance, because now you are fiddling with other people's websites, not just your own. All of the security measures you should take merit their own article, so keep an eye out for an upcoming piece on how to secure your plugin. For now, let's just look at the theory in a nutshell; you can worry about implementation once you grasp that.

The safety of your plugin usually depends on the stability of its two legs. One leg makes sure that the plugin does not help spread naughty data. Guarding against this entails filtering the user's input, escaping queries to protect against SQL injection attacks and so on. The second leg makes sure that the user has the authority and intention to perform a given action. This basically means that only users with the authority to delete data (such as administrators) should be able to do it. Guarding intention ensures that visitors aren't misled by a hacker who has managed to place a malicious link on your website.

All of this is much easier to do than you might think, because WordPress gives you many functions to handle it. A number of other issues and best practices are involved, however, so we'll cover those in a future article. There is plenty to learn and do until then; if you're just starting out, don't worry about all that for now.

Cleaning Up After Yourself

Many plugins are guilty of leaving a lot of unnecessary data lying around. Data that only your plugin uses (such as meta data for posts or comments, database tables, etc.) can wind up as dead weight if the plugin doesn't clean up after itself.

WordPress offers three great hooks to help you take care of this:

- [register_activation_hook\(\)](#)

This hook allows you to create a function that runs when your plugin is activated. It takes the path to your main plugin file as the first argument, and the function that you want to run as the second argument. You can use this to check the version of your plugin, do some upgrades between versions, check for the correct PHP version and so on.

- [register_deactivation_hook\(\)](#)

The name says it all. This function works like its counterpart above, but it runs whenever your plugin is deactivated. I suggest using the next function when deleting data; use this one just for general housekeeping.

- [register_uninstall_hook\(\)](#)

This function runs when the website administrator deletes your plugin in WordPress' back end. This is a great way to remove data that has been lying around, such as database tables, settings and what not. A drawback to this method is that the plugin needs to be able to run for it to work; so, if your plugin cannot uninstall in this way, you can create an *uninstall.php* file. Check out this function's documentation for more information.

If your plugin tracks the popularity of content, then deleting the tracked data when the user deletes the plugin might not be wise. In this case, at least point the user to the location in the back end where they can find the plugin's data, or give them the option to delete the data on the plugin's settings page before deleting the plugin itself.

The net result of all our effort is that a user should be able to install your plugin, use it for 10 years and then delete it without leaving a trace on the website, in the database or in the file structure.

Documentation And Coding Standards

If you are developing for a big community, then documenting your code is considered good manners (and good business). The conventions for this are fairly well established — [phpDocumentor](#) is one example. But as long as your code is clean and has some documentation, you should be fine.

I document code for my own benefit as well, because I barely remember what I did yesterday, much less the purpose of functions that I wrote months back. By documenting code, you force good practices on yourself. And if you start working on a team or if your code becomes popular, then documentation will be an inevitable part of your life, so you might as well start now.

While not quite as important as documentation, following coding standards is a good idea if you want your code to comply with [WordPress' guidelines](#).

Putting It Into Practice

All work and no play makes Jack a dull boy, so let's do something with all of this knowledge that we've just acquired. To demonstrate, let's build a quick plugin that tracks the popularity of our articles by storing how many times each post has been viewed. I will be using hooks, which we'll cover in an upcoming installment in this series. Until then, as long as you grasp the logic behind them, all is well; you will understand hooks and plugins before long!

PLANNING AHEAD

Before writing any code, let's think ahead and try to determine the functions that our plugin will need. Here's what I've come up with:

- A function that registers a view every time an individual post is shown,
- A function that enables us to retrieve the raw number of views,
- A function that enables us to show the number of views to the user,
- A function that retrieves a list of posts based on their view count.

PREPARING OUR FUNCTION

The first step is to create the folder and file structure. Putting all of this into one file will be fine, so let's go to the `plugins` folder and create a new folder named `awesomely_popular`. In this folder, create a file named `awesomely_popular.php`. Open your new file, and paste some meta data at the top, something like this:

```
<?php
/*
Plugin Name: Awesomely Popular
Plugin URI: http://awesomelypopularplugin.com
Description: A plugin that records post views and contains
functions to easily list posts by popularity
Version: 1.0
Author: Mr. Awesome
Author URI: http://mayawesomefillyourbelly.com
License: GPL2
*/
?>
```

RECORDING POST VIEWS

Without delving too deep, WordPress hooks enable you to (among other things) fire off one of your functions whenever another WordPress function runs. So, if we can find a function that runs whenever an individual post is viewed, we are all set; all we would need to do is write our own function that records the number of views and hook it in. Before we get to that, though, let's write the new function itself. Here is the code:

```

/**
 * Adds a view to the post being viewed
 *
 * Finds the current views of a post and adds one to it by updating
 * the postmeta. The meta key used is "awepop_views".
 *
 * @global object $post The post object
 * @return integer $new_views The number of views the post has
 */
function awepop_add_view() {
    if(is_single()) {
        global $post;

        $current_views = get_post_meta($post->ID, "awepop_views", true);
        if(!isset($current_views) OR empty($current_views) OR !
is_numeric($current_views) ) {
            $current_views = 0;
        }

        $new_views = $current_views + 1;
        update_post_meta($post->ID, "awepop_views", $new_views);
        return $new_views;
    }
}

```

As you can see, I have added phpDocumentor-style documentation to the top of the function, and this is a pretty good indication of what to expect from this convention. First of all, using a conditional tag, we determine whether the user is viewing a post on a dedicated page.

If the user is on a dedicated page, we pull in the `$post` object, which contains data about the post being shown (ID, title, posting date, comment count, etc.). We then retrieve the number of views that the post has already gotten. We will add 1 to this and then overwrite the original value with the new one. In case something goes wrong, we first check whether the current view count is what it should be.

The current view count must be set; it cannot be empty. And it must be numeric in order for us to be able to add 1 to it. If it does not meet these criteria, then we could safely bet that the current view count is 0. Next, we add 1 to it and save it to the database. Finally, we return the number of views that the post has gotten, together with this latest number.

So far, so good. But this function is never called, so it won't actually be used. This is where hooks come in. You could go into your theme's files and call the function manually from there. But then you would lose that functionality if you ever changed the theme, thus defeating the entire purpose. A hook, named `wp_head`, that runs just before the `</head>` tag is present in most themes, so we can just set our function to run whenever `wp_head` runs, like so:

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

That's all there is to the "mysticism" of hooks. We are basically saying, whenever `wp_head` runs, also execute the `awepop_add_view` function. You can place the code before or after the function itself.

RETRIEVING AND SHOWING THE VIEWS

In the function above, I already use the WordPress `get_post_meta()` function to retrieve the views, so writing a separate function for this might seem a bit redundant. In this case, it might well be redundant, but it promotes some object-oriented thinking, and it gives us greater flexibility when further developing the plugin. Let's take a look:

```
/**
 * Retrieve the number of views for a post
 *
 * Finds the current views for a post, returning 0 if there are none
 *
 * @global object $post The post object
 * @return integer $current_views The number of views the post has
 */
function awepop_get_view_count() {
    global $post;

    $current_views = get_post_meta($post->ID, "awepop_views", true);
    if(!isset($current_views) OR empty($current_views) OR !
is_numeric($current_views) ) {
        $current_views = 0;
    }

    return $current_views;
}
```

This is the same piece of code that we used in the `awepop_add_view()` function, so you could just use this to retrieve the view count there as well. This is handy, because if you decide to handle the 0 case differently, you only need to change it here. You will also be able to extend this easily and provide support for cases when we are not in the loop (i.e. when the `$post` object is not available).

So far, we have just retrieved the view count. Now, let's show it. You might be thinking this is daft—all we need is `echo awpop_get_view_count() . "views",` no? That would certainly work, but what if there is only 1 view. In this case, we would need to use the singular “view.” You might also want the freedom to add some leading text or some other tidbit, which would be difficult to do otherwise. So, to allow for these scenarios, let's write another simple function:

```
/**
 * Shows the number of views for a post
 *
 * Finds the current views of a post and displays it together with
 some optional text
 *
 * @global object $post The post object
 * @uses awepop_get_view_count()
 *
 * @param string $singular The singular term for the text
 * @param string $plural The plural term for the text
 * @param string $before Text to place before the counter
 *
 * @return string $views_text The views display
 */
function awepop_show_views($singular = "view", $plural = "views",
$before = "This post has: ") {
    global $post;

    $current_views = awepop_get_view_count();

    $views_text = $before . $current_views . " ";

    if ($current_views == 1) {
        $views_text .= $singular;
    }
    else {
        $views_text .= $plural;
    }

    echo $views_text;
}
```

SHOW A LIST OF POSTS BASED ON VIEWS

To show a list of posts based on view count, we create a function that we can place anywhere in our theme. The function will use a custom query and loop through the results, displaying a simple list of titles.

```

/**
 * Displays a list of posts ordered by popularity
 *
 * Shows a simple list of post titles ordered by their view count
 *
 * @param integer $post_count The number of posts to show
 *
 */
function awepop_popularity_list($post_count = 10) {
    $args = array(
        "posts_per_page" => 10,
        "post_type" => "post",
        "post_status" => "publish",
        "meta_key" => "awepop_views",
        "orderby" => "meta_value_num",
        "order" => "DESC"
    );

    $awepop_list = new WP_Query($args);

    if($awepop_list->have_posts()) {
        echo "
<ul>";
    }

    while ( $awepop_list->have_posts() ) : $awepop_list->the_post();
        echo "
<li><a href='\".get_permalink($post->ID).\">\".the_title('\", \"\",
false).\"</a></li>

";
    endwhile;

```

```
if($awepop_list->have_posts()) {  
    echo "</ul>  
  
";  
}  
  
}
```

We start by passing a bunch of parameters to the `WP_Query` class, in order to create a new object that contains some posts. This class will do the heavy lifting for us: it finds 10 published posts that have `awepop_views` in the meta table, and orders them according to this property in descending order.

If posts meet this criterion, we create an unordered list element. Then, we loop through all of the posts that we have retrieved, showing each title as a link to the relevant post. We cap things off by adding a closing tag to the list when there are posts to show. The code is below, and the explanation follows. Placing the `awepop_popularity_list()` function anywhere in your theme should now generate a simple list of posts ordered by popularity.

As an added precaution, place the call to this function in your theme, like so:

```
if (function_exists("awepop_popularity_list")) {  
    awepop_popularity_list();  
}
```

This ensures that if the plugin is disabled (and thus the function becomes undefined), PHP won't throw a big ol' error. It just won't show the list of most popular posts.

Overview

By following the theory laid down in this article and using only a handful of functions, we have created a rudimentary plugin to track our most popular posts. It could be vastly improved, but it shows the basics of using plugins perfectly well. Moreover, by learning some patterns of WordPress development (plugins, hooks, etc.), you are gaining skills that will serve you in non-WordPress environments as well.

You should now be able to confidently follow tutorials that start with “First, create a WordPress plugin...” You now understand things not just on a need-to-know basis, but at a deeper level, which gives you more flexibility and power when coding your own plugins. Stay tuned for the upcoming article on hooks, actions and filters for an even more in-depth resource on the innards of plugins.

How To Integrate Facebook With WordPress

Thiemo Fetzner

Facebook is one of those Web phenomena that impress everyone with numbers. To cite some: about 250 million users are on Facebook, and together they spend more than 5 billion minutes on Facebook... every day. These numbers suggest that we should start thinking about how to use Facebook for blogging or vice versa.

We did some research to **find out how the integration of Facebook with WordPress and vice versa works**, or — in other words — how you can present your WordPress blog on Facebook or use the functionality of Facebook on your WordPress-powered blog. Both of these can be achieved with a set of WordPress plug-ins, a couple of which we'll present here in detail.

1. Integrating A WordPress Blog Into Facebook

Integrating a WordPress blog into Facebook is actually quite simply achieved via the **Facebook API**. The Facebook API makes programming applications that can be spread via Facebook almost a piece of cake. A lot of interactive browser games are on Facebook, such as the currently popular “Mafia Wars.” This game allows users to start a mafia family with their friends, with the goal of becoming an important figure in the virtual underground crime scene. To start a clan, you invite other friends on the network to join. This is the growth strategy of any application on Facebook: the simple snowball effect.

The applications sustain themselves through earnings generated by displaying advertisements, which also makes Facebook an even more attractive platform to develop on. This symbiosis generates growth for both Facebook and its applications.

PLUG-IN INSTALLATION AND CONFIGURATION

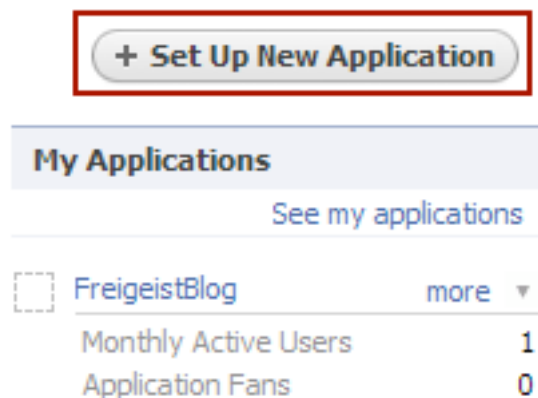
John Eckman developed the WordPress plug-in [Wordbook](#) in early 2009. This plug-in allows WordPress blog owners to integrate their blog in Facebook. This gives a blog two lives: one as an application on Facebook (such as, for example, my private FreigeistBlog) and one at the original URL (<http://freigeist.devmag.net>).

To access a blog via Facebook, you need to first grant access to the application. To do this, go to the so-called “canvas” page, which is where the Facebook twin of the blog lives ([my example](#)). However, granting access to the blog’s application means that the administrator of the blog also has access to information about you as a user (this is what most applications aim for: information such as date of birth, gender and educational status allows them to display quite targeted ads).

The application we’re dealing with is a simple blog and, in this sense, quite innocent, but we should state outright that the **Facebook API as it is now treats blogs and websites as applications**, which may not be appropriate, given the issue mentioned above. Facebook users who authorize the blog application can now easily send comments and share posts from within Facebook. The data, however, is still stored in the original database on the server where the blog is installed.

This makes it look as though Facebook serves merely as a simple feed reader. Yet, we get some other benefits. The blog on Facebook can be used to create a community around it by taking advantage of Facebook's snowball effect, because friends of the blog's users will see in their activity stream that they have been participating on the blog. Furthermore, it means that Facebook users will see new posts from your blog whenever they sign in to Facebook and can easily follow comments, making your blog more accessible.

To create a Facebook twin of your blog, first you have to [set up a new application](#). To do this, you need a Facebook account and have to register as a developer using the preceding link. All of this can be done in a few minutes.



Create a new Facebook application

Once you have agreed to the terms of use, give your application a name. Then you will receive your API key and a secret, which you will need later.

Basic	Essential Information	
Authentication	Application Name	FreigeistBlog
Profiles		
Canvas	Application ID	143305737165
Connect	API Key	afad0e000e000000
Widgets	Secret	afad0e000e000000
Advanced	Basic Information	

Settings for the Facebook application. [Large view](#).

Then, you have to submit a so-called “post-authorize callback URL.” This is the address on your server to which Facebook will send a notice whenever a user authorizes access to the application. By the same logic, there is also the “post-remove callback URL,” which receives a notice when a user removes the application. Both of these events are handled by the Wordbook plug-in. You merely need to write the address of the blog’s root directory with a trailing slash.

The screenshot displays the Facebook application settings interface. On the left is a sidebar with navigation links: Basic, Authentication (highlighted with a blue arrow), Profiles, Canvas, Connect, Widgets, and Advanced. The main content area is titled 'Authentication Settings'. It includes a section 'Installable to?' with checkboxes for 'Users' (checked) and 'Facebook Pages' (unchecked). Below this is a section titled 'Authentication Callback URLs'. It contains two text input fields: 'Post-Authorise Callback URL' and 'Post-Remove Callback URL', both containing the value 'http://freigeist.devmag.net/'. A 'Save' button is visible in the bottom right corner.

Facebook application settings: Define callback URLs.

The third step is to claim your canvas page, which is the page through which a Facebook user accesses your blog, and a canvas callback URL, which is the page from which content is retrieved. Again, include a trailing slash, or else internal links on your blog won't work with their Facebook twin.

Now you have some choices to make, namely, how to set up your canvas page. You have a choice between iFrame and FBML. FBML is a Facebook XML scheme with which you can use specific Facebook tags (such as tags to display user profiles). You can also use it to access certain Facebook procedures. However, the Wordbook plug-in works with iFrames, which allow Javascript and other tags, and which FBML does not support.

The image shows the Facebook application settings interface, specifically the 'Canvas' tab. On the left is a sidebar with navigation links: Basic, Authentication, Profiles, Canvas (highlighted with a blue arrow), Connect, Widgets, and Advanced. The main content area is divided into sections. The 'Required URLs' section contains 'Canvas Page URL' (http://apps.facebook.com/f) and 'Canvas Callback URL' (http://freigeist.devmag.net/). The 'Optional URLs' section contains 'Bookmark URL' and 'Post-Authorise Redirect URL', both with empty input fields. The 'Canvas settings' section includes 'Render Method' (radio buttons for IFrame, which is selected, and FBML), 'IFrame Size' (radio buttons for Smart size and Resize, with Resize selected), 'Canvas Width' (radio buttons for Full width (760px), which is selected, and another option), and 'Quick Transitions' (radio buttons for On and Off, with Off selected).

Required URLs	
Canvas Page URL	http://apps.facebook.com/f
Canvas Callback URL	http://freigeist.devmag.net/

Optional URLs	
Bookmark URL	
Post-Authorise Redirect URL	

Canvas settings	
Render Method	<input checked="" type="radio"/> IFrame <input type="radio"/> FBML
IFrame Size	<input type="radio"/> Smart size <input checked="" type="radio"/> Resize
Canvas Width	<input checked="" type="radio"/> Full width (760px) <input type="radio"/>
Quick Transitions	<input type="radio"/> On <input checked="" type="radio"/> Off

Facebook application settings: Define canvas page.

To distinguish between them rather crudely, you can say that iFrames give the developer more flexibility but, unlike FBML, restrict access to Facebook procedures.

Another advantage of iFrames is that code that Facebook retrieves from the canvas callback URL need not be parsed by the FBML parser, which could yield a performance gain. With iFrames, only internal links on the blog need to be adjusted. And the “resizable” option allows Facebook’s JavaScript code to adjust the size of the iFrame to Facebook’s layout.

Now the hard work is done. All that’s left is to install the Wordbook [plug-in using the standard WordPress method](#): install and activate. Then you can change the plug-in’s settings on the settings panel, and here you will need your application ID and the secret. You also have to tell the plug-in where the canvas page is located, so that internal links can be adjusted.

Required Options:

To use this app, you must register for an API key at <http://www.facebook.com/developers>. Follow the link and click "set up a new application." After you've obtained the keys, fill in both your application's API and Secret keys as well as your application's Canvas page URL.

Enter Your Facebook Application's API Key:



Enter Your Facebook Application's Secret:

Enter Your Facebook Application's Canvas Page URL, **NOT** INCLUDING "http://apps.facebook.com/"





Customization Options:

These options will allow you to customize wpbook to your liking.

Commenting Options:

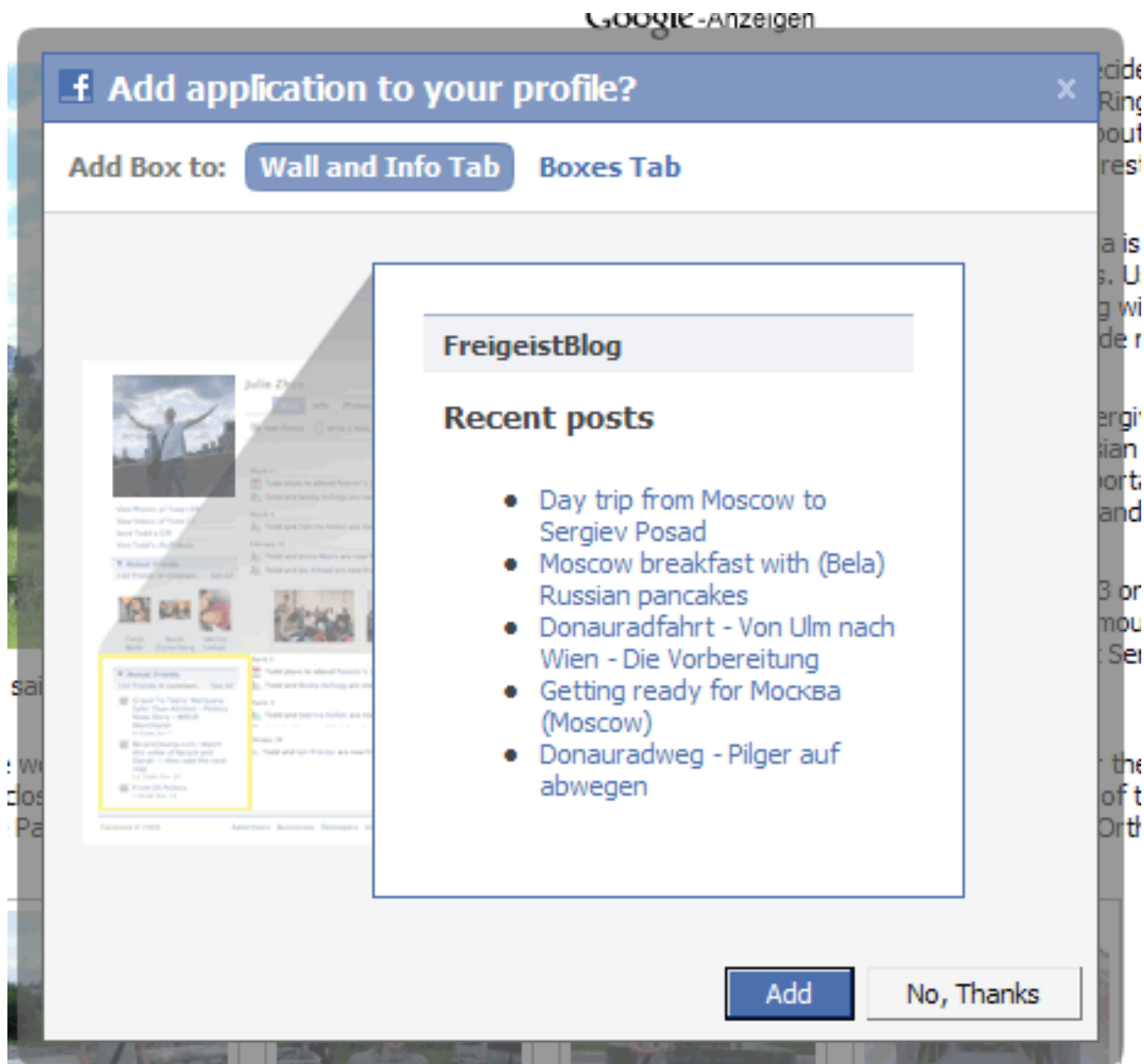
- ☒ Allow comments inside Facebook 
- ☒ Require Comment Authors E-mail Address 

Socialize Options:

- ☒ Show Invite Friends Link 
- ☒ Enable Facebook users to add your app to their profile 
- ☒ Enable "Share This Post" (within Facebook) 
- ☒ Enable "view post at external site" link 

Adjusting the settings of the Wordbook plug-in

And that's it! If you want, you could activate or deactivate some other options, such as the commenting function and whether users can add your application to their profile by displaying the latest posts from your blog in their profile.



Browsing through the blog via Facebook. [Large view.](#)

The plug-in allows you to play around a little bit. But as we said, you are somewhat limited in how fully you can integrate your blog into the Facebook canvas. But the next plug-in we'll look at integrates a bit of Facebook into your blog.

2. Integrating Facebook In A WordPress Blog

To begin, a little history lesson is needed. Many users do not like having to register for each blog where they would like to post comments, especially if they already have accounts on so many other social networks, such as Facebook and MySpace. So a single online ID for several purposes would be ideal, wouldn't it? That's the idea behind the OpenID protocol, which started in 2005. It decentralizes the identification of users for various providers and services. In essence, you can create an account on Facebook and connect it to services such as MySpace or even a personal blog. If you want to change your profile for all of these services, that too is decentralized: you simply change the settings on your Facebook account. OpenID is a chance to make the Web and its services more easily accessible. There are reasonable risks and concerns involved, but also many opportunities.

Facebook announced in 2007 that it would implement OpenID, and others followed, which explains why we now find more and more buttons that say "Connect with Facebook" or "Google Friend Connect." This leads us to our second plug-in, [Facebook Connect WordPress plug-in](#), which almost seamlessly integrates Facebook into your blog.

The plug-in allows users to comment on a blog using their Facebook account; and if they are already signed in or on Facebook, they need not sign in again. Users do not have to register for a unique account on the blog because the plug-in retrieves the user's information directly from the Facebook API. With access to the user profiles on Facebook, you can display your users' profile pictures, which adds a personal touch to your blog.

The plug-in integrates a lot of Facebook functions: for example, users can send invitations and share stories and comments on Facebook, which gives your blog the benefit of word-of-mouth marketing. To do this, you need to activate the plug-in option that publishes a user's activity in their respective activity feed. Last but not least, you can enable a gadget that displays the profile pictures of your blog's most recent visitors, similar to "Google Friend Connect."



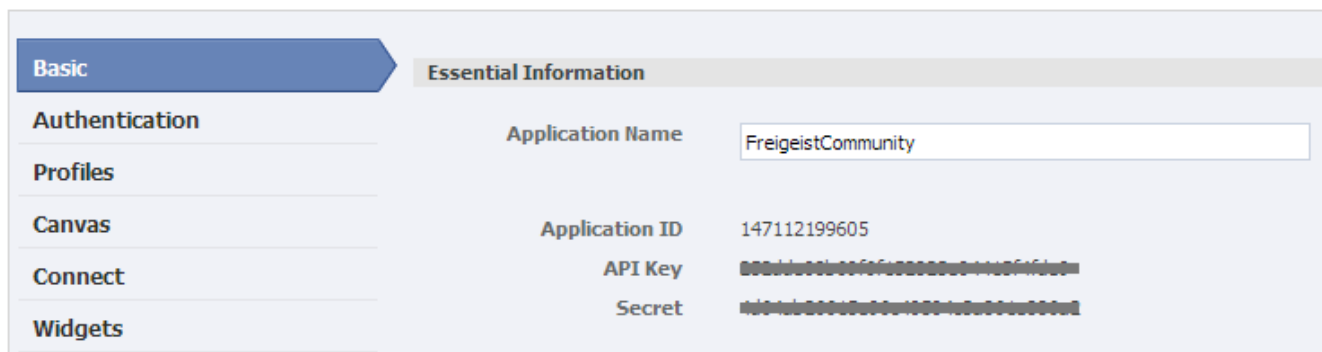
Facebook Connect implemented on sociable.es (in Spanish)

This plug-in essentially does the opposite of Wordbook (which integrates Facebook functionality into your blog).

PLUG-IN INSTALLATION AND CONFIGURATION

Again, as in the previous section, you will need to create a new Facebook application.

 [Edit FreigeistCommunity](#) [Back to My Applications](#)

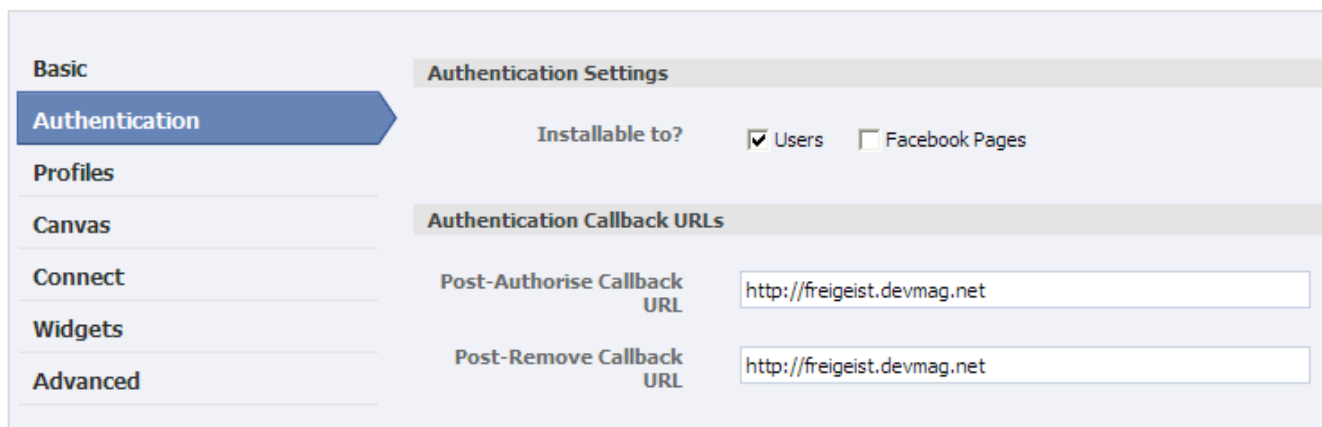


The screenshot shows the 'Essential Information' tab of a Facebook application configuration page. On the left is a sidebar with links: Basic, Authentication, Profiles, Canvas, Connect, and Widgets. The 'Basic' tab is selected. The main content area is titled 'Essential Information' and contains the following fields:

Field	Value
Application Name	FreigeistCommunity
Application ID	147112199605
API Key	[Redacted]
Secret	[Redacted]

Creating a new Facebook application.

You will also have to define the callback URLs, which point to the root of your blog.



The screenshot shows the 'Authentication' tab of the same Facebook application configuration page. The sidebar now has 'Authentication' selected. The main content area is titled 'Authentication Settings' and includes the following settings:

Installable to? ☒ Users ☐ Facebook Pages

Authentication Callback URLs

Field	Value
Post-Authorise Callback URL	http://freigeist.devmag.net
Post-Remove Callback URL	http://freigeist.devmag.net

Defining callback URLs for the new application.

Again, most of the work is now done, and you can soon start having fun and being creative. Just a few steps remain. First, download the plug-in from the website (see link above) and upload and enable it. A big part of the plug-in consists of the “Facebook Connect” library, which is provided by Facebook. You might stumble over the two `xd_reciever` files, one in HTML and one in PHP. They play a key role: enabling the so-called **cross-domain communication** (hence, the `xd`) between your blog and Facebook.

So why are these needed? Typically, HTTP requests are sent with the **XMLHttpRequest** object. However, the typical security settings on browsers allow XMLHttpRequest to send requests only to the domain where the original request was sent to. In our case, requests are sent to and from Facebook. This cross-domain communication is achieved with **iFrame cross-domain communication**. With this, the application opens an iFrame on facebook.com with the relevant requests; for example, to retrieve information on whether a user is logged into Facebook.

These requests are sent to Facebook through the iFrame via the URL, with which the iFrame is opened. The request is checked, and now the Facebook script that was called via the iFrame opens an iFrame on the application page, where the outcome of the request is sent to, again with the query string of the URL. The result of any requests lands in the query string of the `xd_receiver.htm` file on your own server. This circumvents the problem of being unable to use XMLHttpRequest.

Now back to the plug-in. Once you have installed and activated the plug-in, you can add the plug-in as a widget to your blog’s sidebar. However, you first need to enter your API key and secret.

Facebook Connector Options

Active Facebook template ID:247652790467

Facebook Application Configuration

Facebook App. Config.

[Go to Facebook Developer App](#)

Facebook API Key:

Facebook API Secret:

Automatic Approval:

☒ Enable comment auto-approval

Add share button:

☒ Add Facebook share button

Comment Form:

☒ Allow send user comments to Facebook.

Secure login:

☐ Facebook Connect Via SSL.[Update Configuration »](#)

Settings for Facebook Connect WordPress plug-in.

As you will see, a whole lot of options are enabled by default, such as automatically publishing comments if they are posted through a Facebook account (the rationale being that you don't have to moderate them because they come from actual people using Facebook and not spambots).

If you activate the sharing function, the plug-in adds a “Share” button automatically below each post. You can also activate the option that publishes a user’s comments in their activity feed on Facebook, thus making their activity on your blog visible to their friends.

After you have adjusted the settings, you will be notified that you need to define templates for the presentation. These need to be “synchronized” with Facebook. Scroll down a bit to generate and activate these templates. You can change the language manually here as well.

Template ID:253738180467 [Activate template](#) [Delete template](#) <-Active Facebook template

One line story: { *actor* } commented on { *blogname* }, post title: { *post_title* }, { *body_short* }

Short story title: { *actor* } commented on { *blogname* }, post title: { *post_title* }

Short story body: { *body_short* }

Full story title:

Full story body:

One line stories template:

{ *actor* } commented on { *blogname* }, post title: { *post_title* }, { *body_short* }

Short stories template:

Title

{ *actor* } commented on { *blogname* }, post title: { *post_title* }

Body

{ *body_short* }

Full stories template:

Title

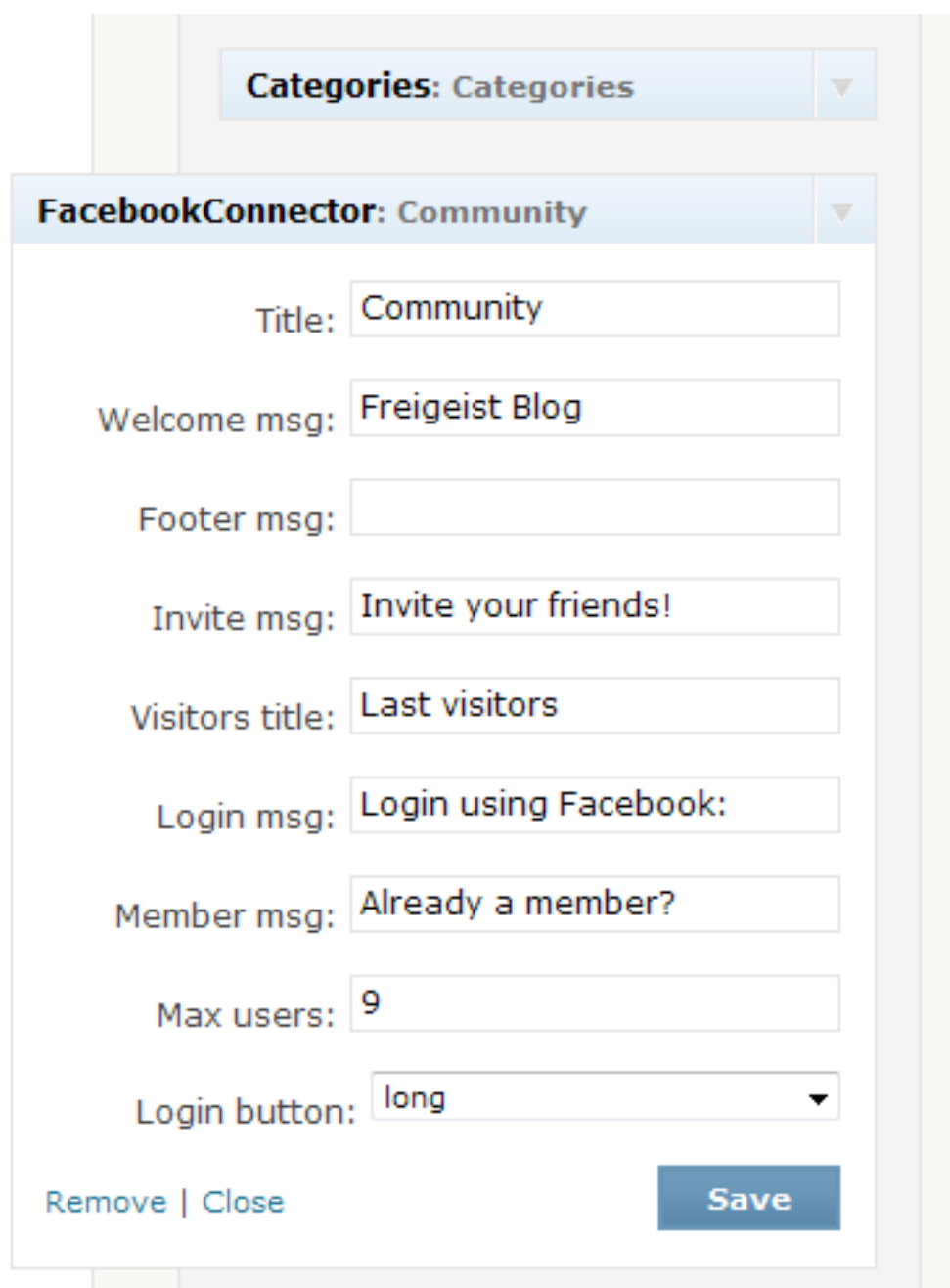
{ *actor* } commented on { *blogname* }, post title: { *post_title* }

Body

{ *body* }

Adjust Facebook Connect’s template settings.

The first template controls how a user's activity is posted in their activity feed on Facebook. However, you can also include the activity feed of your users in your gadget, as done on sociable.es (see link above). The last step is to go to the widget set-up page and include the gadget as a widget in your sidebar.



The image shows a configuration window for a widget titled "FacebookConnector: Community". At the top, there is a "Categories" dropdown menu with "Categories" selected. Below this, the "FacebookConnector: Community" dropdown is also set to "Community". The configuration fields include:

- Title: Community
- Welcome msg: Freigeist Blog
- Footer msg: (empty)
- Invite msg: Invite your friends!
- Visitors title: Last visitors
- Login msg: Login using Facebook:
- Member msg: Already a member?
- Max users: 9
- Login button: long (dropdown menu)

At the bottom left, there are links for "Remove" and "Close". At the bottom right, there is a blue "Save" button.

Including the Facebook Connect widget in the sidebar

Here again, you have some choice over the configuration, especially with regard to changing the language, showing a big or small “Connect to Facebook” button, etc. After installing the plug-in, you may want to see what else you can do with it. The implementation on sociable.es is quite a nice one.

Further Resources

These two plug-ins are quite specific in what they do. However, a wide variety of Facebook plug-ins are available for platforms other WordPress. Here is a list, certainly not comprehensive:

- [Movable Type](#)
A plug-in by Six Apart for adding Facebook Connect to a Movable Type blog, allowing any Facebook user to sign in. It is still in beta.
- [WordPress-FacebookConnect](#)
This plug-in is quite similar to the one on sociable.es. It has the same features, such as single sign-on, publishing comments to news feeds and displaying profile pictures. However, it has not been updated since the beginning of the year. Still, there is a nice tutorial by the developer Adam Breckler.
- [WordPress Fotobook](#)
With this WordPress plug-in, you can import all of your photo albums from Facebook onto a WordPress page.
- [Drupal's Facebook Connect module](#)
With this Drupal module, you can allow Facebook users to connect to your blog through their account. Similar to the plug-in by sociable.es.
- [Gigya WordPress plug-in](#)
This plug-in integrates not only Facebook but Twitter, MySpace and other OpenID providers into your blog for community building.

- [StatusPress](#)

This small plug-in displays your Facebook, Twitter or Last.fm status on your blog.

- [Quailpress](#)

Integrate Facebook-sharing functionality on your blog with this plug-in. However, it has not been actively developed for some time. And with the sociable.es plug-in, it is practically redundant.

How To Use AJAX In WordPress

Daniel Pataki

In the last few years, AJAX has crept onto websites and slowly become the way to create dynamic, user-friendly, responsive websites. AJAX is the technology that lets you update the contents of a page without actually having to reload the page in the browser. For example, Google Docs utilizes this technology when saving your work every few minutes.



While there are a number of ways to use AJAX in WordPress — and all are “correct,” in the loose sense of the word — there is one method that you should follow for a few reasons: WordPress supports it, it is future-proof, it is very logical, and it gives you numerous options right out of the box.

What Is AJAX?

If you're not familiar with AJAX, I suggest continuing to the end of this article and then reading the [Wikipedia article on AJAX](#), followed by some [tutorials](#). This is a rare case when I recommend reading as little about it as possible before trying it out, because it confused the heck out of me at first; and the truth is, you will rarely interact with AJAX in its “raw” state—you will usually use helpers such as jQuery.

In a nutshell, AJAX is a combination of HTML, CSS and JavaScript code that enables you to send data to a script and then receive and process the script's response without needing to reload the page.

If you are creating a page on your website where users can modify their profile, you could use AJAX to update a user's profile without needing to constantly reload the page whenever they submit the form. When the user clicks the button, the data they have entered into the form is sent via AJAX to the processing script, which saves the data and returns the string “data saved.” You can then output that data to the user by inserting it onto the page.

The thing to grasp about AJAX is how not different it is from what you're already doing. If you have a contact form, chances are that the form is marked up with HTML, some styles are applied to it, and a PHP script processes the information. The only difference between this and AJAX is how the information that the user inputs gets to the script and back to the user—everything else is the same.

To exploit the full potential of AJAX and get the most out of this article, you will need to be familiar with JavaScript (jQuery will suffice), as well as HTML, CSS and PHP. If your JavaScript knowledge is a bit iffy, don't worry; you'll still be able to follow the logic. If you need a hand, just submit a comment, and I'll try to help out.

HOW NOT TO USE AJAX

One method that has been around, and which I used myself back in the bad old days, is to simply load the `wp-load.php` file at the top of your PHP script. This lets you use WordPress functions, detect the current user and so on. But this is basically a hack and so is not future-proof. It is much less secure and does not give you some of the cool options that the WordPress system does.

How AJAX Works In WordPress Natively

Because AJAX is already used in WordPress' back end, it has been basically implemented for you. All you need to do is use the functions available. Let's look at the process in general before diving into the code.

Every AJAX request goes through the `admin-ajax.php` file in the `wp-admin` folder. That this file is named `admin-ajax` might be a bit confusing. I quite agree, but this is just how the development process turned out. So, we should use `admin-ajax.php` for back-end and user-facing AJAX.

Each request needs to supply at least one piece of data (using the `GET` or `POST` method) called `action`. Based on this action, the code in `admin-ajax.php` creates two hooks, `wp_ajax_my_action` and `wp_ajax_nopriv_my_action`, where `my_action` is the value of the `GET` or `POST` variable `action`.

Adding a function to the first hook means that that function will fire if a logged-in user initiates the action. Using the second hook, you can cater to logged-out users separately.

Implementing AJAX In WordPress

Let's build a rudimentary voting system as a quick example. First, create an empty plugin and activate it. If you need help with this part, look at the tutorial on [creating a plugin](#). Also, find your theme's single.php file, and open it.

PREPARING TO SEND THE AJAX CALL

Let's create a link that enables people to give a thumbs up to our articles. If a user has JavaScript enabled, it will use JavaScript; if not, it will follow the link. Somewhere in your single.php file, perhaps near the article's title, add the following code.

```
<?php
    $votes = get_post_meta($post->ID, "votes", true)
    $votes = ($votes == "") ? 0 : $votes;
?>
This post has <div id='vote_counter'><?php echo $votes ?></div>
votes<br />
<?php
    $nonce = wp_create_nonce("my_user_vote_nonce");
    $link = admin_url('admin-ajax.php?action=my_user_vote&post_id=' .
$post->ID . '&nonce=' . $nonce);
    echo '<a class="user_vote" data-nonce="' . $nonce . '" data-
post_id="' . $post->ID . '" href="' . $link . '">vote for this
article</a>';
?>
```

First, let's pull the value of the votes meta key related to this post. This meta field is where we will store the total vote count. Let's also make sure that if it doesn't exist (i.e. its value is an empty string), we show 0.

We've also created an ordinary link here. The only extra bit is a pinch of security, using nonces, to make sure there is no foul play. Otherwise, this is simply a link pointing to the `admin-ajax.php` file, with the action and the ID of the post that the user is on specified in the form of a query string.

To cater to JavaScript users, we have added a `user_vote` class, to which we will attach a click event, and a `data-post_id` property, which contains the ID of the post. We will use these to pass the necessary information to our JavaScript.

HANDLING THE ACTION WITHOUT JAVASCRIPT

If you click this link now, you should be taken to the `admin-ajax.php` script, which will output `-1`. This is because no function has been created yet to handle our action. So, let's get cracking!

In your plugin, create a function, and add it to the new hook that was created for us. Here's how:

```
add_action("wp_ajax_my_user_vote", "my_user_vote");
add_action("wp_ajax_nopriv_my_user_vote", "my_user_vote");

function my_user_vote() {

    if ( !wp_verify_nonce( $_REQUEST['nonce'], "my_user_vote_nonce") )
    {
        exit("No naughty business please");
    }

    $vote_count = get_post_meta($_REQUEST["post_id"], "votes", true);
    $vote_count = ($vote_count == '') ? 0 : $vote_count;
    $new_vote_count = $vote_count + 1;

    $vote = update_post_meta($_REQUEST["post_id"], "votes",
    $new_vote_count);

    if($vote === false) {
        $result['type'] = "error";
        $result['vote_count'] = $vote_count;
    }
    else {
        $result['type'] = "success";
        $result['vote_count'] = $new_vote_count;
    }

    if(!empty($_SERVER['HTTP_X_REQUESTED_WITH']) &&
    strtolower($_SERVER['HTTP_X_REQUESTED_WITH']) == 'xmlhttprequest')
    {
        $result = json_encode($result);
        echo $result;
    }
    else {
```

```
    header("Location: ".$_SERVER["HTTP_REFERER"]);  
}  
die();  
}  
  
function my_must_login() {  
    echo "You must log in to vote";  
    die();  
}
```

First of all, we've verified the nonce to make sure that the request is nice and legit. If it isn't, we simply stop running the script. Otherwise, we move on and get the vote count from the database; make sure to set it to 0 if there is no vote count yet. We then add 1 to it to find the new vote count.

Using the `update_post_meta()` function, we add the new vote count to our post. This function creates the post's meta data if it doesn't yet exist, so we can use it to create, not just update. The function returns `true` if successful and `false` for a failure, so let's create an array for both cases.

I like to create these result arrays for all actions because they standardize action handling, giving us good debugging information. And, as we'll see in a second, the same array can be used in AJAX and non-AJAX calls, making error-handling a cinch.

This array is rudimentary. It contains only the type of result (error or success) and the vote count. In the case of failure, the old vote count is used (discounting the user's vote, because it was not added). In the case of success, we include the new vote count.

Finally, we detect whether the action was initiated through an AJAX call. If so, then we use the `json_encode()` function to prepare the array for our JavaScript code. If the call was made without AJAX, then we simply send the user back to where they came from; obviously, they should be shown the updated vote count. We could also put the array in a cookie or in a session variable to return it to the user the same way, but this is not important for this example.

Always end your scripts with a `die()` function, to ensure that you get back the proper output. If you don't include this, you will always get back a `-1` string along with the results.

The function to handle logged-out users is obviously poor, but it is meant merely as an example. You can expand on it by having it redirect the user to a registration page or by displaying more useful information.

ADDING JAVASCRIPT TO THE MIX

Because we've now handled the user's action using regular methods, we can start building in the JavaScript. Many developers prefer this order because it ensures graceful degradation. In order for our system to use AJAX, we will need to add jQuery, as well as our own JavaScript code. To do this, WordPress-style, just go to your plugin and add the following.

```
add_action( 'init', 'my_script_enqueuer' );

function my_script_enqueuer() {
    wp_register_script( "my_voter_script", WP_PLUGIN_URL.'/my_plugin/
my_voter_script.js', array('jquery') );

    wp_localize_script( 'my_voter_script', 'myAjax', array( 'ajaxurl'
=> admin_url( 'admin-ajax.php' ) ));

    wp_enqueue_script( 'jquery' );
    wp_enqueue_script( 'my_voter_script' );
}
```

This is the WordPress way of including JavaScript files. First, we register the JavaScript file, so that WordPress knows about it (so make sure to create the file and place it somewhere in the plugin). The first argument to the `wp_register_script()` function is the “handle” of our script, which is a unique identifier. The second is the location of the script. The third argument is an array of dependencies. Our script will require jQuery, so I have added it as a dependency. WordPress has already registered jQuery, so all we needed to add was its handle. For a detailed list of the scripts that WordPress registers, look at the [WordPress Codex](#).

Localizing the script is not strictly necessary, but it is a good way to define variables for our script to use. We need to use the URL of our `admin-ajax.php` file, but because this is different for every domain, we need to pass it to the script. Instead of hard-coding it in, let’s use the `wp_localize_script()` function. We add the script handle as the first argument, an object name as the second argument, and we can add object members as an array in the third parameter. What this all boils down to is that, in our `my_voter_script.js` file, we will be able to use `myAjax.ajaxurl`, which contains the URL of our `admin-ajax.php` file.

Once our scripts have been registered, we can actually add them to our pages by enqueueing them. We don't need to follow any particular order; WordPress will use the correct order based on the dependencies.

Once that's done, in the `my_voter_script.js` JavaScript file, paste the following code:

```
jQuery(document).ready( function() {

    jQuery(".user_vote").click( function() {
        post_id = jQuery(this).attr("data-post_id")
        nonce = jQuery(this).attr("data-nonce")

        jQuery.ajax({
            type : "post",
            dataType : "json",
            url : myAjax.ajaxurl,
            data : {action: "my_user_vote", post_id : post_id, nonce:
nonce},
            success: function(response) {
                if(response.type == "success") {
                    jQuery("#vote_counter").html(response.vote_count)
                }
                else {
                    alert("Your vote could not be added")
                }
            }
        })
    })

})
```

Let's go back to the basics. This would be a good time for those of us who are new to AJAX to grasp what is going on. When the user clicks the vote button without using JavaScript, they open a script and send it some data using the GET method (the query string). When JavaScript is used, it opens the page for them. The script is given the URL to navigate to and the same parameters, so apart from some minor things, from the point of view of the script being run, there is no difference between the user clicking the link and an AJAX request being sent.

Using this data, the `my_user_vote()` function defined in our plugin should process this and then send us back the JSON-encoded result array. Because we have specified that our response data should be in JSON format, we can use it very easily just by using the response as an object.

In our example, all that happens is that the vote counter changes its value to show the new vote count. In reality, we should also include some sort of success message to make sure the user gets obvious feedback. Also, the alert box for a failure will be very ugly; feel free to tweak it to your liking.

Conclusion

This concludes our quick tutorial on using AJAX in WordPress. A lot of functionality could still be added, but the main point of this article was to show how to properly add AJAX functionality itself to plugins. To recap, the four steps involved are:

1. Make the AJAX call;
2. Create the function, which will handle the action;
3. Add the function to the hook, which was dynamically created for us with the action parameter;
4. Create success handlers as needed.

As mentioned, make sure everything works well without JavaScript before adding it, so that the website degrades properly for people who have disabled it.

Keep in mind that, because we are using hooks, we can also tie existing WordPress functions to our AJAX calls. If you already have an awesome voting function, you could just tie it in after the fact by attaching it to the action. This, and the ease with which we can differentiate between logged-in states, make WordPress' AJAX-handling system very powerful indeed.

Better Image Management With WordPress

Daniel Pataki

With the advent of sophisticated and user-friendly content management systems like WordPress, textual content has become increasingly easier to manage. The architecture of these systems aims to deliver a well-formed code foundation; this means that if you are a good writer, then your content will be just as awesome as the structure and quality of the code that runs it.



However, media handling is, by nature, not the greatest. In many cases, images are used merely to make the website look good, not to supplement the content. Little care is usually taken to make these elements as useful as their textual counterparts. They are often tacked on as an afterthought; the owner thinks, “If all of my posts have an image, surely I should find something quickly for this next one as well.”

Because **the content of images cannot be parsed by search engines**, making sure they are rich in meta information before publishing them is important. Here are a few ways to enrich your blog using some common sense, best practices and the power of WordPress.

Understanding And Using Images

To get the most out of your graphic content, you’ll need to be familiar with how they work in HTML. To put an image on a page, you would add an image tag, with the appropriate attributes, like so:

```

```

As you can see, the tag has three attributes that contain information about the image:

- `src` is the URL source of the image file;
- `alt`, or alternative, text is shown when an image can’t load (whether because of a loading error, text-only browser, etc.);
- `title` is the title attribute, where you can add a short description of the image, which will pop up after hovering over the image for a second.

The `src` and `alt` attributes are both required; the HTML is invalid without them. However, HTML is not a strict language. Your post will still render just fine if you leave out the `alt` text, which is one of the negative aspect of loose languages: it doesn't force best practices.

WHY USE ALT AND TITLE ATTRIBUTES?

The most useful aspect of `alt` and `title` is that they allow you to add text-based information to an element on your website that would otherwise be invisible to search engines. If you sell umbrellas, Google won't see that one particular image on your page is of the coolest umbrella it's ever seen. You'll have to add that information yourself.

Also, `alt` attribute can be a huge help to the disabled, because this is how they know what is in an image. So, **use the `title` attribute to write something snappy about the image, and use the `alt` attribute to describe it.** Sticking with our umbrella example, the incorrect way to do this would be:

```

```

And the correct way would be:

```

```

Remember, the `alt` attribute is descriptive not only for the visually impaired, but for Google as well. Your website might even rank better if it's image-heavy.

While not as critical, it is probably worth optimizing the file name as well. The name `o290rjf.jpg` won't get in the way showing the image, but `super-sleek-umbrella.jpg` is a parsable bit of text, and there is a chance that some search engines would take it into account. Also, if someone downloads the image from your website, they will be able to find it more easily in their "Downloads" folder. And user satisfaction translates into more visits.

ADDING IMAGES PROPERLY WITH WORDPRESS

WordPress allows you to attach media to posts very easily through the "Add media" modal window, which you can access by clicking one of the icons over the editing toolbar in a post. You can select multiple images and upload them to the post with a click. Because this is so easy, adding the meta attributes is often overlooked and regarded as a hassle.

When uploading images, make sure to fill out the form which is displayed. Add the `title` and `alt` attribute at a bare minimum, but also consider filling in the caption and description fields. If you want a short, nicely formatted caption to appear under the image (which is a good idea), type one in. We'll look later at harnessing the description field, so writing a paragraph or so about the image might be a good idea.

Once done, all you need to do is insert the image, and the correct HTML tag will be plopped in by WordPress automatically. By taking an extra minute, you will have added a sizable bit of text to your image, making it SEO-friendly and in turn making your website that much more informative. If this is all you have time for, then you have done the most important step. But let's look at some more advanced image-handling techniques.

Managing Image Sizes

If you display an image at a size of 450×300 pixels, then having an image file of roughly the same size is a good idea. If the source file is 2250×1500 pixels, the image will show up just fine, but instead of loading a 50 KB image, you would be loading a 500 KB image, while achieving the same effect.

WordPress is super-smart, though, taking care of this for you by churning out different sizes for each image you upload. See the dimensions it creates by going to the media settings in the back end. You can modify these once you have the final layout, which I would advise.

For an image-centric website, you might want to add a couple of more sizes, to make sure you never serve an image that is bigger than needed. By putting the following code in your theme's functions.php file, you create two extra sizes:

```
add_image_size( 'large_thumb', 75, 75, true );  
add_image_size( 'wider_image', 200, 150 );
```

The first line defines an image that is cropped to exactly 75×75 pixels, and the second line defines an image whose maximum dimension is 200×150, while maintaining the aspect ratio. Note the name given in the first argument of the function, because you will be referring to it when retrieving the images, which you can do like so:

```
wp_get_attachment_image_src( 325, 'wider_image' );
```

The first argument is the ID of the attachment that we want to show. The second argument is the size of the image.

REBUILDING YOUR THUMBNAILS

If you have been blogging for a while now, you probably have a ton of images. Adding an image size now will not create new thumbnails of your existing images. If you specify an image size—for example, our `wider_image` format—WordPress will fetch a resolution that is close to it, but it won't create a thumbnail especially for this size.

Using a plug-in, however, you can go back and regenerate the thumbnails to make sure that all of the images are optimized, thus **minimizing server load**. I can personally vouch for [AJAX Thumbnail Rebuild](#), which goes through all of your images and regenerates the selected sizes for you.

Using Featured Images

A featured image can capture the message of a post. Featured images have many uses: for adding flare in a magazine-style layout, underlining a point made in an article, or substituting for an article's title (in the sidebar, for example).

Featured images have been built into WordPress since version 2.9, so you don't need any special plug-ins. If you are using the new default WordPress theme, TwentyTen, or the cutting-edge TwentyEleven (which is right now only in development versions), then featured images are already enabled. Otherwise, you might need to switch them on manually. To enable them, just open your theme's `functions.php` file, paste in the code below, and voila!

```
add_theme_support( 'post-thumbnails' );  
set_post_thumbnail_size( 115, 115 )
```

The first line of code tells WordPress to enable featured images, while the second sets the default size for featured thumbnails. The `set_post_thumbnail_size()` bit works just like the `add_image_size()` function we looked at above. You can give it a width, a height and, optionally, a third boolean parameter (`true` or `false`) to indicate whether it should be an exact crop.

Once that's done, go into the back end and edit a post. You should see a featured image widget in the right sidebar; click it to add an image. Or navigate to the media section of the post, view an image's details, and click the "Use as featured image" link.

The only thing left to do is make these featured images show up! You will need to edit the code for the loop in your theme's files, which is usually found in `index.php` or in some cases in `loop.php`. Look for something like this:

```
<?php while ( have_posts() ) : the_post(); ?>
The code to display a post is inside here, it can be quite long
<?php endwhile; ?>
```

Wherever you want to show the images, add the following in the loop:

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

In some cases, you may want to show the featured image at a size different than the default. If so, you can pass the desired size as an argument, like so:

```
<?php the_post_thumbnail("wider_image"); ?>
```

You can name a size that you have previously created using `add_image_size()`, as I have done above, or you can use an array to specify a size on the fly: `array(225, 166)`.

Creating Galleries

The easiest way to show multiple images in a post is to upload the images to the post and then use the gallery short code to display them all.

Gallery Settings

Link thumbnails to: ☐ Image File ☒ Attachment Page

Order images by:

Order: ☒ Ascending ☐ Descending

Gallery columns:

[Update gallery settings](#)

Simply open the “Upload/insert” media screen, click on “Galleries,” and scroll down to the gallery settings. Make sure the links point to the attachment pages (more on this later), and then insert the gallery. Now, thumbnails of all the images you have uploaded to that post will be displayed, each linked to its attachment page.

INCLUDING AND EXCLUDING IMAGES

You can easily include images from other posts or exclude certain images from the current post by modifying the gallery short code. If you switch the editor to the HTML view, you should see `[gallery]` where the gallery would show up. You can add options to it using the following format:

```
[ gallery option_1="value" option_2="value" ].
```

To include a specific image, you will need to know its attachment ID. You can find that by going to the “Media” section of the WordPress admin area, finding the image you need, hovering over it, and reading the target from the URL or status bar. It should be something like `http://webtastique.net/wp-admin/media.php?attachment_id=92&action=edit`. The number after `attachment_id` is what you need.

You can include multiple items like so: `[gallery include="23,39,45"]`. And exclude items the same way: `[gallery exclude="87,11"]`.

EXCLUDING THE FEATURED IMAGE

Sometimes you will want to use all of the images attached to a post except the featured one. You could find the ID of the image and enter it in the exclude options of the gallery shortcode every time, but that would be a hassle (especially if you change the featured image later). Let’s automate this.

Regrettably, the only way to do this is by replacing a core function in WordPress with our own, using the `remove_shortcode()` and `add_shortcode()` functions. The large chunk of code below may be off-putting, but implementing it is as easy as copying, pasting and adding two lines of code. The reason we need to add all this is that we can’t just go around editing a WordPress core file; we need to replace core functions with built-in functions.

First, open your theme’s `functions.php` file (if it doesn’t exist, simply create it), and add the following code to it:

```
// remove the WordPress function
remove_shortcode('gallery', 'gallery_shortcode');
// add our own replacement function
add_shortcode('gallery', 'myown_gallery_shortcode');
```

This removes the `gallery_shortcode()` function that WordPress uses to display galleries and replaces it with our own function, called `myown_gallery_shortcode()`.

The code below is almost exactly the same as the default, but we are adding a line to exclude our featured image. Paste the code below into the `functions.php` file, and then read the explanation further down:

```

function myown_gallery_shortcode($attr) {
    global $post, $wp_locale;

    static $instance = 0;
    $instance++;

    // Allow plugins/themes to override the default gallery
    template.

    $output = apply_filters('post_gallery', '', $attr);
    if ( $output != '' )
        return $output;

    // We're trusting author input, so let's at least make sure it
    looks like a valid orderby statement
    if ( isset( $attr['orderby'] ) ) {
        $attr['orderby'] = sanitize_sql_orderby( $attr['orderby'] );
        if ( !$attr['orderby'] )
            unset( $attr['orderby'] );
    }
    extract(shortcode_atts(array(
        'order'      => 'ASC',
        'orderby'    => 'menu_order ID',
        'id'         => $post->ID,
        'itemtag'    => 'dl',
        'icontag'    => 'dt',
        'captiontag' => 'dd',
        'columns'    => 3,
        'size'       => 'thumbnail',
        'include'    => '',
        'exclude'    => $default_exclude
    ), $attr));

```

```

$default_exclude = get_post_thumbnail_id($post->ID);
$exclude .= ",".$default_exclude;

$id = intval($id);
if ( 'RAND' == $order )
    $orderby = 'none';

if ( !empty($include) ) {
    $include = preg_replace( '/[^0-9,]+/', '', $include );
    $_attachments = get_posts( array('include' => $include,
'post_status' => 'inherit', 'post_type' => 'attachment',
'post_mime_type' => 'image', 'order' => $order, 'orderby' =>
$orderby) );

    $attachments = array();
    foreach ( $_attachments as $key => $val ) {
        $attachments[$val->ID] = $_attachments[$key];
    }
} elseif ( !empty($exclude) ) {
    $exclude = preg_replace( '/[^0-9,]+/', '', $exclude );
    $attachments = get_children( array('post_parent' => $id,
'exclude' => $exclude, 'post_status' => 'inherit', 'post_type' =>
'attachment', 'post_mime_type' => 'image', 'order' => $order,
'orderby' => $orderby) );
} else {
    $attachments = get_children( array('post_parent' => $id,
'post_status' => 'inherit', 'post_type' => 'attachment',
'post_mime_type' => 'image', 'order' => $order, 'orderby' =>
$orderby) );
}

if ( empty($attachments) )
    return '';

```



```

if ( is_feed() ) {
    $output = "\n";
    foreach ( $attachments as $att_id => $attachment )
        $output .= wp_get_attachment_link($att_id, $size, true) .
"\n";
    return $output;
}

$itemtag = tag_escape($itemtag);
$captions = tag_escape($captions);
$columns = intval($columns);
$itemwidth = $columns > 0 ? floor(100/$columns) : 100;
$float = is_rtl() ? 'right' : 'left';

$selector = "gallery-{$instance}";

$output = apply_filters('gallery_style', "
    <!--          #{$selector} {                margin:
auto;                }                #{$selector} .gallery-item {
float: {$float};                margin-top: 10px;
text-align: center;                width: {$itemwidth}
%;                }                #{$selector} img {                border:
2px solid #cfcfcf;                }                #{$selector} .gallery-
caption {                margin-left: 0;                } -->

    <!-- see gallery_shortcode() in wp-includes/media.php -->
<div id=\"\$selector\" class=\"gallery galleryid-{$id}\">
\");

    $i = 0;

    foreach ( $attachments as $id => $attachment ) {
        $link = isset($attr['link']) && 'file' == $attr['link'] ?
wp_get_attachment_link($id, $size, false, false) :
wp_get_attachment_link($id, $size, true, false);

```

```

$output .= "<{$itemtag} class='gallery-item'>";
$output .= "
    <{$icontag} class='gallery-icon'>
        $link
    <!--{$icontag}-->";
if ( $captiontag && trim($attachment->post_excerpt) ) {
    $output .= "
        <{$captiontag} class='gallery-caption'>
            " . wptexturize($attachment->post_excerpt) . "
        <!--{$captiontag}-->";
    }
$output .= "<!--{$itemtag}-->";
if ( $columns > 0 && ++$i % $columns == 0 )
    $output .= '<br style="clear: both;">';
}

$output .= "
    <br style="clear: both;"></div>

\n";

return $output;
}

```

In lines 18 through 29, WordPress is determining the default attributes. By default, nothing is excluded; so under this bit of code, we add two more lines, and that's it:

```

$default_exclude = get_post_thumbnail_id($post->ID);
$exclude .= ", ".$default_exclude;

```

The first line here finds the featured image of the post in question, while the second appends it to the exclude list. The rest of the code is the same as the default.

Using Attachment Pages



Probably my favorite double-meaning logos, the way that something as complex as a person hitting a golf ball (complete with flexed elbows) could be transformed into another pretty detailed object, a spartan warriors head.

COMMENTS	LOGO AUTHOR	COLOR PALETTE
<i>0 Comments</i>	<i>lexlogo40513</i>	

In my opinion, attachment pages are the single best tool for creating richer, more informative image-driven websites. They enable you to create separate pages for each and every media item you have, affording you considerably more power in managing them.

Attachment pages exist in WordPress by default, but people seem to rarely link to them. Linking thumbnails directly to their full-sized versions (i.e. without the website framework) is much more common. I am not a huge fan of this because it throws the user into a completely new environment without prior warning. Attachment pages allow you to show the user a wealth of information about the image; and for those who need a bigger version, you can display download links for different sizes.

ENABLING ATTACHMENT PAGES

As stated, you don't need to do anything to enable attachment pages. Just make sure to link your images to them instead of to the original files. For galleries, link to the attachment page using the radio buttons before inserting them. When inserting a single image, point the link's URL field to the "Post URL" by clicking the relevant button below it.

STYLING ATTACHMENT PAGES

If your theme doesn't have an `attachment.php` file, then `single.php` will handle the display of attachment pages by default. If you have a decent theme, chances are this will work fine without your needing to touch any code. When clicking on an image, you should arrive on a page that shows the title and description of the image and the image itself.

To add additional information to this page, you will need an `attachment.php` file. I suggest duplicating `single.php` and going from there, because in most cases it will have most of what you need.

Adding Image Data

To make the attachment pages more informative, add a bunch of meta data to your images. To help with this, I have created a plug-in especially for Smashing Magazine readers, which you can download from the WordPress Plugins page, or just search for “media custom fields” in WordPress’ back end where you “Add new” plug-ins.

This plug-in lets you create your own custom fields, like the photographer’s name, coordinates, color palette, etc. What you add is up to you. You can easily manage all of the information on the plug-in’s admin page.

In the video below, I’ll walk you through how I did this on my own blog. You’ll learn about basic usage and see an example.

“[Better Media Management With WordPress Using the Media Custom Fields Plugin](#),” by [Daniel Pataki](#).

Creative Attachment Page Uses

DOWNLOAD LINKS FOR IMAGE SIZES

Using the `add_image_size()` function mentioned above, you could create five or six image sizes and show Flickr-style download options that allow users to **choose the dimensions** of their preference. This is helpful when showcasing desktop backgrounds and large photographs. So, let’s do that:

```
// If we are on an attachment page, the $post object will be
available and the $post->ID variable will contain the ID of the
image in question.

// Find the meta data field from the postmeta table, which contains
the sizes for a given image. This is the '_wp_attachment_metadata'
field, which contains a serialized array. Take care, because if you
use 'true' as the third parameter, the function will unserialize
the string for you, so that you don't need to do it.
$image_meta = get_post_meta( $post->ID, '_wp_attachment_metadata',
true);

// Put all the image sizes and file names into an array for ease of
use
$image_sizes = $image_meta['sizes'];
$image_sizes['original']['width'] = $image_meta['width'];
$image_sizes['original']['height'] = $image_meta['height'];
$image_sizes['original']['file'] = $image_meta['file'];

// Display a list of links for these images
echo '
<h3>This image is available in the following formats</h3>
'
;
echo '
<ul>';

foreach ( $image_sizes as $size_name => $size ) {
    $url = wp_get_attachment_image_src( $post->ID, $size_name );
    $anchortext = $size['width'] . 'x' . $size['height'];
    echo "<a href=\"". $link[0] . "\">". $anchortext . "</a>";
}
echo '</ul>
'
;
```

Adding Color Palettes

By adding some creativity to the mix, you can come up with some nifty features. The screencast above and the code below shows you how to display color blocks of the dominant colors in each of your photos.

COMMENTS	LOGO AUTHOR	COLOR PALETTE
<i>0 Comments</i>	<i>lexlogo40513</i>	

To accomplish this, you will first need to create a custom field using the Media Custom Fields plug-in and name it something like “Color Palette.” Remember to look at the field name that the system generates; it is displayed in parentheses next to the title you chose. It should be something like tqmcf_color-palette.

Once that’s done, edit the image you’d like, and add the following in the custom field: `color_1,color_2,color_3`, where `colors_x` should be hex values. In my case, I entered the following string:
`f0e9bf,e4dc99,000000`.

Open up the `attachment.php` file in a code editor. Wherever you want to display the colors, you’ll need to add something like this:

```
// Retrieve the field value from the database
$color_palette = get_post_meta( $post->ID, 'tqmfc_color-palette',
true );

// Turn the string into an array of values, where each value is one
of the colors
$colors = explode( ',', $color_palette );

echo '
<h2>Logo Colors</h2>
'
;

// Loop through all the colors and create the color blocks, which
will actually be links pointing the the color's page on
Colourlovers.com
foreach ( $colors as $color ) {
    $link = 'http://www.colourlovers.com/color/ '.$color.'/';
    echo '<a class="color-block" style="background: #'.$color.';"
href="'.$link.'"></a>';
}
```

You will also need to style the link element so that it shows up. Because anchors are inline elements by default, if they have no content, they won't show up. Here's the CSS I used, but you'll need to change it to match your website:

```
.color-block {
    display: block;
    float: left;
    height: 20px;
    margin-right: 3px;
    width: 30px;
}
```

Conclusion

As you can see, even with minimal effort, you can create a much more robust system for storing and showing images. And with some copying and pasting, you can take it one step further.

The first and most important step is to add meta data like alt text to images, give them meaningful file names and so on. By doing so, you lay a foundation for any media management system. You can easily add other meta data to your files by using the [Media Custom Fields plugin](#) for WordPress.

With this foundation in place and a few simple code tweaks, you can show images based on any of the custom fields you wish, displaying relevant and interesting information about them. Creating download buttons for multiple sizes and creating multiple color palettes are only the tip of the iceberg. The techniques showcased here can be used for so much more!

Using HTML5 To Transform WordPress' TwentyTen Theme

Richard Shepherd

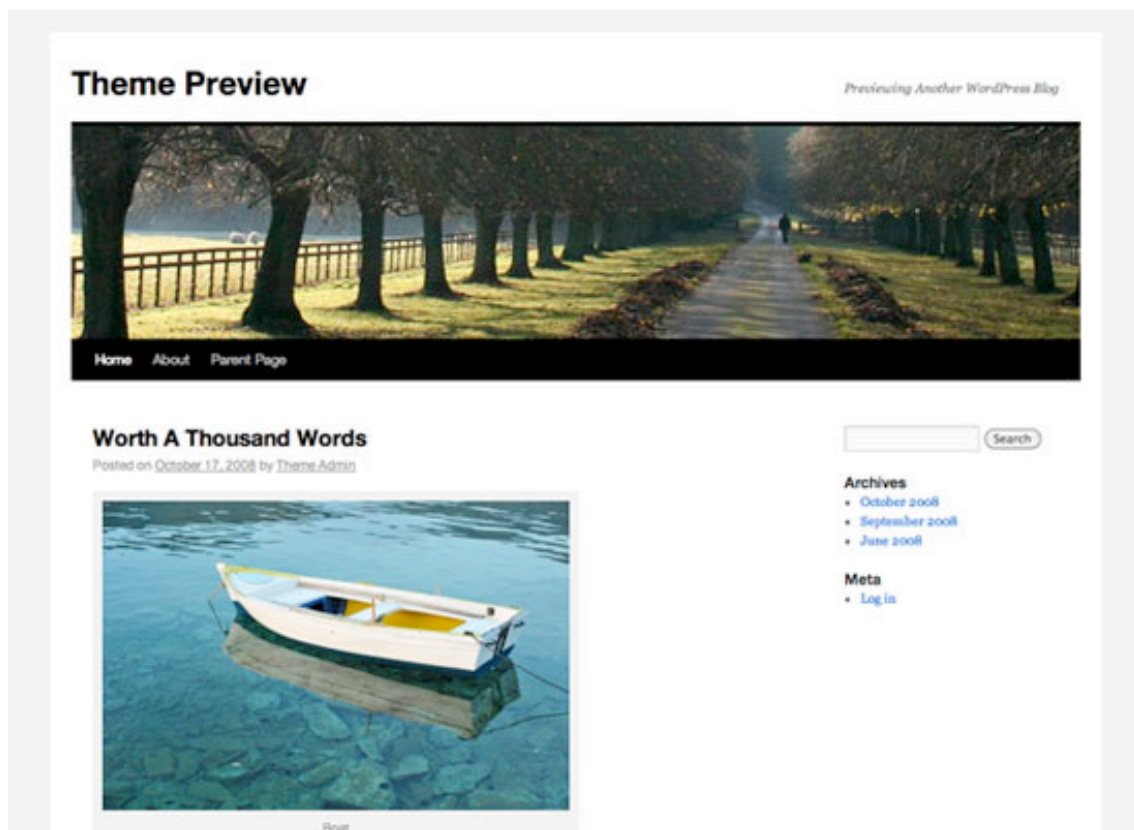
Last year, WordPress launched arguably its biggest update ever: WordPress 3.0. Accompanying this release was the brand new default theme, TwentyTen, and the promise of a new default theme every year. Somewhat surprisingly, TwentyTen declares the HTML5 doctype but doesn't take advantage of many of the new elements and attributes that HTML5 brings.



Now, HTML5 does many things, but you can't just add `<!doctype html>` to the top of a document and get excited that you're so 2011. Mark-up, as they say, is meaning, and HTML5 brings a whole bunch of meaning to our documents.

In a recent survey by Chris Coyier over at CSS-Tricks, almost two thirds of respondents said they would not use HTML5 in new projects. In a similar survey by Smashing Magazine the results were almost identical: only 37% of voters said they use HTML5. This is depressing reading. Perhaps developers and designers are scared off by cross-browser incompatibility and the chore of learning new mark-up. The truth is that with a pinch of JavaScript, HTML5 can be used safely today across all browsers, back to IE6.

WordPress seems to sympathize with the majority of CSS-Tricks' readers. TwentyTen is a fine theme that already validates as HTML5; but in order to cater to users without JavaScript, it has to forgo a large chunk of HTML5 elements. The reason? Our old friend Internet Explorer doesn't support most of them prior to version 9.



The default TwentyTen WordPress Theme.

For example, you've probably already heard of the `<section>` and `<article>` tags, both of which are champing at the bit to be embedded in a WordPress template. But to use these HTML5 elements in IE8 (and its predecessors), you need JavaScript in order to create them in the DOM. If you don't have JavaScript, then the elements can't be styled with CSS. Turn off JavaScript and you turn off the styling for these elements; invariably, this will break the formatting of your page.

I assume that WordPress decided to exclude these problematic tags so that its default theme would be supported by all browsers — not just those with JavaScript turned on.

While I understand this decision, I also think it's a mistake. Three core technologies make the Web work: HTML, CSS and JavaScript. All desktop browsers support them (to some degree), so if any one of them off is disabled the user will have to expect a degraded experience. JavaScript is now fundamental to the user experience and while we should support users who turn off JavaScript, or have it turned off for them and have no chance to turn it on again as they don't have the right to do so, I question just how far we should support them.

WHY USING JAVASCRIPT MAKES SENSE

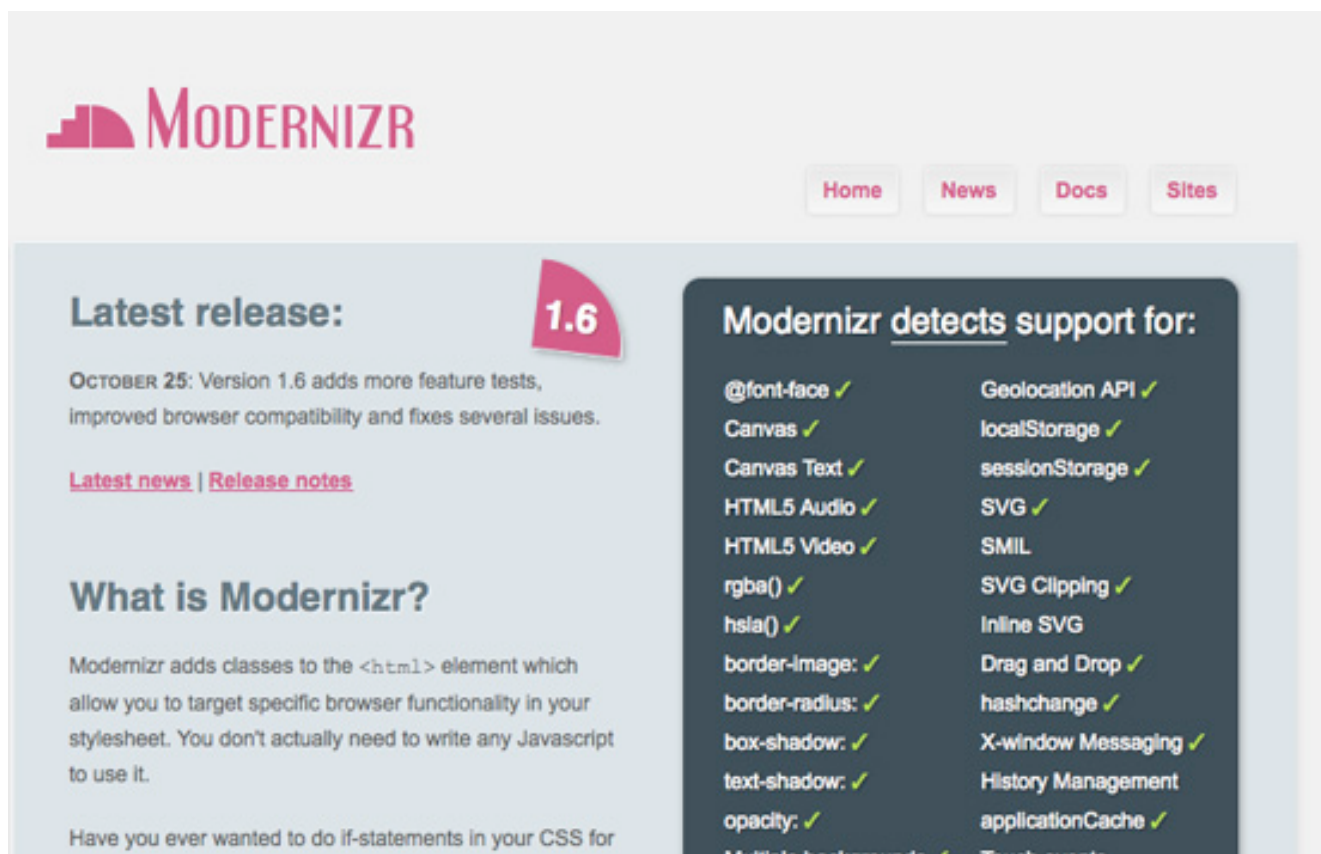
Yahoo gives compelling evidence that less than 1.5% of its users turn off JavaScript. My own research into this, ably assisted by Greig Daines at eConversions, puts the figure below 0.5% (based on millions of visitors to a UK retail website).

Whilst it's true that JavaScript should be separated from a site's content, design and structure the reality is no longer black and white. I strongly believe that the benefits and opportunities HTML5 brings, together with related technologies such as CSS3 and media queries (both of which sometimes rely on JavaScript for cross-browser compatibility), is more than enough reason to use JavaScript to 'force' new elements to work in Internet Explorer. I am a passionate advocate for standards-based design that doesn't rely on JavaScript; HTML5 is the one structural exception.

Yes, we should respect a user's decision to deactivate JavaScript in their browser. However, I don't believe that this is a good enough reason for not using modern technologies, which would provide the vast majority of users with a richer user experience. After all, in the TwentyTen example, if the theme had HTML5 tags in it, everything would look fine in modern browsers (latest versions of Safari, Firefox, Opera, Chrome and IE9), with or without JavaScript.

If the browser is IE6 – IE8, and JavaScript is turned off, then users would see the content but it will not be styled correctly. If the content would not be displayed at all, we'd have a completely different discussion. If you are still not convinced, I will briefly discuss another option for those who absolutely must support users with JavaScript turned off.

To make TwentyTen play fair with IE, I suggest Remy Sharp's HTML5 shim or, if you want to sink your teeth into CSS3, Modernizr. Modernizr not only adds support for HTML5 elements in IE but also tells you which CSS3 properties are supported by the user's browser by adding special classes to the `<html>` element.



The screenshot shows the Modernizr website homepage. At the top left is the Modernizr logo, which consists of a stylized 'M' made of steps followed by the word 'MODERNIZR' in a bold, sans-serif font. To the right of the logo are four navigation buttons: 'Home', 'News', 'Docs', and 'Sites'. Below the navigation is a main content area. On the left, there's a section titled 'Latest release:' with a red badge showing '1.6'. The text below says 'OCTOBER 25: Version 1.6 adds more feature tests, improved browser compatibility and fixes several issues.' and includes links for 'Latest news' and 'Release notes'. Below this is a section titled 'What is Modernizr?' with a paragraph explaining that Modernizr adds classes to the `<html>` element to target specific browser functionality. On the right side of the main content area is a dark blue box titled 'Modernizr detects support for:'. It contains a list of features with green checkmarks indicating support: @font-face, Canvas, Canvas Text, HTML5 Audio, HTML5 Video, rgba(), hsla(), border-image, border-radius, box-shadow, text-shadow, opacity, Multiple backgrounds, Geolocation API, localStorage, sessionStorage, SVG, SMIL, SVG Clipping, Inline SVG, Drag and Drop, hashchange, X-window Messaging, History Management, applicationCache, and Touch events.

Modernizr.js

So, let's assume you've rightly banished non-JavaScript users with a polite message in a `<noscript>` tag. We can now start tinkering under the hood of TwentyTen to bring some more HTML5 to WordPress.

Upgrading To HTML5

TwentyTen gets a number of things spot on. First of all, it declares the right doctype and includes the abbreviated meta charset tag. It also uses other semantic goodness like Microformats and great accessibility features like WAI-ARIA. But we can go further.

Important notes:

- I am referencing the HTML generated at <http://wp-themes.com/twentyten/>, rather than the simple “Hello World” clean installation of WordPress 3.
- For this article, I'll be editing the files directly in the `/wp-content/themes/twentyten/` directory. I've provided all the updated HTML5 theme source files for you to download from TwentyTen Five.
- Line numbers may change over time, so when I reference one, I'll usually say “on or around line...” The version of WordPress at the time of writing is 3.0.4.

ARTICLES

One of the more confusing parts of the HTML5 spec is the `<section>` and `<article>` tags. Which came first, the chicken or the egg? The easiest way to remember is to refer to the specification. The HTML5 spec may be dry at the best of times, but its explanation of articles will always point you in the right direction:

The article element represents a self-contained composition in a document, page, application, or site and that is, in principle, independently distributable or reusable, e.g. in syndication.

If the piece of content in question can be, and most likely will be, syndicated by RSS, then there's a good chance it's an `<article>`. A blog post in WordPress fits the bill perfectly.

On the TwentyTen home page, we get the following HTML:

```
<div id="post-19">
...
</div>
```

Semantically this means very little. But with the simple addition of an `article` tag, we're able to transform it into mark-up with meaning.

```
<article id="post-19">
...
</article>
```

Note that we retain the `id` to ensure that this `<article>` remains unique.

To make this change in the TwentyTen theme, open `loop.php`, which is in `/wp-content/themes/twentyten/`. On or around line 61, you should find the following code:

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

We'll need to change that `<div>` to an `<article>`, so that it reads:

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

And then we close it again on or around line 97, so that...

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

... becomes:

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

There are also instances on lines 32, 101 and 124. Opening some of the other pages in the theme, for example `single.php`, and making the same change is worthwhile. Thus, line 22 in `single.php` would change from...

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

... to

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

And line 55 would change from...

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

... to:

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

So far, so good. These are simple changes, but they already serve to overhaul the semantics of the website.

TIME AND DATE

According to the HTML5 spec:

The <time> element either represents a time on a 24-hour clock, or a precise date on the proleptic Gregorian calendar, optionally with a time and a time-zone offset.

This means we can give the date and time of an article's publication more context with HTML5's `<time>` tag. Look at the code that WordPress generates:

```
<a href="http://wp-themes.com/?p=19" title="4:33 am"
rel="bookmark"><span>October 17, 2008</span></a>
```

We can add meaning to our mark-up by transposing this to:

```
<a href="http://wp-themes.com/?p=19" title="4:33 am"
rel="bookmark"><time datetime="2008-10-17T04:33Z"
pubdate>October 17, 2008</time></a>
```

This time is now machine-readable, and the browser can now interact with the date in many ways should we so wish. I've also added the boolean attribute `pubdate`, which designates this as the date on which the article or content was published.

Time in the `datetime` attribute is optional, but because WordPress includes it when you post an article, we can too. Implementing this in TwentyTen requires us to dig a little deeper. In `loop.php`, the following function on or around line 65 calls for the date to be included:

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

To make our HTML5 changes, let's head over to `/wp-content/themes/twentyten/` and open `functions.php`. On or around line 441, you'll see this:

```
function twentyten_posted_on() {
printf( __( '<span>Posted on</span> %2$s <span>by</span> %3$s',
'twentyten' ),
meta_prep_meta_prep_author',
sprintf( '<a href="%1$s" title="%2$s" rel="bookmark"><span>%3$s</
span></a>',
get_permalink(),
esc_attr( get_the_time() ),
get_the_date()
),
```

If you don't know what that means, don't worry. We're focusing on the [sprintf](#) function, which basically takes a string and inserts the variables that are returned by the three functions listed: that is, `get_permalink()`, `get_the_time()` and `get_the_date()` are inserted into `%1$s`, `%2$s` and `%3$s`, respectively.

We need to change how the date is formatted, so we'll have to add a fourth function: `get_the_date('c')`. WordPress will then return the date in Coordinated Universal Time (UTC) format, which is exactly what the `<time>` element requires. Our finished code looks like this:

```
printf( __( 'Posted on %2$s by %3$s', 'twentyten' ),
meta-prep meta-prep-author',
sprintf( '<a href="%1$s" rel="bookmark"><time datetime="%2$s"
pubdate>%3$s</time></a>',
get_permalink(),
get_the_date( 'c' ),
get_the_date()
),
```

I've included `get_the_date()` twice because we need two different formats: one for the `<time>` element and one that's displayed to the user. I've also removed `title="[time published]"` because that information is already included in the `<time>` element.

For more details on WordPress' date and time functions, check out:

- [Function Reference/get the time](#),
- [Formatting Date and Time](#).

FIGURES

A figure—for our purposes at least—is a piece of media that you upload in WordPress to embed in a post. The most obvious example would be an image, but it could be a video, too, of course. WordPress 3 is helpful enough to add captions to images when you first import the images, but it doesn't display those captions using the new HTML5 `<figure>` and `<figcaption>` tags.

The spec defines `<figure>` as follows:

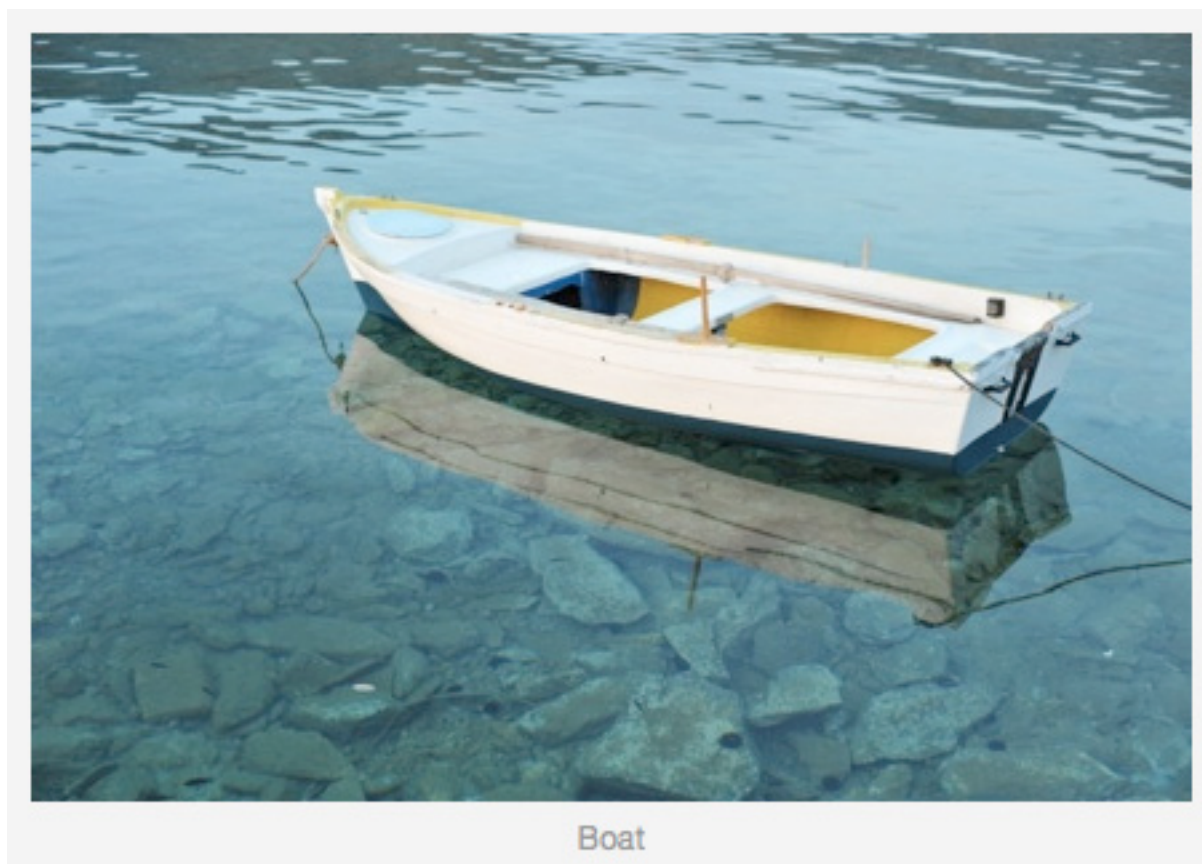
The figure element represents a unit of content, optionally with a caption, that is self-contained, that is typically referenced as a single unit from the main flow of the document, and that can be moved away from the main flow of the document without affecting the document's meaning.

And it defines `<figcaption>` like so:

The figcaption element represents a caption or legend for the rest of the contents of the figcaption element's parent figure element, if any.

Currently an image with a caption is rendered like this:

```
<div class="wp-caption" style="width: 445px;">
<p class="wp-caption-text">Boat</p>
</div>
```



A

WordPress image with a caption.

Changing this HTML to include HTML5 elements requires us to first look at `media.php` in the `/wp-includes/` directory, where this code is generated. On or around line 739, you'll find:

```
return '<div ' . $id . 'class="wp-caption ' . esc_attr($align) . '"  
style="width: ' . (10 + (int) $width) . 'px">  
' . do_shortcode( $content ) . '<p>' . $caption . '</p></div>';
```

To upgrade this to HTML5, we need to define a new function that outputs our `<figure>`-based HTML and assign this function to the same shortcode that calls `img_caption_shortcode()`. I've done this in `/wp-content/themes/twentyten/functions.php` by adding the following to the bottom of the file:

```

add_shortcode('wp_caption', 'twentyten_img_caption_shortcode');
add_shortcode('caption', 'twentyten_img_caption_shortcode');

function twentyten_img_caption_shortcode($attr, $content = null)
{

extract(shortcode_atts(array(
    'id'      => '',
    'align'    => 'alignnone',
    'width'    => '',
    'caption' => ''
), $attr));

if ( 1 > (int) $width || empty($caption) )
return $content;

if ( $id ) $idtag = 'id="' . esc_attr($id) . '" ';

return '<figure ' . $idtag . 'aria-describedby="figcaption_' .
$id . '" style="width: ' . (10 + (int) $width) . 'px">'
    . do_shortcode( $content ) . '<figcaption id="figcaption_' .
$id . '">' . $caption . '</figcaption></figure>';
}

```

First, we point the shortcodes for `wp-caption` and `caption` to our new function `twentyten_img_caption_shortcode()`. Then, we simply copy the original function from `media.php`, and change the last few lines to include our `<figure>` element. This now renders our `boat.jpg` example from above like so:

```
<figure id="attachment_64" style="width: 445px;">
</a>
<figcaption id="figcaption_attachment_64">Boat</figcaption>
</figure>
```

THE COMMENTS FORM

One of the biggest improvements introduced in HTML5 is how form fields work and respond to user input. We can take advantage of these changes by using HTML5 form elements in the default WordPress comments form in three ways:

1. We can set the text-input type to `email` and `url` for the relevant fields. This not only more accurately describes the input field, but also adds better keyboard functionality for the iPhone, for example.
2. We can add the boolean attribute required to our `required` form fields. This goes beyond WAI-ARIA's `aria-required='true'` because it invokes the browser's own `required` behavior.
3. We can add placeholder text to our form fields, a popular JavaScript method that is now handled in-browser. Placeholder text allows you to go into more detail about what information is required than a form label generally allows.

Before adding HTML, a typical comment input field might look like this:

```
<label for="email">Email</label> <span>*</span>
<input id="email" name="email" type="text" value=""
size="30" aria-required='true' />
```

After our HTML5 changes, it would look like this:

```
<label for="email">Email</label> <span>*</span>
<input id="email" name="email" type="email" value=""
size="30" aria-required='true'
placeholder="How can we reach you?" required />
```

To make these improvements in the code, we need to do two things. First, we need to change the HTML for the default fields (name, email address and website URL), and then we need to change it for the comment's `<textarea>`. We can achieve both of these changes with additional filters and custom functions.

To change the HTML for the default form fields, we need to add the following filter to the bottom of `functions.php`:

```
add_filter('comment_form_default_fields', 'twentytenfive_comments');
```

And then we have to create our custom function `twentytenfive_comments()` to change how these fields are displayed. We can do so by creating an array containing our new form fields and then returning it to this filter. Here's the function:


```

function twentytenfive_comments() {

$req = get_option('require_name_email');

$fields = array(
author' => '<p>' . '<label for="author">' . __( 'Name' ) . '</label>' .
' . ( $req ? '<span>*</span>' : '' ) .
<input id="author" name="author" type="text" value="" .
esc_attr( $commenter['comment_author'] ) . '" size="30"' .
$aria_req . ' placeholder = "What should we call you?"' . ( $req ? '
required' : '' ) . '</p>',

email'  => '<p><label for="email">' . __( 'Email' ) . '</label>' .
( $req ? '<span>*</span>' : '' ) .
<input id="email" name="email" type="email" value="" . esc_attr(
$commenter['comment_author_email'] ) . '" size="30"' . $aria_req . '
placeholder="How can we reach you?"' . ( $req ? ' required' : '' ) .
' /></p>',

url'    => '<p><label for="url">' . __( 'Website' ) . '</label>' .
<input id="url" name="url" type="url" value="" .
esc_attr( $commenter['comment_author_url'] ) . '" size="30"
placeholder="Have you got a website?" /></p>'

);
return $fields;
}

```

You can see here that each element in the form has a name in the `array()`: `author`, `email` and `url`. We then type in our custom code, which contains the new HTML5 form attributes. We have added placeholder text to each of the elements and, where `required`, added the boolean `required` attribute (and we need to check if the admin has made these fields required using the `get_option()` function). We've also added the correct input type to the inputs for author, email address and website URL.

Finally, we need to add some HTML5 to the `<textarea>`, which is home to the user's comments. We have to use another filter here, also in `functions.php`:

```
add_filter('comment_form_field_comment',  
'twentytenfive_commentfield');
```

We follow this with another custom function:

```
function twentytenfive_commentfield() {  
  
    $commentArea = '<p><label for="comment">' . _x( 'Comment',  
    'noun' ) . '</label><textarea id="comment" name="comment" cols="45"  
    rows="8" aria-required="true" required placeholder="What\'s on your  
    mind?" ></textarea></p>';  
  
    return $commentArea;  
  
}
```

This is more or less the same as the default `<textarea>`, except with `placeholder` and `required` attributes.

You can control exactly which fields appear in your form with these two filters, so feel free to add more if you want to collect more information.

Although relatively simple, these changes to the comment form provide additional (and useful!) features to users with latest-generation browsers. Look in Opera, Chrome (which doesn't yet support `required`) or Firefox 4 to see the results.

HEADER, NAVIGATION AND FOOTER

We finally get around to inserting the new `<header>`, `<nav>` and `<footer>` elements. Currently, the code in `/wp-content/themes/twentyten/header.php` looks more or less like this:

```
<div id="header">
<div id="masthead">
<div id="branding" role="banner">
...
</div><!-- #branding -->

<div id="access" role="navigation">
...
</div><!-- #access -->
</div><!-- #masthead -->
</div><!-- #header -->
```

It doesn't take a genius to see that we can easily make this HTML5-ready by changing some of those divs to include `<header>` and `<nav>`.

```
<header id="header">
<section id="masthead" >
<div id="branding" role="banner">
...
</div><!-- #branding -->

<nav id="access" role="navigation">
...
</nav><!-- #access -->
</section><!-- #masthead -->
</header><!-- #header -->
```

You can see that we've left the WAI-ARIA role of `navigation` assigned to the `nav` element—simply to offer the broadest possible support to all browsers and screen readers.

I have replaced the `#masthead` div with a `<section>` because all of the elements in this area relate to one another and are likely to appear in a document outline. It seems you could delete this section altogether and just apply 30 pixels of `padding-top` to the header to maintain the layout. I've maintained the elements' `ids` in case more than one of each are on the page—multiple headers, footers and navs (and more) are all welcome in HTML5.

While we're editing the header, we can introduce the new `<hgroup>` element. This element enables us to include multiple headings in a section of our document, while they would be treated as just one heading in the document outline. Currently, the code on or around line 65 in `header.php` looks like this:

```
<?php $heading_tag = ( is_home() || is_front_page() ) ? 'h1' :  
'div'; ?>  
<<?php echo $heading_tag; ?> id="site-title">  
<span>  
<a href="<?php echo home_url( '/' ); ?>" title="<?php echo  
esc_attr( get_bloginfo( 'name', 'display' ) ); ?>" rel="home"><?php  
bloginfo( 'name' ); ?></a>  
</span>  
</<?php echo $heading_tag; ?>>  
<div id="site-description"><?php bloginfo( 'description' ); ?></  
div>
```

We can edit this to include the `<hgroup>` tag, and also change `<div id="site-description">` to an `<h2>` element:

```

<hgroup>
<?php $heading_tag = ( is_home() || is_front_page() ) ? 'h1' :
'div'; ?>
<<?php echo $heading_tag; ?> id="site-title">
<span>
<a href="<?php echo home_url( '/' ); ?>" title="<?php echo
esc_attr( get_bloginfo( 'name', 'display' ) ); ?>" rel="home"><?php
bloginfo( 'name' ); ?></a>
</span>
</<?php echo $heading_tag; ?>>
<h2 id="site-description"><?php bloginfo( 'description' ); ?></h2>
</hgroup>

```

In `/wp-content/themes/twentyten/footer.php`, we have:

```

<div id="footer" role="contentinfo">
  <div id="colophon">

    <div id="site-info">
      <a href="<?php echo home_url( '/' ) ?>" title="<?php echo
esc_attr( get_bloginfo( 'name', 'display' ) ); ?>" rel="home">
        <?php bloginfo( 'name' ); ?>
      </a>
    </div><!-- #site-info -->
    <div id="site-generator">

    </div><!-- #site-generator -->

  </div><!-- #colophon -->
</div><!-- #footer -->

```

We can easily edit this to include a `<footer>` and another `<section>` element:

```
<footer role="contentinfo">
<section id="colophon">
...
<div id="site-info">
<a href="<?php echo home_url( '/' ) ?>" title="<?php echo
esc_attr( get_bloginfo( 'name', 'display' ) ); ?>" rel="home">
<?php bloginfo( 'name' ); ?>
</a>
</div><!-- #site-info -->

<div id="site-generator">
...
</div><!-- #site-generator -->

</section><!-- #colophon -->
</footer><!-- #footer -->
```

JAVASCRIPT AND CSS

As mentioned, we should include an HTML5 shim or Modernizr.js to make sure that all of our new elements render correctly in Internet Explorer prior to version 9. I added the following line to header.php:

```
<script src="<?php bloginfo('stylesheet_directory'); ?>/js/
Modernizr-1.6.min.js"></script>
```

A couple of things to note here. First, we no longer need `type="text/javascript"` because the browser will assume that a script is JavaScript unless it's told different. Secondly, we have to use the WordPress `bloginfo()` function to point the source URL to our theme directory.

Although we are including Modernizr partly to make sure that IE can deal with the new HTML5 elements, I am serving it to all browsers because of the CSS3-checking functionality it provides.

In style.css, we need to make sure that our HTML5 elements have a `display: block` attribute, because some older browsers will treat them as inline elements. For our purposes, the following line at the top of the CSS file will do:

```
header, nav, section, article, aside, figure, footer {  
display: block; }
```

While we're talking about CSS, remember that we can now remove `type="text/css"` from our `<link>` tags. The simplified code looks like this:

```
<link rel="stylesheet" href="<?php bloginfo( 'stylesheet_url' );  
?>" />
```

That should be enough for now. Remember, though, that changing the structure of the page by replacing older HTML elements with new ones might require some additional CSS.

We should let the small minority of users know that we've stopped supporting browsers that have JavaScript turned off. A polite message just below the opening `<body>` tag in header.php should suffice:

```
<noscript><strongdisplayed correctly. Please enable JavaScript before continuing...</  
strong></noscript>
```

Add some very basic styling in style.css to make this message unmissable.

```
/* A message for users with JavaScript turned off */
noscript strong {
display: block;
font-size: 18px;
line-height: 1.5em;
padding: 5px 0;
background-color: #ccc;
color: #a00;
text-align: center; }
```

Still Not Convinced? A Cross-Browser Alternative

There is another option for those of you who absolutely must support users with JavaScript turned off, as suggested by Christian Heilmann. Simply wrap your HTML5 elements with divs which share the same ID name. For example:

```
<article id="post-123">
...
</article>
```

becomes

```
<div class="article">
<article id="post-123">
...
</article>
</div>
```

Then it's just a case of adding `.article` to your article CSS definition. It might look like this:

```
.article,
article { display: block; background-color: #f7f7f7; }
```

It's worth noting that this adds another layer of markup to your code which isn't needed for most users. I'd only recommend it if non-JavaScript users are a significant proportion of your users and/or sales.

Final Thoughts

TwentyTen was a huge step forward for WordPress; and as a piece of HTML, it is a beacon of best practice. By including some simple JavaScript, we can now open up the theme to the world of HTML5—and the additional meaning and simpler semantic code that it offers.

While we've addressed a good number of new HTML5 elements in this article, it really is just a starting point and you can add many more yourself. For example, you could add headers and footers to individual posts, or you might like to add the new `<aside>` element.

DOWNLOAD TWENTYTEN WITH HTML5

To complement this article, I have created a new version of TwentyTen, with the HTML5 elements we have discussed. Download this theme from [TwentyTen Five](#).



TWENTYTEN FIVE

<Bringing HTML5 to WordPress>

Upgrade WordPress to HTML5! Download and install TwentyTen Five, or use it as a framework for your own HTML5 themes. It's FREE!

[DOWNLOAD NOW](#)

What is it?

Built to accompany an [article in Smashing Magazine](#), TwentyTen Five is an HTML5 upgrade of the default WordPress TwentyTen theme.

It's completely free, easy to build on, and brings brand new HTML5 elements and functionality to WordPress.

```
/**
 * Prints HTML with meta information for the current post-date/time and author.
 *
 * @since Twenty Ten Five 1.0
 */
function twentyten_posted_on() {
    printf( __( 'Posted on %2$s by %3$s', 'twentyten' ),
        'meta-prep meta-prep-author',
        sprintf( 'a href="%1$s" rel="bookmark"><time datetime="%2$s" pubdate=
get_permalink(),
get_the_date('c'),
get_the_date()
    ),
    sprintf( '<span class="author vcard"><a class="url fn n" href="%1$s" title=
get_author_posts_url( get_the_author_meta( 'ID' ) ),
```

The Authors

Daniel Pataki

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

Peter Wilson

Peter Wilson is a Web developer based in Melbourne, Australia, and started making Websites in 1994. Peter co-founded web production studio [Soupgiant](#) in 2009 and forms opinions on all things web at [Big Red Tin](#).

Ryan Olson

Ryan is a front-end developer who believes in an enjoyable web for all and created [BrowsingBetter](#). He loves WordPress, jQuery, learning new web skills, and short walks far from beaches. Check out his newest project [TextMateUser](#) and tweet with him [@ryanolson](#).

Sawyer Hollenshead

Sawyer Hollenshead is a web designer and digital entrepreneur — [Freelancing](#) all day and running [Shaken & Stirred Web](#) all night. Follow him at [@sawyerh](#).

Thiemo Fetzner

Thiemo Fetzner is pursuing a PhD in Economics at the London School of Economics. He has been publishing on web development and data analysis for more than 10 years in German print and online magazines such as Dr.Web, his own website Devmag and on his blog Freigeist.