

A



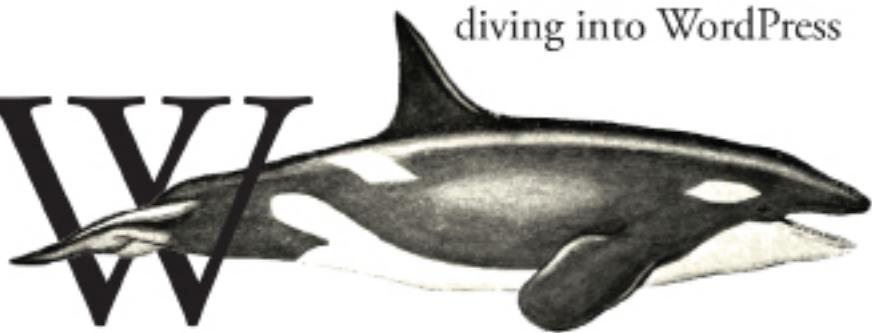
stands for Antelope,
catching up with Ajax

RONALD HUERECA

WORDPRESS & AJAX

An in-depth guide on using Ajax with WordPress

W



is the Killer Whale,
diving into WordPress

The people that made this free release possible

For those who have purchased this book in e-book format, a big “Thanks!” goes out to you.

The following companies have also made the free release possible by advertising for the book and/or supporting me while writing the book:

VG Universe Design



[VG Universe Design](#) is a Web & Graphic Design Studio owned and operated by this book’s designer, Vivien Anayian. Her work speaks for itself, and she was instrumental in this book’s public release.

WebDesign.com



[WebDesign.com](#) provides premium, professional web development training to web designers, developers, and WordPress website owners. The [WebDesign.com](#) Training Library currently holds over 230 hours of premium training developed by seasoned experts in their field, with approximately 20 hours of new training material added each month.

WebDevStudios



[WebDevStudios](#) is a website development company specializing in WordPress. We have a staff of developers and designers who live and breathe WordPress and pride ourselves in being WordPress experts. We have the resources to support any client from a single blog to a WordPress Multisite network with thousands of sites to a BuddyPress site with thousands of members.

WPCandy



[WPCandy](#) is an unofficial community of WordPress users and professionals. They are best known for their solid WordPress reporting and also publish a growing weekly podcast and regular community features. If you love WordPress and want to stay plugged in to the community, [WPCandy](#) is the place to be.

RONALD HUERECA

WORDPRESS AJAX

An in-depth guide on using Ajax with WordPress

cover  book design by Vivien Anayian

WordPress and Ajax - 2nd Edition

Copyright © 2010 by Ronald Huereca

All Rights Reserved under International and Pan-American
Copyright Conventions

This book is protected under U.S. and international copyright law. No part of this book may be used or reproduced in any manner without the written permission of the copyright holder except where allowed under fair use.

The rights of this work will be enforced to the fullest extent possible, including under the Digital Millennium Copyright Act of 1998 (DMCA) of the United States, the European Directive on Electronic Commerce (EDEC) in the European Union and other applicable U.S. and international laws.

All source code used in this book is licensed under the GNU Public License (GPL). For more information about the GPL or to obtain a copy of the full license, please visit <http://www.gnu.org/copyleft/gpl.html>.

BOOK DESIGNER

Vivien Anayian

COVER DESIGNER

Vivien Anayian

COVER ILLUSTRATIONS

Antelope by Haekel, *Vintage educational plate*

Killer Whale by W. Sidney Berridge from *A Book Of Whales, 1900. Plate XXI (From cast in National History Museum)*

PRIMARY TYPEFACES

Adobe Garamond Pro, *designed by Claude Garamond, Robert Slimbach*

Futura, *designed by Paul Renner*

Myriad Pro, *designed by Robert Slimbach, Carol Twombly*

Monaco, *designed by Susan Kare, Kris Holmes, Charles Bigelow*

Table of Contents

Acknowledgments 11

About Ronald Huereca - Author 12

Credits and Thanks 12

Introduction..... 15

How About a Little Ajax? 16

The Book's Beginning 16

The Goal of the Book 17

Helpful Tools for the Journey 18

Online Resources 19

A Word of Warning 19

Chapter 1

What is Ajax? 21

Chapter 2

Adding Scripts Properly to WordPress 27

So How Does wp_enqueue_script Work? 29

 Handle 29

 Src 29

 Deps 30

 Ver 31

 In_footer 31

Great, I have wp_enqueue_script down. Now what? 32

Naming Your Handlers 35

Loading Scripts Conclusion 36

Chapter 3

Localizing Your Scripts 37

wp_localize_script	39
Handle	39
Object_name	39
l10n	40
wp_localize_script Example	40
Other Localization Techniques	42
The JSON Technique	42
A Custom Function	44
Localization Conclusion	45

Chapter 4

Adding Styles Properly to WordPress 47

How Does wp_enqueue_style Work?	48
Handle, Src, Deps, and Ver	49
Media	49
The wp_enqueue_style Hooks	50
The init Technique	51
Conditional Comments	53
Loading Styles Conclusion	54

Chapter 5

Page Detection in WordPress..... 55

WordPress Conditionals	57
Loading Scripts on the Front-End	58
Loading Scripts on the Home Page	59
Loading Scripts on the Front Page	60
Loading Scripts on Posts or Pages	60
Loading Scripts on Comment Pages	61
Loading Scripts for Post Types	62
Loading Scripts for Custom Taxonomies	62
Loading Scripts for Detecting Shortcodes	62
Load Scripts for a Specific Platform	63
Conditionals Conclusion	63
Page Detection in the Admin Area	64
Page Detection for Specific Admin Pages	65
Page Detection for Menu Items	66
Page Detection Conclusion	68

Chapter 6

WordPress Loading Techniques 69

Overriding Styles (and Scripts)	71
Disabling Styles (and Scripts)	73
Loading Just Your Scripts	75
Creating Standalone Pages	77
Loading WordPress Manually Using wp-load	77
Loading WordPress Using a Template Redirect	79

Preventing Plugins From Loading 83
WordPress Loading Techniques Conclusion 86

Chapter 7

Properly Formatting jQuery for WordPress Use.. 87

Namespacing 88
Public Properties/Functions 91
Private Variables/Functions 92
Including Your jQuery Script in Your Theme 95
Conclusion for Properly Formatting jQuery for WordPress 96

Chapter 8

Nonces and the Escape API..... 97

WordPress Nonces 98
 Nonces and Forms 101
 Nonces and URLs 104
 Nonces and Ajax 106
The Escape API 107
 Validating Numeric Inputs 108
 Escaping HTML 109
 Escaping Attributes 110
 Escaping JavaScript 111
 Escaping URLs 111
 Filtering HTML 112
 Escape API Conclusion 114

Chapter 9

Sending Our First Ajax Request 115

Set Up the PHP Class to Handle Back-end Operations 117

Setting up the Interface 119

Setting Up the JavaScript File 122

Setting up the Ajax Object 124

Finalizing the functions.php Class 126

 The get_comments Method 126

 Add in our JavaScript Files 127

 Add in JavaScript Localization 128

 Add in String Localization 131

 Add in Query Variable Support 131

Finalizing the Ajax Request 133

Chapter 10

Processing Our First Ajax Request 139

Securing Your Ajax Processor 140

 Performing a Nonce Check 141

Server-Side Processing of Ajax Requests 143

Sending an Ajax Response 146

Client-Side Processing/Parsing 150

 Parsing the XML Document Object 150

 Processing the Data 151

The Output 156

Chapter 11

WordPress and Admin Ajax..... 165

WordPress' admin-ajax.php	166
Registering the Ajax Processor	168
Getting the Location of the Ajax Processor	170
Passing Data to the Ajax Processor	171
The wp_ajax Callback Method	173
Finalizing the Ajax Processor	173
Admin Ajax Conclusion	175

Chapter 12

Example 1: WP Grins Lite 177

The WPGrins Class	181
The Constructor	182
add_scripts and add_scripts_frontend	187
get_js_vars	188
add_styles and add_styles_frontend	190
add_admin_pages	191
print_admin_page	192
ajax_print_grins	193
wp_grins	193
get_admin_options	195
save_admin_options	196
Our Template Tag	197
The Admin Panel (admin-panel.php)	198
The JavaScript File (wp-grins.js)	203

Chapter 13

Example 2: Static Random Posts209

Creating the Static Random Posts Widget 212

static_random_posts	214
form	215
update	218
get_admin_options	219
save_admin_options	221
init	221
add_admin_pages	222
print_admin_page	222
add_post_scripts	223
get_js_vars	224
widget	225
get_posts	229
build_posts	231
print_posts	233
ajax_refresh_static_posts	234

The Admin Panel (admin-panel.php) 237

Updating the Options 238

The User Interface 242

The JavaScript File (static-random-posts.js) 246

Static Random Posts Conclusion 253

Chapter 14

Example 3: Ajax Registration Form255

Creating the Ajax_Registration Class 257

 rform_shortcode 260

 post_save 263

 has_shortcode 265

 add_scripts 266

 add_styles 267

The Script File (registration.js) 269

 Capturing the Form Data 270

 Building the Ajax Object 272

 Parsing the Ajax Response 273

The Ajax Processor 280

 Parsing the Passed Form Data 280

 Data Validation 282

 Creating the User 287

 Sending the Response 288

Ajax Registration Form Conclusion 289

Now You Begin Your Own Journey291

Acknowledgments



About Ronald Huereca - Author

People describe Ronald as a highly opinionated (and sometimes funny) writer.

He's worked with WordPress since 2006 and has released several WordPress plugins, his most popular being Ajax Edit Comments.

Ronald currently calls Austin, Texas home. His education includes a Master's in Business Administration and a degree in Electronics Engineering Technology.

When he's not killing trees or reindeer, he blogs at his personal site, ronalfy.com.

Credits and Thanks

First, I would like to thank my family, friends, and God for getting me through a very tough per-

sonal situation in 2009. Without them, this book would not have been possible.

I'd like to thank Vivien Anayian for the inspiration behind Ajax Edit Comments, which consequently started me with Ajax and WordPress.

I'd also like to thank Ajay D'souza and Mark Ghosh. Both have helped me tremendously along my WordPress journey.

Finally, I'd like to thank the folks at Automattic and the contributors behind WordPress.

Introduction

How About a Little Ajax?

I'm rather biased when it comes to Ajax. One of my first WordPress plugins is Ajax-based. As soon as I started using Ajax, I fell in love with the possibilities.

You see, Ajax is what helps achieve that “rich” Internet experience. Ajax helps eliminate unnecessary page loads, can streamline a user interface, and can make a task that is cumbersome run gracefully behind the scenes.

As with every piece of technology, Ajax can be used for good or for bad. There are those that will use and abuse Ajax (and they should be spanked unmercilessly).

The Book's Beginning

When I was learning Ajax with WordPress, finding good documentation was hard to find. My education was basically ripping code from other plugins, mashing them together, and hoping everything worked.

I've grown a lot in my journey. While I am still far from perfect, I felt it necessary to share what I have learned over the years.

This book began humbly. I thought to myself, “Why not write a quick group of articles on using Ajax with WordPress?”

I began working on an outline, and it became quite obvious this wasn't going to be a short series of articles. In order to adequately cover the topic, there needed to be much, much more.

The Goal of the Book

The goal of this book is to provide you a rock-solid foundation for using Ajax with WordPress. After the foundation has been laid, you will be walked through several real-world examples. By the end of the book, you should not only have a thorough understanding of Ajax, but how Ajax functions within WordPress itself.

The code examples I present are from my own (sometimes painful) experiences with using Ajax. I've cried. I've bled. And I hope to share my agony (err, joy).

Prerequisites For This Book

This book gets right down to business (sorry, no foreplay), so I'm assuming the following if you intend to follow me throughout this journey:

- You are using WordPress version 3.0 and above.
- You have “some” knowledge of JavaScript and some knowledge of the uber-awesome jQuery library (you’re free to disagree with me on this, but keep in mind I carry a big stick).
- You already know PHP. This book gets heavy in places, and it helps tremendously if you have a good grasp on PHP.
- You have some experience with WordPress theme or plugin development (and their respective APIs).
- Or, you have none of the above, but want to read anyway (you’re the one that ignores the “falling rocks” signs on the highway, right?)

Helpful Tools for the Journey

- Firebug for Firefox: this tool is one of those where you wonder how you ever got along without it. It’ll assist you with debugging any JavaScript errors.
- XAMPP: this tool allows you to host WordPress locally. It’s great for testing scripts and themes without modifying a production install.

Online Resources

To access the code and various resources mentioned in this book, please visit:

- <http://www.wpajax.com/code/> - Downloadable code samples.
- <http://www.wpajax.com/links/> - Links and resources mentioned in this book.

A Word of Warning

Ok, perhaps I should have said “words.”

My writing style throughout this book is very casual. I wrote this book as if we’re going on a journey together to achieve Ajax nirvana.

So what are you waiting for? Let’s begin.

What is Ajax?

What is Ajax?

First things first... What the heck is Ajax anyways besides a lovely acronym (Asynchronous JavaScript and XML)? It's definitely not a common household cleaner (at least, not on the Internet).

Let's just say that Ajax consists of a bunch of tubes... Okay, bad joke, but let's go on.

Ajax, in simple terms, allows a user on the client-side (the web browser) to indirectly interact with the server-side (the back-end that you will never see).

Ajax allows for complex processing of data that would be impossible to accomplish on the client-side.

Ajax is event based. Some kind of event occurs on the client side (a page load, a click, a form submission) and this event is captured.

JavaScript captures this event, forms the Ajax request, and sends the request to the server. The server sees this request, processes it, and sends back an Ajax response.

It's really that simple. Perhaps too simple for you techno-geeks, but I digress. I mean, I could go

deep into technical mumbo-jumbo, but all it will get you is weird stares from the people you're explaining it to (and perhaps a slap from the girl you're trying to pick up when you start talking about sending and receiving requests).

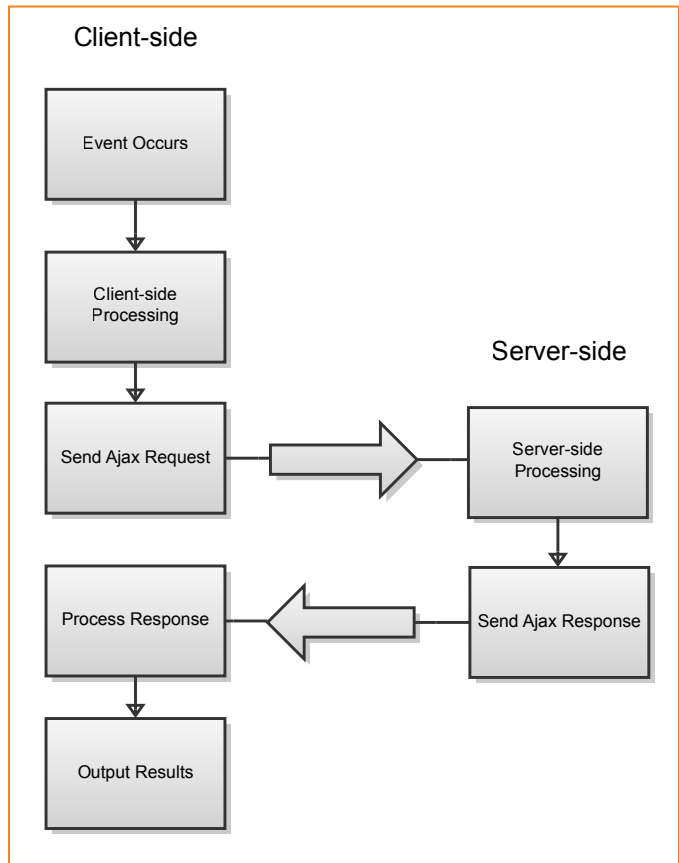


Figure 1. Ajax Process

Now implementing Ajax, and implementing it well, is another story. So let's read on.

Ajax Best Practices

As with any technology, Ajax can easily be abused. I've seen too many sites that depend on Ajax for navigation or to access crucial and/or critical content without alternatives.

From my perspective, Ajax should be absolutely transparent and unobtrusive to the user. I'm thinking of Flickr and Netflix when I say this. Both sites mentioned use Ajax to add to the user experience, rather than detract from it.

When you click on that caption or description, or leave that star rating, it's simple and to the point. All we should know from the end-user's perspective is that it works (and that's all we should know).

Hitting the back-button doesn't ruin the experience; it just means you're "done".

I'll leave Ajax best practices to just one statement: it should be absolutely and completely transparent to the user. The user shouldn't even know he's using Ajax.

Ajax should be used to accomplish quick and simple tasks. That's it. Anything more is just asking

All we should know from the end-user's perspective is that it works

Ajax should be absolutely and completely transparent to the user

to be beat down by the Internet troll (and he's mean).

Here are some good examples of when to use Ajax:

- Reduce the need to move away from the current page. I'm thinking of Netflix's star rating and "Add to Queue" features. The user gets to keep browsing and isn't interrupted.
- Allow quick editing of options. Observe how easy it is to edit captions and descriptions in Flickr.
- Allow quick functionality without a page refresh. I'm thinking of WordPress' use of comment moderation functions that allow you to stay on the same page without having to refresh.

And here are some bad examples:

- Providing multi-page Ajax operations. If the user hits the back button or refreshes, then what? He has to start all over from scratch (and he's about to call the Internet troll on you).
- Loading content on websites. Ever been to one of those sites that load the content when you

click on a button or drop-down menu option? Not good. There's no back button functionality, no way to bookmark the content, and if you refresh, you're back at the beginning. In other words, they might as well have used Flash (ducking for cover).

- Using Ajax upon a page load. This last one is more of a guideline as there are many exceptions. Notable exceptions include scripts that track statistics and would like to avoid caching plugins.

If you are implementing an Ajax process and the project delves into the realm of faking browser history and remembering user states, then Ajax is likely not the ideal solution.

Remember, *keep Ajax simple*. If the user realizes you are using it, you have failed.

Adding Scripts Properly to WordPress

Adding Scripts Properly to WordPress

Starting in WordPress 2.1 (if I remember correctly), the awesome folks at Automattic gave us the even awesomer function of `wp_enqueue_script`.

Before that, it was every plugin or theme author for himself. If you wanted to add in a script, it was hard-coded in.

As you might imagine, this presented a ton of problems. Scripts were loaded twice, out of order, or even when they weren't needed at all.

Furthermore, some themes and plugins had the JavaScript embedded within the plugin's or theme's PHP file just to capture a few PHP variables. Not good! In order to add scripts properly to JavaScript, you must always keep your PHP and JavaScript separate. And by separate, I mean separate files. There's just no excuse anymore (I'll get into this more when I cover localizing scripts).

Always keep PHP and JavaScript separate.

The `wp_enqueue_script` function is the first step in loading your scripts properly. Not only can you add your script, but you can also specify the dependencies (e.g., jQuery), and the version number.

The function prevents duplicate JavaScript files from loading, which is always a good thing.

So How Does `wp_enqueue_script` Work?

The function `wp_enqueue_script` takes in five arguments, with the last four being optional.

```
<?php wp_enqueue_script('handle', 'src', 'deps', 'ver', 'in_footer'); ?>
```

Handle

The `handle` argument is a string that names your script, or references someone else's.

For example, you can add the jQuery script to any page by calling:

```
wp_enqueue_script('jquery');
```

Likewise, if you have previously registered your script (using `wp_register_script`), you can call your script by calling:

```
wp_enqueue_script('my_script');
```

Src

The `src` argument is asking for the URL of the file. If you supply the `src` of the script, the

`wp_enqueue_script` function automatically registers your script for others to use (no need to use `wp_register_script`).

An example of `wp_enqueue_script` in use is:

```
wp_enqueue_script('my_script', plugins_url('my_plugin/my_script.js'));
```

Themers would use:

```
wp_enqueue_script('my_script', get_bloginfo('stylesheet_directory') . '/my_script.js');
```

Deps

The `deps` argument is an array of dependencies. For example, if your script requires jQuery or other JavaScript files, it'll load these files before your plugin's JavaScript file.

```
wp_enqueue_script('my_script', plugins_url('my_plugin/my_script.js'), array('jquery', 'another_script'));
```

See that array up there (it's pretty, I know)? That's what's calling your dependencies.

jQuery is built into WordPress by default, so calling it via dependency is no problem as long as you queue it via `wp_enqueue_script`.

The `another_script` dependency assumes you have already used `wp_enqueue_script` to assign it a `handle` and a `src`.

The outcome in this scenario is that jQuery and `another_script` will load before `my_script`.

Ver

The `ver` argument is simply the version of the JavaScript file for caching purposes. If you are supplying your own script, it's a good idea to supply a version.

The version is string based and looks like this:

```
wp_enqueue_script('my_script', plugins_url('my_plugin/my_script.js'), array('jquery', 'another_script'), '1.0.0');
```

As you update your JavaScript file, update the `ver` argument. If you don't, you'll inevitably run into caching issues with WordPress.

In_footer

The `in_footer` argument (since WordPress 2.8) determines if the script will load in the header or footer (header is the default).

To load your script in the footer, simply pass it a `1` (or `true`).

```
wp_enqueue_script('my_script', plugins_url('my_plugin/my_
script.js'), array('jquery', 'another_script'), '1.0.0', true);
```

Great care should be used when setting the scripts to load in the footer as some themes may not have this functionality built in (the same problem exists for loading scripts in the header).

Great, I have `wp_enqueue_script` down. Now what?

You have to call `wp_enqueue_script` from the appropriate hook in WordPress.

Fortunately, WordPress has two such action hooks you can tap into: `admin_print_scripts` and `wp_print_scripts`.

The `admin_print_scripts` hook allows you to add scripts specifically in the admin area, while the `wp_print_scripts` hook allows you to add scripts everywhere else (it should be noted that you can still use `wp_print_scripts` to load scripts in the admin area).

Adding your `wp_enqueue_script` code is as simple as adding an action inside your plugin or theme code.

An example of the code being used inside of a class constructor is:

```
add_action('admin_print_scripts', array(&$this, 'add_admin_scripts'));
```

And for those not using a class (classes help avoid conflicts with other plugins):

```
add_action('admin_print_scripts', 'add_admin_scripts');
```

I'm particularly fond of working with classes over standard functions. There's less potential for naming conflicts, and the class methods can be given more intuitive names.

Here's an example class structure (see next page):


```

<?php
if (!class_exists('myPlugin')) {
    class myPlugin {
        function myPlugin(){$this->__construct();}
        function __construct() {
            //JavaScript
            add_action('wp_print_scripts', array(&$this,'add_
scripts'),1000);
        } //end constructor
        //Adds the appropriate scripts to WordPress
        function add_scripts(){
            wp_enqueue_script('another_script', plugins_url(
'another_script.js', __FILE__ ), '', '1.0.0');
            wp_enqueue_script('my_script', plugins_url( 'my_
script.js', __FILE__ ), array('jquery', 'another_script'),
'1.0.0');
        }//End add_scripts
    }//End class myPlugin
} //End if class
//Further avoid conflict
if (class_exists('myPlugin')) {
    //instantiate the class
    $mypluginvar = new myPlugin();
}
?>

```

Once you venture into the `add_scripts` method, you'll see the use of the `wp_enqueue_script` function in all its glory.

For themers, just include the class in your `functions.php` file and replace the `wp_enqueue_script` `src` argument with the code such as shown next:

```

wp_enqueue_script('my_script', get_bloginfo('stylesheet_
directory') . '/my_script.js');

```

Naming Your Handlers

Of great importance is naming your handler for `wp_enqueue_script`. If you are using a third-party script (say, Colorbox), use common sense in naming the handler. There's no reason to name it "lightbox" or "mybox". The handler name should be obvious, right?

Although I don't recommend the following technique for all plugins, here's a neat way to resolve naming conflicts: give a way for plugin authors and themers a way to override your name. How? Perform an `apply_filters` for your handler name and provide the necessary documentation.

Here's an example:

```
wp_enqueue_script(apply_filters('my_plugin_script',  
'my_script'), plugins_url('my_plugin/my_script.js'),  
array('jquery', 'another_script'), '1.0.0', true);
```

Using the above example, someone in their theme's `functions.php` or plugin code could add your filter and overwrite the name to resolve naming conflicts.

```
add_filter('my_plugin_script', create_function('$name', 'return  
"my_new_name";'));
```

Loading Scripts Conclusion

Using `wp_enqueue_script` is a great way to avoid the dreaded dilemma of loading multiple copies of the same scripts.

To avoid an angry mob of plugin authors at your door with pitchforks, I'd highly recommend making use of the `wp_enqueue_script` function.

Up next is how to access dynamic content using JavaScript localization.

Localizing Your Scripts

Localizing Your Scripts

When adding scripts to WordPress, you will inevitably run into a small, but painful, issue of localization.

Localizing a plugin or theme is relatively straightforward, but JavaScript presents its own difficulties since we can't easily call the PHP functions necessary (which is one reason authors embed JavaScript in PHP files).

Since embedding JavaScript in PHP files is never a good technique, we use localization to save the day.

When using Ajax with WordPress, the JavaScript file will need to know the exact location of the WordPress site it is to interact with. You can't hard-code this in unless you embed, so how does one get the location?

Furthermore, if you display anything with JavaScript, chances are your users will want the strings to be localized.

Fortunately, WordPress provides the ultra-handly `wp_localize_script` function.

`wp_localize_script`

The `wp_localize_script` takes three arguments:

- `handle`
- `object_name`
- `l10n`

Handle

The `handle` argument will be the same handle you use for your script name.

For example, if you have a handle of `my_script`, you would use the same name when calling the `wp_localize_script` function.

Object_name

The `object_name` argument is a string that tells WordPress to create a JavaScript object using the name you specify.

It's important that the string you pass is as unique as possible in order to minimize naming conflicts with other scripts.

For the upcoming example, our object name will be `my_unique_name`.

l10n

The `l10n` argument is an array of strings you would like to localize.

Within this array, you will want to take advantage of the `__` (yes, those are two underlines) WordPress function.

`wp_localize_script` Example

For the purpose of this example, let's create a function called `localize_vars` and have it return an array.

```
<?php
function localize_vars() {
    return array(
        'SiteUrl' => get_bloginfo('url'),
        'AjaxUrl' => admin_url('admin-ajax.php'),
        'OtherText' => __('my text', "my_localization_
name")
    );
} //End localize_vars
?>
```

Please note the use of the `__()` function. It takes in the text we want to localize, and our localization name. This will be the same name you use if you take use localization within WordPress.

The variable `SiteUrl` gives us the http path to our WordPress site, and `AjaxUrl` gives us the path to WordPress' built-in Ajax processor.

From another area in our code, we call the `localize_vars` function:

```
<?php
wp_enqueue_script('my_script', plugins_url('my_plugin/my_
script.js'), array('jquery'), '1.0.0');
wp_localize_script('my_script', 'my_unique_name', localize_
vars());
?>
```

WordPress then conveniently adds localization JavaScript immediately before our main script is included. Viewing the page source will reveal:

```
<script type='text/javascript'>
/*  */
    my_unique_name = {
        SiteUrl: "http://www.mydomain.com",
        AjaxUrl: "http://www.mydomain.com/wp-admin/admin-
ajax.php",
        OtherText: "my localized text"
    }
/* ]]&gt; */
&lt;/script&gt;</pre></div><div data-bbox="294 684 926 789" data-label="Text"><p>With the <code>localize</code> example, you can use PHP magic to add just about anything to your localization object. Hence, no need to ever embed JavaScript within a PHP file.</p></div><div data-bbox="294 812 926 893" data-label="Text"><p>Now you can call your localized JavaScript variables from your <code>my_script.js</code> file. Here's an example of an alert:</p></div>
```



```
alert(my_unique_name.SiteUrl);  
alert(my_unique_name.OtherText);
```

It's really as easy as that. You can now localize JavaScript strings and get the coveted Ajax URL (which we'll need for Ajax later).

Other Localization Techniques

While the `wp_localize_script` function does great work, it has one inherent flaw: each localized string is on a new line. For plugins that require a lot of localized strings, the size of the page source can easily balloon to unacceptable levels.

To remedy this, we can use two additional localization techniques: one uses JSON, and the other is a custom function.

The JSON Technique

The JSON Technique uses WordPress' built-in JSON class in order to parse our localized variables.

We would use the same `localize_vars` function, but would modify the way we queue our scripts.

First, let's create a helper function that will instantiate the JSON class and spit out our localized variables to screen.

```
function js_localize($name, $vars) {
    ?>
    <script type='text/javascript'>
    /*  */
    var &lt;?php echo $name; ?&gt; =
    &lt;?php
    require_once(ABSPATH . '/wp-includes/class-json.php');
        $wp_json = new Services_JSON();
        echo stripslashes($wp_json-&gt;encodeUnsafe($vars));
    ?&gt;
;
    /* ]]&gt; */
    &lt;/script&gt;
&lt;?php
}</pre>
</div>
<div data-bbox="295 471 926 550" data-label="Text">
<p>The <code>js_localize</code> function takes in a <code>$name</code> (our object name) and an array of our localized variables (<code>$vars</code>).</p>
</div>
<div data-bbox="295 572 926 625" data-label="Text">
<p>The function then instantiates the JSON class and encodes the variables for output.</p>
</div>
<div data-bbox="295 646 927 699" data-label="Text">
<p>Here's how the code would look when queuing up your scripts.</p>
</div>
<div data-bbox="94 726 823 815" data-label="Text">
<pre>&lt;?php
js_localize('my_unique_name', localize_vars());
wp_enqueue_script('my_script', plugins_url('my_plugin/my_script.js'), array('jquery'), '1.0.0');
?&gt;</pre>
</div>
<div data-bbox="295 850 926 902" data-label="Text">
<p>Please note that the <code>js_localize</code> function is run <i>before</i> the script is queued.</p>
</div>
```

While this technique does eliminate the newlines and creates cleaner source code, it does have one major flaw. It doesn't work for all languages.

For example, the Turkish language causes the above technique to come crashing down.

However, if you don't plan on having additional languages and want localization purely for the ability to access the JavaScript variables, then I would recommend this technique.

A Custom Function

For those wanting to eliminate the newlines caused by `wp_localize_scripts`, and still have the ability to handle complex languages, then a custom function will have to suffice.

We'll use the same exact code to queue our scripts, but the `js_localize` function will change a bit.

My technique is to iterate through the localized variables, save them to an array, and output the array to screen.

It might not be the most poetic thing you've ever seen, but it works pretty well, even for those complex languages.

```
function js_localize($name, $vars) {
    $data = "var $name = {";
    $arr = array();
    foreach ($vars as $key => $value) {
        $arr[count($arr)] = $key . " : '" . esc_js($value)
        . "'";
    }
    $data .= implode(",",$arr);
    $data .= "};";
    echo "<script type='text/javascript'>\n";
    echo "/* <![CDATA[ */\n";
    echo $data;
    echo "\n/* ]]> */\n";
    echo "</script>\n";
}
```

Localization Conclusion

Within this chapter you learned the how and the why of JavaScript localization.

The benefits of localizing your JavaScript are:

- No need to embed JavaScript and PHP.
- Can capture PHP variables without having to load the WordPress environment.
- Can enable others to translate your JavaScript strings.

You also learned three different techniques to achieve localization.

- Using `wp_localize_script` - Recommended for general use.
- Using JSON - Recommended for non-complex localization and performance.
- Using a Custom Function - Recommended for complex localization and performance.

In the next chapter we'll be talking about how to load styles (CSS) properly. What do styles have to do with Ajax? Well, you want to make it look pretty, don't you?

Adding Styles Properly to WordPress

Adding Styles Properly to WordPress

Let's give WordPress some style with some awesome CSS (Cascading Style Sheets, but you probably knew that already).

Just as `wp_enqueue_script` can prevent duplicate scripts from loading, `wp_enqueue_style` can do the same. But the biggest advantage of using `wp_enqueue_style` over other techniques is the use of dependencies.

Unfortunately, the use of `wp_enqueue_style` isn't as widely adopted as `wp_enqueue_script`, but in my non-humble opinion, it's just as important to use.

The biggest reasons to use `wp_enqueue_style`?

- Allows for dependencies and media targeting (screen, print, etc.)
- Allows for others to overwrite your styles (an uber-awesome reason if you ask me)

So what are the arguments, and what hooks should we use?

How Does `wp_enqueue_style` Work?

The `wp_enqueue_style` function takes in five arguments, with the last four being optional.

```
<?php wp_enqueue_style('handle', 'src', 'deps', 'ver',  
'media'); ?>
```

Handle, Src, Deps, and Ver

Since I already covered loading scripts properly, and `wp_enqueue_script` and `wp_enqueue_style` are so similar, I'm going to breeze over the first four arguments (they act the same, really).

So rather than being repetitive, let's get to the new argument, which is `media`.

Media

The `media` argument is asking for the type of media the stylesheet is defined for.

```
wp_enqueue_style('my_css', plugins_url('my_plugin/my_css.css'),  
array('another_css_file'), '1.0.0', 'screen');
```

In this particular example, we're just setting the media type to `screen`, which is for most browsers.

Some other common media types you may use are: `all`, `print`, `screen`, `handheld`.

The `wp_enqueue_style` Hooks

In order to use `wp_enqueue_style` properly, you need to call it using the appropriate hooks. There are two main hooks (`admin_print_styles` and `wp_print_styles`), but a third will come in handy for theme developers (`init`, which I'll cover last).

The `admin_print_styles` hook allows you to add styles in the admin area, while the `wp_print_styles` hook allows you to add styles on both the front-end and admin area.

To add your styles, use the `add_action` function and call the appropriate hook and provide a callback function.

```
<?php
add_action('wp_print_styles', 'my_styles_callback');
function my_styles_callback() {
    wp_enqueue_style('my_css', plugins_url('my_plugin/my_
css.css'), array('another_css_file'), '1.0.0', 'screen');
}
?>
```

The above code will place a stylesheet on each front-end and admin area provided that the “`another_css_file`” dependency has been previously registered (using `wp_register_style`). If you want to place the stylesheet in the admin area only, you should use `admin_print_styles` instead.

The init Technique

One technique I'm rather fond of when it comes to themes is to use the `init` action to register all the styles, and then selectively load the styles in the theme's `header.php` file.

First, let's dive into the theme's `functions.php` and register some styles:

```
<?php
add_action('init', 'my_theme_register_styles');
function my_theme_register_styles() {
    //Register styles for later use
    wp_register_style('my_theme_style1', get_stylesheet_
directory_uri() . '/style1.css', array(), '1.0', 'all');
    wp_register_style('my_theme_style2', get_stylesheet_
directory_uri() . '/style2.css', array('my_theme_style1'),
'1.0', 'all');
    wp_register_style('my_theme_style3', get_stylesheet_
directory_uri() . '/style3.css', array('my_theme_style1', 'my_
theme_style2'), '1.0', 'all');
}
?>
```

I registered three styles for later inclusion, and cascaded the dependencies.

To show these on the front-end, you would venture into `header.php` and add the following code before the `</head>` tag:

```
<?php wp_print_styles(array('my_theme_style1', 'my_theme_
style2', 'my_theme_style3')); ?>
</head>
```

The `wp_print_styles` function is passed an array of handlers, and will push our styles out for the world to see.

Since I cascaded the dependencies, however, you could (in this particular case) get away with just passing the handler of the last registered style (both output the three registered styles):

```
<?php wp_print_styles(array('my_theme_style3')); ?>
```

Using the `init` method, you can even skip loading styles via `header.php` and instead load them for a particular page template:

```
<?php
/*
Template Name: Custom Style Template
*/
?>
<?php wp_enqueue_style('my_theme_style3'); ?>
<?php get_header(); ?>
<!--loop stuff-->
<?php get_footer(); ?>
```

The above technique assumes that your theme makes use of the `wp_head()` call in `header.php`. And in an ideal reality (let's be optimistic, shall we?), all themes should make use of the `wp_head()` and

`wp_footer()` calls (in `header.php` and `footer.php` respectively).

Conditional Comments

One of the biggest arguments I've seen regarding not using `wp_enqueue_style` is its lack of support for conditional comments.

But `wp_enqueue_style` does allow for conditional comments (so now there's no excuse, ha!).

So how does one achieve the holy grail of conditional comments for inferior (oops, did I say that?) browsers?

It's pretty simple really. Declare a `$wp_styles` global, and add conditional comment data (we'll modify our `functions.php` example).

```
add_action('init', 'my_theme_register_styles');
function my_theme_register_styles() {
    //Register styles for later use
    wp_register_style('my_theme_style1', get_stylesheet_directory_uri() . '/style1.css', array(), '1.0', 'all');
    wp_register_style('my_theme_style2', get_stylesheet_directory_uri() . '/style2.css', array('my_theme_style1'), '1.0', 'all');
    wp_register_style('my_theme_style3', get_stylesheet_directory_uri() . '/style3.css', array('my_theme_style1', 'my_theme_style2'), '1.0', 'all');
    global $wp_styles;
    $wp_styles->add_data( 'my_theme_style3', 'conditional', 'lte IE 7' );
}
```

In the previous example, we used the `$wp_styles` global to add a conditional to load for Internet Explorer versions less than 7.

If you were to view the source where the styles loaded, you would see something like this:

```
<link rel='stylesheet' id='my_theme_style1-css' href='http://
yourdomain.com/wp-content/themes/gravy/style1.css?ver=1.0'
type='text/css' media='all' />
<link rel='stylesheet' id='my_theme_style2-css' href='http://
yourdomain.com/ronalfy/wp-content/themes/gravy/style2.
css?ver=1.0' type='text/css' media='all' />
<!--[if lte IE 7]>
<link rel='stylesheet' id='my_theme_style3-css' href='http://
yourdomain.com/wp-content/themes/gravy/style3.css?ver=1.0'
type='text/css' media='all' />
<![endif]-->
```

Loading Styles Conclusion

Using `wp_enqueue_style` is a terrific way of loading styles for both themes and plugins.

Using the techniques mentioned in this chapter, you should be able to build some fairly advanced and scalable solutions.

Up next is page detection, because we really don't want our scripts and styles to load on every single page, right?

Page Detection in WordPress

Page Detection in WordPress

Let's say you have a WordPress site that has ninety-six (yes, I know this person) plugins installed. Each plugin performs a specialized function, and each installs their own CSS and JavaScript. Now imagine that all of these CSS and JavaScript files are loaded for each and every single page load. Catastrophe?

Perhaps the above scenario is a little far-fetched, but I've come across sites where the CSS and JavaScript overhead has approached half a megabyte per page load. All I have to say is, "Ick!"

One solution to the above dilemma is to install a plugin such as W3 Total Cache or WP Minify to compress the scripts and styles. While extremely helpful, this is like putting a band-aid on a lost limb. It's not solving the real issue: there are too many damn scripts running. And, in reality, it's probably not your fault.

A plugin's (or theme's) scripts and styles should only load where absolutely needed. Failure to do so can (and will) cause script conflicts, weird CSS bugs, and (of course) increase site loading time.

This is where page detection comes to the rescue. With page detection, the scripts and styles load exactly where needed and nowhere else.

The lack of proper page detection is often why plugins get a bad rap. People accuse plugins of slowing down a site considerably, and if many lack any form of page detection, these people are right. It will slow down a site. And then you'll see yet another post on "How to do XYZ without a plugin."

Let's get over this hump, use some proper page detection, and get these plugins (and themes) running as efficient as possible.

The first step in page detection is learning the many WordPress conditionals, so let's start there.

WordPress Conditionals

Let's begin with some basic starter code.

We'll be using the `wp_print_scripts` action and a generic `wp_enqueue_script` call to load the jQuery library.


```
<?php
add_action('wp_print_scripts', 'my_plugin_load_scripts');
function my_plugin_load_scripts() {
    //Loads jQuery on every front-end and admin page
    wp_enqueue_script('jquery');
}
?>
```

This code will run the jQuery library on every single front-end and admin page. If you use a theme that requires jQuery on every single page, perhaps this is acceptable. However, what if you would like to be more selective?

What if you only want the script to run on the front page? Or perhaps only on a page with a specific post type? You can do so with WordPress conditionals (please note that all of the examples can be interchanged with the `wp_print_styles` action and the `wp_enqueue_style` functions).

Loading Scripts on the Front-End

Since `wp_print_scripts` runs scripts on both the front-end and admin area, it may be a good idea to check the `is_admin` conditional (the `is_admin` conditional returns true if you are in the admin area).

If you would like to run a script everywhere but the admin area, you would use something like this (building on our starter code):

```
<?php
add_action('wp_print_scripts', 'my_plugin_load_scripts');
function my_plugin_load_scripts() {
    if (is_admin()) return;
    //Loads jQuery on every single front-end page
    wp_enqueue_script('jquery');
}
?>
```

Loading Scripts on the Home Page

The `is_home` conditional (returns `true` or `false`, as with most conditionals) is useful for determining if a page is the home page of your site's blog. If you choose to have a static page as your front page, then the `is_home` conditional will only return `true` for the page you select as your Posts page (options are found in Administration > Settings > Reading).

If you would like a script to run only when `is_home` is true, you would use the following (the remaining conditional examples assume we're already in the `my_plugin_load_scripts` function):

```
if (!is_home()) return;
//Loads jQuery on the home page of a site's blog
wp_enqueue_script('jquery');
```

As seen in the above code, we check to see if the `is_home` conditional is `true`. If it's not, we exit the function.

Loading Scripts on the Front Page

The `is_front_page` conditional will return `true` on the page you have set as your front page (in Administrative > Settings > Reading).

Here's a quick example:

```
if (!is_front_page()) return;
wp_enqueue_script('jquery');
```

Loading Scripts on Posts or Pages

Need a script to run on a specific post or page? WordPress conditionals have you covered.

Here is an example of the `is_single` conditional, which detects if you are on a post or not (within the code are several examples of usage).

```
if (!is_single()) return;
//or use:
if (!is_single(2043)) return; //Checks for post ID 2043
//or use:
if (!is_single('my-post-slug')) return; //checks for a post
slug of my-post-slug
//or use:
if (!is_single(array(2043, 'my-post-slug', 'my post title')))
return; //Pass an array of matches you'd like to check (behaves
like an "or" statement)
wp_enqueue_script('jquery');
```

What about pages? You would use the `is_page` conditional (again, this example shows several uses).

```

if (!is_page()) return;
//or use:
if (!is_page(22)) return; //Checks for page ID 22
//or use:
if (!is_page('my-page-slug')) return; //checks for a page slug
of my-page-slug
//or use:
if (!is_page(array(22, 'my-page-slug', 'my page title')))
return; //Pass an array of matches you'd like to check (behaves
like an "or" statement)
wp_enqueue_script('jquery');

```

Loading Scripts on Comment Pages

Need to run a script on pages with comments open?

```

if (!comments_open()) return;
wp_enqueue_script('jquery');

```

What if you want to only run scripts when there are comments on a post?

```

global $post;
if (!(is_single() || is_page()) || !is_object($post) || $post->comment_count == 0) return;
wp_enqueue_script('jquery');

```

Several sanity checks are run here. We first want to check if we're on a post or page, since these are where we'll find comments. Next, we make sure the `$post` variable is an object. And finally, we make sure the post has comments.

Loading Scripts for Post Types

Need a script to run only for a specific post type? Easy.

```
if (get_post_type() != 'movies') return;
wp_enqueue_script('jquery');
```

Loading Scripts for Custom Taxonomies

Do you need to run a script for only a taxonomy page (typically an archive page)?

```
if (get_query_var("taxonomy") != 'rating') return;
wp_enqueue_script('jquery');
```

Loading Scripts for Detecting Shortcodes

Here's a decently hard one. How in the heck do you do page detection on a shortcode that is embedded within a post's content?

For those not acquainted with shortcodes, they are small snippets of text (e.g., `[gallery]`) placed within a post's (or page's) content. Shortcodes are very useful for placing executable code inside what is typically non-dynamic content. However, performing page detection for a specific shortcode inside the content isn't exactly straightforward.

```
global $post;
if (!(is_single() || is_page()) || !is_object($post)) return;
//Perform a test for a shortcode called yourshortcode in the
content
preg_match('/\[yourshortcode[^\]]*\]/is', $post->post_content,
$matches); //replace yourshortcode with the name of your
shortcode
if (count($matches) == 0) return;
wp_enqueue_script('jquery');
```

Now is the above solution ideal? Not really. In fact, you can make it super complicated and make sure the regular expression is only performed when a post is saved and cache the results as a custom field. But for quick demonstration purposes, this does the trick.

Load Scripts for a Specific Platform

Would you like to load a script that requires the iPhone? What about a script that runs on the Safari browser, but not Internet Explorer?

Fortunately, WordPress has several global variables you can use to achieve this.

Here's a quick snippet that has all of the globals, but does a check for the iPhone. If the iPhone isn't detected, we load an overlay script called Thickbox (you may be familiar with Thickbox if you've ever used the Media Uploader on a post or page).

```
global $is_lynx, $is_gecko, $is_IE, $is_opera, $is_NS4, $is_safari, $is_chrome, $is_iphone;
if ($is_iphone) return;
wp_enqueue_script('thickbox');
```

Conditionals Conclusion

There are many, many other conditionals to work with, so if you fancy some documentation, please

check out the official WordPress Codex page on conditionals: http://codex.wordpress.org/Conditional_Tags

Page Detection in the Admin Area

Page detection is a tad bit more difficult in the admin area. There aren't really any conditionals you can use, and the predictability of what page your on is pretty much nil. Or is it?

Enter the `admin_print_scripts` action (for the examples in this section, you can also interchange `admin_print_scripts` with `admin_print_styles` for stylesheets).

To load a script for all of the admin pages, you would use the following code:

```
<?php
add_action('admin_print_scripts', 'my_plugin_load_scripts');
function my_plugin_load_scripts() {
    wp_enqueue_script('jquery');
}
?>
```

That's great. But what if you only want to load a script for a certain admin page (such as when editing a post)? What about for plugin settings pages, or top-level menu pages?

Fortunately, `admin_print_scripts` takes a suffix parameter in the format of:

```
admin_print_scripts-suffix
```

Page Detection for Specific Admin Pages

The suffix can be several different things, so let's first go over how to load a script for a specific admin page.

Say, for example, you would like a script to run when someone is creating or editing a post. There are two specific files that allow you to do this, which happen to be `post.php` (editing posts or pages) and `post-new.php` (new posts or pages). As a result, you can use `post.php` and `post-new.php` as the suffix parameter.

Here's an example:

```
<?php
add_action('admin_print_scripts-post.php', 'my_plugin_load_
scripts');
add_action('admin_print_scripts-post-new.php', 'my_plugin_load_
scripts');
function my_plugin_load_scripts() {
    if (get_post_type() == 'post')
        wp_enqueue_script('jquery');
}
?>
```

In the example, our script will run for files `post.php` and `post-new.php`. Since the script would also run for

pages, we do a quick type-check to make sure it's just a post.

You can translate the above example to just about any admin page.

Need a script to run when editing a comment? Use the suffix `comment.php`. What about when adding a new user? Use the suffix `user-new.php`.

There are many of these types of pages in the admin panel, and loading scripts or styles is as easy as adding the page name as a suffix (to `admin_print_scripts` and `admin_print_styles` respectively).

Page Detection for Menu Items

Many themes and plugins come with their own settings pages in the admin area. Wouldn't it be nice to load scripts for these specific areas?

You can, by using the page hook as the suffix.

When you register an admin menu (using functions such as `add_options_page` and `add_menu_page`), the function returns a hook you can use for page detection.

In this next example, we'll be making use of the `admin_menu` action, which is useful for initializing all of the custom settings pages.

```
<?php
add_action('admin_menu', 'my_admin_menu');
//Function to initialize the admin menu
function my_admin_menu() {
    $page_hook = add_menu_page( "My Plugin Name Options",
    "My Plugin", 'administrator', 'my_plugin', 'my_plugin_admin_
    settings');
    add_action("admin_print_scripts-$page_hook", 'my_plugin_
    load_scripts');
}
//Build the admin menu interface
function my_plugin_admin_settings() {
    echo "My Plugin Page";
}
//Load our scripts
function my_plugin_load_scripts() {
    wp_enqueue_script('jquery');
}
?>
```

The `admin_menu` action uses callback function `my_admin_menu`. Within `my_admin_menu`, we assign variable `$page_hook` with the results of the function `add_menu_page` (this adds a top-level settings page).

Afterwards, we use the `admin_print_scripts` action with the `$page_hook` suffix to load a script on our specific plugin page.

You can use the above technique for all of your custom menus. For more ways to add menus to

WordPress, please see the WordPress Codex page on administration menus: http://codex.wordpress.org/Adding_Administration_Menus

Page Detection Conclusion

In this chapter you learned how to perform page detection on the front-end and admin area of a WordPress site.

Page detection allows you to load scripts and styles exactly where necessary. Page detection also helps with page load since unnecessary files are prevented from loading.

For this next chapter, let's move away from scripts and styles a bit, and instead concentrate on some page-loading techniques that will help you on your Ajax journey.

WordPress Loading Techniques

WordPress Loading Techniques

So far we've gone through loading scripts and styles properly, localizing scripts, and how to use page detection.

What's next? Let's just call this chapter the orgy of various techniques for disabling scripts, loading the WordPress environment manually, disabling plugins, and other questionable hacks.

Let's get started and figure out how to override the scripts and styles that other plugins or themes have provided.

Overriding Scripts and Styles

The beauty of queuing scripts and styles is that a rogue plugin author (such as myself) can override and/or disable the queued items at will.

Why in the world would anyone want to do this?

Say, for example, that you have installed a plugin that provides its own styles (let's use WP PageNavi as an example, since it is quite popular).

For those not familiar with WP PageNavi, it's a neat little plugin that gets rid of the Previous/

Next buttons on a theme and instead gives you nice numeric options so you can easily navigate through pages.

I often see themes that support PageNavi include this as a hard-coded plugin, sometimes because the CSS has been heavily customized to suit the theme.

It should be noted that WP PageNavi includes an option to disable its styles, but that's just way too easy. Let's override the styles instead.

I'll show you two ways to override this plugin's styles: one where you completely override the CSS with your own file, and one where you disable the CSS from loading (and include your own in the theme's default CSS file).

Overriding Styles (and Scripts)

Styles make use of the `wp_print_styles` action and the `wp_enqueue_style` function for queuing.

All WordPress actions have a priority argument, which makes it possible to run your code when needed.

In this case, we want to load our code before PageNavi queues its stylesheet.

Assuming we have taken PageNavi's CSS file and placed it in our theme directory, let's set up an action and register the style using the same handler (so that when PageNavi queues the stylesheet, it uses our version).

The following could be placed within your plugin's `init` action, or in the theme's `functions.php` file:

```
<?php
add_action('wp_print_styles', 'wp_pagenavi_style_override',1);
function wp_pagenavi_style_override() {
    wp_register_style('wp-pagenavi', get_stylesheet_
directory_uri() . '/pagenavi-css.css', array(), '2.7', 'all');
}
?>
```

See the “1” parameter there for `add_action`? That's our priority. Since PageNavi doesn't declare a priority (default is 10), we set our priority to run before its style inclusion function.

By registering our style, we reserve the handler name. Even though PageNavi queues its style with its own stylesheet, WordPress sees the handler that we previously registered, and uses that one instead.

Now what if PageNavi declared a priority of one (1) like we did? We can declare our priority to

run after its code and totally delete PageNavi's style reference (so cruel, right?).

However, since PageNavi has a priority of ten (default), let's set ours to eleven (it really could be any integer above ten).

```
<?php
add_action('wp_print_styles', 'wp_pagenavi_style_override',11);
function wp_pagenavi_style_override() {
    wp_deregister_style('wp-pagenavi');
    wp_enqueue_style('wp-pagenavi', get_stylesheet_
directory_uri() . '/pagenavi-css.css', array(), '2.7', 'all');
}
?>
```

Since our code is run after PageNavi's, we de-register its handler (using `wp_deregister_style`) and queue up the style (we could have used a different handler name here, but it's a good idea to stick with the same one).

Overriding styles is that simple. For scripts, it's pretty much the same way (albeit, with different function and action names).

Disabling Styles (and Scripts)

What if you want to completely disable PageNavi's styles and include them instead in the theme's default CSS file?

We'll make use of priorities once more and call `wp_deregister_style`, which will completely disable the style. It's now up to you to provide your own styles within your theme's CSS.

```
<?php
add_action('wp_print_styles', 'wp_pagenavi_style_disable',11);
function wp_pagenavi_style_disable() {
    wp_deregister_style('wp-pagenavi');
}
?>
```

An alternative to de-registering styles is just to remove the action the plugin uses. Here's an example:

```
<?php
add_action('init', 'wp_pagenavi_style_disable');
function wp_pagenavi_style_disable() {
    remove_action('wp_print_styles', array(PageNavi_Core,
'stylesheets'));
}
?>
```

Normally `remove_action` takes a function name as its second argument, but since PageNavi uses a class, we pass an array with the class reference, and the method name.

Loading Just Your Scripts

For advanced users, there may be occasions where you only want to load *just your* scripts. You would do this in order to minimize potential conflicts with other script files.

Situations where you may want to do this are:

- On certain page templates.
- When using ThickBox or Colorbox as an inline frame.

For themers, doing this is relatively simple. Just remove the `wp_head()` call from your `header.php` file.

From there, you queue your script, and pass the `wp_print_scripts` function an array of script handlers.

Here's some example code:

```
<?php
/*Use variants of this code before the </head> tag in your
header.php file*/
wp_enqueue_script('my_script', get_stylesheet_directory_uri() .
'/my_script.js', array('jquery'), '1.0.0');
wp_print_scripts(array('my_script'));
?>
```

Removing the `wp_head` call in `header.php`, however, will cause many plugins to crash and burn (a lot of plugins need the `wp_head` call to load their scripts and styles). Consequently, removing `wp_head` is a fairly bad idea.

What about conditionally removing `wp_head` and printing our scripts for specific templates? Now we're talking.

First, we have to get the post or page ID. Next, we have to see if a custom field named `_wp_page_template` is set. After that, we check for a particular template name and disable `wp_head` accordingly.

```
<?php
$custom_template = false;
if (is_page()) {
    global $wp_query;
    $post_id = (int) $wp_query->get_queried_object_id();
    $template = get_post_meta($post_id, '_wp_page_template',
true);
    switch($template) {
        case "custom-page-fullwidth.php":
            $custom_template = true;
            wp_enqueue_script('jquery');
            wp_print_scripts(array('jquery'));
            break;
    }
}
if (!$custom_template)
    wp_head();
?>
```

In the example, we retrieved the page template name. If one of our pages uses the Full Width template, we disable `wp_head` and load

all our scripts manually (you can do the same with styles using `wp_print_styles`).

Creating Standalone Pages

For those needing to create standalone pages outside of the normal templating system that WordPress provides, you will first need to manually load the WordPress environment.

Uses for this technique include:

- Providing help pages for plugins or themes.
- Providing an Ajax processor for your script file (more on that later).
- Providing a manually created page (not a permalink) that needs WordPress as a framework.

We'll first go over how to call `wp-load.php` directly. However, there is a more elegant way, which I will go over next.

Loading WordPress Manually Using `wp-load`

WordPress' `wp-load.php` is the basic starting point for WordPress. It loads the settings, the config file, gets your plugins working, theme loaded, and then some.

Including `wp-load.php` in any local file will load the WordPress environment, meaning you now have access to all of the back-end code that powers WordPress.

Here's some example code on loading WordPress manually:

```
<?php
/*The use of the dirname functions depend on the hierarchy of
your file. Adjust them as needed.*/
//Code should be used before the </head> tag of your file.
$root = dirname(dirname(dirname(dirname(dirname(__FILE__))))));
if (file_exists($root.'/wp-load.php')) {
    // > WP 2.6
    require_once($root.'/wp-load.php');
    /*Run custom WordPress stuff here */

    //Output header HTML, queue scripts and styles, and
include BODY content
    wp_enqueue_script('my_script', get_stylesheet_directory_
uri() . '/my_script.js', array('jquery'), '1.0.0');
    wp_print_scripts(array('my_script'));
}
?>
```

Please note that the `$root` variable will have to be modified depending on where your page lies in the directory structure. You may have to use more, or less, `dirname` function calls (trial and error is your good, albeit always drunk, friend).

This technique assumes you know where `wp-load.php` is. If you're working on a client site, chances are you know where it is.

If you're a plugin or theme author writing for the masses, however, the location is unpredictable.

This is because users can choose to move their `wp-content` directory wherever they choose. Since your theme and/or plugin resides in this directory, finding `wp-load.php` is at best a guessing game.

The more elegant way to accomplish standalone pages is through query variables and a template redirect.

Loading WordPress Using a Template Redirect

When you are performing a template redirect, you are (in effect) taking control and dictating to WordPress which path to take. You can load custom templates, perform 301 redirects, set headers, and much more.

In this case, we'll be using a template redirect to load a custom page based on query variables. Since the WordPress environment has already loaded by the time it gets to our redirect, we no longer have to worry about including `wp-load.php` directly.

First, let's begin by setting up one action and one filter and pointing them to some helper functions (`load_page` and `query_trigger` respectively).

```
<?php
add_action('template_redirect', 'load_page');
add_filter('query_vars', 'query_trigger');
?>
```

You can place the above code in your theme's `functions.php` file, or include it in your plugin's base file (if writing a plugin). Please keep in mind I made the function names rather generic. It's up to you to come up with something more unique in order to avoid name conflicts.

The `query_vars` filter enables us to add a query variable to WordPress. The `template_redirect` action allows us to capture this query variable and perform an action on it.

Let's start with the `query_trigger` function first.

```
<?php
function query_trigger($queries) {
    array_push($queries, 'my_page');
    return $queries;
}
?>
```

With the `query_trigger` function, we are automatically passed a list of the existing WordPress query vari-

ables. It's our job to add in our query (in this case, `my_page`) and return the `$queries` array.

Now we have to do something with our query variable, so let's move on to the `load_page` function.

```
<?php
function load_page() {
    $pagepath = WP_PLUGIN_DIR . '/my-plugin-dir/';
    switch(get_query_var('my_page')) {
        case 'help.php':
            include($pagepath . 'help.php');
            exit;
        case 'changelog.php':
            include($pagepath . 'changelog.php');
            exit;
        default:
            break;
    }
}
?>
```

The first thing the `load_page` function does is establish an include path, which is captured in the `$pagepath` variable.

The function then performs a `switch` statement in order to capture the query variable (if applicable).

Right now, we have two cases: one for `help.php` and another for `changelog.php`. If any of those are found, we load the appropriate page.

So here's how it works. Instead of pointing your browser to a WordPress page, you would point it to a query.

Using `http://mydomain.com/?my_page=help.php` would load the `help.php` file.

Using `http://mydomain.com/?my_page=changelog.php` would load `changelog.php`.

The benefits of this technique? No having to search for `wp-load.php`. You also have access to all the available WordPress functions, classes, actions, filters, and so on.

Please note in both `help.php` and `changelog.php`, you will have to queue your scripts like in the `wp-load` example shown earlier.

So in order to load scripts on a standalone page, you will need to:

- Load the WordPress environment (via `wp-load.php` or using a template redirect).
- Queue your scripts for later inclusion.
- Call the `wp_print_scripts` function to load your scripts.

Preventing Plugins From Loading

To call this next technique a hack is giving it a big compliment.

There may be cases where you will want to prevent WordPress plugins from loading when creating a standalone page (to prevent caching, conflicts, or for some other weird and insane reason).

If you look at your `wp-settings.php` file, there is a call to a function named `wp_get_active_and_valid_plugins` right before it loads your active plugins.

Within this function, it checks to see if the constant `WP_INSTALLING` is defined. If it is, WordPress doesn't load any plugins.

So to “trick” WordPress into not loading plugins, you have to define the `WP_INSTALLING` constant. Afterwards, you load the WordPress environment.

After that, we can manually load any plugins desired.

Here's some sample code demonstrating the concept:

```

<?php
header('Content-Type: text/html');
define('WP_INSTALLING', true);
//Adjust the dirnames to match the path to your wp-load file.
$root = dirname(dirname(dirname(dirname(dirname(__FILE__))))));
if (file_exists($root.'/wp-load.php')) {
    // WP 2.6
    require_once($root.'/wp-load.php');
} else {
    // Before 2.6
    require_once($root.'/wp-config.php');
}
$plugin = 'your-plugin-directory/your-plugin-file.php';

// Validate plugin filename
if ( !validate_file($plugin) && '.php' == substr($plugin, -4) &&
file_exists(WP_PLUGIN_DIR . '/' . $plugin)) {
    include_once(WP_PLUGIN_DIR . '/' . $plugin);
}
unset($plugin);
?>

```

The above code demonstrates how to load just one plugin in order to use its functions if necessary.

As you can see, we set the content-type, define `WP_INSTALLING`, and load the WordPress environment manually.

Afterwards, we specify an absolute path to our plugin file that we want to load, validate it, make sure the file exists, and then include the file as part of WordPress.

Trial and error is your good, albeit always drunk, friend.

If you have multiple plugins to load, you can use a `foreach` statement.

```
$current_plugins = get_option( 'active_plugins' );
if ( is_array($current_plugins) ) {
    foreach ( $current_plugins as $plugin ) {
        // $plugin looks like: your-plugin-dir/your-plugin-file.
php
        switch($plugin) {
            case: 'yourplugin':
            case: 'anotherplugin':
                break;
            default:
                continue;
        }
        if ( !validate_file($plugin) && '.php' ==
substr($plugin, -4) && file_exists(WP_PLUGIN_DIR . '/' .
$plugin) ) {
            include_once(WP_PLUGIN_DIR . '/' . $plugin);
        }
        unset($plugin);
    }
unset($current_plugins);
```

I would advise that you use this technique on a case-by-case basis.

If writing code for the masses, do not use this technique as the path to `wp-load.php` is unpredictable. However, for personal use and/or client sites, this technique should be acceptable since you'll likely know where `wp-load.php` can be found.

WordPress Loading Techniques Conclusion

Within this chapter you learned how to:

- Override scripts and styles
- Load just your scripts
- Create a standalone page
- Prevent plugins from loading

Some of these techniques are just out-right hacks, but I'll leave it up to you to decide which ones.

Since we've gotten all of the basics out of the way, let's move on to doing a little bit of jQuery.

Properly Formatting jQuery for WordPress Use

Properly Formatting jQuery for WordPress Use

Did I mention we would be using the uber-awesome jQuery library for these examples?

Since we are, it's a good idea to make sure your jQuery code is formatted appropriately for WordPress.

Topics covered in this section will be:

- Namespacing.
- Public properties/functions.
- Private variables/functions.

You may have your own preference on how to code your own jQuery scripts, but the examples I give are based on various painful experiences of trial and error (remember my drunk friend?). In my examples, I'm not going for creative programming; I'm going for readability.

Namespacing

For those not familiar with namespacing, let me give you a brief example.

Say that there are two plugins, both having the function of `readCookie`, which is not that un-

common since it is a very usable function from QuirksMode.

The result would be a JavaScript error and a possibly non-functioning script.

Namespacing resolves this issue by enclosing the `readCookie` function inside of a unique name that can only be called from within the namespace. The result? No conflicts!

So how does one namespace?

Let's start off with some basic jQuery starter code that will spit out an alert box after the page has finished loading:

```
jQuery(document).ready(function() {  
    var $ = jQuery;  
    alert("HI");  
});
```

The `jQuery(document).ready()` code starts initializing before a window has finished loading.

An additional benefit is you can use the above code ad-inifitum, unlike the selfish `onLoad` event (theme and plugin authors using the `onLoad` event are just asking to get their butts kicked by other developers).

One thing you might notice is the `$` symbol. It's a symbol that jQuery, Prototype, and MooTools use. Since WordPress runs jQuery in no-conflict mode, you must use the jQuery object instead of the `$` symbol.

As a sidenote, I highly recommend not using MooTools with WordPress simply because of the conflicts it has with the jQuery and Prototype libraries (I'm not knocking MooTools or anything, but it just doesn't play well with the libraries WordPress primarily uses).

Since I like using the `$` symbol for the sake of simplicity, I declare the `$` symbol inside the scope of the load event and assign it the jQuery object.

From there, we use the jQuery plugin authoring guidelines to assign our namespace.

Here's an example:

```
jQuery(document).ready(function() {  
    var $ = jQuery;  
    $.my_script_namespace = {};  
});
```

As you can see from the above code, I created a namespace called `my_script_namespace` using the `$`

Never, ever, use the `onLoad` event to initialize a script. You might just find some nasty "surprises" in your mail.

reference. Creating a namespace is as simple as that.

Public Properties/Functions

Since we're not creating a jQuery plugin, we'll stray slightly from the "guidelines".

There are certain variables, properties, and functions you will want others to be able to publicly call.

Here's an example of each type:

```
jQuery(document).ready(function() {
  var $ = jQuery;
  $.my_script_namespace = {
    init: function() {
      alert("Initializing");
    },
    my_function: function(obj) { /*do some stuff*/
      _my_function($(obj));},
    vars: {}
  }; //end my_script_namespace
});
```

Namespacing prevents function naming conflicts.

The `init` function is your publicly accessible initialization function call. You execute code directly inside this function without referencing other functions.

The `my_function` function is probably as close to a property as you'll get in JavaScript. You can exe-

cute code, and call a private function (which we'll declare later).

The variable `vars` is declared as an empty object, which can be added to from an additional private variable of the same name (not declared yet).

Private Variables/Functions

Private variables and functions are necessary for when you don't want other scripts to call your functions directly (use a property for indirect access).

A script with private variables and functions looks like this:

```
jQuery(document).ready(function() {
    var $ = jQuery;
    $. my_script_namespace = {
        init: function() {
            alert("Initializing");
        },
        my_function: function(obj) { /*do some stuff*/
            _my_function($(obj));},
        vars: {}
    }; //end my_script_namespace
    //Private variables/functions are declared outside of
the namespace
    var myvars = $.my_script_namespace.vars; //private
variable
    function _my_function(obj) { /*private function*/
        //my private code here
    }
});
```

Notice that our namespace ended before the declaration of the private functions and variables. Also note the commas separating the properties, functions, and variables in the namespace.

All of the functions and variables (that are private) are now within the scope of the `jQuery(document).ready()` enclosure. Since they exist only within the scope of the enclosure, you will be able to avoid the conflict of having duplicate JavaScript functions.

The beauty of this is that the namespace'd functions can call the in-scope functions. Furthermore, the private variables can reference the namespace'd variables.

If you've been looking closely at the code, you'll observe that nothing is being done here. After the namespace is added, no function is being called. The namespace is there, but nothing is happening.

It's as if the namespace doesn't exist (cue scary music).

The best way I've found to load code is to call a public initialization function immediately before the ending of the loading enclosure:

```
jQuery(document).ready(function() {
    var $ = jQuery;
    $. my_script_namespace = {
        init: function() {
            alert("Initializing");
        },
        my_function: function(obj) { /*do some stuff*/
            _my_function($(obj));},
        vars: {}
    }; //end my_script_namespace
    //Private variables/functions are declared outside of
the namespace
    var myvars = $.my_script_namespace.vars; //private
variable
    function _my_function(obj) { /*private function*/
        //my private code here
    }
    $.my_script_namespace.init();
});
```

By placing the `$.my_script_namespace.init()`; towards the end, you ensure that the rest of the code has finished loading. From there, you can call the public initialization function (which currently displays an alert box to show it's working).

Advanced users (such as other plugin authors) may need to call this from outside the original script.

In this case, you'll want to use the above script as a dependency, and use your own load event to call the script using the regular jQuery namespace.

```
jQuery(document).ready(function() {  
    jQuery.my_script_namespace.init();  
});
```

Including Your jQuery Script in Your Theme

Let's use the jQuery code we developed in this chapter and put it into a WordPress theme.

Yes, it just spits out an alert box, but this will show you how to take a script and add it in.

Now one question I often get regarding scripts is why not to put the script directly into `header.php` using the `<script>` tags? Well, for one, it's incredibly difficult to specify dependencies. And another reason, it's impossible to disable the script programmatically, which is a major benefit of queueing scripts.

That being said, let's take our final JavaScript code and place it in your theme's root directory with a file named `init.js` (I always like having an `init.js` file with my theme just so I can initialize events and such).

Now open up `functions.php` and we'll set up a script that only loads on the front-end of a site.

Here's the code we'll use in `functions.php`:

```
<?php
add_action('wp_print_scripts', 'my_theme_js_init');
function my_theme_js_init() {
    if (is_admin()) return;
    wp_enqueue_script('my_theme_init', get_stylesheet_
directory_uri() . '/init.js', array('jquery'));
}
?>
```

With the above code, you'll see an alert box when you visit each front-end page.

Conclusion for Properly Formatting jQuery for WordPress

In this section you learned how to properly format jQuery code for WordPress.

The examples given are my personal guidelines from painful experience, but you're welcome to experiment with your own techniques.

So the foundation has been laid. You know how to load scripts and styles properly. You know how to localize scripts and perform page detection. And now you know some pretty nifty loading tricks.

Before we move to our first Ajax request, let's learn a little about WordPress security.

Nonces and the Escape API

Nonces and the Escape API

Helping secure WordPress is always a good thing. The golden rule of WordPress security is, “Never trust anyone, even yourself.”

Security involves making sure requests have originated from the correct location, and escaping untrusted sources of information (which can come from user input, malicious localization, and even your own database).

The use of WordPress nonces and the Escape API (as I’ll call it) assist in securing your WordPress plugin or theme. And using them with Ajax requests is a must.

Let’s begin with nonces, since you’re probably wondering what the heck that is.

WordPress Nonces

A WordPress nonce (a number used once, get it?) is a way of ensuring a request originated from the correct source.

Okay, so that didn’t make a whole lot of sense, did it?

Let me explain with an analogy.

You have checked into a hotel and are given an electronic room key for room 215.

What if, while you're out on the town, you drop your room key? Someone could pick it up, but (magnetic hacking aside) that person would have no idea what room the key went to.

Furthermore, this room key expires the moment you check out, or even as you request a new room key, thus rendering the key good for only a short amount of time.

The room key is your nonce, and each nonce is associated with an action, which in this example would be, "Open room 215."

If someone retrieved this room key, all they would be able to potentially do is enter your room. The person couldn't steal your car, enter your house, or steal your child's candy.

Nonces behave almost the same way. They are associated with a specific action, have an expiration, and require a specific key (in the form of a unique string) that will allow the action to proceed.

WordPress nonces help protect you from a common exploit called Cross-Site Request Forgery (or CSRF).

Say, for example, you are logged in as an admin user for a WordPress site. Somebody you trust sends you an e-mail and asks you to visit the site. You click on the link.

What you didn't realize, however, was the link was set up to exploit a user-related WordPress plugin that would add a user to your WordPress site.

Since a nonce wasn't used, the plugin has no way of knowing that you "intentionally" wanted to add this user. Instead, the plugin is simply relying on your admin status to perform the action.

If a nonce were used, this request would have been stopped cold. The person sending the e-mail would have to know the nonce key, and also have registered the action within WordPress. And there's a time limit, as the nonces are valid for only 24 hours (the keys change every 12 hours).

So what are ideal situations when nonces should be used?

- When performing actions in an administrative panel (plugin settings pages, theme settings pages, custom meta for posts, etc.)
- When storing items in the WordPress database
- When retrieving items from the WordPress database that require a logged-in status (profile information, options)

As a reminder, nonces are action based. If this action needs protection, then nonces are necessary to ensure the request originated legitimately.

Now that the nonce terminology is out of the way, what are some of the applications for using nonces?

Nonces and Forms

When using nonces with forms, you can make use of the WordPress function `wp_nonce_field`.

The function `wp_nonce_field` takes in four arguments, all of which are optional.

```
<?php wp_nonce_field( $action, $name, $referer, $echo ) ?>
```

- `$action` - The nonce action you would like to take. An example is: `my-form_submit`.
- `$name` - The nonce name (default is `_wpnonce`). I recommend always using this parameter since WordPress' default is `_wpnonce` and you wouldn't want to conflict (especially when creating forms in the WordPress admin area).
- `$referer` - A boolean on whether to set the referer field or not. I usually just set this to true.
- `$echo` - Whether to echo out the result, or return it (default is `true`). If you are creating a form as a shortcode or as a string and need the result returned, set this option to false.

Here's some example code that I've placed in a WordPress page template (you could use a shortcode instead).

```
<form id='registration-form' method="post" action="<?php echo
$_SERVER["REQUEST_URI"]; ?>">
<?php wp_nonce_field( 'submit-form_registration', '_
registration_nonce', true, true ); ?>
<!--form data-->
<input type='submit' value='Submit Registration'
name='registration-submit' id='registration-submit' />
</form>
```

The above code has a nonce field that is echoed out. The action name is `submit-form_registration` and the nonce name is `_registration_nonce`.

The form posts to itself (i.e., the same page), so when we begin processing the form data, we would make use of the `check_admin_referer` WordPress function to verify the nonce.

```
<?php
if ( isset( $_POST['registration-submit'] ) ):
    check_admin_referer( 'submit-form_registration', '_
registration_nonce' );
    echo "Success!";
endif;
?>
```

The `check_admin_referer` function takes in two arguments: the nonce action and the nonce name. If the nonce isn't valid, WordPress outputs a message (either a "-1" or a "Are you sure you want to do this?" message).

An alternative to `check_admin_referer` is the WordPress function `wp_verify_nonce`.

The function `wp_verify_nonce` takes in two arguments: the actual nonce and the nonce action.

Here's an example of the same type of verification using `wp_verify_nonce`.

```
<?php
if ( isset( $_POST['registration-submit'] ) ):
    if ( !wp_verify_nonce( $_REQUEST['_registration_nonce'],
'submit-form_registration' ) ) {
        die("Request is not valid");
    }
endif;
?>
```

Both `check_admin_referer` and `wp_verify_nonce` accomplish the same thing: stopping an unintentional request cold.

I prefer using `wp_verify_nonce` since I have more control over the output. However, `check_admin_referer` is simpler and requires less code. It's your call on how you want to verify nonces.

Nonces and URLs

If you have comments enabled on your site, chances are you have received e-mails when someone leaves a comment on a post. These comment e-mails contain several links, notably the ones to delete or mark the comment as spam. These links have nonces attached to them.

There are two functions that will help you create a link that has a nonce attached: `wp_nonce_url` and `wp_create_nonce`.

Once again, I just created a page template that spits out a link. The link (when clicked) goes to the same page, but with a nonce variable attached as a GET variable.

The `wp_nonce_url` function takes in two arguments: the URL to visit, and the nonce action.

Here's some sample code that would be placed inside the loop:

```
<?php
if ( isset( $_REQUEST['_wnonce'] ) ):
    if ( !wp_verify_nonce( $_REQUEST['_wnonce'], 'sample-
link-action' ) ) {
        die("Request is not valid");
    }
endif;
?>
<?php
$my_site_url = wp_nonce_url( get_permalink() , 'sample-link-
action' );
?>
```

The nonce action used here is called `sample-link-action`. When clicked, the nonce is checked.

Let's modify the above code to use `wp_create_nonce` instead. The `wp_create_nonce` function only takes in one argument, which is the nonce action name.

It's up to us to define the query variable (i.e., a `GET` variable) that will be appended to the URL.


```
<?php
if ( isset( $_REQUEST['_sample_link_nonce'] ) ):
    if ( !wp_verify_nonce( $_REQUEST['_sample_link_nonce'],
'sample-link-action' ) ) {
        die("Request is not valid");
    }
endif;
?>
<?php
$nonce = wp_create_nonce( 'sample-link-action' );
$my_site_url = get_permalink() . '?_sample_link_nonce=' .
$nonce;
?>
```

As you can see from the code, a query variable called `_sample_link_nonce` is created and assigned the value from `wp_create_nonce`.

So when would you use `wp_create_nonce` over `wp_nonce_url`? It's all about control, as in the case of `check_admin_referer` versus `wp_verify_nonce`.

Both will get you to the same result, but `wp_create_nonce` gives you more control on naming the nonce name and forming the URL.

Nonces and Ajax

Nonces and Ajax aren't exactly straightforward, but you'll see how Ajax nonces are implemented throughout the various Ajax examples in this book.

Basically, you would create either a nonce in a form field or as a URL (using the techniques already presented in this chapter) and use JavaScript to capture the nonce value.

You would then use Ajax to pass this nonce value to your Ajax processor. The Ajax processor will then verify the nonce.

The two functions to assist you in verifying Ajax nonces are: `check_ajax_referer` (same arguments as `check_admin_referer`) and (slightly repeating myself) `wp_verify_nonce`.

Once again, it's about control. The `check_ajax_referer` function assumes a `$_REQUEST` variable of `_ajax_nonce`. With `wp_verify_nonce`, the nonce variable can be defined in the function call.

So which is better? It's your call.

The Escape API

The WordPress Escape API (a.k.a. Data Validation) is a way for you to sanitize untrusted pieces of information.

So what information can't be trusted?

- Information you input (yes, you can't even trust yourself)

- Information others input
- Information in your own database (just because it's already there doesn't mean it's safe)

In summary, you can't trust anyone or anything that comes in contact with your WordPress installation. This is why data validation is so crucial.

Let's go over several functions you would use to ensure the integrity of your data using both the Escape API and several PHP functions.

Validating Numeric Inputs

If you create a function or code snippet that takes a value that should be an integer, it's a good idea to ensure that the data is what you think it is.

To verify that an input is an integer, you can use the PHP function `intval`.

```
<?php
function my_test_input( $post_id = 0 ) {
    echo "Before: $post_id";
    $post_id = intval( $post_id );
    echo "After: $post_id";
}
my_test_input("sdlkjflkjdf");
?>
```

In the above example, I pass a string variable. The result is:

```
Before: sdlkjflkjdf  
After: 0
```

You can use the WordPress function `absint` to ensure that the integer isn't negative:

```
<?php  
echo absint( "-40" ); //Output is 40  
?>
```

The PHP `is_numeric` function can also be used to verify that the input is indeed a number:

```
<?php  
echo is_numeric( 9.4 ); //Output is 1 or true  
?>
```

Escaping HTML

Escaping HTML input is useful for storing data in the WordPress database, for outputting HTML text to the screen, and for placing HTML text in textareas.

Here's an example:

```
<?php  
$string = "<strong><em>Test HTML Input</em></strong>";  
echo esc_html( $string );  
//Outputs: &lt;strong&gt;&lt;em&gt;Test HTML Input&lt;/em&gt;&lt;/strong&gt;  
?>
```

As you can see from the output, `esc_html` encodes the various HTML tags (`esc_html` also encodes ampersands, single quotes, and double quotes).

If you wish to use translations with `esc_html`, you would append `__` (for returning) or `_e` (for echoing) to the function.

```
<?php
$string = "<strong><em>Test HTML Input</em></strong>";
esc_html_e( $string, 'UniqueLocalizationName' );
//Outputs: &lt;strong&gt;&lt;em&gt;Test HTML Input&lt;/em&gt;&lt;/strong&gt;
?>
```

Escaping Attributes

In general, every piece of text that goes within an attribute should be escaped using the WordPress function `esc_attr`. As with `esc_html`, you can append `__` or `_e` for translations.

Here's an example of using `esc_attr` to output to a form input.

```
<?php
$string = "'My <> String in quotes'";
?>
<input type="text" value="<?php echo esc_attr( $string ); ?>"
/>
<?php
//Outputs: <input type="text" value="&quot;My &lt;&gt; String
in quotes&quot;" />
?>
```

Using `esc_attr` shouldn't be limited to forms. If it goes in an attribute, a WordPress best practice is to escape the text using `esc_attr`.

Escaping JavaScript

If you're creating a JavaScript variable with a PHP value, the `esc_js` function will escape the value. The main uses of the function are for escaping single quotes and double quotes.

The best case I've seen for using `esc_js` is when localizing JavaScript variables (using `wp_localize_script`).

```
<?php
function add_scripts() {
    global $post;
    wp_enqueue_script( 'ajax-registration-js', plugins_url(
'js/registration.js' ,__FILE__ ), array( 'jquery', 'wp-ajax-
response' ), '1.0' );
    wp_localize_script( 'ajax-registration-js',
'ajaxregistration',
    array( 'Ajax_Url' => esc_url( admin_url( 'admin-
ajax.php' ) ),
        'DatabaseText' => esc_js( intval( $post->ID ) ),
        'TranslationText' => esc_js( __( 'My Text',
'UniqueLocalizationName' ) ) )
    ) );
}
?>
```

Escaping URLs

You shouldn't trust URLs people enter as well. If a URL is formed maliciously, it can wreak all

kinds of havoc. Fortunately, WordPress has the `esc_url` function.

One thing to take note of here is it is generally a good practice to escape all URLs (even if they come from your own database). Here's an example:

```
<?php
echo "<a href='" . esc_url ( site_url( "/<sampletext'&%" ) ) .
"'>Escaped URL</a>";
//The output: <a href='http://localhost:8888/ronalfy/sampletex
t&#039;&#038;%>Escaped URL</a>
?>
```

Please observe how the ”<” character was removed and the single quote and ampersand were encoded.

Filtering HTML

If you are accepting input from users that is HTML, it's a good idea to filter the input.

The WordPress function `wp_kses` allows you to filter the HTML input. You can use the WordPress global `$allowedposttags` to use the WordPress list. Or, you can create your own.

Let's filter the following link a user has submitted:

```
<a href='http://www.wordpress.org' class='link' id='wordpress-link' title='WordPress.org'>WordPress.org</a>
```

Let's get a little strict with our attributes. Let's only accept the "href" and "title" attributes. Everything else must be stripped out.

The technique is to build our own `KSES` array and use the `wpkses` function to filter the input.

The `KSES` array will only allow one HTML tag and the two attributes we want to allow.

```
<?php
$customkses = array(
    'a' => array(
        'href' => array (),
        'title' => array ()
    ) );
echo wpkses( "<a href='http://www.wordpress.org' class='link' id='wordpress-link' title='WordPress.org'>WordPress.org</a>",
$customkses );
?>
```

And here's the resulting output:

```
<a href='http://www.wordpress.org' title='WordPress.org'>WordPress.org</a>
```

As you can see from the `KSES` example, you can filter the HTML so only the tags and attributes you allow are accepted.

Escape API Conclusion

Within this chapter I've covered several PHP and WordPress functions to assist in data validation.

There are many more functions available, and I encourage you to read over the Data Validation section at the WordPress Codex: http://codex.wordpress.org/Data_Validation

Sending Our First Ajax Request

Sending Our First Ajax Request

By now, you should know how to do the following:

- To properly add scripts to WordPress.
- To create well-formed jQuery scripts.
- To localize scripts.

It's now time to work on our first Ajax request.

We'll be doing a simple Ajax call to the WordPress back-end to get the total number of comments on a website. It's simple, but it'll demonstrate what's needed to make that first Ajax request (gotta crawl before you walk).

In this example, I'll be making use of themes. The later (and more advanced) examples will make use of plugins (so don't go crying to mama just yet plugin authors).

This example spans two chapters and will cover how to manually create an Ajax processor. However, WordPress has its own built-in Ajax processor, and I'll demonstrate how to use it after we have our bases covered here.

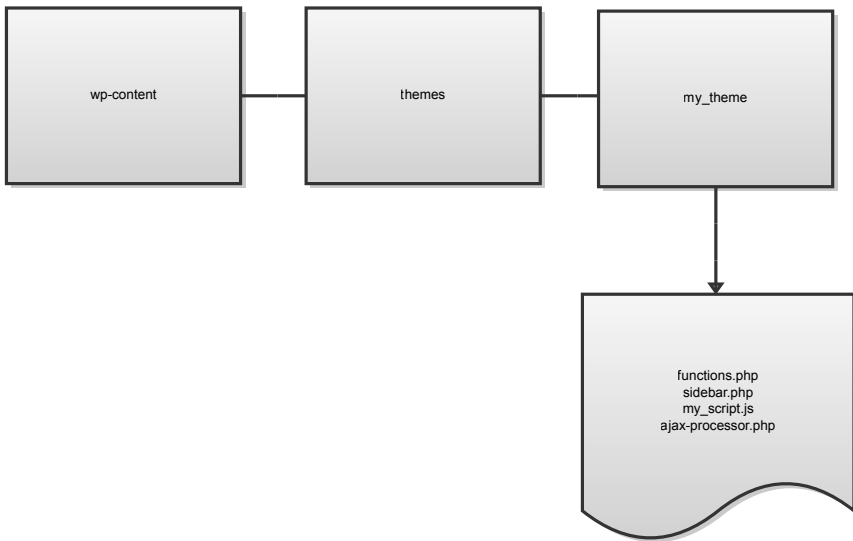


Figure 2. Base Files

Here are the files we'll be working with:

- `functions.php`
- `sidebar.php`
- `my_script.js`
- `ajax-processor.php`

Set Up the PHP Class to Handle Back-end Operations

We'll first place our PHP class in our `functions.php` file. For those not familiar with `functions.php`, it allows a theme to have access to a handful of custom functions throughout the theme. It's also a good way to hard-code in plugins.

I've created the following class to include in our `functions.php` file.

```
<?php
if (!class_exists('myThemeClass')) {
    class myThemeClass {
        /**
         * PHP 4 Compatible Constructor
         */
        function myThemeClass(){$this->__construct();}
        /**
         * PHP 5 Constructor
         */
        function __construct(){
            //initialization code here
        }
        function get_comments() {
            //get comment code here
        }
    } //End Class myPluginClass
}
//instantiate the class
if (class_exists('myThemeClass')) {
    $myClassVar = new myThemeClass();
}
?>
```

The above PHP class is very scaled down, but it sets up a decent structure for future growth.

Since we are going to be retrieving comments using the WordPress back-end, I've added in the `get_comments` method.

Use *functions.php* for theme-wide custom functions.

Setting up the Interface

The next step in sending our Ajax request is to set up some kind of interface that the user will interact with on the client-side.

In this particular case, it's just a link. Yep, a simple link!

We're going to add this link in our `sidebar.php` file.

First I'm going to assign a variable our `href` portion of the link.

```
<?php
$link_url = esc_url(wp_nonce_url( site_url('?my_page=ajax-processor&action=getcomment'), "my-theme_getcomment"));
?>
```

There are a few things going on here. The `wp_nonce_url` function is necessary for securing links. It prevents you from following a malicious script and performing an unintended action on the back-end.

Since we're not accessing anything serious, the `wp_nonce_url` is overkill, but learning it now will help you later if you want to secure your Ajax requests.

In this case, we pass `wp_nonce_url` our URL and a unique identifier that combines the name of our theme to the action it wants to take.

The `site_url` function is used because we want our site's base URL. What's passed is a relative path, and WordPress formats the URL appropriately.

Finally, we use the `esc_url` function, since it's a WordPress best-practice to sanitize data coming from the database.

The `$link_url` now has a value similar to this:

```
http://www.yourdomain.com/?my_page=ajax-processor&
action=getcomments&_wpnonce=1d8a910922
```

We now have two variables to work with when we deal with our Ajax request: `action`, and `_wpnonce`. Please keep in mind the `_wpnonce` value was generated based on the unique string we used with `wp_nonce_url`, which in this case is `my-theme_getcomments`.

Also of note is our `action=getcomment` variable. Since it's likely that our Ajax processor will be taking various inputs, we specify what action we want it to take.

Our final link code in the `sidebar.php` file would look like this:

```
<?php
$link_url = esc_url(wp_nonce_url( site_url('?my_page=ajax-processor&action=getcomment'), "my-theme_getcomment"));
?>
<a href='<?php echo $link_url; ?>' id='get-comments'><?php
_e('Get Comments', 'get-comments'); ?></a>
<div id="get-comments-output"></div>
```

I gave the link an `id` of `get-comments` in order to capture it later for events inside of jQuery. I also placed an empty `div` with an `id` of `get-comments-output` for displaying our output (more on that later).

Notice that I used another localization function, `_e`. This echoes out the string rather than returning it, as opposed to the `__` function. We'll add in our localization code later.

When viewing your theme, you should now see the `Get Comments` link.

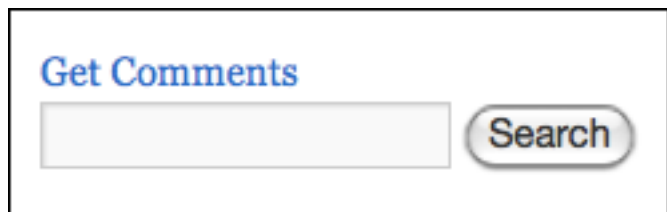


Figure 3. Get Comments Link

Now it's time to set up our JavaScript file.

Setting Up the JavaScript File

We'll now be working with our `my_script.js` file, which should be empty at this point.

First, let's add in our loading code foundation:

```
jQuery(document).ready(function() {  
    var $ = jQuery;  
});
```

Now, let's create a namespace of `get_my_comments` with a public `init` function. We'll also be adding a call to the `init` function as well.

```
jQuery(document).ready(function() {  
    var $ = jQuery;  
    $.get_my_comments = {  
        init: function() { /*my init code */  
    }; //End of get_my_comments namespace  
    $.get_my_comments.init();  
});
```

Now we have a good foundation going. As soon as the document has loaded, our initialization function will run. As of right now, the script hasn't been added to your theme yet, but we'll get to that soon.

The next step is adding an event to capture when the user clicks on our `Get Comments` link. The event code will be placed inside our `init` function.

Within the event, we'll call a private function named `_do_ajax` and pass it our link object.

```
jQuery(document).ready(function() {
    var $ = jQuery;
    $. get_my_comments = {
        init: function() {
            $("#get-comments").bind("click",
function() {
                _do_ajax(this); //Call to our private
function
                return false;
            });
        }
    }; //End of get_my_comments namespace
    function _do_ajax(obj) {
        var element = $(obj); //our link object
    } //end _do_ajax
    $.get_my_comments.init();
});
```

We have now captured the click event for our link, but nothing is happening yet other than calling our `_do_ajax` function.

Let's begin setting up our Ajax object.

In our `init` function, we call the private function `_do_ajax` and pass it our `link` object.

We then create the `_do_ajax` function, and create the variable `element` to hold our object. We place the `$()` around the `obj` variable in order to use jQuery actions with it.

```
function _do_ajax(obj) {  
    var element = $(obj); //our link object  
    var url = wpAjax.unserialize(element.attr('href'));  
}
```

The `wpAjax` object comes from one of our script dependencies that we haven't defined yet (`wp-ajax-response`).

It takes the `href` attribute of the passed element and creates an object called `url` with it.

The `url` object now has two variables we can access via JavaScript (`url._wponce` and `url.action`). We can also use the `href` attribute to capture the path to our Ajax processor, but I want to show you how to capture this path via localized JavaScript since not all Ajax requests involve links.

Setting up the Ajax Object

We'll be building on the existing `_do_ajax` function to set up our Ajax object.

I'll first create an object with the name of `s` to hold our Ajax data. Why `s`? Because it's short and sweet (like my alliteration?).

With our `s` object, we'll define our parameters for use with jQuery's Ajax options.

```
function _do_ajax(obj) {
    var element = $(obj); //our link object
    var url = wpAjax.unserialize(element.attr('href'));
    var s = {};
    s.response = 'ajax-response';
    s.type = "POST";
    s.url = mythemegetcomments.ajax_url;
    s.data = $.extend(s.data, { action: url.action, _ajax_
nonce: url._wpnonce });
    s.global = false;
    s.timeout = 30000;
} //end _do_ajax
```

There's a lot going on here. First, I define the `s` object. I then set the response type using `s.response`.

I also set the Ajax request to be POST, give it a URL to the Ajax processor (using `mythemegetcomments.ajax_url`, which we'll create using localization later), and assign data that will be passed to our Ajax processor.

The `s.data` object is used to pass variables to our Ajax processor (in this case as POST variables). The variables passed will be `action` and `_ajax_nonce` (holds the values of our action name and nonce value respectively).

We have two objects in our script that are undefined right now: `wpAjax` and `mythemegetcomments`.

Let's move back to our `functions.php` file and add them in.

Finalizing the `functions.php` Class

Earlier in this chapter, I gave you the foundation for a PHP class with a name of `myThemeClass`.

Here's what we need to finalize this class:

- Finish the `get_comments` method.
- Add in our JavaScript files.
- Add in JavaScript localization.
- Add in String localization.

The `get_comments` Method

Let's concentrate on our `get_comments` method first.

What we'll do is call the `wp_count_comments` function and return its content.

```
function get_comments() {  
    return wp_count_comments();  
    /*wp_count_comments returns an object with values of  
    trash, spam, approved, and moderated*/  
}
```

The `wp_count_comments` function returns an object with variables `approved`, `moderated`, `spam`, and `trash`.

Add in our JavaScript Files

In the constructor, let's add the `wp_print_scripts` action to point to a method conveniently named `add_scripts`.

```
function __construct(){
    add_action('wp_print_scripts', array(&$this, 'add_
scripts'));
}
```

Now we can concentrate on our `add_scripts` method.

```
function add_scripts() {
    if (is_admin()) return;
    wp_enqueue_script('my_script', get_stylesheet_directory_
uri() .'/my_script.js', array("jquery", "wp-ajax-response" ,
"2.3"));
    wp_localize_script( 'my_script', 'mythemegetcomments',
$this->get_js_vars());
} //End add_scripts
```

You might remember from our `my_scripts.js` file that we had an undefined object of `wpAjax`. The dependency `wp-ajax-response` is the script that initializes that object.

The next thing going on is we queue our script (giving it a handle of `my_script`) and give it the dependencies of `jquery` and `wp-ajax-response`.

Finally, we call the `wp_localize_script` function, pass it our script's handler name, tell it to create a JavaScript object named `mythemegetcomments`, and pass it a list of variables to localize (via the `get_js_vars` method, which will return an array of strings to localize).

Let's move on to JavaScript localization.

Add in JavaScript Localization

We'll be calling several strings that need to be localized in our `my_scripts.js` file. Here are the strings that need to be localized:

- You have
- approved
- comments
- in moderation
- trashed
- spam

When we used `wp_localize_script`, we called a method named `get_js_vars`. Let's define this method and have it return an array of localized strings.

```
function get_js_vars() {
    return array(
        'ajax_url' => site_url('?my_page=ajax-processor'),
        'you_have' => __('You have', 'get-comments'),
        'approved' => __('approved', 'get-comments'),
        'comments' => __('comments', 'get-comments'),
        'in_moderation' => __('in moderation', 'get-
comments'),
        'trashed' => __('trashed', 'get-comments'),
        'spam' => __('spam', 'get-comments')
    );
} //end get_js_vars
```

Do you remember how earlier we used the JavaScript object `mythemegetcomments.ajax_url` to get the path to our Ajax processor? This is where that object comes from.

Since we passed `wp_localize_script` an object name of `mythemegetcomments`, WordPress will create that JavaScript object and assign a variable named `ajax_url` with the URL to our Ajax processor.

We also again make use of the `__` function with the localization name `get-comments`. We will use this name in a little bit to add in some localization so that our JavaScript variables can be translated into other languages.

When viewing the HTML source, you should now see something similar to this example:


```
<script type='text/javascript' src='http://www.yourdomain.com/
wp-includes/js/jquery/jquery.js?ver=1.4.2'></script>
<script type='text/javascript'>
/*  */
var wpAjax = {
    noPerm: "You do not have permission to do that.",
    broken: "An unidentified error has occurred."
};
try{convertEntities(wpAjax);}catch(e){};
/* ]]&gt; */
&lt;/script&gt;
&lt;script type='text/javascript' src='http://www.yourdomain.com/
wp-includes/js/wp-ajax-response.js?ver=20091119'&gt;&lt;/script&gt;

&lt;script type='text/javascript'&gt;
/* <![CDATA[ */
var mythemegetcomments = {
    ajax_url: "http://www.yourdomain.com/?my_page=ajax-
processor",
    you_have: "You have",
    approved: "approved",
    comments: "comments",
    in_moderation: "in moderation",
    trashed: "trashed",
    spam: "spam"
};
/* ]]&gt; */
&lt;/script&gt;
&lt;script type='text/javascript' src='http://www.yourdomain.com/
wp-content/themes/twentyten/my_script.js?ver=2.3'&gt;&lt;/script&gt;</pre></div><div data-bbox="81 747 700 797" data-label="Text"><p>Beautiful, right? Everything's in just the right order.</p></div><div data-bbox="81 820 700 874" data-label="Text"><p>You now have access to the <code>mythemegetcomments</code> and <code>wpAjax</code> objects in your <code>my_script.js</code> file.</p></div>
```

Add in String Localization

We're almost done with the localization. We just need to add support for localization into our class.

Let's go back to our constructor and add in an `init` action and point it to a method named (what else?) `init`.

```
function __construct(){
    add_action('wp_print_scripts', array(&$this, 'add_
scripts'));
    add_action('init', array(&$this, 'init'));
}
```

We next create the `init` method and initialize the localization.

```
function init() {
    load_theme_textdomain('get-comments');
}
```

Add in Query Variable Support

Finally, we need to add in query variable support for our Ajax processor. We'll start by adding the `load_page` and `query_trigger` methods.

The `query_trigger` method adds in our query variable and the `load_page` handles the loading of the Ajax processor.

```
function load_page() {
    $pagepath = STYLESHEETPATH . '/';
    switch(get_query_var('my_page')) {
        case 'ajax-processor':
            if (file_exists($pagepath . 'ajax-processor.
php'))
                include($pagepath . 'ajax-processor.
php');
            else
                header("HTTP/1.0 404 Not Found");
                exit;
        default:
            break;
    }
} //end function load_page
function query_trigger($queries) {
    array_push($queries, 'my_page');
    return $queries;
} //end function query_trigger
```

If the value for query variable `my_page` is `ajax-processor`, we check for the existence of the file. If it exists, the file is included. If not, we return a 404 header (not really necessary, but will assist with errors later on).

Ideally, instead of returning the 404 error, you would just `break` out of the `switch` and allow for normal WordPress templating.

Please note that the constant `STYLESHEETPATH` is what gives us the absolute directory path to your active parent or child theme.

Up next is adding in the action for `load_page` and the filter for `query_trigger`. We'll be adding the action and filter right after the class is instantiated.

```
//instantiate the class
if (class_exists('myThemeClass')) {
    $myClassVar = new myThemeClass();
    add_action('template_redirect',
array(&$myClassVar, 'load_page'));
    add_filter('query_vars', array(&$myClassVar, 'query_
trigger'));
}
```

We're now done with our class. Phew! Let's move on to finalizing our first Ajax request.

Finalizing the Ajax Request

Let's move back to our `my_script.js` file.

The last thing we did with it is add in some data for our `s` object.

Let's add in a `success` and `error` function for our Ajax request.

```
function _do_ajax(obj) {
    //[...] s object initialization
    s.success = function(r) {
        alert("Success!");
    }
    s.error = function(r) {
        alert("Epic Fail!");
    }
} //end _do_ajax
```

As seen in the above code, our `success` and `error` functions are each defined to `alert` us to which one occurs.

Now it's time to add in our Ajax call, which is the last step in sending our first Ajax request.

```
function _do_ajax(obj) {
    //[...] s object initialization
    s.success = function(r) {
        alert("Success!");
    }
    s.error = function(r) {
        alert("Epic Fail!");
    }
    $.ajax(s);
} //end _do_ajax
```

Sending an Ajax request is that simple!

Now here's a look at our finalized code for the first Ajax request (we'll fill in the `success` and `error` functions later):

```
jQuery(document).ready(function() {
var $ = jQuery;
$.get_my_comments = {
    init: function() {
        $("#get-comments").bind("click", function() {
            _do_ajax(this); //Call to our private function
            return false;
        });
    }
}; //End of get_my_comments namespace
function _do_ajax(obj) {
    var element = $(obj); //our link object
    var url = wpAjax.unserialize(element.attr('href'));
    var s = {};
    s.response = 'ajax-response';
    s.type = "POST";
    s.url = mythemegetcomments.ajax_url;
    s.data = $.extend(s.data, { action: url.action, _ajax_
nonce: url._wpnonce });
    s.global = false;
    s.timeout = 30000;
    s.success = function(r) {
        alert("Success");
    } //End success
    s.error = function(r) {
        alert("Epic Fail!");
    }
    $.ajax(s);
} //end _do_ajax
$.get_my_comments.init();
});
```

So what now?

Well, head to your website and click on the **Get Comments** link.

And what are you met with? **EPIC FAILURE!**

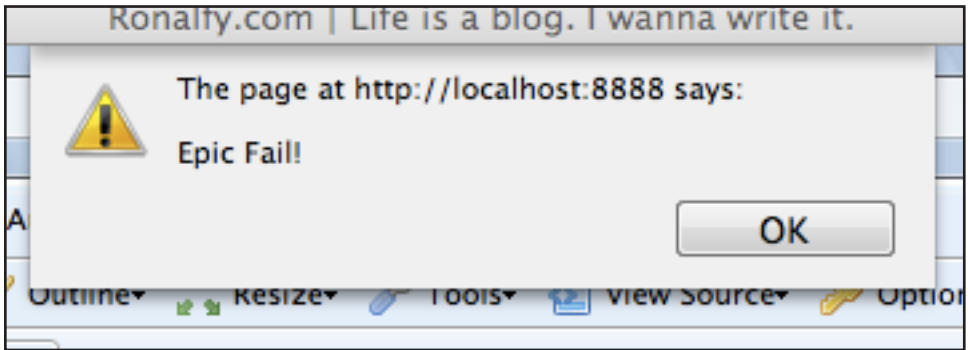


Figure 4. Epic Fail!

I can hear you now, “You mean we just did all that work just to end up in failure?”

Yep, it’s like watching one of those good movies that have a horrible ending (Matrix trilogy anyone?).

Before you grab your stick and beat down the author, be reminded that we have yet to create our `ajax-processor.php` file.

The file doesn’t exist, and I set up the `load_page` method to spit out a `404 error` so that our `s.error` function runs when clicking on the `Get Comments` link (yes, I’m sneaky).

So in a way, we were successful. We successfully sent our first request, but hit a brick wall.

So what's next? Well obviously we have to resolve our epic failure (yes, I'll take some of the blame).

Let's move on to Part 2 of this example and process the Ajax request.

Processing Our First Ajax Request

Processing Our First Ajax Request

We've successfully sent our first Ajax request. Hurrah! But we ran into a brick wall because our Ajax processor didn't exist. Let's change that.

Securing Your Ajax Processor

Let's go ahead and create our empty `ajax-processor.php` file and place it in your theme's root directory.

Once you have done that, go ahead and click on the [Get Comments](#) link.

Success! Well, sort of. Nothing is happening, but at least we're talking to our Ajax processor.

To get our Ajax processor ready for business, we need to add a few things.

```
<?php
header('Content-Type: text/html; charset=UTF-8');
define('DOING_AJAX', true);
?>
```

In the above example, there are a few things going on.

We first specify a `header` with a content type and charset. Since JavaScript sends requests using

UTF-8, it's important to set our header to the same charset.

The second thing done here is we define a constant called `DOING_AJAX` and assign it a value of `true`. The constant `DOING_AJAX` is mainly used to avoid Cron jobs, which can slow an Ajax request down.

Now we're set to move onto the next step: checking our passed nonce.

Performing a Nonce Check

As a reminder, we passed two POST variables to our Ajax processor:

- `action` - The action we are taking.
- `_ajax_nonce` - The nonce we will be verifying against.

Another reminder is that we used the string `my-theme_getcomment` to create our nonce back in the `sidebar.php` file.

Thankfully, WordPress provides an easy function to verify Ajax nonces. The function, you ask? It's called `check_ajax_referer`.

If we were to send an Ajax request without a nonce and included the function `check_ajax_referer` in

our script handler, the Ajax processor would send back a string with value “-1”.

Effectively, when `check_ajax_referer` fails, the Ajax server-side processing is stopped cold.

Adding in the `check_ajax_referer` function in our Ajax processor will look like this:

```
//Check the AJAX nonce
check_ajax_referer('my-theme_getcomment');
```

Let’s now go back to our `my_script.js` file and update the `success` function to alert us to the response that `ajax-processor.php` file sends back.

```
s.success = function(r) {
    alert(r);
}
```

Now what happens when we go back to our website and click on that `Get Comments` link?

If our nonce checking is working, you should receive a blank alert box. If the nonce failed, you should receive a “-1”.

You can test this out by changing the string in the `check_ajax_referer` function call. Click again on the `Get Comments` link and observe that the alert box now shows a “-1”.

We should be good to go as far as performing some basic security checks in our Ajax processor, so let's move on to some server-side processing fun.

Pssst... Don't forget to change the string in `check_ajax_referer` back to `my-theme_getcomment`.

Server-Side Processing of Ajax Requests

Previously, we added in a security check for our passed nonce. Now it's time to do some processing of the passed data.

We'll be interacting with our class defined in the `functions.php` file.

Now you may be wondering, "Why don't we just include the code needed in `myThemeClass` inside the Ajax processor?"

There's absolutely no problem with that, and later on in this book, I'll show you exactly how to do that, but I want to show you how you can easily interact with classes from within your Ajax processor.

Let's move on. Next up is capturing our `action` variable.

Since the `action` variable was passed as a POST variable, we check to see if it's set. If it's set, we assign it to the `$action` variable.

```
if (isset($_POST['action'])) {  
    $action = $_POST['action'];  
}  
die(''); //Place at the end of the file  
>
```

The `die` function is placed at the end of the file for one main reason: some servers add in code at the end of files (the `die` function prevents this).

Within the `$_POST` conditional, we add in a `switch` statement to determine what action is being taken.

```
if (isset($_POST['action'])) {  
    $action = $_POST['action'];  
    switch($action) {  
        case 'getcomment':  
            break;  
        case 'otheraction':  
            break;  
        default:  
            break;  
    } //end switch  
} //end if
```

A `switch` statement for only one value is extreme overkill, but it demonstrates how you can check for various actions and perform different tasks.

Now that we've added in our `switch` statement, it's time to interact with our `myThemeClass` class.

Adding a 'die' function at the end of your Ajax processor prevents your host from appending unnecessary code.

Since the Ajax processor is technically being included within the `load_page` method, the Ajax processor code is in scope of the `myPluginClass` instantiation.

As a result, we can use the `$this` variable to access the methods within `myPluginClass`.

We'll be calling `$this->get_comments()` to retrieve all of our comment counts.

```
case 'getcomment':  
    $comment_count = $this->get_comments();  
    break;
```

The above code assigns the variable `$comment_count` with the result from the `get_comments` method.

Remember, the `get_comments` method returns an object with four variables:

- approved
- moderated
- spam
- trash

We'll be dealing with these variables in the next section when we work on sending our Ajax response.

Sending an Ajax Response

A super easy way of sending back a request to our `my_script.js` file is just to `echo` out the variables.

But, we have four keys we have to deal with, and parsing that in JavaScript will be a pain.

So what's one to do?

Fortunately, WordPress has the `WP_Ajax_Response` class.

Let's go ahead and instantiate the class and assign it to the `$response` variable.

```
case 'getcomment':  
    $comment_count = $myClassVar->get_comments();  
    $response = new WP_Ajax_Response();  
    break;
```

Now it's time to add some data to our `$response` variable by calling the `add` method.

When we call the `add` method, we pass it an associative array with various parameters. Here are the parameters we'll be dealing with:

- **what** - A string that is the XMLRPC response type. In our case, we'll just call it "getcomments".
- **supplemental** - An associative array of strings. We'll be passing our `$comment_count` keys here.

When calling our **add** method, we'll have access to all of the various parameters with JavaScript, so it's best to add them in appropriately.

There is an additional parameter for the **add** method we could be using called **data**. This parameter is used when sending just a string of data.

But we have four pieces of data we need to send back, so that means if we used the **data** parameter, we'd have to create four separate responses by calling the **add** method each time.

The **supplemental** parameter allows us to return multiple values with just one response. We'll find out later just how easy it is to get these values via JavaScript.

Alright, enough talking. Let's get back to the code and call the **add** method already.

```
case 'getcomment':
    $comment_count = $this->get_comments();
    $response = new WP_Ajax_Response();
    $response->add(array(
        'what' => 'getcomments',
        'supplemental' => array(
            'awaiting_moderation' => $comment_count-
>moderated,
            'approved' => $comment_count->approved,
            'spam' => $comment_count->spam,
            'trashed' => $comment_count->trash
        )
    ));
```

As you can see from the above code, we pass to the `add` method an array with parameters `what` and `supplemental`.

The `supplemental` parameter itself takes another array, which we assign our values from `$comment_count`.

We're almost done. All that's left is to send the response.

```
$response->send();
```

It really is as simple as that.

WordPress does all the dirty work and returns a nice XML object that you can parse through rather easily using JavaScript.

Here's the full code for the Ajax processor:

```
<?php
header('Content-Type: text/html; charset=UTF-8');
define('DOING_AJAX', true);
//Check the AJAX nonce
check_ajax_referer('my-theme_getcomment');
if (isset($_POST['action'])) {
    $action = $_POST['action'];
    switch($action) {
        case 'getcomment':
            $comment_count = $this->get_comments();
            $response = new WP_Ajax_Response();
            $response->add(array(
                'what' => 'getcomments',
                'supplemental' => array(
                    'awaiting_moderation' => $comment_count-
>moderated,
                    'approved' => $comment_count->approved,
                    'spam' => $comment_count->spam,
                    'trashed' => $comment_count->trash)
                ));
            $response->send();
            break;
        case 'otheraction':
            break;
        default:
            break;
    } //end switch
} //end if
die('');
?>
```

If you click on your [Get Comments](#) link, you're alerted that you have an `[object XMLHttpRequest]`.

It's time to go back to our `my_scripts.js` file and do some client-side processing on this returned object.

Client-Side Processing/Parsing

In this section we'll be working with our `my_script.js` file to do some client-side processing/parsing.

We'll be completing the `s.success` function, which currently looks like this:

```
s.success = function(r) {  
    alert(r);  
}
```

Nothing spectacular is going on here other than a simple `alert` box that doesn't show us anything useful.

Right now as it stands, the `r` variable is an XML Document object, which we can do nothing with at the moment.

Let's change that.

Parsing the XML Document Object

Do you remember our JavaScript dependency of `wp-ajax-response`? We'll be using that same script to parse the passed XML Document.

The sub function from the `wpAjax` object that is used for parsing is called `parseAjaxResponse`, which takes two arguments:

- Our passed XML Document.
- Our response type (in our case, `ajax-response`, which was defined in the `s` object as `s.response`).

The call to `wpAjax.parseAjaxResponse` would look like this:

```
s.success = function(r) {  
    res = wpAjax.parseAjaxResponse(r, this.response);  
}
```

The keyword `this` is used instead of the object `s` since the `s` object is out of scope. However, all of the values captured in the `s` object are now available with the `this` keyword.

Our parsed response is now stored inside the `res` variable.

Getting to that data is another matter. Let's move on to processing the parsed data.

Processing the Data

We could process the data using the easy (but not scalable way) by assigning another variable the object we are after.

This would be fine, but what if you had multiple responses? In our particular case, we don't, but I'm still going to show you how to handle multiple responses for scalability purposes.

The way to handle multiple responses is by using the `jQuery.each` function.

The `$.each` function acts as an iterator through an object or array. It'll go through each of the responses and allow us to take action on each one.

Within the `$.each` function, we'll be using the `this` object to access each response.

The `this` object should contain the `what` and `supplemental` data we sent back using the `ajax-processor.php` file.

As a reminder, here's what the `supplemental` data should contain:

- `moderated`
- `approved`
- `spam`

- trash

Here's what we should expect the `this` object to contain:

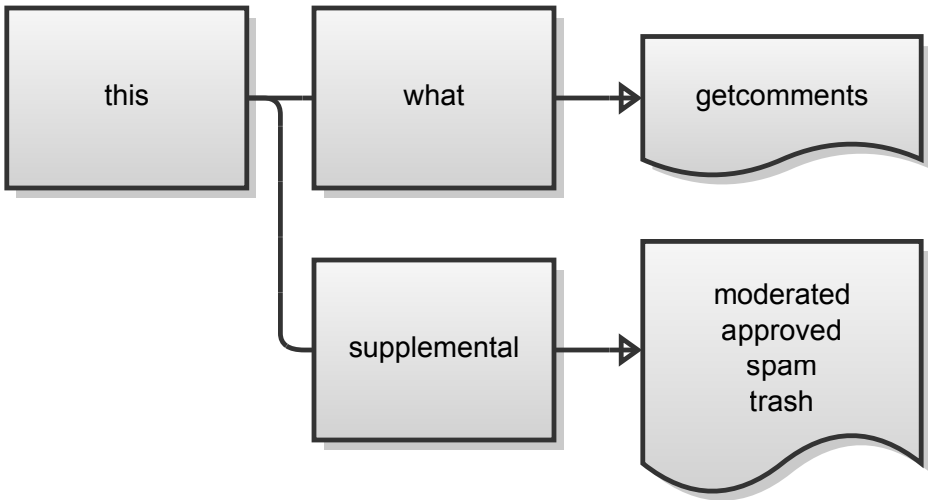


Figure 5. The "this" Object

Let's now concentrate on the `this.what` value, which should be `getcomments`.

A simple `switch` statement in the `$.each` function will aid us in picking out the `what` argument.

Again, a `switch` statement for one value is extreme overkill, but it'll show you how to handle multiple responses.


```
s.success = function(r) {
    var res = wpAjax.parseAjaxResponse(r,this.response);
    $.each( res.responses, function() {
        switch(this.what) {
            case "getcomments":
                //do stuff here
                break;
            case "something else":
                break;
            default:
                break;
        } //end switch
    }); //end each
} //End success
```

The `switch` statement checks the `what` variable for several values. In our case, we check for a case of “getcomments”.

It’s within the “getcomments” case that we’ll access our `supplemental` data.

Getting the `supplemental` data is super easy thanks to the `parseAjaxResponse` function.

Accessing the `supplemental` data is as simple as calling the key values we assigned in `ajax-processor.php`:

- `this.supplemental.moderated`
- `this.supplemental.approved`
- `this.supplemental.spam`

- `this.supplemental.trash`

Let's go ahead and assign the `supplemental` data to four JavaScript variables:

- `moderation_count`
- `approved_count`
- `spam_count`
- `trashed_count`

Our code would look like this:

```
switch(this.what) {
  case "getcomments":
    var moderation_count = this.supplemental.awaiting_
moderation;
    var approved_count = this.supplemental.approved;
    var spam_count = this.supplemental.spam;
    var trashed_count = this.supplemental.trashed;
    break;
  case "something else":
    break;
  default:
    break;
} //end switch
```

Our four variables are now assigned the values retrieved from the Ajax request.

We're done with the client-side processing/parsing.

Let's move on to the output.

The Output

This is the grand finale of sending our Ajax request.

Here's a brief summary of what we've accomplished so far:

- We sent our first Ajax request and ended up in EPIC FAILURE (not my fault, I swear).
- We secured our Ajax request with the use of nonces.
- We processed the Ajax request and prepared for a response.
- We sent the response back to our JavaScript file.
- We parsed the response and processed it for later use.

So what's left to do? Output the data, of course.

As a reminder, we have four JavaScript variables we'll be using for the output:

- `moderation_count`
- `approved_count`
- `spam_count`
- `trashed_count`

And do you remember the localized JavaScript variables we defined way back when? Ah, you don't? Well, rather than force you to thumb your way back 20 or so pages, here they are again (I know, you're welcome).

```
<script type='text/javascript'>
/*  */
var mythemegetcomments = {
  ajax_url: "http://www.yourdomain.com/?my_page=ajax-processor",
  you_have: "You have",
  approved: "approved",
  comments: "comments",
  in_moderation: "in moderation",
  trashed: "trashed",
  spam: "spam"
};
/* ]]&gt; */
&lt;/script&gt;</pre>
</div>
<div data-bbox="295 609 501 635" data-label="Text">
<p>So let's get to it.</p>
</div>
<div data-bbox="295 656 766 681" data-label="Text">
<p>Let's first create our <code>moderation</code> string:</p>
</div>
<div data-bbox="93 697 862 771" data-label="Text">
<pre>//Strings
var moderation = mythemegetcomments.you_have + " " +
moderation_count + " " + mythemegetcomments.comments + " " +
mythemegetcomments.in_moderation + ".&lt;br /&gt;";</pre>
</div>
<div data-bbox="295 794 926 875" data-label="Text">
<p>It's slightly involved here, but don't be intimidated. All we're doing here is building our string with a combination of the strings stored in the</p>
</div>
```

`mythemegetcomments` object and the JavaScript variables we previously defined.

The `moderation` string should contain something similar to:

```
"You have x comments in moderation."
```

Since we have that down, let's go ahead and create the rest of the strings.

```
//Strings
var moderation = mythemegetcomments.you_have + " " +
moderation_count + " " + mythemegetcomments.comments + " " +
mythemegetcomments.in_moderation + "<br />";

var approved = mythemegetcomments.you_have + " " +
approved_count + " " + mythemegetcomments.approved + " " +
mythemegetcomments.comments + "<br />";

var spam = mythemegetcomments.you_have + " " + spam_count + " "
+ mythemegetcomments.spam + " " + mythemegetcomments.comments +
"<br />";

var trashed = mythemegetcomments.you_have + " " + trashed_count
+ " " + mythemegetcomments.trashed +
```

Another small problem: what if we have a lot of `spam` and `approved` comments? Would you like to have an output such as, “You have a total of 1023930 approved comments”?

Um, no, right? Since adding in the comma separation for groups of thousands is tedious using JavaScript, let's get PHP to do it (ah, the convenience of Ajax).

Let's venture back to our `ajax-processor.php` file and make use of the `number_format` function.

The `number_format` function will be wrapped around our `$comment_count` keys to transform a count such as 4209094 into 4,209,094.

```
$response->add(array(
    'what' => 'getcomments',
    'supplemental' => array(
        'awaiting_moderation' => number_format($comment_
count->moderated),
        'approved' => number_format($comment_count->
approved),
        'spam' => number_format($comment_count->spam),
        'trashed' => number_format($comment_count->trash)
    )
));
$response->send();
```

We now have our strings finalized.

There's two ways to go about an output:

- Use `alert` boxes.
- Output to the theme.

Alert boxes are messy and should be used rarely, if ever. I personally only use them for debugging purposes.

Outputting to the theme is the correct way to go. But first we must find a box (err, `div`) on the theme to output to.

Do you remember how in our `sidebar.php` file we added an empty `div` with an `id` of “get-comments-output”?

This placeholder `div` is what we’ll use for the output. Please also note that the `br` tags were added into the JavaScript variables since we are going to be outputting to HTML and want to include some line breaks.

Groovy. Now we’re ready to output. And it’s going to be a lot easier than you think.

```
//JavaScript Output
$("#div#get-comments-output").html(moderation + approved + spam
+ trashed);
```

Ah, I love it when things are simple, and that above code snippet is as easy as it gets.

We simply capture the `div` with the `id` of “get-comments-output”. We then alter its inner-HTML and fill it with our strings.

Now when you click on the `Get Comments` link, you should see something similar to this:

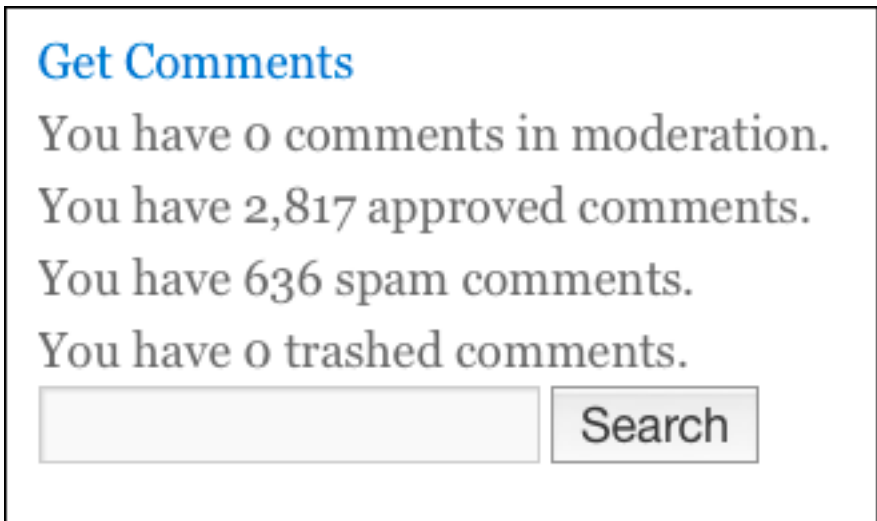


Figure 6. The Output

I know, all that work to get just four lines of text to output to a theme, right? Hey, I never said Ajax wasn't tedious at times, but this example hopefully demonstrated to you how many moving parts Ajax has.

I'll leave you with the final `s.success` function code for your viewing pleasure.


```

s.success = function(r) {
    var res = wpAjax.parseAjaxResponse(r,this.response);
    $.each( res.responses, function() {
        switch(this.what) {
            case "getcomments":
                var moderation_count = this.
supplemental.awaiting_moderation;
                var approved_count = this.
supplemental.approved;
                var spam_count = this.supplemental.
spam;
                var trashed_count = this.
supplemental.trashed;

                //Strings
                var moderation = mythemegetcomments.
you_have + " " + moderation_count + " " + mythemegetcomments.
comments + " " + mythemegetcomments.in_moderation + ".<br />";

                var approved = mythemegetcomments.
you_have + " " + approved_count + " " + mythemegetcomments.
approved + " " + mythemegetcomments.comments + ".<br />";

                var spam = mythemegetcomments.you_
have + " " + spam_count + " " + mythemegetcomments.spam + " " +
mythemegetcomments.comments + ".<br />";

                var trashed = mythemegetcomments.you_
have + " " + trashed_count + " " + mythemegetcomments.trashed +
" " + mythemegetcomments.comments + ".";
                $("div#get-comments-output").
html(moderation + approved + spam + trashed);
                break;
            case "something else":
                break;
            default:
                break;
        } //end switch
    }); //end each
} //End success

```

We're done! The output is finalized. You may drink a cerveza now.

But after you're done drinking, let's move on to how to use WordPress' built-in Ajax processor. We'll be modifying this example slightly, so brace yourself and let's proceed.

WordPress and Admin Ajax

WordPress and Admin Ajax

I recently led you on a journey of sending and processing an Ajax request to retrieve comment information from WordPress. In the example, I showed you how to manually create an Ajax processor.

WordPress, however, has its own Ajax processor built-in, and it's incredibly simple to use. While knowing how to manually create an Ajax processor is still useful, I'm going to show you how to use WordPress' built-in Ajax processor.

We'll be modifying our [Get Comments](#) example a bit here so we don't have to start from scratch.

WordPress' admin-ajax.php

If you venture into the “wp-admin” folder of WordPress, one of the files you'll see is [admin-ajax.php](#). This file is a WordPress Ajax processor on steroids. It has all the major Ajax actions WordPress needs, and allows you to define your own as well.

One main misconception to admin-ajax is that it is used for admin Ajax requests. Get the term “admin” out of your head as admin-ajax can be used on both the front-end and admin area of your site.

The admin-ajax is completely action based. Basically, the Ajax request contains an action. The `admin-ajax.php` file acts as a director and matches up an action with a callback function.

Once the callback function is called, the function is responsible for returning any data (in string format, XML, or JSON). So think of admin-ajax as an intermediary between the Ajax request and Ajax response.

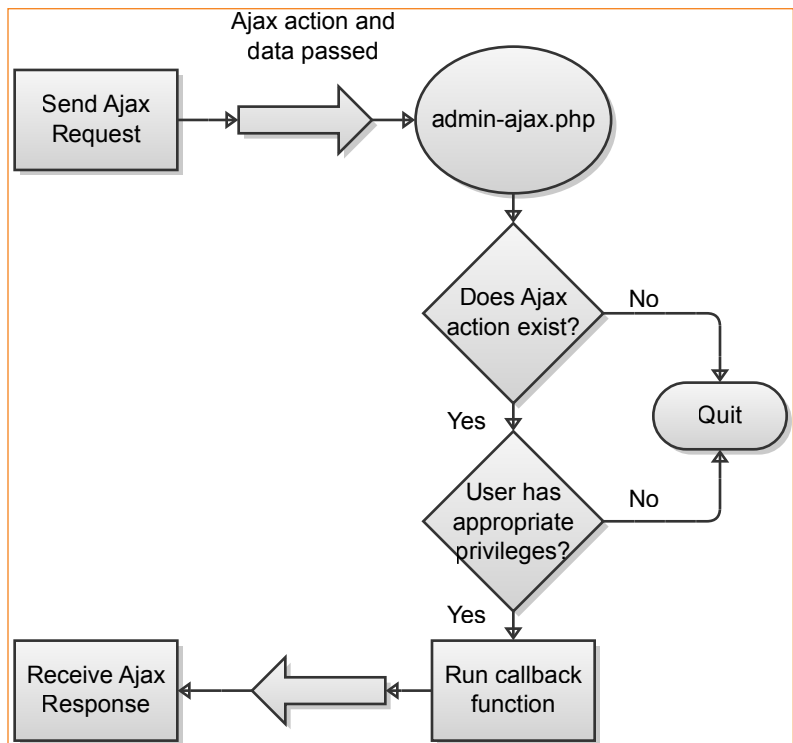


Figure 7. WordPress admin-ajax.php process

You may remember from the `Get Comments` example that the Ajax action we used was called `getcomment`. This action name will come in handy when we register our Ajax processor function.

In the `Get Comments` example, we created a separate file that served as our Ajax processor. With `admin-ajax`, however, it'll be a separate method within our class.

Let's go ahead and register the method that will serve as the new Ajax processor.

Registering the Ajax Processor

Registering the Ajax processor function is as simple as adding a WordPress action.

To tell WordPress about an Ajax processor, you must use the `wp_ajax` action.

The `wp_ajax` action is structured like this:

```
<?php
add_action('wp_ajax_getcomment', 'callback_function');
?>
```

You'll notice in the `wp_ajax` action that there is a suffix of `getcomment`. This is our Ajax action name.

When the action name gets passed to `admin-ajax.php`, WordPress will know to call the function named `callback_function`.

However, the above snippet will only run for privileged (logged-in) users. What if you want to run code for all users? You would add the `nopriv` suffix as well. Here's an example:

```
<?php
add_action('wp_ajax_getcomment', 'callback_function');
add_action('wp_ajax_nopriv_getcomment', 'callback_function');
?>
```

Now both privileged and unprivileged users can access the Ajax processor (indirectly, of course).

Let's go ahead and modify the PHP constructor in our `Get Comments` example to point to a method named `ajax_get_comments`.

```
<?php
function __construct(){
    add_action('wp_print_scripts', array(&$this, 'add_
scripts'));
    add_action('init', array(&$this, 'init'));
    add_action('wp_ajax_getcomment', array(&$this, 'ajax_
get_comments'));
    add_action('wp_ajax_nopriv_getcomment', array(&$this,
'ajax_get_comments'));
}
?>
```


Since we want all users to be able to access the `getcomment` action, the second action makes use of the `nopriv` suffix.

Getting the Location of the Ajax Processor

In the `Get Comments` example, we localized a JavaScript variable to give us the URL to our Ajax processor. We also used query variables to load the WordPress environment manually.

With `admin-ajax`, we don't have to worry about query variables, but we do have to worry about the location of `admin-ajax.php`.

In the admin area, there is a JavaScript variable called `ajaxurl`. This variable contains the URL to `admin-ajax.php`.

This variable is handy when in the admin area only, however. If you're on the front-end, you're out of luck.

Fortunately, all we have to do is update one area in our code to point to the new location, which is in the `get_js_vars` method (the method we use to return the localized JavaScript variables).

Let's modify our localized variable `ajax_url` to point to the new location of the Ajax processor.

```
function get_js_vars() {
    return array(
        'ajax_url' => admin_url('admin-ajax.php'),
        'you_have' => __('You have', 'get-comments'),
        'approved' => __('approved', 'get-comments'),
        'comments' => __('comments', 'get-comments'),
        'in_moderation' => __('in moderation', 'get-
comments'),
        'trashed' => __('trashed', 'get-comments'),
        'spam' => __('spam', 'get-comments')

    );
} //end get_js_vars
```

We make use of the function `admin_url` and pass it the filename `admin-ajax.php`. The resulting URL would be like:

```
http://www.yourdomain.com/wp-admin/admin-ajax.php
```

Since we make use of the same JavaScript variable in our JavaScript file, there's no need to update our script (hooray for small victories!).

Passing Data to the Ajax Processor

Passing data to the Ajax processor is the same route we took in the `Get Comments` example.

Let's look at our `sidebar.php` code again and modify the link location to point to the new Ajax processor location.

```
<?php
$link_url = esc_url(wp_nonce_url( admin_url('admin-ajax.
php?action=getcomment'), "my-theme_getcomment"));
?>
<a href='<?php echo $link_url; ?>' id='get-comments'><?php
_e('Get Comments', 'get-comments'); ?></a>
```

Based on the above code, we created a link with a URL pointing to `admin-ajax.php`. The URL contains two query variables (an action and a nonce) that can be captured via JavaScript.

As a reminder, here's where we capture and pass the variables via JavaScript in our `my_script.js` file.

```
s.data = $.extend(s.data, { action: url.action, _ajax_nonce:
url._wponce });
```

Via JavaScript, we create a data variable that contains the action name and a nonce. Whenever the action is detected via the Ajax processor, our action method is called.

Let's go ahead and create this method.

The `wp_ajax` Callback Method

When we used the `wp_ajax` WordPress action, we provided a callback method called `ajax_get_comments`.

Let's go ahead and create this method in our class. We'll use the `check_ajax_referer` function to do a nonce check.

As a reminder, the name we used when creating the nonce was `my-theme_getcomment`.

```
function ajax_get_comments() {  
    check_ajax_referer('my-theme_getcomment');  
    exit;  
}
```

The biggest difference between both Ajax processor techniques is the use of the `exit` keyword. The `exit` keyword is often used for template redirects in order to terminate a script (stop it from proceeding any further).

An `exit` at the end of the Ajax processor prevents any more items from being outputted erroneously.

Finalizing the Ajax Processor

Now that our nonce check is complete, let's go ahead and flesh out the rest of the Ajax processor.

```
function ajax_get_comments() {
    check_ajax_referer('my-theme_getcomment');
    $comment_count = $this->get_comments();
    $response = new WP_Ajax_Response();
    $response->add(array(
        'what' => 'getcomments',
        'supplemental' => array(
            'awaiting_moderation' => number_
format($comment_count->moderated),
            'approved' => number_format($comment_count-
>approved),
            'spam' => number_format($comment_count-
>spam),
            'trashed' => number_format($comment_count-
>trash)
        )
    ));
    $response->send();
    exit;
} //end ajax_get_comments
```

If you compare the code of this Ajax processor to that of the `Get Comments` method, you'll find that they are both very similar.

The biggest differences between this Ajax processor and the old one are:

- I got rid of the switch statement and conditionals. The reason? The `getcomment` action is now associated with the `ajax_get_comments` method. It's a valid assumption that if the method is being called, the appropriate action is being taken.
- There is no need to set the content-type or any constants. The `admin-ajax.php` file does all this for you.

Admin Ajax Conclusion

WordPress' `admin-ajax.php` is a powerful and easy-to-use technique for easily adding in Ajax processors.

Everything is action based using the WordPress action `wp_ajax`.

Each Ajax action can now have its own associated method (or function), and you can assign privileges so that the Ajax processor is only run when needed.

We'll be making more use of `admin-ajax` in the examples, so read on if you'd like to see more implementation techniques.

Example 1: WP Grins Lite

Example 1: WP Grins Lite

The WordPress plugin WP Grins is very simple in what it accomplishes: it provides clickable smilies that one can insert into a post or comment.



Figure 8. WP Grins Screenshot - Comment Area

WP Grins is a great way to spice up a rather dull looking comments form. The biggest drawback WP Grins has for me, however, is that it uses the heavy-weight Prototype JavaScript framework.

None of my other installed plugins used Prototype, and neither did my theme. Prototype (the version included with WordPress) weighs in at 143 KBs per page load. For just adding simple “grins” to a page, this was unacceptable overhead.

As I explored into the code, I found it rather trivial to port it over to the much lighter-weight jQuery library. Thus, the name was born for the new plugin: WP Grins Lite.

As I explored more into the code, I found several more issues (no disrespect to Alex King here):

- The JavaScript was embedded within the PHP code. Not a huge issue, but it's a lot cleaner when separated. In fact, my first version of WP Grins Lite still used the embedded JavaScript.
- There was no class structure to prevent function name conflicts. Granted, the function names were prefixed with "wp_grins", so no big deal here.
- The smilies showed up in weird places. Limited page detection was used, but more was needed.
- No option to display smilies manually in the post comments section. This wasn't a big deal to me, but it was to one of the users. Adding this option in would be a moderate challenge.

Adding an option to manually insert the smilies wasn't exactly trivial. After some brainstorming, I determined what was needed:

- A separation of PHP and JavaScript. With the original WP Grins, the two were interspersed, and there was no way to call the grins manually without them also showing up via JavaScript.
- An admin panel for the manual option. I could have gone with having the user manually fill in a config option in the raw file, but a good plugin author never makes his users delve into the code. It's like trying to pick up single women at a marriage seminar. Not a good idea.
- Have a dedicated function for printing out the grins. This would aid manual showing and JavaScript use.
- For JavaScript use, have the smilies returned via Ajax. I thought about this one for a while, and finally determined Ajax was best used for this one. I could have printed the smilies as a localized variable, but the inner-programmer in me started screaming at this possibility.

So now you have my justifications for the changes I made to the plugin.

In order to begin this journey, let's take a look at WP Grin Lite's file structure.

Forcing users to manually edit code is like trying to pick up single women at a marriage seminar. Not a good idea.

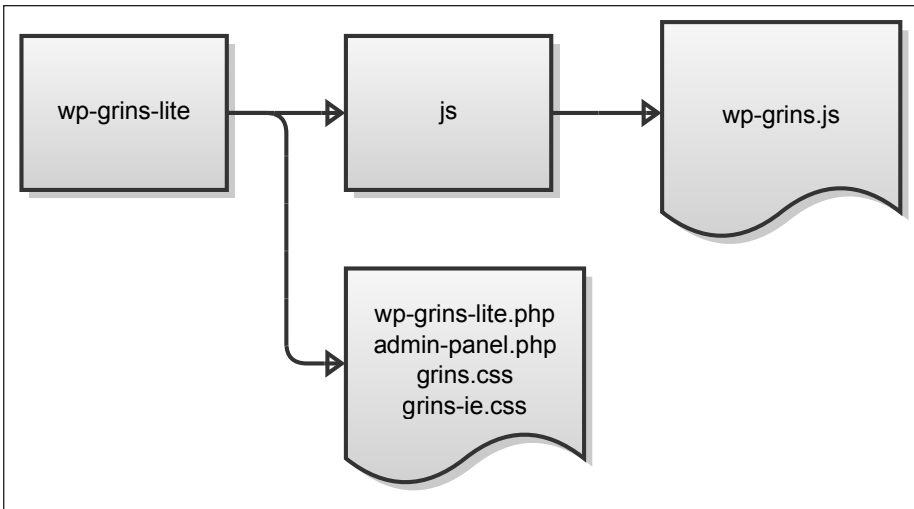


Figure 9. WP Grins Lite File Structure

Let's begin with the file `wp-grins.php`.

The WPGrins Class

The importance of using classes as part of your plugin structure should not be understated.

First, it allows you to enclose all of your functions (as methods) within the class, and avoid naming conflicts. If you've ever ventured into plugin code that isn't using a class, chances are the functions are prefixed.

While prefixed functions help, a class structure allows you to have generic and more intuitively-named function names.

Let's look at the beginning's of our class structure.

```
<?php
if (!class_exists('WPGrins')) {
    class WPGrins{
        var $adminOptionsName = "wpgrinslite";
        /**
         * PHP 4 Compatible Constructor
         */
        function WPGrins(){$this->__construct();}

        /**
         * PHP 5 Constructor
         */
        function __construct(){

        }
    }
}
//instantiate the class
if (class_exists('WPGrins')) {
    $GrinsLite = new WPGrins();
}
?>
```

Shown above is our basic skeleton class called *WPGrins*.

There's nothing in it as of this point, so let's add in some methods so we can get some smiley goodness.

The Constructor

First up is the constructor.

```
function __construct(){
    //Scripts
    add_action('admin_print_scripts-post.php',
array(&$this, 'add_scripts'),1000);
    add_action('admin_print_scripts-post-new.php',
array(&$this, 'add_scripts'),1000);
    add_action('admin_print_scripts-page.php',
array(&$this, 'add_scripts'),1000);
    add_action('admin_print_scripts-page-new.php',
array(&$this, 'add_scripts'),1000);
    add_action('admin_print_scripts-comment.php',
array(&$this, 'add_scripts'),1000);
    add_action('wp_print_scripts', array(&$this, 'add_scripts_
frontend'),1000);
    //Styles
    add_action('admin_print_styles-post.php',
array(&$this, 'add_styles'),1000);
    add_action('admin_print_styles-post-new.php',
array(&$this, 'add_styles'),1000);
    add_action('admin_print_styles-page.php',
array(&$this, 'add_styles'),1000);
    add_action('admin_print_styles-page-new.php',
array(&$this, 'add_styles'),1000);
    add_action('admin_print_styles-comment.php',
array(&$this, 'add_styles'),1000);
    add_action('wp_print_styles', array(&$this, 'add_styles_
frontend'));

    //Ajax
    add_action('wp_ajax_grins', array(&$this, 'ajax_print_
grins'));
    add_action('wp_ajax_nopriv_grins', array(&$this, 'ajax_
print_grins'));

    //Admin options
    add_action('admin-menu', array(&$this, 'add_admin_pages'));
    $this->adminOptions = $this->get_admin_options();
}
```

Here are the actions we added:

- `admin_print_scripts` and `wp_print_scripts`. These are needed to add in the JavaScript files.
- `wp_print_styles` and `admin_print_styles`. These are needed to load the plugin's CSS.
- `admin_menu`. This is needed to add in our admin panel.
- `wp_ajax`. This is needed to point to our Ajax-processor method.

The action `admin_print_scripts` has a callback method of `add_scripts`, whereas `wp_print_scripts` has a callback method called `add_scripts_frontend`. The action `wp_print_styles` has a callback method of `add_scripts_frontend`, while `admin_print_styles` has a callback method of `add_styles`. The menu action, `admin_menu`, has a callback method of `add_admin_pages`. Finally, the `wp_ajax` action has a callback method of `ajax_print_grins`.

For the `admin_print_scripts` and `admin_print_styles` actions, we use several suffixes to assist us in page detection.

We know we don't want the scripts to be loaded where WP Grins isn't needed. But where exactly do we want WP Grins to load?

Let's do a little brainstorming. We obviously want WP Grins to load when a comment form is present (which are on single posts and pages on the front-end). There's also a comment form on the back-end, so we want that too.

For posts, we want the scripts to load when someone is creating or editing a post. Likewise for pages.

Now let's go over our dependencies:

- Load only front-end pages where there is a comment form (we'll need to perform some page detection in the `add_scripts` and `add_styles` methods).
- Only load when a person is editing or creating a post (use `post.php` and `post-new.php` as a suffix to our action).
- Only load when a person is editing or creating a new page (use `page.php` and `page-new.php` as a suffix to our action).

- Load when someone is editing a comment in the admin area (use `comment.php` as a suffix to our action)

For the `wp_ajax` action, we use “grins” as the Ajax action name. And since we want all users to see the grins, we had a `nopriv` suffix to the `wp_ajax` action.

The last thing going on in the code is that we assign a class variable called `adminOptions` with the result of the `get_admin_options` method.

The `adminOptions` variable will contain an array of our various admin-panel options (which we’ll define later). The `adminOptions` variable can be referenced using the `$this` predefined PHP variable.

Just looking at the constructor alone, we know we have to define the following methods:

- `add_scripts`
- `add_scripts_frontend`
- `add_styles`
- `add_styles_frontend`
- `add_admin_pages`
- `ajax_print_grins`
- `get_admin_options`

add_scripts and add_scripts_frontend

The `add_scripts` method is performing two duties: it's loading JavaScript for both the front-end and admin area of the WordPress website.

We have one JavaScript dependency, which is jQuery.

Let's move on to the code for `add_scripts` and `add_scripts_frontend`.

```
function add_scripts(){
    wp_enqueue_script('wp_grins_lite', plugins_url('wp-
grins-lite/js/wp-grins.js'), array("jquery"), 1.0);
    wp_localize_script( 'wp_grins_lite', 'wpgrinslite',
$this->get_js_vars());
}
function add_scripts_frontend() {
    //Make sure the scripts are included only on the front-
end
    if (!is_admin()) {
        if ((!is_single() && !is_page()) || 'closed' ==
$post->comment_status) {
            return;
        }
        $this->add_scripts();
    }
}
```

The `add_scripts` method queues the scripts for inclusion, and is called by the `admin_print_scripts` action (with our various page detection suffixes).

The `add_scripts_frontend` method is called by `wp_print_scripts`. The first conditional checks to see if we're on the front-end of a site. If we are, we check if we're on a post or a page. If we happen to be on a post or page, but comments are turned off (no comment form), we get out of the method. If all is good, we call the `add_scripts` method to queue the scripts.

We then queue our scripts and assign some localization via the `get_js_vars` method. So let's move on to that method next.

get_js_vars

```
function get_js_vars() {
    if (is_admin()) {
        return array(
            'Ajax_Url' => admin_url('admin-ajax.php'),
            'LOCATION' => 'admin',
            'MANUAL' => 'false'
        );
    }
    return array(
        'Ajax_Url' => admin_url('admin-ajax.php'),
        'LOCATION' => 'post',
        'MANUAL' => esc_js($this->
adminOptions['manualinsert'])
    );
} //end get_js_vars
```

The `get_js_vars` method returns all of the localized text needed for JavaScript.

What we want is to determine, in JavaScript, whether we're in the admin panel or on a post (determined by the `is_admin` conditional).

A second thing we need is the path to WordPress' Ajax processor.

The third thing we need is to determine if the smilies on a post are going to be inserted manually, or via JavaScript (this is determined by our admin options).

One thing to point out is the reference to the class variable `adminOptions` (using `$this->adminOptions`). It's calling a key called `manualinsert`. We will define this later when we retrieve our admin options. You've probably also noticed that we used the `esc_js` method to sanitize our output. Since we're retrieving information from the database for use in a JavaScript variable, `esc_js` is the logical choice for data validation.

Alright, we got the scripts down. Let's now move on to the styles using the `add_styles` method.

add_styles and add_styles_frontend

The `add_styles` method will be used to insert the CSS for the smilies.

The `add_styles` queues the styles for inclusion is called by the `admin_print_styles` action.

Within `add_styles`, we queue two styles, and add a conditional comment for the handler `wp-grins-ie`.

```
function add_styles() {
    wp_enqueue_style('wp-grins', plugins_url('wp-grins-lite/
grins.css'));
    wp_enqueue_style('wp-grins-ie', plugins_url('wp-grins-
lite/grins-ie.css'));
    global $wp_styles;
    $wp_styles->add_data( 'wp-grins-ie', 'conditional', 'IE'
);
}
function add_styles_frontend() {
    if (!is_admin()) {
        if ((!is_single() && !is_page()) || 'closed' ==
$post->comment_status) {
            return;
        }
        $this->add_styles();
    }
}
```

For `add_styles_frontend`, we use the same exact conditional check as in the `add_scripts_frontend` method. If all is well, we call the `add_styles` method to queue up the styles.

The styles needed for this plugin simply ensure that the cursor is in the shape of a hand when hovering over the smilies.

For `grins.css`, the styles needed in the file are:

```
#wp_grins img {  
    cursor: pointer;  
}
```

For `grins-ie.css`, the styles needed are:

```
#wp_grins img {  
    cursor: hand;  
}
```

Now separate files for six lines of CSS is a bit much, but you must remember that queueing the styles allows others to disable or override your styles.

Hard-coding in the styles in the PHP doesn't allow for this flexibility.

Let's go ahead and work on our `add_admin_pages` method.

add_admin_pages

The `add_admin_pages` is the callback method for the `admin_menu` action.

The code to add a menu is quite simple.

```
function add_admin_pages(){
    add_options_page('WP Grins Lite', 'WP Grins Lite', 9,
    basename(__FILE__), array(&$this, 'print_admin_page'));
}
```

We pass to the `add_options_page` function the page title, menu title (shown in the Settings section in the admin panel), the access level (in our case, admin only), the current file, and a callback method that will show the admin page.

The callback method printing out the admin page is `print_admin_page`. Let's move on to that method next.

print_admin_page

Our `print_admin_page` method will perform one simple task: reference an external PHP file that holds our admin panel.

I like to keep my admin panel scripts separate just because it's a little bit cleaner. I also like to edit my admin panel scripts without having to dig into the main plugin's PHP file. It's just my preference, and you are welcome to include the admin panel code directly in your class if desired.

Here's our code:

```
//Provides the interface for the admin pages
function print_admin_page() {
    include dirname(__FILE__) . '/admin-panel.php';
}
```

We reference the external script `admin-panel.php`, which we'll get to a bit later.

Let's work on our Ajax processor method, which is called `ajax_print_grins`.

ajax_print_grins

The `ajax_print_grins` method is our callback method for the `wp_ajax` WordPress action.

Since no security is needed and we're only returning one data set, we will simply echo out the result.

```
function ajax_print_grins() {
    echo $this->wp_grins();
    exit;
}
```

Since we echo out the result of the `wp_grins` method, let's move on to that one next.

wp_grins

We have a few methods to go before we're done with the `WPGrins` class. For now, we need to work on outputting those beloved smilies.

There's a WordPress PHP `global` variable called `wpsmiliestrans`, which is an array of all of the smilies available.

What's needed is a `foreach` loop that will build one long string. Once the string is built, we return it.

The `$tag` variable holds the smiley code that will be used in a post (e.g., `:shock:`). The `$grin` variable holds the filename (e.g., `icon_redface.gif`) to the smiley.

```
function wp_grins() {
    global $wpsmiliestrans;
    $grins = '';
    $smiled = array();
    foreach ($wpsmiliestrans as $tag => $grin) {
        if (!in_array($grin, $smiled)) {
            $smiled[] = $grin;
            $tag = esc_attr(str_replace(' ', '',
$tag));
            $src = esc_url(site_url("wp-includes/
images/smilies/{$grin}"));
            $grins .= "<img src='$src' alt='$tag'
onclick='jQuery.wpgrins.grin(\"$tag\")';' />";
        }
    }
    return $grins;
} //end function wp_grins
```

Did you notice that within the `img` tag there is an `onclick` reference? Hmmmm, I wonder what that `jQuery.wpgrins.grin()` thing is?

I'm just guessing, but I think it's a reference to a namespace called `wpgrins`, and a public function within that namespace called `grin`. Perhaps a hint of things to come?

Now let's work on the `get_admin_options` method.

get_admin_options

The `get_admin_options` method holds our default admin panel variables.

I've devised a clever function that checks for existence of new keys, and keys that have been deleted.

```
function get_admin_options() {
    if (empty($this->adminOptions)) {
        $adminOptions = array(
            'manualinsert' => 'false'
        );
        $options = get_option($this->adminOptionsName);
        if (!empty($options)) {
            foreach ($options as $key => $option) {
                if (array_key_exists($key,
                    $adminOptions)) {
                    $adminOptions[$key] = $option;
                }
            }
        }
        $this->adminOptions = $adminOptions;
        $this->save_admin_options();
    }
    return $this->adminOptions;
}
```

For our particular plugin, we only need to know the value of one variable, which is if the user has decided to manually insert the smilies on a post or a page.

The key, `manualinsert`, is assigned a value of false by default.

The admin options snippet might seem overly complicated for just one key, but it's very scalable. I have upwards of fifty keys for some of my plugins, and it works just fine, even when I add or delete keys for updated versions.

It also needs to be called only once, so it saves on database calls.

The final part of the code calls the method `save_admin_options`. Let's move on to that method.

save_admin_options

The `save_admin_options` method does one task: it saves the admin options for later use (obvious, right?).

```
function save_admin_options(){
    if (!empty($this->adminOptions)) {
        update_option($this->adminOptionsName, $this->adminOptions);
    }
}
```

The code should just about explain itself.

We call the WordPress function `update_option`, and pass it our option name (defined as a class variable with value `wpgrinslite`) and our admin options.

We're done with our class, but since we're allowing manual inclusion of the smilies, we need a template tag that a user could call.

Our Template Tag

Let's go ahead and call our template tag `wp_print_grins` (yes, I'm unoriginal, and if you listen to my dad, quite lazy).

We're going to use this template tag to access our class and return the grins to the end user (the template tag would be placed towards the end of the `wp-grins.php` file).

```
if (!function_exists('wp_print_grins')) {  
    function wp_print_grins() {  
        global $GrinsLite;  
        if (isset($GrinsLite)) {  
            return $GrinsLite->wp_grins();  
        }  
    }  
}
```

The above snippet is rather straight forward. We reference the class reference variable `GrinsLite` by using the PHP reserved word `global`.

We then check to see if the variable is set. If it is, we call the class method `wp_grins` and return the result to the user.

The user, when calling the template tag, would use:

```
<?php
if (function_exists('wp_print_grins')) {
    echo wp_print_grins();
}
?>
```

Congratulations! We're done with `wp-grins.php`. Let's move on to the admin panel.

The Admin Panel (`admin-panel.php`)

The first step to creating an external admin panel is to do a basic security check so that malicious users can't access it directly.

We check for the existence of the class variable `adminOptionsName`. Accessing the file directly will cause the file to crash and burn.

```
<?php
if (empty($this->adminOptionsName)) { die(''); }

$options = $this->get_admin_options(); //global settings
```

Next, we check to see if the user is admin. This is perhaps unnecessary, but you can't ever have too much security (airlines, anyone?).

```
//Check to see if a user can access the panel
if ( !current_user_can('administrator') )
    die('');
```

You can leave a nastier message if you like, but I chose something benign.

Next up is the code that will execute when a user clicks the “Update Settings” button.

We'll have to do a little foresight here and determine what should be updated.

First we'll have to do a nonce check, so we know in advance we'll have to include a nonce in our form field.

Second, we'll have to update the options and show a success message.

Alright, we know what we want, so let's get to the code.

```

<?php
//Update settings
if (isset($_POST['update'])) {
    check_admin_referer('wp-grins-lite_options');
    $options['manualinsert'] = $_POST['manual'];
    $this->adminOptions = $options;
    $this->save_admin_options();
    ?>
    <div class="updated"><p><strong><?php _e('Settings
successfully updated.', $this->localizationName) ?></strong></
p></div>
    <?php
}
?>

```

We first check for the existence of a POST variable called `update` (which will be the name of our submit button). Up second is our nonce check, which has a name of `wp-grins-lite_options`.

We then declare an associative array with a key called `manualinsert` and assign it the POST variable `manual`.

The last thing we do is display a success message to the user.

Now it's time to work on our user interface. Simple radio box options for the manual insert will suffice.

First, let's add our nonce field to the form.

```
<div class="wrap">
    <h2>WP Grins Lite Options</h2>
    <form method="post" action="<?php echo $_
SERVER["REQUEST_URI"]; ?>">
        <?php wp_nonce_field('wp-grins-lite_options') ?>
```

Keep in mind we created the nonce field with a name of `wp-grins-lite_options`, which is the same name we used above in our nonce check.

The `action` attribute in our form is used to self-post to itself, so we expect that clicking on “Update Settings” will return us to the same page (the reason we have the update code on the same page).

Let’s get to work on our interface:

```
<table class="form-table">
    <tbody>
        <tr valign="top">
            <th scope="row"><?php _e('Manually insert grins? You will
have to call wp_print_grins()', $this->localizationName) ?></
th>
            <td><p><label for="manual_yes"><input
type="radio" id="manual_yes" name="manual" value="true"
<?php if ($options['manualinsert'] == "true") {
echo('checked="checked"'); } ?> /> <?php _e('Yes', $this-
>localizationName); ?></label>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<
label for="manual_no"><input type="radio" id="manual_no"
name="manual" value="false" <?php if ($options['manualinsert']
== "false") { echo('checked="checked"'); } ?>/> <?php
_e('No', $this->localizationName); ?></label></p></td>
        </tr>
    </tbody>
</table>
```


Nothing spectacular is going on here. We output in HTML two radio boxes with the same `name` attribute. One has a value of `true`, and the other `false`. When the user clicks on “Update Settings”, we can capture that value. We also do a conditional check using `$options['manualinsert']` to check the correct radio box by default.

All that is left to do now is create our “Update Settings” button.

```
<div class="submit">
  <input type="submit" name="update" value="<?php
_e('Update Settings', $this->localizationName) ?>" />
</div>
</form>
</div><!--/wrap-->
```

That’s it! We’re done with our admin panel. Now it’s time to test it.

Select “Yes” or “No” and click “Update Settings.”

If everything worked fine, you should see a message that says, “Settings successfully updated.”

If not, it’s time to get out your stick and beat the author.

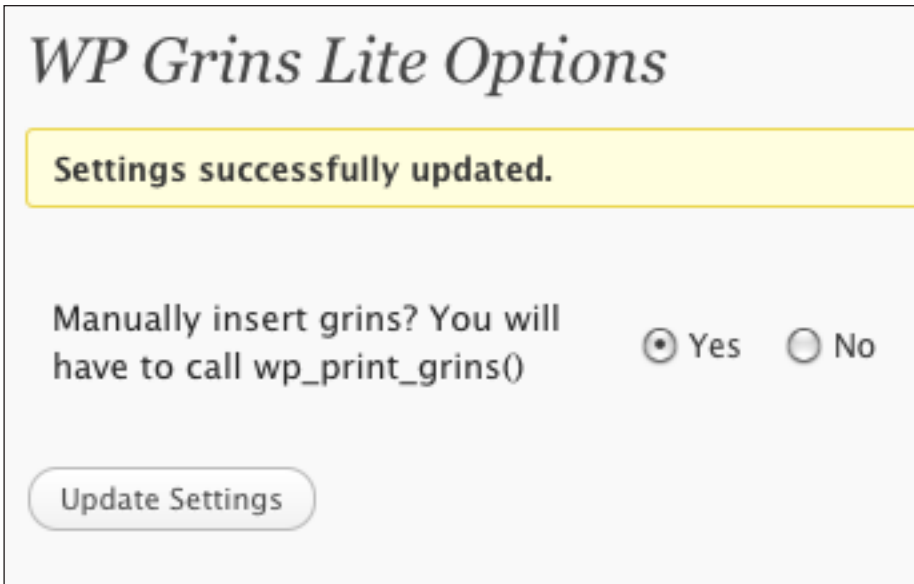


Figure 10. WP Grins Admin Panel

If everything worked fine, make sure you select the “Manual” option back to “No” so that the grins will automatically be included.

Since we have the admin panel up and running, let’s work on our final piece: the JavaScript file.

The JavaScript File (`wp-grins.js`)

The base for our `wp-grins.js` file will contain two functions: one to print out each grin as it is clicked (`grin`) and another to initiate our Ajax call to print out the grins where needed (`init`).

Our base code will look like the following:

```
jQuery(document).ready(function() {  
var $j = jQuery;  
$j.wpgrins = {  
    grin: function(tag) {  
  
    },  
    init: function() {  
  
    }  
};  
$j.wpgrins.init();  
});
```

Now the code for printing out each grin (using the `grin` function) in the appropriate text box is monstrous. How about you just nod your head and assume that it “just works”? Okay, then (phew!).

What we’re really here for is Ajax anyways, so let’s get to the `init` function.

```
init: function() {  
    if (wpgrinslite.MANUAL == "true") { return; }  
    var s = {};  
    s.response = 'ajax-response';  
    s.type = "POST";  
    s.data = $j.extend(s.data, {action: 'grins'});  
    s.global = false;  
    s.url = wpgrinslite.Ajax_Url;  
    s.timeout = 30000;  
    s.success = function(r) {  
        //success stuff here  
    }  
    $j.ajax(s);  
}
```

Please keep in mind that I defined the variable `$j` as our reference to jQuery. I also called the `init` function (using `$j.wpgrins.init()`) upon loading (the goal is to load the grins upon a page load).

The first thing we check for is if the user is going to manually insert the smilies. This will be `false` in the admin panel (remember our localized JavaScript variables?), so no worries there. On a post, it will be what the user has decided upon in the admin panel.

We then build our `s` object and build a `data` object with an `action` called `grins`. The action variable is needed to interface with the WordPress `wp_ajax` action, which will call our class method `ajax_print_grins` when the `grins` action is detected. After the `s` object is built, we make our Ajax call.

What we expect to receive back from our Ajax processor is a string of all the available smilies.

Let's move into our currently empty `success` function.

Via our `success` function, we will print out the string filled with smilies.

The string with our smilies will be contained in variable `r`.

```
s.success = function(r) {
    var grinsDiv = '<div id="wp_grins">'+r+'</div>';
    if ($j('#postdiv').length > 0) {
        var type = 'after';
        var node = $j('#postdiv');
    } else if ($j('#postdivrich').length > 0) {
        var type = 'after';
        var node = $j('#postdivrich');
    } else if ($j('#comment').length > 0) {
        var type = 'before';
        var node = $j('#comment');
    } else {
        return;
    }
    switch (type) {
        case 'after':
            node.after(grinsDiv);
            $j("#wp_grins").css("paddingTop", "5px");
            break;
        case 'before':
            node.before(grinsDiv);
            break;
    }
} //end success
```

We perform several conditional checks to see if we're in a post view (in the admin panel). If so, we assign a variable named `type` with the value `after` (since we want the smilies to appear after the edit area).

If we're in a comment area, we assign `type` with value `before`.

Up next is a simple `switch` statement that determines if the grins are displayed before or after

each node. Again, if we're in a comment area, the grins are displayed before the edit box. If we're in a post area, the grins are displayed after the edit box.

Coolness! We're done with our script (and consequently) the plugin.

WP Grins Lite Conclusion

WP Grins Lite is a simple plugin in concept, but allowing the manual inclusion of the smilies presented a dilemma that was solved via Ajax.

With the new functionality, the grins will now load upon a page load. And since JavaScript is needed to interface with the grins in the first place, using Ajax here isn't a huge disadvantage.

If anything, Ajax does the job well here, since users who have JavaScript disabled won't even see the grins.

Example 2: Static Random Posts

Example 2: Static Random Posts

The WordPress Plugin Static Random Posts was built out of frustration with other “random posts” plugins.

I would often visit a site that had displayed random posts. If I clicked on one, I’m taken to the post. But if I click the back button, all of the posts are refreshed (unless the website makes use of a caching plugin).

To combat this refreshing problem, I thought to myself, “What if the random posts were static for a certain amount of time?”

Another requirement for me is that these random posts could be “refreshed” manually by the blog’s admin. For example, if I set the time to refresh for one week (10,080 minutes in a week), I could manually refresh the posts until I found a suitable combination that I liked.

After some brainstorming, I came up with the main requirements for the plugin:

- The plugin will be a widget that can be included just about anywhere.

- The plugin will have an option to set the number of posts displayed and the title of the widget.
- The plugin will have an admin panel to set the refresh time (minutes) and to exclude certain categories if necessary.
- The plugin will have a “Refresh” link on the blog’s front-end so that admin can refresh the posts at will.

Based on those requirements, we know we need an admin panel, a script to process the “Refresh” link, and a main plugin file to create and process the widget.

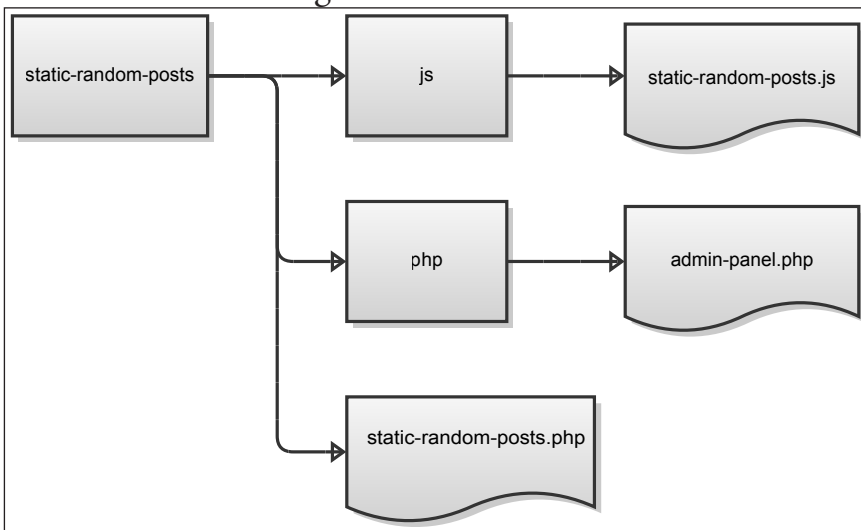


Figure 11. Static Random Posts File Structure

Let's get started on the `static-random-posts.php` file.

Creating the Static Random Posts Widget

Let's first look at the class skeleton for our widget. Please note that we inherit the class `WP_Widget`, which aids tremendously in creating widgets.

```
<?php
if (!class_exists('static_random_posts')) {
    class static_random_posts extends WP_Widget {
        var $localizationName = "staticRandom";
        var $adminOptionsName = "static-random-posts";

        function static_random_posts(){
        }
        // widget - Displays the widget
        function widget($args, $instance) {
        }
        //Updates widget options
        function update($new, $old) {
        }
        //Widget form
        function form($instance) {
        } //End function form
    } //End class
}
add_action('widgets_init', create_function('', 'return
register_widget("static_random_posts");' ) );
?>
```

From looking at the above code, we know we have several standard widget methods that need some work.

- `static_random_posts` - Our class constructor.
- `widget` - Displays the widget to the end user.
- `update` - Updates the widget when the admin modifies the widget's settings.
- `form` - The widget interface in the admin panel.

We'll also need to define some additional methods that will assist us with our widget output.

- `ajax_refresh_static_posts` - Will return HTML via Ajax with updated random posts.
- `init` - An initialization method that will assist with plugin localization.
- `print_posts` - Will return the HTML of the random posts.
- `get_posts` - Will return a comma-separated list of post IDs to display.
- `build_posts` - Will return a widget instance with new random posts.
- `add_admin_pages` - Allows us to add an admin settings page.
- `get_admin_options` - Retrieves the stored admin settings for the widget.

- `add_post_scripts` - Adds the scripts necessary for the plugin's Ajax.

Let's get started on the `static_random_posts` method first, which will serve as our class constructor.

static_random_posts

We'll use the class constructor to load all of our WordPress actions and admin options.

```
/**
 * PHP 4 Compatible Constructor
 */
function static_random_posts(){
    $this->adminOptions = $this->get_admin_options();

    //Initialization stuff
    add_action('init', array(&$this, 'init'));

    //Admin options
    add_action('admin_menu', array(&$this, 'add_admin_
pages'));
    //JavaScript
    add_action('wp_print_scripts', array(&$this, 'add_post_
scripts'),1000);
    //Ajax
    add_action('wp_ajax_refreshstatic', array(&$this, 'ajax_
refresh_static_posts'));
    //Widget stuff
    $widget_ops = array('description' => __('Shows Static
Random Posts.', $this->localizationName) );
    //Create widget
    $this->WP_Widget('staticrandomposts', __('Static Random
Posts', $this->localizationName), $widget_ops);
}
```

You probably recognize all of the callback methods since they were explained earlier.

Of note here is the initialization of the `wp_ajax` action. We used an Ajax action name of `refreshstatic`, which we'll need later for JavaScript use. The `nopriv` suffix isn't used here since we don't want unprivileged users to have access to the script.

Now that the constructor is finished, let's work on the interface that will be displayed to the admin when adding the widget (under Administration > Appearance > Widgets).

form

The `form` method will be automatically called when the widget is added in the admin panel.

Within this method, we will create the user interface that the admin will see.

Let's get started on this method's initialization code:

```
//Widget form
function form($instance) {
    $instance = wp_parse_args((array)$instance,
array('title'=> __("Random Posts", $this->localizationName), 'postlimit'=>5, 'posts'=>', 'time'=>'));
    $postlimit = intval($instance['postlimit']);
    $posts = $instance['posts'];
    $title = esc_attr($instance['title']);
```

The `form` method is automatically passed an `instance` of the widget. We use the function `wp_parse_args` to merge the `instance` with our defaults.

Our defaults are as follows:

- `title` - The title of the widget that will be displayed on the blog's front end.
- `postlimit` - How many posts to retrieve.
- `posts` - A comma separated string with our stored posts.
- `time` - The time the admin sets for post refreshing.

Creating the user interface will fall after the initialization.

When creating our labels and form inputs, we'll make use of class methods `get_field_id` and `get_field_name` and pass it the names of our defaults (e.g., `postlimit`). These methods are part of the `WP_Widget` class that our class inherited.

The `get_field_id` and `get_field_name` methods echo out the unique IDs and form names necessary for the widget.

```

?>
  <p>
    <label for="<?php echo esc_attr($this->get_field_
id('title')); ?>"><?php _e("Title", $this->localizationName);
?><input class="widefat" id="<?php echo esc_attr($this-
>get_field_id('title')); ?>" name="<?php echo esc_attr($this-
>get_field_name('title')); ?>" type="text" value="<?php echo
esc_attr($title); ?>" />
    </label>
  </p>
  <p>
    <label for="<?php echo esc_attr($this->get_field_
id('postlimit')); ?>"><?php _e("Number of Posts to Show",
$this->localizationName); ?><input class="widefat" id="<?php
echo esc_attr($this->get_field_id('postlimit')); ?>" name="<?php
echo esc_attr($this->get_field_name('postlimit')); ?>"
type="text" value="<?php echo esc_attr($postlimit); ?>" />
    </label>
  </p>
  <p><?php _e("Please visit", $this->localizationName) ?> <a
href="options-general.php?page=static-random-posts.php"><?php
_e("Static Random Posts", $this->localizationName) ?></a> <?php
_e("to adjust the global settings", $this->localizationName)
?>.</p>
  <?php
} //End function form

```

Towards the end of our user interface, we provide a link to our global options, which is a link to our admin page (`options-general.php?page=static-random-posts.php`). This link is only for convenience and lets the admin know there are other options for the widget that affect all the widgets globally.

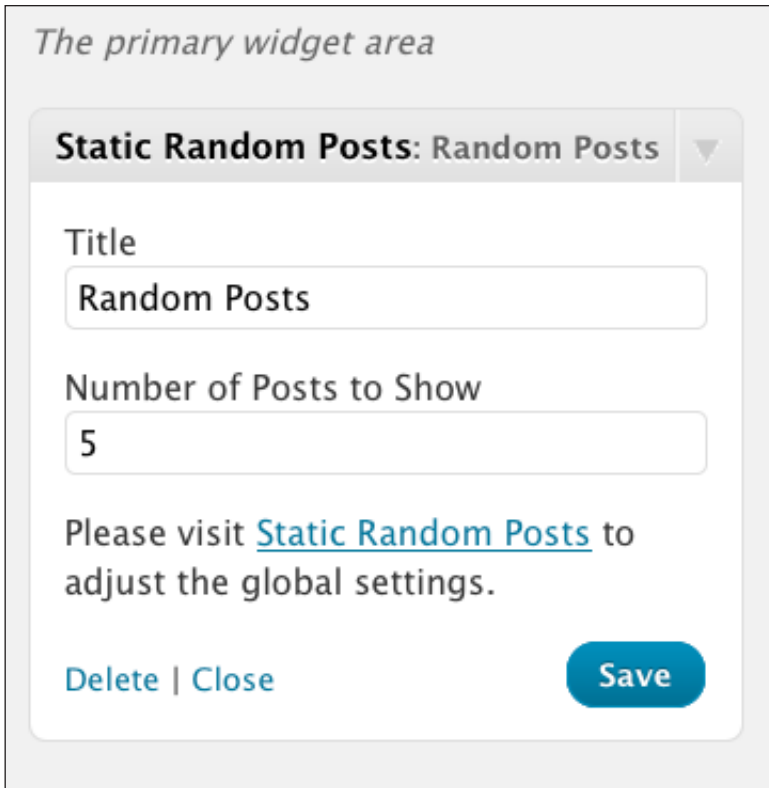


Figure 12. Widget User Interface

As you can see from the screenshot, there is a “Save” button. When the button is clicked, the `update` method runs, so let’s move on to that next.

update

The `update` method performs one simple task: it updates the widget with the options the admin selects.

When the admin clicks “Save”, a *new* and *old* instance is sent to the *update* method.

We simply assign the variable *\$instance* with the *\$old* instance. We then pass the *\$instance* variable our new data and return the instance. Not much to it.

```
//Updates widget options
function update($new, $old) {
    $instance = $old;
    $instance['postlimit'] = intval($new['postlimit']);
    $instance['title'] = esc_attr($new['title']);
    return $instance;
}
```

Let’s now work on our class variable *\$this->adminOptions*, which gets initialized via the *get_admin_options* method.

get_admin_options

The *get_admin_options* method is practically identical to the method we used in Example 1, with the exception of the default keys used.

Our defaults in this case are:

- *minutes* - Number of minutes until the posts are refreshed.

- `categories` - The categories of the posts we would like to exclude.

Both of these options will be used in the admin panel, as well as in several other methods in the class that we have yet to define.

Let's get to the code.

```
//Returns an array of admin options
function get_admin_options() {
    if (empty($this->adminOptions)) {
        $adminOptions = array(
            'minutes' => '5',
            'categories' => ''
        );
        $options = get_option($this->adminOptionsName);
        if (!empty($options)) {
            foreach ($options as $key => $option) {
                if (array_key_exists($key,
                    $adminOptions)) {
                    $adminOptions[$key] = $option;
                }
            }
        }
        $this->adminOptions = $adminOptions;
        $this->save_admin_options();
    }
    return $this->adminOptions;
}
```

Again, we set the defaults, and merge the keys with any saved ones.

Since this method calls `save_admin_options`, let's work on that next.

save_admin_options

Once again, this code is practically identical to the method in Example 1.

```
//Saves for admin
function save_admin_options(){
    if (!empty($this->adminOptions)) {
        update_option($this->adminOptionsName, $this->adminOptions);
    }
}
```

The code is fairly simple. It takes the class variable `$this->adminOptions` and updates it according to our `$this->adminOptionsName` variable.

So what's next?

How about we work on some of our actions.

init

The `init` method will perform our plugin localization. We called it by adding this action in our constructor: `add_action('init',array(&$this, 'init'));`

The only thing to note here is that all of the localization files will be included in the “languages” sub-directory, which I've included in the final plugin version.

```

/* init - Run upon WordPress initialization */
function init() {
    /** Begin Localization Code */
    $static_random_posts_locale = get_locale();
    $static_random_posts_mofile = WP_PLUGIN_DIR . "/static-
random-posts/languages/" . $this->localizationName . "-".
$static_random_posts_locale.".mo";
    load_textdomain($this->localizationName, $static_random_
posts_mofile);
    /** End Localization Code */
} //end function init

```

add_admin_pages

The next action is the action for adding in our admin page. Here is the action we used: `add_action('admin_menu', array(&$this, 'add_admin_pages'))`;

And here's the resulting code:

```

function add_admin_pages(){
    add_options_page('Static Random Posts', 'Static Random
Posts', 9, basename(__FILE__), array(&$this, 'print_admin_
page'));
}

```

Simple and to the point. We call the method `print_admin_page`, so let's move onto that one next.

print_admin_page

This method will output our admin panel. As with Example 1, I use an external PHP file for the admin panel.

```
//Provides the interface for the admin pages
function print_admin_page() {
    include dirname(__FILE__) . '/php/admin-panel.php';
}
```

Let's move onto the next action, which will use the `add_post_scripts` method.

add_post_scripts

The `add_post_scripts` method is called by the following action: `add_action('wp_print_scripts', array(&$this, 'add_post_scripts'), 1000);`

What we want to do within this function is register our JavaScript files and do some simple detection to ensure the scripts don't load unless necessary.

```
//Add scripts to the front-end of the blog
function add_post_scripts() {
    //Only load the widget if the widget is showing
    if ( !is_active_widget(true, $this->id, $this->id_base)
    || is_admin() ) { return; }
    //queue the scripts
    wp_enqueue_script("wp-ajax-response");
    wp_enqueue_script('static_random_posts_script', plugins_
url('static-random-posts') . '/js/static-random-posts.js',
array("jquery", "wp-ajax-response") , 1.0);
    wp_localize_script( 'static_random_posts_script',
'staticrandomposts', $this->get_js_vars());
}
```

What we first do in this method is ensure that the widget is indeed being shown on the front-end (via the `is_admin` conditional).

We call the `is_active_widget` WordPress function to determine if the widget is being shown. If it isn't, we get out of the method.

We then load our scripts and provide some script localization via `wp_localize_script`.

When we localize, we call the method `get_js_vars`. Let's move on to that method next.

get_js_vars

All that we're doing in the `get_js_vars` method is returning an array of variables that need to be localized for use in our JavaScript file. The `get_js_vars` output is used as one of the arguments for the WordPress function `wp_localize_script`.

Since we will have a "Refresh..." button, we need a variable to hold that text.

We also need a "Loading..." message for when the button is clicked.

Finally, we need the URL to WordPress' `admin-ajax.php` file.

```
//Returns various JavaScript vars needed for the scripts
function get_js_vars() {
    return array(
        'SRP>Loading' => esc_js(__('Loading...', $this->localizationName)),
        'SRP>Refresh' => esc_js(__('Refresh...', $this->localizationName)),
        'SRP>AjaxUrl' => admin_url('admin-ajax.php')
    );
} //end get_js_vars
```

We're done with most of our actions, but let's go ahead and move to the `widget` method, which will display the random posts on the front-end.

widget

The `widget` method is automatically called when displaying the widget on the front-end.

The method is passed two variables:

- `$args` - An array of arguments.
- `$instance` - The instance of the widget.

The beginning of the method will look like this:

```
// widget - Displays the widget
function widget($args, $instance) {
    extract($args, EXTR_SKIP);
```

The first thing we do is extract the `$args` array using the PHP function `extract`.

The `$args` array is filled with some useful keys. By using the `extract` function, we now have the following variables available:

- `$before_title` - What code and/or words to display before the widget title.
- `$after_title` - What code and/or words to display after the widget title.
- `$before_widget` - What code and/or words to display before the widget.
- `$after_widget` - What code and/or words to display after the widget.

As a plugin author, we shouldn't modify these variables, but in order to ensure maximum theme compatibility, using these variables is a must.

A themer defines these variables somewhere in their theme (most likely in `functions.php`) by registering a sidebar (example):

```
if ( function_exists('register_sidebar') ) {
    register_sidebar(array(
        'name' => 'Sidebar1',
        'before_widget' => '<div class="sidebar-box"><div
class="sidebar-box-inside">',
        'after_widget' => '</div></div>',
        'before_title' => '<h3>',
        'after_title' => '</h3>',
    ));
}
```

We have to assume that the themer knows what he's doing and rely on his code for the look and feel of our widget.

Let's look at the rest of this code for the widget method.

```
// widget - Displays the widget
function widget($args, $instance) {
    extract($args, EXTR_SKIP);
    echo $before_widget;
    $title = empty($instance['title']) ? __('Random Posts',
$this->localizationName) : apply_filters('widget_title',
$instance['title']);

    if ( !empty( $title ) ) {
        echo $before_title . $title . $after_title;
    };
    //Get posts
    $post_ids = $this->get_posts($instance);
    if (!empty($post_ids)) {
        echo "<ul class='static-random-posts' id='static-
random-posts-$this->number'>";
        $this->print_posts($post_ids);
        echo "</ul>";
        if (current_user_can('administrator')) {
            $refresh_url = esc_url( wp_nonce_url(admin_
url("admin-ajax.php?action=refreshstatic&number=$this-
>number&name=$this->option_name"), "refreshstaticposts"));
            echo "<br /><a href='$refresh_
url' class='static-refresh'>" . __("Refresh...", $this-
>localizationName) . "</a>";
        }
    }
    echo $after_widget;
}
```

There's four major things going on here in this method:

- We add in the default variables before/after the widget and before/after the title.
- We retrieve the post IDs to display by calling our `get_posts` method (not yet defined). We pass it the current `$instance` variable.
- We output an unordered list and call the `print_posts` method (not yet defined) with the list of our post IDs.
- If the user is admin, we output a link to `admin-ajax.php` for refreshing the random posts (we'll capture this via JavaScript later).

When all is said and done, our front-end output should be a widget filled with random links.

We have four more methods to go before we are done with the class. Let's go over them quickly:

- `get_posts` - Returns a comma separated string of post IDs to include in our output.
- `build_posts` - Returns an instance that includes the post IDs and the time of when the posts will need to be refreshed.
- `print_posts` - Prints out the posts to the front-end.

- `ajax_refresh_static_posts` - Our Ajax processor.

`get_posts`

The `get_posts` method is what we use to return a string of post IDs.

It takes in two arguments:

- `$instance` - The widget instance.
- `$build` (`true` or `false`) - Whether to build new posts or not.

Before we get into the code, let me explain the main functionality of this method.

We retrieve the number of posts to display (this value is saved in our widget instance). We then do a quick check to see if there are stored post IDs. If not, we build the IDs.

We then check to see if the time has expired for the static random links. If the time has expired, we build new random links.

We then check to see if the `$build` variable is `true`. If it is, we build the posts.

In all three conditionals, we update the main instance.

Let's get to the code:

```
//Returns the post IDs of the posts to retrieve
function get_posts($instance, $build = false) {
    //Get post limit
    $limit = intval($instance['postlimit']);

    $all_instances = $this->get_settings();
    //If no posts, add posts and a time
    if (empty($instance['posts'])) {
        //Build the new posts
        $instance = $this->build_posts($limit,$instance);
        $all_instances[$this->number] = $instance;
        update_option( $this->option_name, $all_instances
    );
    } elseif(($instance['time']-time()) <=0) {
        //Check to see if the time has expired
        //Rebuild posts
        $instance = $this->build_posts($limit,$instance);
        $all_instances[$this->number] = $instance;
        update_option( $this->option_name, $all_instances
    );
    } elseif ($build == true) {
        //Build for the heck of it
        $instance = $this->build_posts($limit,$instance);
        $all_instances[$this->number] = $instance;
        update_option( $this->option_name, $all_instances
    );
    }
    if (empty($instance['posts'])) {
        $instance['posts'] = '';
    }
    return $instance['posts'];
}
```

Pay attention to the variable `$all_instances`. This variable holds all of our Static Random Posts wid-gets.

When we are updating our instance, we update with this code:

```
$instance = $this->build_posts($limit,$instance);  
$all_instances[$this->number] = $instance;  
update_option( $this->option_name, $all_instances );
```

We have to assume that the `build_posts` method updates the `$instance['posts']` key. We then update only our widget by using:

```
$all_instances[$this->number] = $instance;
```

Finally, we call the WordPress function `update_option` and pass it our widget's `option_name`, and pass it all of the instances.

Since there is such heavy use of the `build_posts` method, let's move onto that method next.

build_posts

The `build_posts` method takes in two arguments:

- `$limit` - The number of posts to retrieve.
- `$instance` - The instance to update and return.

First, we get our saved categories IDs as specified in the admin panel options. Since the categories are in an array, we `implode` them and separate the categories by a comma. The `$cats` variable will

hold all of the category IDs to exclude (e.g., -3,-4,-5).

We then retrieve our list of posts using the WordPress function `get_posts`. We pass it our categories to exclude, the limit on posts, and the `rand` option (for our posts to be random).

After that, we build our `$post_ids` array, and `implode` this array to be a comma-separated string.

We update our instance with the `$post_ids` and a new time.

Finally, we return the instance.

```
//Builds and saves posts for the widget
function build_posts($limit, $instance) {
    //Get categories to exclude
    $cats = @implode(',', $this->adminOptions['categories']);

    $posts = get_posts("cat=$cats&showposts=$limit&orderby=r
and"); //get posts by random
    $post_ids = array();
    foreach ($posts as $post) {
        array_push($post_ids, $post->ID);
    }
    $post_ids = implode(',', $post_ids);
    $instance['posts'] = $post_ids;
    $instance['time'] = time()+(60*intval($this-
>adminOptions['minutes']));

    return $instance;
}
```

print_posts

The `print_posts` method was called in the `widget` method when outputting the random posts.

The `print_posts` method takes a string of comma-separated post IDs. It then retrieves a list of posts based on those IDs, and outputs the result.

```
//Prints or returns the LI structure of the posts
function print_posts($post_ids,$echo = true) {
    if (empty($post_ids)) { return ''; }
    $posts = get_posts("include=$post_ids");
    $posts_string = '';
    foreach ($posts as $post) {
        $posts_string .= "<li><a href='" . get_
permalink($post->ID) . "' title='". esc_attr($post->post_title)
.'">" . esc_html($post->post_title) . "</a></li>\n";
    }
    if ($echo) {
        echo $posts_string;
    } else {
        return $posts_string;
    }
}
```

As you can see from the code, it retrieves the posts, and goes through them one-by-one to build a string full of posts.

If the `$echo` variable is set to `true` (default), it `echoes` the string rather than returning it.

The `$echo` variable will come in handy when we're using our Ajax processor to retrieve the list of posts.

Speaking of the Ajax processor, let's move on to our Ajax-processing method named `ajax_refresh_static_posts`.

`ajax_refresh_static_posts`

The `ajax_refresh_static_posts` method is a callback method for the `wp_ajax` WordPress action. When the `refreshstatic` Ajax action is detected (and the user has the appropriate privileges), the `ajax_refresh_static_posts` method is called.

The first thing we need to do is to perform a nonce check.

```
//Build new posts and send back via Ajax
function ajax_refresh_static_posts() {
    check_ajax_referer('refreshstaticposts');
```

If you venture way back to the widget method, you will notice that we used the nonce name `refreshstaticposts` when we called the `wp_nonce_url` function to build our “Refresh...” URL.

Assuming this nonce gets passed to us via JavaScript (it's coming, I swear!), we perform a nonce check

using `check_ajax_referer`. If the nonce check fails, a `"-1"` is sent back and nothing progresses further.

Let's now capture some `$_POST` variables and perform some data validation.

```
if (isset($_POST['number']) && current_user_
can('administrator')) {
    $number = intval($_POST['number']);
    $action = addslashes(preg_replace("/^[a-z0-9]/i", '',
strip_tags($_POST['action'])));
    $name = addslashes(preg_replace("/^[a-z0-9]/i", '',
strip_tags($_POST['name'])));
```

With our `$number` and `$name` variables at hand, we have enough information to retrieve our widget, so let's do that next.

```
//Get the SRP widgets
$settings = get_option($name);
$widget = $settings[$number];
```

First, we get the widgets based on our widget name (there could be more than one of our widgets displayed).

We then narrow down the selection to just one widget by using our `$number` variable with the `$settings` array.

Our next task is to build the new posts:

```
//Get the new post IDs
$widget = $this->build_posts(intval($widget['postlimit']),$widget);
$post_ids = $widget['posts'];
```

We've gone over the `build_posts` method before, so we know it returns an `instance` of the updated widget with brand new posts.

We then assign `$post_ids` with the new posts for later use.

We now need to save the widget and refresh the cache (if available).

```
//Save the settings
$settings[$number] = $widget;
update_option($name, $settings);

//Let's clean up the cache
//Update WP Super Cache if available
if(function_exists("wp_cache_clean_cache")) {
    @wp_cache_clean_cache('wp-cache-');
}
```

Now it's time to build and return our Ajax response.

```
//Build and send the response
$response = new WP_Ajax_Response();
$response->add( array(
    'what' => 'posts',
    'id' => $number,
    'data' => $this->print_posts($post_ids,
false)));
    $response->send();
}
exit;
} //end ajax_refresh_static_posts
```

We assign the `data` key with the result of the `print_posts` method (which we pass our `$post_ids` variable). The `false` we send tells the method not to `echo` out the variables and return them as a string instead.

We're now done with creating the widget! Let's move on to the admin panel.

The Admin Panel (`admin-panel.php`)

There are three things we wish to accomplish with our admin panel for this example.

- Allow the user to select a time (in minutes) for the random posts to be refreshed.
- Allow a user to select the categories he wants to exclude when retrieving the random posts.
- Update the time and categories and save them to our admin options.

First, let's deal with security:

```
<?php
/* Admin Panel Code - Created on April 19, 2008 by Ronald
Huereca
Last modified on October 07, 2010
*/
if (empty($this->adminOptionsName)) { die(''); }

$options = $this->adminOptions; //global settings

//Check to see if a user can access the panel
if ( !current_user_can('manage_options') )
    die("nope");
```

Now that we have the introduction of our file over with, let's get to updating the options.

Updating the Options

The first obstacle we'll tackle is that of time. We want to ensure that the time entered is numeric, and that the time is greater than one minute (not set to zero).

But first, we need to check our nonce, which uses the string `static-random-posts_admin-options`.

```
//Update settings
if (isset($_POST['update'])) {
    check_admin_referer('static-random-posts_admin-
options');
    $error = false;
    $updated = false;
```

We check for the existence of `$_POST['update']`, check our nonce, and set defaults for variables `$error` and `$updated`.

Up next is checking the `$_POST['time']` variable.

```
//Validate the time entered
if (isset($_POST['time'])) {
    $timeErrorMessage = '';
    $timeClass = 'error';
    if (!preg_match('/^\d+$/i', $_POST['time'])) {
        $timeErrorMessage = __("Time must be a numerical
value", $this->localizationName);
        $error = true;
    } elseif($_POST['time'] < 1) {
        $timeErrorMessage = __("Time must be greater than
one minute.", $this->localizationName);
        $error = true;
    } else {
        $options['minutes'] = $_POST['time'];
        $updated = true;
    }
    if (!empty($timeErrorMessage)) {
        ?>
        <div class="<?php echo $timeClass;
?>"><p><strong><?php _e($timeErrorMessage, $this-
>localizationName); ?></p></strong></div>
        <?php
    }
}
```

We use a regular expression to ensure that all values are a digit (you could have also used the PHP `is_num` function). If the time has non-numerical values, we set the error message.

We next check to see if the time value is less than one. If it is, we set the error message.

After passing the first two conditionals, we assume we're good and update the `$options` array with the new time (we'll save it later).

If the `$timeErrorMessage` isn't empty (one of the first two conditionals has failed), then an error message is spit out for the user to see.

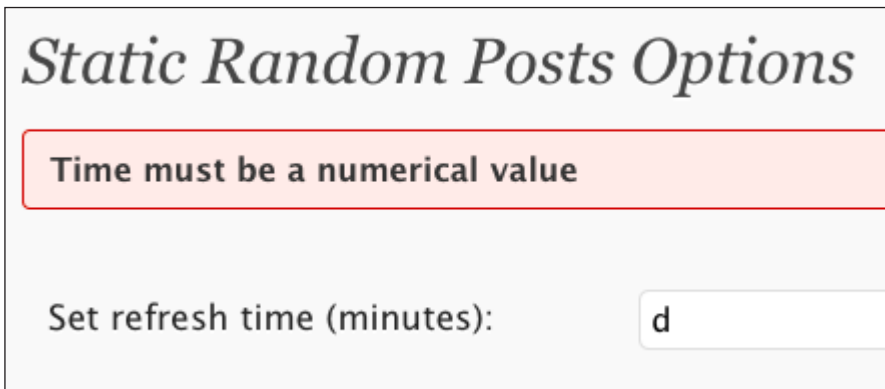


Figure 13. SRP Error Message

Up next is building our categories. Since we want to “exclude” them, we need to add a minus (-) sign before each one.

To perform this exclusion, we do a `for` loop and update each item in the array with a minus (-) sign. After the loop, we assign our `$options` array the new values.

```

//categories (add a "-" sign for exclusion)
for ($i=0; $i<sizeof($_POST['categories']); $i++) {
    $_POST['categories'][$i] = "-" . $_POST['categories']
[$i];
}
$options['categories'] = $_POST['categories'];

$updated = true;
if ($updated && !$error) {
    $this->adminOptions = $options;
    $this->save_admin_options();
    ?>
    <div class="updated"><p><strong><?php _e('Settings
successfully updated.', $this->localizationName) ?></strong></
p></div>
        <?php
    }
}
?>

```

Finally, we save the options and output a “success” message.

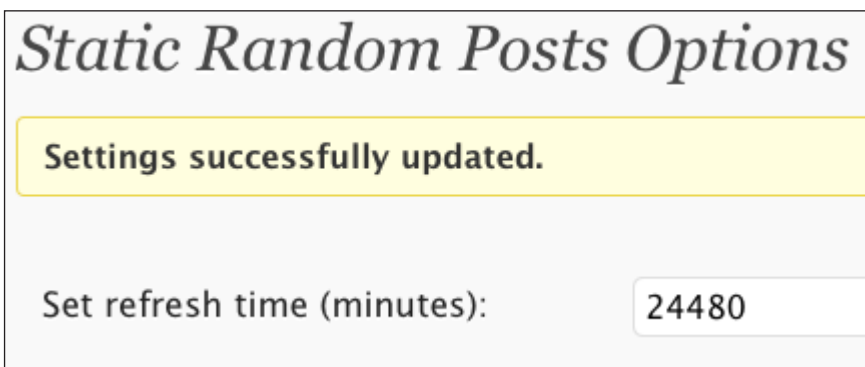


Figure 14. SRP Success Message

Now it’s time to work on the user interface.

The User Interface

The user interface will contain a simple text box to enable users to enter the minutes desired before the random posts are refreshed (you've already seen a few screenshots with this already included).

Additionally, we'll output rows of checkboxes that will contain the blog's categories. By default, none of these will be checked, but we'll have to remember previously checked boxes.

First, let's insert our nonce field.

```
<div class="wrap">
  <h2>Static Random Posts Options</h2>
  <form method="post" action="<?php echo $_
SERVER["REQUEST_URI"]; ?>">
    <?php wp_nonce_field('static-random-posts_admin-options')
?>
```

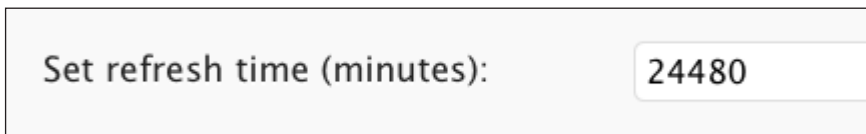
Keep in mind we use the same exact string that we used in our nonce check: `static-random-posts_admin-options`.

As with Example 1, we have the form self-post to itself, which is the reason we have “update” code within the same file. In my experience, this is a cleaner way of updating options rather than having too many separate files all over the place.

Up next is our input box for the refresh rate (minutes).

```
<table class="form-table">
  <tbody>
    <tr valign="top">
      <th scope="row"><?php _e('Set refresh time (minutes):',
$this->localizationName) ?></th>
      <td><input type="text" name="time" value="<?php echo
esc_attr($options['minutes']); ?>" id="comment_time"/><p><?php
_e('Your random posts will be refreshed every', $this-
>localizationName); echo " " . $options['minutes'] . "
";_e('minutes.', $this->localizationName); ?></p></td>
    </tr>
```

The output of the above code would look like this:



Set refresh time (minutes):	<input type="text" value="24480"/>
-----------------------------	------------------------------------

Figure 15. Refresh Time Input Box

Now it's time to show the categories as rows of checkboxes.

What we want here is to get all of the categories as a flat list (no hierarchy). We also want to check each category and see if it is in our “exclude” options. If it is, we want to check the checkbox.

After that, we want to print out the categories.

```

<tr valign="top">
<th scope="row"><?php _e('Exclude Categories:', $this-
>localizationName) ?></th>
<td>
<?php
$args = array(
    'type'                => 'post',
    'child_of'            => 0,
    'orderby'             => 'name',
    'order'               => 'ASC',
    'hide_empty'         => false,
    'include_last_update_time' => false,
    'hierarchical'       => 1);
$categories = get_categories( $args );
foreach ($categories as $cat) {
    $checked = '';
    if (is_array($options['categories'])) {
        if (in_array("-" . $cat->term_id,
$options['categories'], false)) {
            $checked = "checked='checked'";
        }
    }
    echo "<input type='checkbox' id='$cat->term_id'
value='$cat->term_id' name='categories[]' $checked /> ";
    echo "<label for='$cat->term_id'>$cat->name</label><br
/>";
}
?>
</td>
</tr>
</tbody>
</table>

```

Within the `foreach` statement, we check to see if the category is excluded in our admin options. If it is, we mark the category as “checked” (checked categories are excluded from Static Random Posts).

The output of this code will look similar to this figure:

Exclude Categories:

- 500 Words
- Administrative
- Analogies and Metaphors
- Articles
- Blogging
- Christian Poems
- Christianity
- Christianity and Fitness
- City Stages
- Contests

Figure 16. Categories Output

Finally, we print out our “Update” button.

```
<div class="submit">
    <input type="submit" name="update" value="<?php esc_
attr_e('Update Settings', $this->localizationName) ?>" />
</div>
</form>
</div>
```

Phew! Done with the admin panel!

Let’s next move on to our JavaScript file, which will capture our “Refresh...” link and initiate the Ajax call.

The JavaScript File (static-random-posts.js)

The JavaScript file will perform one important function: it must capture when the “Refresh...” button is clicked and initiate an Ajax call. Let’s look at the base code for this file.

```
jQuery(document).ready(function() {  
    var $j = jQuery;  
    $j.staticrandomposts = {  
        init: function() { initialize_links(); }  
    };  
    //Initializes the refresh links  
    function initialize_links() {  
  
    }  
    $j.staticrandomposts.init();  
});
```

What we can tell from the code shown above is that a namespace of `staticrandomposts` is created.

There’s a public function called `init`, and this function is called immediately upon a page load.

The `init` function calls the private function `initialize_links`, so we have to assume the `initialize_links` function will contain the code to initialize our “Refresh...” button and to perform and receive the Ajax request.

Let’s start by writing the code to capture button clicks within the `initialize_links` function.

```
function initialize_links() {  
    $j(".static-refresh").bind("click", function() {  
        return false;  
    });  
}
```

First, let's go back to the "Refresh..." link that the user will click on. Here is the full path that is generated for the link:

```
http://www.domain.com/wp-admin/admin-ajax.php?action=refreshstatic&number=5&name=widget_staticrandomposts&_wpnonce=e4dca644d3
```

Our link is pointing towards WordPress' `admin-ajax.php` and contains four variables:

- `action=refreshstatic` - The action we are taking.
- `number=12` - The unique number of our widget (this could be anything).
- `name=widget_staticrandomposts` - This is the name of our widget.
- `_wpnonce=e4dca644d3` - The unique nonce for the link.

Within our script, we must capture all of these variables and pass them via Ajax.

Now let's get back to the event capturing for our button (err, link).

Our link (in case I didn't mention it before) has a class attribute with the value of `static-refresh`. We bind a `click` event to it, and return `false` (making the link at this point utterly useless).

Once inside the `click` event function, we need to capture the four variables mentioned previously, build our `s` object, and initiate our Ajax call.

```
$( ".static-refresh" ).bind( "click", function() {
    //prepare object for AJAX call
    var obj = $( this );
    obj.html( staticrandomposts.SRP_Loading ); //from
localized variable
    var s = {};
    s.response = 'ajax-response';
    var url = wpAjax.unserialize( obj.attr( 'href' ) );
    s.type = "POST";
    s.data = $.extend( s.data, { action: url.action, number:
url.number, name: url.name, _ajax_nonce: url._wponce } );
    s.global = false;
    s.url = staticrandomposts.SRP_AjaxUrl; //from localized
variable
    s.timeout = 30000;
    s.success = function( r ) {
        //Ajax stuff here
    }
    $.ajax( s );
    return false;
});
```

As shown in the above code, we first assign the “clicked” link to a variable named `obj`.

We alter the link's inner-HTML by using one of our localized variables.

We then start building our `s` object, and create the `s.data` object with our four variables. One thing to note here is that we assign our passed `_wprnonce` variable to `_ajax_nonce` (for use in the Ajax processor).

A `s.success` placeholder is used, and then we initiate our Ajax call.

We're done right? Not quite. We still have to build the inner-workings of the `s.success` function.

First, what are we expecting here? When the user clicks the "Refresh..." link, he's expecting new posts. So we can reasonably expect that the Ajax processor will return a string of these new posts ready for us to insert onto the front-end.

Our first goal upon receiving a response is to change the text back to "Refresh..." and parse the Ajax response.

Let's go ahead and hide the "Refresh..." link (which should now say "Loading..." by the way) and update the text back to its original value. Next, we'll parse the XML response.


```
s.success = function(r) {
    obj.hide();
    obj.html(staticrandomposts.SRP_Refresh);
    //Parse the XML response
    var res = wpAjax.parseAjaxResponse(r, s.response);
```

After we have parsed the response, we go through each of the responses.

```
$j.each( res.responses, function() {
    if (this.what == "posts") {
        var data = this.data;
        $j("#static-random-posts-" + url.number).
hide("slow", function() {
        $j("#static-random-posts-" + url.number).
html(data);
        $j("#static-random-posts-" + url.number).
show("slow", function() { obj.show(); });
        return;
    });
    }
});
```

What we're looking for in this case is a *what* value set to *posts*. We'll then update the widget with the new "data".

Now let's think waaaay back (ok, only a few pages) to when we created the code to output the widget. We created an unordered list and filled this list with our posts.

Still don't remember? Okay, here's a refresher:

```
echo "<ul class='static-random-posts' id='static-random-posts-  
$this->number'>";
```

So when you see in the code `$j('#static-random-posts-' + url.number`, that's exactly what we're using.

We first hide this unordered list. We update its inner-HTML with the new data. We then show the unordered list, and make our “Refresh...” link visible via a callback.

Since someone can theoretically include more than one Static Random Posts widget on a page (a benefit of inheriting the `WP_Widget` class), we have to make sure that each widget instance is unique.

Adding the widget number to a common prefix is a great way to ensure that each widget on the page is served by a unique ID.

WordPress itself already provides a container with a unique ID for our widget, but since we don't want to update the entire widget (just a portion), we created our own unique ID for the unordered list that is used.

Here's the entire `intialize_links` function:

```

function initialize_links() {
    $j("#static-refresh").bind("click", function() {
        //prepare object for AJAX call
        var obj = $j(this);
        obj.html(staticrandomposts.SRP>Loading); //from
Localized variable
        var s = {};
        s.response = 'ajax-response';
        var url = wpAjax.unserialize(obj.attr('href'));
        s.type = "POST";
        s.data = $j.extend(s.data, {action: url.action,
number: url.number, name: url.name, _ajax_nonce: url._
wponce});
        s.global = false;
        s.url = staticrandomposts.SRP_AjaxUrl; //from
Localized variable
        s.timeout = 30000;
        s.success = function(r) {
            obj.hide();
            obj.html(staticrandomposts.SRP_Refresh);
            //Parse the XML response
            var res = wpAjax.parseAjaxResponse(r,
s.response);
            $j.each( res.responses, function() {
                if (this.what == "posts") {
                    var data = this.data;
                    $j("#static-random-posts-" +
url.number).hide("slow", function() {
                        $j("#static-random-
posts-" + url.number).html(data);
                        $j("#static-random-
posts-" + url.number).show("slow", function() { obj.show(); });
                    });
                }
            });
        }
        $j.ajax(s);
        return false;
    });
}

```

We're now done with our JavaScript portion, and done completely with our widget.

Static Random Posts Conclusion

Random Posts

- [The Great Immigration Debate](#)
- [Three Types of Negative Commenters](#)
- [So I Married a Footer](#)
- [A Faster Shopping Experience](#)
- [Airplane](#)

[Refresh...](#)

Figure 17. Static Random Posts Output

Within this chapter you learned how to create a WordPress widget with Ajax capabilities.

I walked you through the brainstorming process to determine what requirements are needed.

So you see how it's all falling into place like a puzzle orgi? Okay, maybe I shouldn't have gotten

that descriptive, but you hopefully get the gist of it (it being Ajax and WordPress) now, yes?

Let's move on to to learn how to make an Ajax registration form.

Example 3: Ajax Registration Form

Example 3: Ajax Registration Form

When building an affiliate program for one of my products, I wanted a way for an affiliate to fill out some required information and have WordPress create a WordPress user account automatically behind the scenes.

What resulted was a simple form on the front-end that the user could fill out and have all of the form validation done on the client side without a page refresh.

The Ajax Registration Form is a plugin that demonstrates how to perform data validation, handle errors, and add a user, all using Ajax.

For this example, we'll be using three files:

- `ajax-registration.php` - The main plugin file.
- `js/registration.js` - The plugin's JavaScript file.
- `css/registration.css` - CSS to handle appearance.

While the structure is quite simple (and really, the plugin is fairly simple as well), there's a few advanced concepts involved that demonstrate how

to pass and return complex data to an Ajax processor.

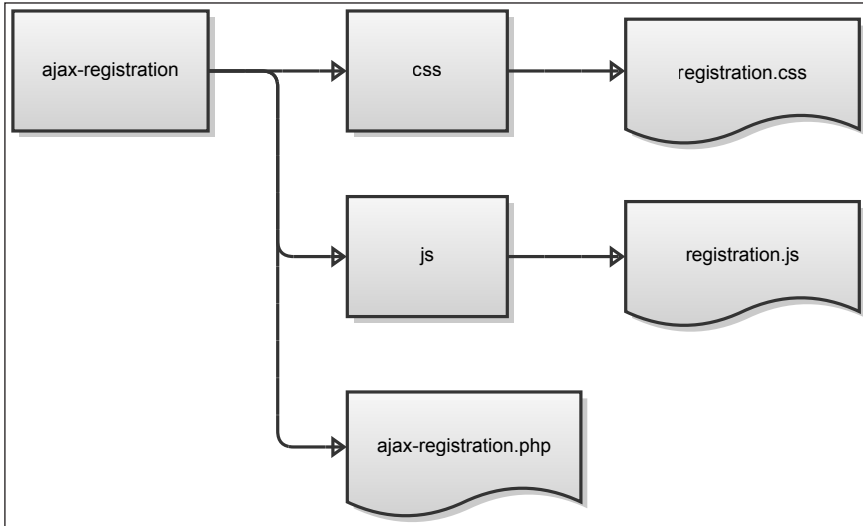


Figure 18. Ajax Registration File Structure

Let's begin with `ajax-registration.php`, which contains all of our plugin logic.

Creating the `Ajax_Registration` Class

The `Ajax_Registration` class will be in charge of:

- Loading scripts and styles.
- Adding a shortcode handler (we'll be using a shortcode to insert a form onto a post).
- Handling the Ajax request.
- Handling page detection for the shortcode.

Knowing these pre-requisites, let's go ahead and check out our class structure.

```
<?php
class Ajax_Registration {

    //Constructors
    function Ajax_Registration() {
        $this->__construct();
    }
    function __construct() {
        //actions and shortcode
    }
    //Add the registration script to a page
    function add_scripts() {
    }
    //Add Styles for the form
    function add_styles() {
    }
    function ajax_process_registration() {
    } //end ajax_process_registration
    //Perform shortcode page detection
    function has_shortcode() {
    }
    //Add/save shortcode information
    function post_save( $post_id ) {
    } //end post_save
    //Print out the shortcode
    function rform_shortcode( ) {
    }
} //end class
//Instantiate
$ajaxregistration = new Ajax_Registration();
?>
```

Let's begin by filling out our constructor. We need actions for scripts and styles (loading on the front-end), an action for when a post is saved (for

caching shortcode data), an action for the Ajax processor, and a shortcode callback.

```
function __construct() {
    //add scripts
    add_action( 'wp_print_scripts', array( &$this, 'add_scripts'
) );
    //add css
    add_action( 'wp_print_styles', array( &$this, 'add_styles' )
);
    //ajax
    add_action( 'wp_ajax_nopriv_submitajaxregistration', array(
    &$this, 'ajax_process_registration' ) );
    add_action( 'wp_ajax_submitajaxregistration', array( &$this,
    'ajax_process_registration' ) );
    //when saving a post
    add_action( 'save_post', array( &$this, 'post_save' ) );
    //shortcode
    add_shortcode( 'rform', array( &$this, 'rform_shortcode' )
);
}
```

If you observe the callback methods, they match most of the methods in our class structure. The `has_shortcode` method isn't referenced, but it's a helper method for adding in scripts and styles (more on that in a bit).

Since our plugin is based on a shortcode for the user interface, let's move into the `rform_shortcode` method first.

rform_shortcode

The `rform_shortcode` method returns a string that consists of our user interface (a registration form, if you haven't guessed by now).

The shortcode used is called `rform`, and you'd insert it into a post or page by using: `[rform]`

When the `rform` shortcode is detected, WordPress calls the `rform_shortcode` method and allows us to return our custom code in string format.

The `rform_shortcode` method will return a basic form as a string.

The form will have the following fields and features (with all having a unique ID for JavaScript use):

- A form - Has an ID of `ajax-registration-form`.
- Input fields - Has IDs of `firstname`, `lastname`, `username`, and `email`.
- A nonce field - Has an ID of `_registration_nonce`.
- A submit button - Has an ID of `ajax-submit`.
- A status message area - Has an ID of `registration-status-message`.

Each input field also has a “name” attribute, which will aid JavaScript in capturing the various values.

Let’s look at the code:

```
function rform_shortcode( ) {
    $return = "<form id='ajax-registration-form'>";
    $return .= wp_nonce_field( 'submit_ajax-registration',
'_registration_nonce', true, false );
    $return .= "<ul id='ajax-registration-list'>";
    $return .= "<li><label for='firstname'>First name:
</label><input type='text' size='30' name='firstname'
id='firstname' /></li>";
    $return .= "<li><label for='lastname'>Last name:
</label><input type='text' size='30' name='lastname'
id='lastname' /></li>";
    $return .= "<li><label for='username'>Desired Username:
</label><input type='text' size='30' name='username'
id='username' /></li>";
    $return .= "<li><label for='email'>E-mail Address: </
label><input type='text' size='30' name='email' id='email' /></
li>";
    $return .= "<li><input type='submit' value='Submit
Registration' name='ajax-submit' id='ajax-submit' /></li>";
    $return .= "<li id='registration-status-message'></li>";
    $return .= "</ul>";
    $return .= "</form>";

    return $return;
}
```

As you can see from the code, it’s just a basic form. The nonce being added (via `wp_nonce_field`) is returned so we could use it with the shortcode. A nonce here isn’t exactly necessary, but is a security

precaution just to make sure the request originating from the correct site.

When all is said and done, the shortcode output should look like this (please note that the output shown already has styles attached).

Register

Please use the form below to register.

First name:

Last name:

Desired Username:

E-mail Address:

Figure 19. Registration Form using the rform Shortcode

Now that the shortcode callback is finished, all an end-user would have to do is include the `[rform]` shortcode on a post or a page.

The next method we'll be tackling is the `post_save` method, which is a callback method for the `save_post` WordPress action.

post_save

The `post_save` method is used to save a custom field when a post or page has the `[rform]` shortcode in its content.

We save this custom field in order to perform some page detection when queueing the plugin's styles and scripts.

Since the `post_save` method is called each time a post or page is saved, we either have to add the custom field (if the shortcode is detected), or remove it (if the shortcode isn't there).

Within the `post_save` method, we'll get the post's content. If the post is a revision, we'll have to get the post's original ID and get the content for that instead.

Once we have the post's content, we'll perform a regular expression check for the `[rform]` shortcode, and save custom field data based on the result.

```
function post_save( $post_id ) {
    //Retrieve the post object - If a revision, get the
    original post ID
    $revision = wp_is_post_revision( $post_id );
    if ( $revision )
        $post_id = $revision;
    $post = get_post( $post_id );

    //Perform a test for a shortcode in the post's content
    preg_match('/^[rform[^\]]*\]/is', $post->post_content,
    $matches);

    if ( count( $matches ) == 0 ) {
        delete_post_meta( $post_id, '_ajax_registration'
    );
    } else {
        update_post_meta( $post_id, '_ajax_registration',
    '1' );
    }
} //end post_save
```

The results of the regular expression check are stored in a variable called `$matches`. If there are any matches, a custom field with the label `_ajax_registration` is saved. If there are no matches, we remove the custom field (even if it never existed).

The custom field, when used with the `post_save` method, ensures that we can perform up-to-date

page detection to load our scripts and styles only where needed (i.e., only when a post or page has the `[rform]` shortcode in its content).

Now that we have two of our shortcode methods down, let's move on to the page-detection portion, which lies within the `has_shortcode` method.

has_shortcode

The `has_shortcode` method is a conditional that returns true if a post or page has our shortcode embedded in it. Otherwise, the method returns false.

When a post is saved and a shortcode is detected in the post's content, a custom field called `_ajax_registration` is set for the post.

This allows us (when queueing our scripts and styles) to just check if this custom field is set. If it is, we allow the scripts and styles to load.

If we didn't use a custom field, we would have to run a regular expression on each page load. This has the potential to slow down a site. While a regular expression alone may not be enough to cripple a site's loading time, it makes more sense to simply cache the result as a custom field.

Let's look at the code for the `has_shortcode` method:

```
//Returns true if a post has the rform shortcode, false if not
function has_shortcode() {
    global $post;
    if ( !is_object($post) ) return false;
    if ( get_post_meta( $post->ID, '_ajax_registration',
true ) )
        return true;
    else
        return false;
}
```

We simply try to get the `_ajax_registration` custom field. If it's set, we return true. If not, we return false. When used with loading the scripts and styles, `has_shortcode` makes sure that the script and style overhead is only loaded on a post that has a shortcode embedded.

All of the shortcode methods are finished, so let's move to adding the plugin's scripts.

add_scripts

The `add_scripts` method is used to load the `registration.js` file on the front-end. We'll have to do some additional page detection as well.

The script has the dependencies of `jquery` and `wp-ajax-response`.

Since we also need to know the location of `admin-ajax.php`, a localized JavaScript variable is added as well.

```
//Add the registration script to a page
function add_scripts() {
    if ( is_admin() || !$this->has_shortcode() ) return;
    wp_enqueue_script( 'ajax-registration-js', plugins_url(
'js/registration.js' ,__FILE__ ), array( 'jquery', 'wp-ajax-
response' ), '1.0' );
    wp_localize_script( 'ajax-registration-js',
'ajaxregistration', array( 'Ajax_Url' => admin_url( 'admin-
ajax.php' ) ) );
}
```

The first conditional, `is_admin`, is used to make sure we're not in the admin area. The second conditional, `has_shortcode`, is used for page detection (to make sure the page in question actually uses the shortcode).

When we localize the script, we provide a JavaScript object name of `ajaxregistration` (JavaScript usage would consist of: `ajaxregistration.Ajax_Url`).

Now that our JavaScript file is properly referenced, let's add some CSS to our plugin.

add_styles

The `add_styles` method will load an external CSS file only for pages with our shortcode embedded.

The CSS for this plugin is fairly simple and you can easily expand upon it if you are willing.

The CSS we need covers the list output, and has some styles for error and status messages.

The following CSS would go into the `registration.css` file:

```
#ajax-registration-list {
    list-style-type: none;
}
#ajax-registration-list label {
    display: block;
}
#ajax-registration-form .error {
    background-color: #FFE8E8;
    border: 1px solid #CC0000;
}
#ajax-registration-form .success {
    background-color: #FFFFE0;
    border: 1px solid #E6DB55;
}
```

And here's the `add_styles` method that queues up the stylesheet:

```
function add_styles() {
    if ( is_admin() || !$this->has_shortcode() ) return;
    wp_enqueue_style( 'ajax-registration-css', plugins_url(
    'css/registration.css', __FILE__ ) );
}
```

Once again, the styles are only loaded on the front-end (via the `is_admin` conditional) and where a shortcode is detected (via the `has_shortcode` conditional).

Our final class method is `ajax_process_registration`, but let's save that for a bit later. For now, let's work on our script file.

The Script File (`registration.js`)

The `registration.js` file will handle our Ajax call and will capture all of the form data.

Let's look at its basic structure:

```
jQuery(document).ready(function() {  
    var $ = jQuery;  
    $.registrationform = {  
        init: function() {  
  
        }  
    }; //end .registrationform  
    $.registrationform.init();  
});
```

We use a jQuery namespace of `registrationform` and all of our code will reside within the `init` function.

Capturing the Form Data

First, let's work on capturing the "submit" event when someone clicks the "Submit" button on the registration form.

```
init: function() {
    $("#ajax-registration-form").submit(function() {
        return false;
    });
}
```

Since our form has an ID of `ajax-registration-form`, we simply attach a "submit" event to it.

The first thing we need to do after a user has hit "Submit" is to clear any error messages.

```
$("#ajax-registration-form").submit(function() {
    //Clear all form errors
    $('#ajax-registration-form input').removeClass('error');
    //Update status message
    $("#registration-status-message").removeClass('error').
    addClass('success').html('Sending...');
    //Disable submit button
    $('#ajax-submit').attr("disabled", "disabled");

    return false;
});
```

If you remember the CSS we used in `registration.css`, there is an error portion for the form inputs (`#ajax-registration-form .error`). Via JavaScript, we need to clear all inputs (e.g., Username, E-mail Address) from displaying this error class.

Our status message must also be cleared, and this has an ID of `registration-status-message`. Once we clear the error for the status message, we change its text to “Sending...” since the user has initiated the registration submission.

Finally, we disable the “Submit” button temporarily.

Up next is retrieving all the form data and serializing it into one string.

```
//Disable submit button
$('#ajax-submit').attr("disabled", "disabled");
//Serialize form data
var form_data = $('#ajax-registration-form input').
serializeArray();
form_data = $.param(form_data);

return false;
```

The variable `form_data` is loaded with all of the form inputs (that have name/value pairs). These values are serialized to an array. If you were to look at `form_data` at this point with a JavaScript debugger, you would see an array with five values. This format is not exactly ideal to pass via Ajax, so we run the jQuery `param` function on the data.

If you were to look at the value of `form_data` now, it would look like:

```
_registration_nonce=0ca8be2b7b&firstname=Ronald&lastname=Huereca  
&username=ronalfy&email=ron%40ronalfy.com
```

This serialized format is perfect for sending via Ajax, and everything is already encoded as well.

Building the Ajax Object

Now that our data is serialized, let's begin building the Ajax object. I'm going to show you a slightly different technique for building the Ajax object, but it really is the same type of object we've built before if you get down to the details.

We'll be making use of jQuery's `post` function. For our particular case, we'll pass it the location to WordPress' `admin-ajax.php`, our data variables, and a callback function for when the Ajax processor sends back a response (sounds similar already, doesn't it?).

Please note that the path to WordPress' `admin-ajax.php` is retrieved from the localized JavaScript variable `ajaxregistration.Ajax_Url`.

For our data parameters, we'll also be passing a nonce (the nonce field has an ID of `_registra-`

tion_nonce), and an Ajax action variable (submitajaxregistration).

```
form_data = $.param(form_data);

//Submit ajax request
$.post( ajaxregistration.Ajax_Url, { action:
'submitajaxregistration', ajax_form_data: form_data, _ajax_
nonce: $('#_registration_nonce').val() },
function(data){
    //Success code goes here
}
);
return false;
```

As you can see from the code, there's a lot less code for building the Ajax object compared to earlier methods shown. However, they both do the same thing.

Parsing the Ajax Response

The `function(data)` portion is the code that parses the Ajax response.

Let's assume that the data variable holds our Ajax response. We now have to parse the data. Let's also build some placeholders for error handling, which I'll explain shortly.


```
function(data){
    var res = wpAjax.parseAjaxResponse(data, 'ajax-
response');
    if (res.errors) {
        //errors
    } else {
        //no errors
    }
}
```

After parsing the Ajax response, the `res` variable holds all of the various responses passed from the `WP_Ajax_Response` class (from the Ajax processor).

If there are any errors present, the `res.errors` variable is set to true. When building the Ajax response object in the Ajax processor, you can pass a `WP_Error` object. If any of these objects are detected, the overall response is flagged as an error.

This error handling will assist us immensely in determining when to spit out an error message (via JavaScript) or a success message.

The `res` object contains whether we have errors, the response objects, and (if applicable), all of the error objects as well.

In addition, the errors object contains two additional variables: `code` and `message`.

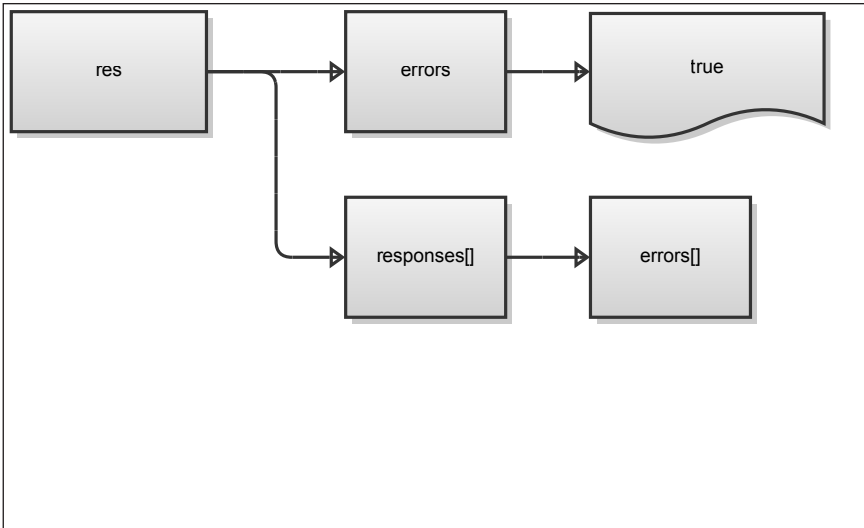


Figure 20. Res Object Structure

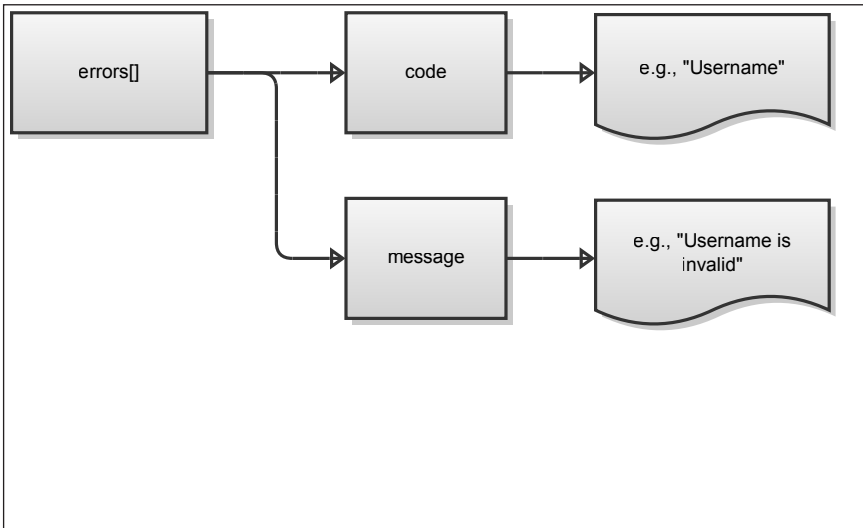


Figure 21. Error Object Structure

Since `res.responses` is an array, and `errors` is also an array, we must perform two `$.each()` loops in order to capture any errors if they are present.

```
if (res.errors) {
    //form errors
    //re-enable submit button
    $('#ajax-submit').removeAttr("disabled");
    var html = '';
    $.each(res.responses, function() {
        $.each(this.errors, function() {
            $("#" + this.code).addClass('error');
            html = html + this.message + '<br />';
        });
    });
    $("#registration-status-message").removeClass('success').
    addClass('error').html(html);
}
```

The first thing we do is re-enable the “Submit” button (since we disabled it when the user first clicks the button).

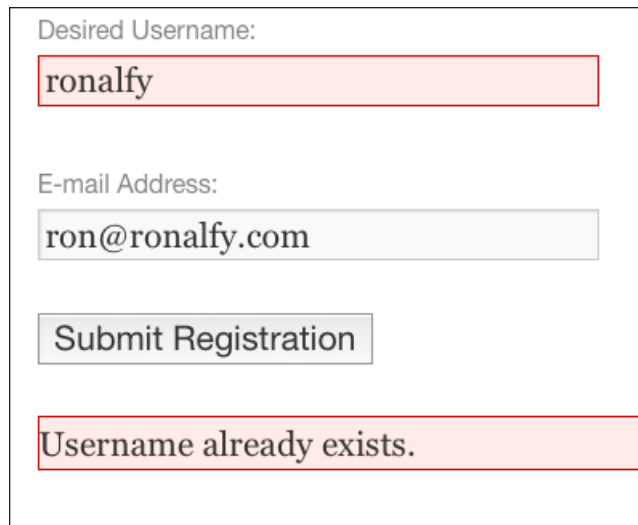
The variable `html` is used here to store the output messages that will be shown in our status area on the form page.

The first `$.each()` statement loops through all of the responses. Within each response is another `$.each()` statement, which loops through all of the errors.

The `this.code` variable stores the form input ID, so we use the value to select the input and add a CSS class called “error” to it. We also add on to

the `html` variable with the contents of `this.message`.

Finally, we clear out the CSS for the status message and output the html error message to the status box.



Desired Username:
ronalfy

E-mail Address:
ron@ronalfy.com

Submit Registration

Username already exists.

Figure 22. Registration Errors

The “else” portion is a little more straightforward. If there are no errors, we go through the first response and output a “success” message to the user informing him that the registration has successfully gone through.

```
} else {  
    //no errors  
    $.each(res.responses, function() {  
        $("#registration-status-message").  
addClass('success').html(this.data);  
        return;  
    });  
}
```

First name:

Last name:

Desired Username:

E-mail Address:

User registration successful. Please check your e-mail.

Figure 23. Registration Successful

Here's the full code for the success portion for parsing the Ajax response:

```
function(data){
    var res = wpAjax.parseAjaxResponse(data, 'ajax-
response');
    if (res.errors) {
        //form errors
        //re-enable submit button
        $('#ajax-submit').removeAttr("disabled");
        var html = '';
        $.each(res.responses, function() {
            $.each(this.errors, function() {
                $("#" + this.code).addClass('error');
                html = html + this.message + '<br
/>';
            });
        });
        $("#registration-status-message").
removeClass('success').addClass('error').html(html);
    } else {
        //no errors
        $.each(res.responses, function() {
            $("#registration-status-message").
addClass('success').html(this.data);
            return;
        });
    }
}
```

Our script is done. We've captured the data, sent the data via Ajax, and have parsed the response. However, we have yet to create the Ajax processor that returns all the necessary data.

Let's move back into the `Ajax_Registration` class and finish the `ajax_process_registration` method.

The Ajax Processor

The first thing we'll be doing in the Ajax processor is verifying the passed nonce. Please note that we created the nonce in the `rform_shortcode` method by passing an action name of `submit_ajax-registration` to the `wp_nonce_field` WordPress function. We'll be using the same action name to verify the nonce.

```
function ajax_process_registration() {
    //Verify the nonce
    check_ajax_referer( 'submit_ajax-registration' );

    exit;
} //end ajax_process_registration
```

Parsing the Passed Form Data

Next, we include a WordPress file called `registration.php`, which contains several helper functions that we'll need (i.e., `validate_username`, `username_exists`). We'll also read in the passed form data and parse it into an array.

```
//Need registration.php for data validation
require_once( ABSPATH . WPINC . '/registration.php');

//Get post data
if ( !isset( $_POST['ajax_form_data'] ) ) die("-1");
parse_str( $_POST['ajax_form_data'], $form_data );
```

Our form data was saved as a `$_POST` variable named `ajax_form_data` (if you don't believe me, take a look back at the JavaScript file).

If this data is available, we use the PHP function `parse_str` and pass it the `$_POST` variable and the name of an array we want the parsed data to be stored into.

The `parse_str` function will turn the serialized string that was passed back into an array. The variable `$form_data` will now have values similar to this:

- `$form_data['firstname']`
- `$form_data['lastname']`
- `$form_data['email']`
- `$form_data['username']`

Please note that you could have used the PHP `extract` function as well to convert all the array keys into PHP variables (e.g., `$firstname`, `$lastname`).

Now that we have our form data as a PHP array, we can perform some data validation on the values.

Data Validation

We'll be making use of the `sanitize_text_field` WordPress function, which is a useful function for sanitizing a text field based on form input.

```
//Get the form fields
$firstname = sanitize_text_field( $form_data['firstname'] );
$lastname = sanitize_text_field( $form_data['lastname'] );
$username = sanitize_text_field( $form_data['username'] );
$email = sanitize_text_field( $form_data['email'] );

$error_response = $success_response = new WP_Ajax_Response();
$errors = new WP_Error();
```

After the inputs have been sanitized, we instantiate an instance of both `WP_Ajax_Response` (for returning the Ajax response) and `WP_Error` (used for error messages).

`WP_Ajax_Response` has already been covered, but the `WP_Error` class needs a little explanation.

The `WP_Error` class has two methods we'll use: `add` (for adding the error codes) and `get_error_codes` (for retrieving all error messages as an array).

As we go through and do some more in-depth data validation, we'll use the `add` method for adding both a unique code and an error message.

For purposes here, we'll be doing a little bit of hacking.

The “code” portion of the `add` method should be unique (and in our case, it will be). However, the “code” portion also doubles for an ID of the form input we want to affect on the client-side.

Let's look at the code to validate the “required” inputs from our form:

```
//Start data validation on firstname/lastname
//Check required fields
if ( empty( $firstname ) )
    $errors->add( 'firstname', 'You must fill out a first
name.', 'firstname' );

if ( empty( $lastname ) )
    $errors->add( 'lastname', 'You must fill out a last
name.' );

if ( empty( $username ) )
    $errors->add( 'username', 'You must fill out a user
name.' );

if ( empty( $email ) )
    $errors->add( 'email', 'You must fill out an e-mail
address.' );
```

If any of our fields are empty, we add an error to the `$errors` object.

Since empty fields are a show stopper, we immediately return an Ajax response if there are errors present.

```
//If required fields aren't filled out, send response
if ( count ( $errors->get_error_codes() ) > 0 ) {
    $error_response->add(array(
        'what' => 'errors',
        'id' => $errors
    ));
    $error_response->send();
    exit;
}
```

Since the `get_error_codes` method returns an array of all error codes, we wrap the PHP `count` function around it to determine if it's empty. If it isn't empty, we return the Ajax response.

The “id” portion of the Ajax response is designed to hold a `WP_Error` object. This is what gives us our `errors` object when parsing via JavaScript.

Assuming there are no further errors, we proceed with validating the username.

The username will be invalid if:

- The username already exists.
- The username doesn't validate.

- The username is reserved.

```
//Add usernames we don't want used
$invalid_usernames = array( 'admin' );
//Do username validation
$username = sanitize_user( $username );
if ( !validate_username( $username ) || in_array( $username,
$invalid_usernames ) ) {
    $errors->add( 'username', 'Username is invalid.' );
}
if ( username_exists( $username ) ) {
    $errors->add( 'username', 'Username already exists.' );
}
```

The `$invalid_usernames` variable is a custom array that will hold username values we don't want users to select. If a username selects a value that matches in this array, the username will be determined to be invalid.

The `sanitize_user` function is used to further sanitize the username (remove tags and unsafe characters). Further assisting in validation is the `validate_username` function, which determines if the username is a valid WordPress username. If not, we add an error.

Finally, the `username_exists` function lets us know if the username already exists as a WordPress user. If the user does exist, we add another error.

Up next is checking whether the e-mail address is valid or not. The e-mail address will be invalid if:

- The e-mail address isn't a proper e-mail address
- The e-mail address already exists.

For the e-mail address validation, we'll be making use of two WordPress functions: `is_email` (checks for a valid e-mail address) and `email_exists` (checks if the e-mail is already registered to another user).

```
//Do e-mail address validation
if ( !is_email( $email ) ) {
    $errors->add( 'email', 'E-mail address is invalid.' );
}
if (email_exists($email)) {
    $errors->add( 'email', 'E-mail address is already in
use.' );
}
```

Again, we add error messages if any of the validation checks fail.

Finally, we check for the existence of errors, and if there are any, we return a response.

Now, if we're at this point, it has to be assumed that none of the fields are blank. Furthermore, an e-mail address couldn't exist and be invalid at the same time (wishful thinking). So while it may

be evident that we're overwriting error messages, we really aren't.

```
//If any further errors, send response
if ( count ( $errors->get_error_codes() ) > 0 ) {
    $error_response->add(array(
        'what' => 'errors',
        'id' => $errors
    ));
    $error_response->send();
    exit;
}
```

Creating the User

The data validation portion is now finished. If there were any errors, we wouldn't be at this point in our code execution. Let's now assume everything is fine and create a new user.

```
//Everything has been validated, proceed with creating the user
//Create the user
$user_pass = wp_generate_password();
$user = array(
    'user_login' => $username,
    'user_pass' => $user_pass,
    'first_name' => $firstname,
    'last_name' => $lastname,
    'user_email' => $email
);
$user_id = wp_insert_user( $user );
```

A new password is generated (via `wp_generate_password`) and a `$user` array is created with all of our validated form inputs. Finally, we pass the

`$user` array to `wp_insert_user` to complete the user registration.

Now, we are at a fork in the road. For example purposes, we'll just be sending out the standard WordPress e-mail that sends a user their new username and password. However, you could choose to not use the next code snippet and instead write your own e-mail message (using `wp_mail`).

```
/*Send e-mail to admin and new user -  
You could create your own e-mail instead of using this  
function*/  
wp_new_user_notification( $user_id, $user_pass );
```

Sending the Response

If we've made it down this far in the Ajax processor, the user has successfully been created and has been sent an e-mail with their credentials. Now it's time to return an output and show this to the user.

```
//Send back a response  
$success_response->add(array(  
    'what' => 'object',  
    'data' => 'User registration  
successful. Please check your e-mail.'  
));  
$success_response->send();  
exit;  
} //end ajax_process_registration
```

Ajax Registration Form Conclusion

Please use the form below to register.

First name:

Last name:

Desired Username:

E-mail Address:

Username is invalid.
E-mail address is invalid.

Figure 24. Ajax Registration Form

Within this chapter you learned how to create a WordPress registration form that works via Ajax.

The data validation and user creation are all done behind-the-scenes without a page refresh. The

user is alerted instantly when there is a validation error.

The applications for this technique are numerous. For example, I have used a variant of it to create affiliate accounts for one of my products.

I've also used another variant to send data to PayPal and register a user when the purchase completes.

You could also customize the form to accept other input fields, such as a website URL, Twitter handler, or custom data such as a country.

The best part of this form is (in my opinion), it still works if you have user registrations disabled on your site. While some may view this as a downside for this technique, I rather enjoy the customization that can be used over the standard registration form.

Now You Begin Your Own
Journey

Now You Begin Your Own Journey

First, I'd like to thank you "so" much for taking the time to read this book. It is my hope that you have gotten something out of it.

So what now?

Don't let your journey stop. I would highly recommend a visit to the book's website, <http://www.wpajax.com>.

Also, check out the following links that are considered extensions of the book.

- <http://www.wpajax.com/code> - For the downloadable code mentioned in this book.
- <http://www.wpajax.com/links> - Links to functions, techniques, and other helpful resources that have been mentioned in the book.
- <http://www.wpajax.com/testimonials> - Leave your feedback so others can see how this book has helped you.

Once again, you are awesome for putting up with me throughout this journey. Thanks so much.

The people that made this free release possible

For those who have purchased this book in e-book format, a big “Thanks!” goes out to you.

The following companies have also made the free release possible by advertising for the book and/or supporting me while writing the book:

VG Universe Design



[VG Universe Design](#) is a Web & Graphic Design Studio owned and operated by this book’s designer, Vivien Anayian. Her work speaks for itself, and she was instrumental in this book’s public release.

WebDesign.com



[WebDesign.com](#) provides premium, professional web development training to web designers, developers, and WordPress website owners. The [WebDesign.com](#) Training Library currently holds over 230 hours of premium training developed by seasoned experts in their field, with approximately 20 hours of new training material added each month.

WebDevStudios



[WebDevStudios](#) is a website development company specializing in WordPress. We have a staff of developers and designers who live and breathe WordPress and pride ourselves in being WordPress experts. We have the resources to support any client from a single blog to a WordPress Multisite network with thousands of sites to a BuddyPress site with thousands of members.

WPCandy



[WPCandy](#) is an unofficial community of WordPress users and professionals. They are best known for their solid WordPress reporting and also publish a growing weekly podcast and regular community features. If you love WordPress and want to stay plugged in to the community, [WPCandy](#) is the place to be.

“This book answered all of my Ajax questions and some I didn’t even know I had!”

— RYAN HELLYER pixopoint.com

“Reading the book is actually kind of fun and not boring for a second. The writing style is casual yet accurate, a bit like attending a very detailed and sharp conference, where you get a glimpse of the author’s personality as well as a bit of his knowledge.”

— OZH RICHARD planetozh.com

“Ronald ingeniously shares his knowledge from his personal and unique experiences with Ajax, PHP and jQuery along with his witty and often humorous writing style that makes very technical tasks as easy as possible on the brain.”

— ROBYN-DALE SAMUDA bloggingpro.com

RONALD HUERECA (ronalfy.com) has worked with WordPress since 2006 and has released several WordPress plugins, his most popular being Ajax Edit Comments (ajaxeditcomments.com).

His education includes a Master’s in Business Administration and a degree in Electronics Engineering Technology.

WORDPRESS & AJAX — wpajax.com

An in-depth guide on using Ajax with WordPress