Table of Contents

WordPress The Right Way

This book is a condensed resource of best practices for and by WordPress developers, intended to fast track developers past common mistakes and painful problems.

This is a living document and will continue to be updated with more helpful information and examples as they become available.

How to Contribute

You can contribute on GitHub. Changes will be pushed to Gitbook.io automatically when the main repository changes.

Editing the book can be done either by updating the markdown files with a text editor, or opening the repository in the Gitbook desktop app. The desktop app will give you a live preview option.

License

Getting Started

Basic PHP

It's assumed that you have a basic knowledge of PHP. This will include a knowledge of:

- functions
- arrays
- variables
- loops and conditionals
- classes and objects
- class inheritance
- polymorphism
- POST and GET
- variable scope

If you don't have a good grasp of those concepts, you should make sure you have a firm understanding before continuing.

It's also assumed you have a code editor that has PHP syntax highlighting, although these will be beneficial:

- Auto Indenting
- Auto-completion
- Brace matching
- Syntax checking

Local Development Environments

It's important to have a local development environment. Gone are the old days of changing a PHP file then updating it on the live server and hoping for the best.

With a local environment, you can work faster, no more uploading and downloading files, being at the mercy of a dodgy internet connection, or waiting for pages to load from the open web. With a local server stack you can work on a train in a tunnel with no wifi or phone signal, and test your work before deploying it to the live server.

Here are a few options for setting up a local development environment. They fall into two categories:

- Virtual Machines
- Native Server Stacks

The first type of environment usually involves projects such as Vagrant, and gives you a standardised consistent virtual machine to work with.

The second, installs the server software directly into your operating system. There are various tools that make this easy, but your environment will be unique and more difficult to debug. These are sometimes called LAMP stacks, which stands for Linux Apache MySQL PHP.

IIS

Microsoft Internet Information Services is the server software that powers Windows based servers. Variants of it come with Windows if you install the appropriate components, but knowledge of IIS setup in the WordPress community is rare. Most remote servers run an Apache or Nginx setup, and developer knowledge is geared in that direction.

IIS is not the easiest route to take.

Version Control

A vital part of working in teams and contributing is version control. Version control systems track changes over time and allow developers to collaborate and undo changes.

Git

Created by Linus Torvalds the creator of Linux, Git is a popular decentralised system, if you've ever been on GitHub, you've encountered git.

Subversion

Also known as svn, this is a centralised version control system, used for the plugin and theme repositories on WordPress.org

Code Style Guide

Clean Code

It's important to keep code readable and maintainable. This prevents small but critical errors from becoming hidden in your code, while making whole classes of bugs incredibly obvious (missing closing braces are easy to spot when you indent consistently).

While it's best to use the same standard as everybody else, if you're more comfortable using a PSR standard, then use that. If you do though, do it consistently.

Indenting

Indenting in WordPress is done using tabs, representing 4 spaces visually. Indenting is important for readable code, and each statement should be on its own line. Without indenting, it becomes very difficult to understand what's happening, and mistakes are easier to make. This also makes support requests on the forums and stack exchange difficult to answer.

A good editor will auto-indent for you, most can re-indent a file if you've older code that needs fixing.

A good way to ensure that all members on a team are using the same styles is to use Editor Config. It contains plugins for different editors, so everyone can use their favorite editor.

For instance, the following `.editorconfig` file enforces the above rule, indentation as tabs of width 4 spaces.

```
[*.php]
indent_style = tabs
indent_size = 4
```

PHP Tag Spam

The `<?php` and `?>` tags should be used sparingly. For example:

```
<?php while( have_posts() ) { ?>
    <?php the_post(); ?>
    <?php the_title(); ?>
    <strong><?php the_date(); ?></strong>
    <?php the_content(); ?>
<?php } ?>
```

Would be easier to read as:

```
<?php
while( have_posts() ) {
    the_post();
    the_title();
    ?>
    <strong><?php the_date(); ?></strong>
    <?php
    the_content();
} ?>
```

A good guideline is to calculate what needs to be displayed, then display it all in one go rather than mixing the two.

Linting

A lot of editors support or have built in syntax checkers. These are called Linters. When using a good editor, syntax errors are highlighted or pointed out.

For example, in PHPStorm, a syntax error is given a red underline.

Coding Standards

WordPress follows a set of coding standards. These differ from the PSR standards. For example, WordPress uses tabs rather than spaces, and places the opening bracket on the same line.

The WordPress Contributor Handbook covers the coding standards in more details. Click below to read more:

- HTML Coding Standards
- PHP Coding Standards
- JavaScript Coding Standards
- CSS Coding Standards

PHP Code Sniffer & PHP CS Fixer

PHP Code Sniffer is a tool that finds violations of the coding standard. Many editors integrate support, including support for a second tool that fixes those violations automatically.

To use this, you will need the WordPress Coding Standards definition.

Debugging

When developing for WordPress, it's important that your code works, but when it fails it can be like a needle in a haystack. It doesn't need to be that way.

This chapter covers:

- How to find out what errors have occurred
- How to debug the issue
- Plugins and tools to make your life easier
- Features in WordPress that make debugging easier
- How to prevent problems from occurring to begin with and easy automated tools to catch mistakes for you

But before you continue, a word on White Screens of Death

White Screens of Death

A common issue with new WordPress developers is the white screen of death. This happens when a fatal error occurs in PHP. Many new developers respond to this by making changes and hoping the problem goes away, but there are better ways of dealing with this.

When an error occurs in PHP, it gets logged somewhere, and you can find out what went wrong and where.

A good starting point for developers is to enable `WP_DEBUG`.

Error Logging

There are several kinds of error logging, but the most basic are:

- Displaying errors on the frontend
- Writing errors to a log file
- Not displaying anything at all

In a production/live environment, you want to write errors to a log file.

Warnings vs Errors

Depending on how PHP is configured, warnings will also be shown. A warning is something that does not stop PHP from running but indicates a problem might have occurred. For example:

```
$my_array = array(
    'alice' => 5,
    'bob' => 6
);
echo $my_array['eve'];
```

Here, I am echoing the 'eve' entry in `$my_array`, but there is no such entry. PHP responds by creating an empty value and logging a warning. Warnings are indicators of bugs and mistakes.

PHP Error Reporting

Depending on what was defined in your `php.ini`, PHP will have an error reporting level. Everything below that level will be ignored or considered a warning. Everything above it will be considered an error. This can vary from server to server.

The `@` operator

Never use the `@` operator. It's used to hide errors and warnings in code, but it doesn't do what people expect it to do.

`@` works by setting the error reporting level on a command so that no error is logged. It doesn't prevent the error from happening, which is what people expect it to do. This can mean fatal errors are not caught or logged. Avoid using the `@` operator, and treat all instances of it with suspicion.

Handling Errors

While using the Core APIs, it's a good idea to check return values. For example, when creating a post, if something goes wrong you should be able to handle that outcome. Not handling errors and failures can lead to unstable code and unpredictable behavior.

Return values

Many functions return error and success values. You should always check these values after making a call. For example `get_post_meta` returns a custom field value, but if that custom field/post meta does not exist, it returns an error value.

Different APIs return different error values, and can include:

- `null` values
- false
- `WP_Error` objects

WordPress API calls at the time of writing do not throw exceptions. However if you hook into actions such as `save_post` and throw an exception, it may not be caught due to this expectation, so do not throw exceptions unless you're sure you know what you're doing.

`WP_Error`

The `WP_Error` object is a catch all error message object returned by some APIs. It has internal storage for multiple error messages and error codes.

`is_wp_error`

This is a helpful method to simplify error checking. It checks if a returned value was a `WP_Error` object, and also checks for a handful of other error values. It returns a true or false value, allowing checks such as these:

This is a helpful method to simplify error checking. It checks if a returned value was a `WP_Error` object, but does not check for other error values. It's shorthand for `if ( get_class( $variable ) == 'WP_Error' )`. For example:

```
if ( !is_wp_error( $value ) ) {
    // do things
} else {
    // display a warning to the user and abort
}
```

While this is a useful function, remember, not every API returns the same error value, and you should check first.

Tools

Debugging tools fall into two categories:

- Tools to diagnose issues when they arise and reveal problems
- Tools that prevent mistakes and errors from ever happening to begin with

The age old adage still applies: **prevention is better than cure**

Debugging Tools / Plugins

Installing the plugin Developer from the WordPress repository will give you quick access to a broad range of debugging tools. The following debugging plugins are quite useful:

- Log Deprecated Notices Logs usage of deprecated functions.
- Debug Bar Provides an interface for debugging PHP Notices/Warnings/Errors, reviewing SQL Queries, analysing caching behaviour and much more. It's also extendable with plugins.
- Debug Console The Debug Console for example is really useful.
- Query Monitor View debugging and performance information on database queries, hooks, conditionals, HTTP requests, redirects and more.

Xdebug and Remote Debugging

The Xdebug PHP Extension allows for enhanced debugging, function and method tracing, and profiling of PHP applications. This is installed with VVV and can be turned on/off.

With PHPStorm, you can install a browser extension to access Xdebug (or Zend Debugger) from within the IDE.

Rather than manually adding `var_dump` statements and reloading the page, you can add a breakpoint anywhere in your PHP code, execution will stop and you can see a stack trace, inspect (and modify) the values of all variables and objects or manually evaluate (test) a PHP expression.

With zero-configuration debugging (controlled via cookies and bookmarklets) you don't need to add `?XDEBUG_SESSION_START` to your URLs and you can also debug HTTP post requests.

PHP Debuggers

- DBG - PHP Debugger and Profiler

Browser Web Inspectors

- Chrome DevTools for Google Chrome
- Firebug for Mozilla Firefox
- F12 developer tools for Internet Explorer
- Opera Dragonfly for Opera

Prevention

There are a number of tools dedicated to analysing code and catching semantic mistakes, or pointing out problems in code.

PHP Mess Detector for example, will highlight long variable names, npath and cyclomatic complexity, classes that are too large, unused variables, and other problems. SCheck is a tool provided by Facebook, and performs similar checks, such as finding dead statements and unused classes.

If you can't type hint, you can make use of a tool such as phantm to infer types and find clashes. Many others exist though, and integrate with your editor/IDE, so look around

Constants of `wp-config.php`

Currently there are several PHP constants on the `wp-config.php` that will allow you to improve you WordPress code and help you debug.

`WP_DEBUG`

This is an Option included in WordPress version 2.3.1.

By default this will be set to `false` which will prevent warnings and errors from been shown, but **all WordPress developers should have this option active**.

Activates the Logs

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

Deactivates the Logs

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

*Check that the values must be **bool** instead of **string***

A minor patch later the on Wordpress version 2.3.2, the system allowed us to have a more granular control over the Database error logs.

Later on in the version 2.5, WordPress raised the error reporting level to E_ALL, that will allow to see logs for Notices and Deprecation messages.

Notes:

If you have this option turned on, you might encounter problems with AJAX requests, this problem is related to Notices been printed on the output of the AJAX response, that **will break XML and JSON**.

`WP_DEBUG_LOG`

When you use `WP_DEBUG` set to `true` you have access to this constant, and this will allow you to log your notices and warnings to a file.

`WP_DEBUG_DISPLAY`

When you use `WP_DEBUG` set to `true` you have access to this constant, with it you can choose to display or not the notices and warnings on the screen.

Note:

If these variables don't produce the output you are expecting check out the Codex Section about ways to setup your logging.

`SCRIPT_DEBUG`

When you have a WordPress plugin or theme that is including the Minified version of your CSS or JavaScript files by default you are doing it wrong!

Following the WordPress idea of creating a file for development and its minified version is very good and you should have both files in your plugin, and based on this variable you will enqueue one or the other.

By default this constant will be set to `false`, and if you want to be able to debug CSS or JavaScript files from WordPress you should turn it to `true`.

Activates the Logs

```
define( 'SCRIPT_DEBUG', true );
```

*Check that the values must be **bool** instead of **string***

WordPress default files `wp-includes` and `wp-admin` will be set to its development version if set to `true`.

`CONCATENATE_SCRIPTS`

On your WordPress administration you will have all your JavaScript files concatenated in to one single request based on the dependencies and priority of enqueue.

To remove this feature all around you can set this constant to `false`.

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

`SAVEQUERIES`

When you are dealing with the database you might want to save your queries so that you can debug what is happening inside of your plugin or theme.

Make `$wpdb` save Queries

```
define( 'SAVEQUERIES', true );
```

***Note:** this will slowdown your WordPress*

Core

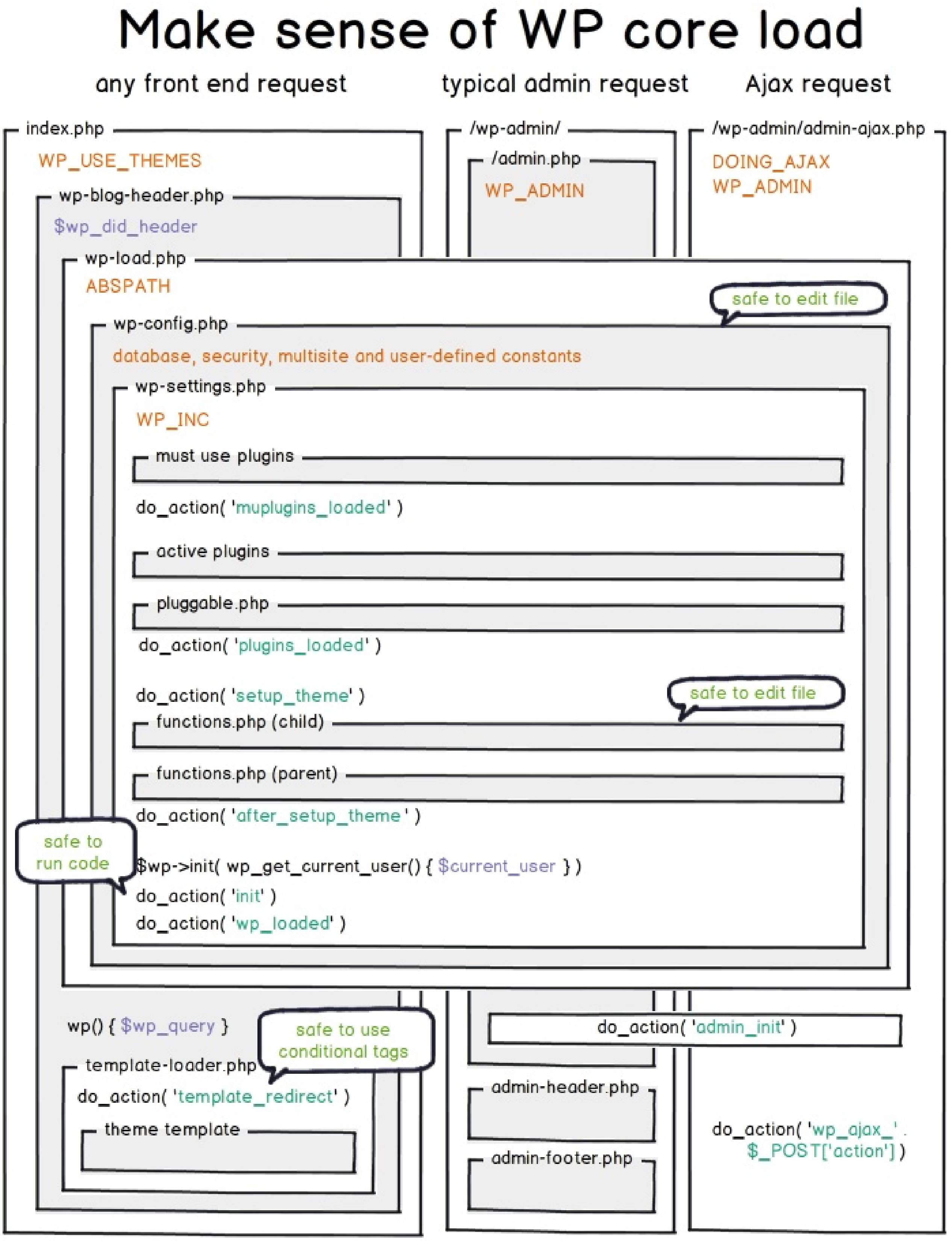
WordPress core is the code that powers WordPress itself. It is what you get when downloading WordPress from wordpress.org, minus the themes and plugins.

Load Process

At the most basic, the WordPress core loading follows this pattern:

- Load MU plugins
- Load Activated plugins
- load theme functions.php
- Run init hook
- Run main query
- Load template

Administration and AJAX requests follow a similar but lighter process. This diagram covers the specifics:



by Rarst.net CC-BY-SA

Deregistering jQuery

Many plugin and theme developers attempt to unregister the jQuery that comes with core, and add their own copy, normally the jQuery on the Google CDN. Do not do this, it can cause compatability issues.

Instead use the copy of jQuery that comes with WordPress and aim for the version used in the latest WordPress when testing. This ensures maximum compatability across plugins.

Modifying Core

It's tempting to modify parts of Core to remove or add things, but this must never be done. When WordPress updates, all your changes will be lost.

Instead, use Hooks/Actions and Filters to modify Core behaviour.

Further Reading

- Making Sense of Core Load

Data

There are multiple data types in WordPress, but the basic data types stored in the database are:

- Post
- Comments
- Terms
- User
- Blogs
- Network
- Links
- Options
- Site Options

Some of these data types can have additional data attached to them in the form of meta data, some of them have types and statuses.

There are also roles and capabilities, which are not considered content and apply exclusively to users.

Meta

Meta data is data with a key/name and a value, attached to another piece of data. Some people will know them as custom fields. Others will know them as user meta.

Post Meta (or Custom fields) are normally shown in the post edit screen, but if the post meta's key/name begins with an underscore, that field is hidden. This way features such as featured images can be built with their own user interfaces.

Post types

Posts, pages, attachments, and menus are all different kinds of posts. You can also register your own post types, and these are referred to as custom post types. Remember to flush permalinks (manually: Dashboard > Settings > Permalinks > Save Changes or programmatically via flush_rewrite_rules) if you modify or add a post type. If you do not do this, some links will generate 404 errors.

*Warning: Developers sometimes attempt to work around the rewrite rules by flushing rewrite rules on the `init` action using the `flush_rewrite_rules` , but this is a mistake. It can lead to unexpected behaviours, and has a large negative performance impact. Rewrite rules are expensive to build.

Default Post Types

The default post types are as follows:

- Post (`post`)
- Page (`page`)
- Attachment (`attachment`)
- Revision (`revision`)
- Navigation menu (`nav_menu_item`)

Menus

Menu items are stored as `nav_menu_item` posts, however, the menu itself is a term in a custom taxonomy that contains `nav_menu_item` posts.

Revisions

A post of type `revision` , revisions are historical copies of posts, and are tied to their original post via the post_parent field. To get a posts revisions, grab all its children of type `revision` . If a database is growing very large or a particular post/page is frequently/automatically edited, you can limit the number of revisions stored.

Uploaded Files & Images

When you upload a file, WordPress does not reference the image or file using its URL, it uses an attachment. Attachments are posts of type `attachment` , and are referred to by their post ID. For example, when you set a featured image on a post, it stores your chosen image's post ID in that post's meta (`_thumbnail_id`).

If a file is uploaded whilst editing a post (rather than in the Media Library itself), it's `post_parent` field is set to that of the post.

WordPress generates and saves resized versions of the original at the time of upload for better performance. They can be cropped and manipulated independently and the dimensions and filenames are stored in the attachment postmeta.

The default image sizes are configured in Dashboard > Settings > Media. If you need more, you can use add_image_size and also control cropping. You can request a specific size of image using the attachment ID and the image size name (e.g. 'medium', 'large'.)

The Regenerate Thumbnails plugin is useful if you change sizes at a later date.

Comments

Comments have their own table, and are attached to a post. Comments are not a type of post however, but they are capable of storing meta data. This is rarely used by developers but allows for interesting things.

Terms and Taxonomies

A taxonomy is a way of categorising or organising things. Items are organised using terms in that taxonomy.

For example, yellow is a term in the colour taxonomy. Big and small are both terms in the size taxonomy.

The Tags and categories that come with WordPress are both taxonomies. Individual tags and categories are called terms.

You can register your own taxonomies, but remember to flush permalinks (see *Post Types* above) if you make changes.

Taxonomy terms are tied to Object IDs, where an object ID can be any kind of data. This includes posts, users, or comments. These IDs are normally post IDs, but this is purely convention. There is nothing preventing a user or a comment taxonomy. A user taxonomy would be useful for grouping users into locations or job roles.

Options

Options are stored as key value pairs in their own table. Some options have an autoload flag set and are loaded on every page load to reduce the number of queries.

Transients

Transients are stored as options and are used to cache things temporarily

Object Cache

By default WordPress will use in memory caching that does not persist between page loads. There are plugins available that extend this to use APC or Memcache amongst others.

Data Overview

Here's a table showing the full spectrum of data types in WordPress that are stored in the database:

	Post	**Comments**	**Term**	**User**	**Bl**
Description	Content, e.g. articles	Commentary on a post	A type of objects, for classifying	Users and Authors	A webs
Supported	Yes	Yes	Yes	Yes	Yes
Meta	Custom fields	Comment meta	Planned	User Meta	Option
Meta Access	`get_post_meta`	`get_comment_meta`	Planned	get_user_meta	get_op
Type	Post type	Comment type	Taxonomy	Roles and Capabilities	No
Type registration	`register_post_type`	defined on use	`register_taxonomy`	`add_role` `add_cap`	n/a
Taxonomy UI?	Yes	No	Yes	No	No
Default Types	post page attachment nav_menu nav_menu_item	none pingback trackback	cat tag	admin editor author contributor subscriber	n/a
Query Class	`WP_Query`	`WP_Comment_Query`	`get_terms`	`WP_User_Query`	`wp_get`
Has Archives	Yes	No	Per blog	Yes	No
Has Widget	Recent Posts	Recent Comments	Categories and Tags	No	No
Data Availability	Per blog	Per blog	Per blog	Per install	Per ne
Set current	`setup_postdata`	n/a	n/a	n/a	switch_

Database Table	`wp_posts`	`wp_comments`	`wp_terms`
			`wp_term_relationships`
			`wp_term_taxonomy`
				`wp_users`	`wp_blo`

Meta Table	`wp_postmeta`	`wp_commentmeta`	Planned	`wp_usermeta`	`wp_opt`

Queries

This chapter talks about several kinds of query. Post queries, taxonomy queries, comment queries, user queries, and general SQL queries.

Whenever possible, use the query APIs that WordPress provides, rather than directly calling the database. This allows the internal cache system to speed up your queries, and for caching plugins to help out.

Not using the WordPress APIs to perform queries means that 3rd party plugins are unable to intercept and modify requests, leading to compatibility issues, and broken or incomplete functionality.

Query Limits and Performance

Some queries are more expansive than others, they simply do more work and don't scale. No amount of MySQL optimisation will fix them. For example, complex meta queries are more expensive.

One issue that most developers don't realise is scale. For example, you are listing terms in a custom taxonomy in a dropdown, and you have 5 or 10 terms. In that example the query will be fast, however, if 10,000 terms are added 6 months later, that dropdown is going to take a very long time to generate.

So always add limits to your queries, even if you don't think they're needed. Place an unrealistically high number you never expect to hit them, e.g. 100 or 1000.

Post Queries

Post queries retrieve posts from the database so that they can be processed or displayed on the frontend. This section covers some vital concepts, and methods of generating these queries.

The Main Loop

Every page displayed by WordPress has a main query. This query grabs posts from the database, and is used to determine what template should be loaded.

Once that template is loaded, the main loop begins, allowing the theme to display the posts found by the main query. Here is an example main loop:

```
if ( have_posts() ) {
    while ( have_posts() ) {
        the_post();
        // display post
    }
} else {
    // no posts were found
}
```

The Main Query and Query Variables

The main query is created using the URL, and is represented by a `WP_Query` object.

This object is told what to fetch using Query Variables. These values are passed into the query object at the start, and must be part of a list of valid query variables.

For example, the query variable 'p' is used to fetch a specific post type, e.g.

```
$posts = get_posts( 'p=12' );
```

Fetches the post with ID 12. The full list of options are available on the `WP_Query` codex entry.

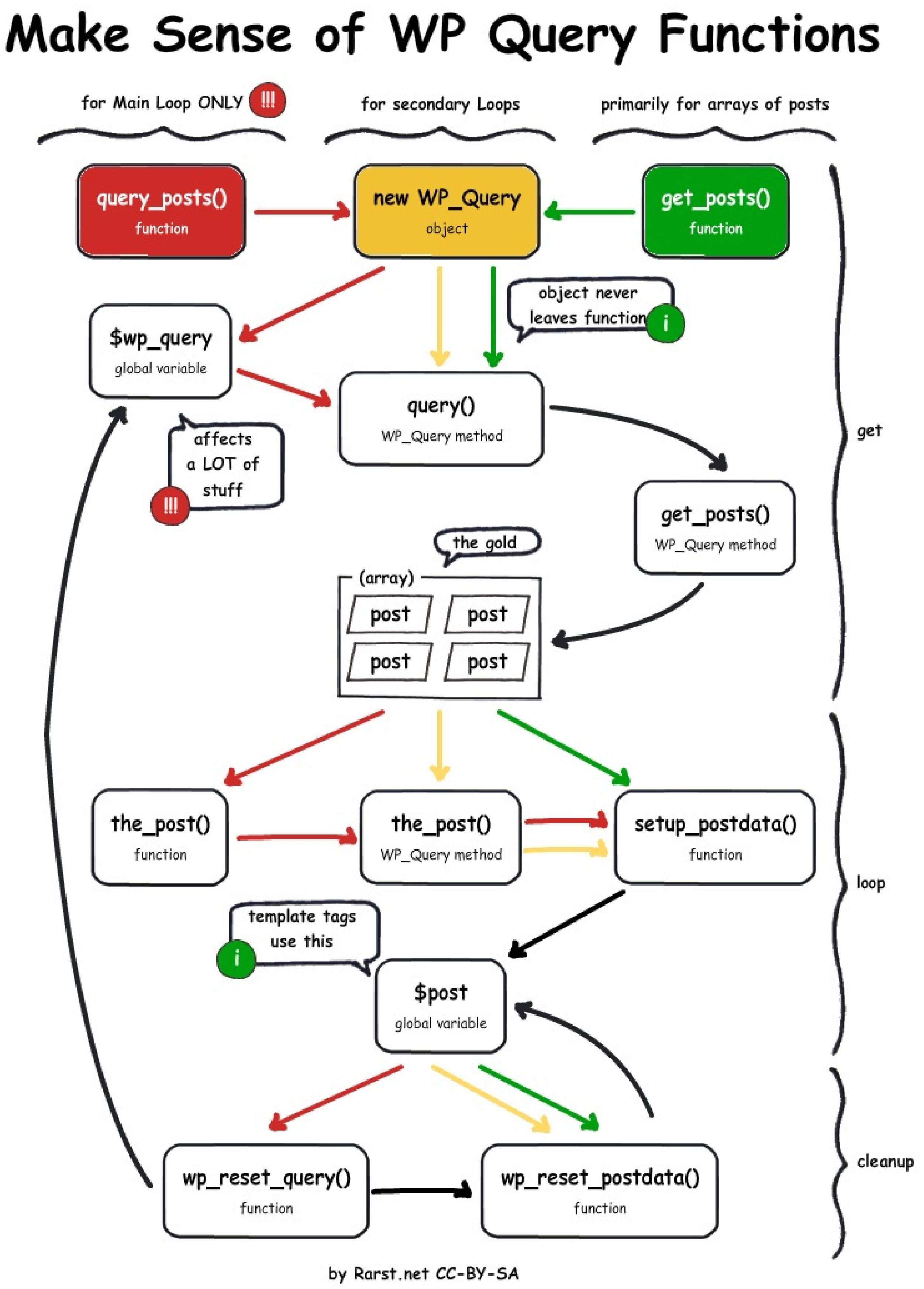
Making a Query

To retrieve posts from the Database, you need to make a post query. All methods of getting posts are layers on top of the `WP_Query` object.

There are 3 ways to do this:

- `WP_Query`
- `get_posts`
- `query_posts`

This diagram explains what happens in each method:



WP_Query

```
$query = new WP_Query( $arguments );
```

All post queries are wrappers around `WP_Query` objects. A `WP_Query` object represents a query, e.g. the main query, and has helpful methods such as:

```
$query->have_posts();
$query->the_post();
```

etc. The functions `have_posts();` and `the_post();` found in most themes are wrappers around the main query object:

```
function have_posts() {
    global $wp_query;

    return $wp_query->have_posts();
}
```

`get_posts`

```
$posts = get_posts( $arguments );
```

`get_posts` is similar to `WP_Query`, and takes the same arguments, but it returns an array containing the requested posts in full. You shouldn't use `get_posts` if you're intending to create a post loop.

While `get_posts` is conceptually simpler than `WP_Query` for novice programmers to understand, it does have a downside. `get_posts` doesn't make extensive use of the object cache in the way that `WP_Query` does, and may not be as performant.

Don't use `query_posts`

`query_posts` is an overly simplistic and problematic way to modify the main query of a page by replacing it with new instance of the query.

It is inefficient (re-runs SQL queries) and will outright fail in some circumstances (especially often when dealing with posts pagination). Any modern WordPress code should use more reliable methods, such as making use of the `pre_get_posts` hook, for this purpose. Do not use `query_posts()`.

Meta Queries and Performance

When performing a query involving meta keys, there can be performance issues. This is because there is no index on the post meta tables. As a result post meta queries have the potential to be very expensive. Queries involving both post meta keys and post meta values can be even more expensive.

`NOT IN` Queries

Queries looking for posts that are not in a category or don't have a post meta key can be very expensive, and should be avoided. The nature of the query means that they are expensive, as the database has to figure out which posts do have the term/meta key, then subtract those results from the full list of posts. These queries don't scale and are resource intensive.

Cleaning up after Queries

`wp_reset_postdata`

When using `WP_Query` or `get_posts`, you may set the current post object, using `the_post` or `setup_postdata`. If you do, you need to clean up after yourself when you finish your while loop. Do this by calling `wp_reset_postdata`.

A common mistake is to call `wp_reset_postdata` after the if statement. This is incorrect, as the post hasn't changed if the if statement is false, leading to potentially unexpected behavior. Always call the function before the closing brace, not after, e.g.

```
if ( $q->have_posts() ) {
    while( $q->have_posts() ) {
        $q->the_post();
    }
    wp_reset_postdata();
}
```

`wp_reset_query`

When you call `query_posts`, you will need to restore the main query after you've done your work. Failure to do so can lead to a large number of issues and unexpected behavior. You can do this with `wp_reset_query`. Always do this after calling `query_posts`, and only do it when necessary.

The `pre_get_posts` Filter

If you need to change the main query and display something else on the page, you should use the `pre_get_posts` filter.

Many people will want to use this for things such as removing posts from an archive, changing the post types for search, excluding categories, and others

Here is the Codex example for searching only for posts:

```
function search_filter($query) {
    if ( !is_admin() && $query->is_main_query() ) {
        if ($query->is_search) {
            $query->set('post_type', 'post');
        }
    }
}

add_action( 'pre_get_posts', 'search_filter' );
```

These filters can go in a themes `functions.php`, or in a plugin.

Further Reading

- You don't know query, a talk by Andrew Nacin
- When you should use WP_Query vs query_posts, Andrei Savchenko/Rarst

Taxonomy and Term Queries

When dealing with taxonomies (including post categories and tags), it's safer to rely on the generic APIs rather than the legacy helper APIs. These include:

- `get_taxonomies`
- `get_terms`
- `get_term_by`
- `get_taxonomy`
- `wp_get_object_terms`
- `wp_set_object_terms`

It's easier to learn one set of APIs, and think of categories and tags as just another taxonomy, rather than mixing and matching older functions such as `get_category` etc.

Comment Queries

You can retrieve comments using the `WP_Comment_Query` class. When WordPress tries to load a single post, it constructs one of these objects in order to retrieve the number of comments it has, ready for when it's displayed later on.

This is a basic comment query:

```
$args = array(
   // args here
);

// The Query
$comments_query = new WP_Comment_Query();
$comments = $comments_query->query( $args );

// Comment Loop
if ( $comments ) {
    foreach ( $comments as $comment ) {
        echo '<p>' . $comment->comment_content . '</p>';
    }
} else {
    echo 'No comments found.';
}
```

Comment queries can find comments of different types across multiple or single posts. Using a comment query can be faster than a raw SQL command thanks to the built cache system.

User Queries

Similar to comment queries, user queries can be used to find individual users, users with specific roles, and other parameters.

Here is a basic User query:

```
$args = array(
    //
);

// The Query
$user_query = new WP_User_Query( $args );

// User Loop
if ( ! empty( $user_query->results ) ) {
    foreach ( $user_query->results as $user ) {
        echo '<p>' . $user->display_name . '</p>';
    }
} else {
    echo 'No users found.';
}
```

Note that the user query class may not be available yet if your code runs very early.

SQL

WPDB

It can be tempting for the uninformed to resort to a raw SQL query to grab posts. Only do this as a last resort.

But if you have to make an SQL query, use `WPDB` objects.

dbDelta and Table Creation

The `dbDelta` function examines the current table structure, compares it to the desired table structure, and either adds or modifies the table as necessary, so it can be very handy for updates.

The `dbDelta` function is rather picky, however. For instance:

- You must put each field on its own line in your SQL statement.
- You must have two spaces between the words `PRIMARY KEY` and the definition of your primary key.
- You must use the key word `KEY` rather than its synonym `INDEX` and you must include at least one KEY.
- You must not use any apostrophes or backticks around field names.
- `CREATE TABLE` must be capitalised.

With those caveats, here are the next lines in our function, which will actually create or update the table. You'll need to substitute your own table structure in the $sql variable.

Further Reading

- Creating Tables With Plugins - Codex

Routing

Common questions asked by new developers revolve around a misconception. They believe that single.php is what made WordPress load a single post, and talk about making WordPress load archive.php instead so that they can view multiple posts rather than an individual post.

That viewpoint is confusing, the truth is that such a viewpoint is completely upside down. The template does not determine the content. The content determines the template used.

This chapter will go into more depth regarding how WordPress breaks down a URL, creates a query, then figures out which template to load.

- An explanation of how rewrite rules generate a query, which loads a template, which displays a page
- Custom Query variables & routing
- Adding a rewrite rule
- Flushing rewrite rules
- Debugging rewrite rules
- Clashes & slugs

The Main Loop & Template Loading

- The main query is a WP_Query object
- The main loop is using that main query
- The main query is done before the template is even loaded
- The template is loaded based on what the main query is
- The main query is determined by parameters called query variables
- Which template is loaded when is shown on the template hierarchy diagram
- All templates are just custom ways of showing the main post loop

Where Query Variables Come From

- Mention that query variables come from the URL
- URLs are broken down using rewrite rules into query vars
- extra query vars can be added to any WordPress URL which is how searches work
- Query vars have a whitelist, and they're the same as the parameters passed into WP_Query

Rewrite Rules

- Rewrite rules are based on regular expressions
- Regular expressions map nice URLs on to uglier query var based URLs that can be parsed
- Rewrite rules have priority/order
- Rewrite rules are generated then stored in the database
- Rebuilding rewrite rules is expensive

Clashes, Slugs, & Debugging

- Slugs must be unique
- 1 URL can't be 2 things, there must be no ambiguity
- Showing different things depending on where the user has been before is terrible for caching
- Monkey rewrite tools plugin for debugging

Templates

Loading templates via `get_template_part`

When including templates in your theme, it's tempting to use code such as this:

```
include( 'customloop.php' );
```

However, doing this breaks support for child themes. Instead using `get_template_part` will do the job better, while giving extra flexibility. For example:

```
get_template_part( 'custom', 'loop' );
```

This way WordPress will attempt to load `custom-loop.php`. If the file does not exist, it will load `custom.php`, and if a child theme exists, it will load the child theme version of the file.

This allows fallback templates and specialised templates based on post meta and other data such as post type. For example:

```
get_template_part( 'loop', get_post_type() );
```

This will load `loop.php`, but if a custom version of the template exists for that post type, it will load that instead. e.g. `loop-page.php`

Internally, `get_template_part` uses the `locate_template` function. This function finds the appropriate file, and returns its name. This is useful for finding a template, without loading it.

`locate_template is also useful for plugin theming. For example:

```
// if the theme has a custom template for my plugin
if ( locate_template( 'mycustomplugin.php') != '' ) {
    // load the custom template for my plugin from the theme
    get_template_part( 'mycustomplugin.php' );
} else {
    // fallback to the plugins default theme
    include( 'defaulttemplates/mycustomplugin.php' );
}
```

If you wish to provide such a system in your plugin though, it's advised you use the `template_include` filter. Scroll down for a more in depth look at the `template_include` filter.

How Templates are Chosen, and The Template Hierarchy

- Notes on how a template is chosen using the main query. How templates are chosen and loaded, how child themes are involved. Show the template hierarchy diagram

Functions.php and Plugins

`functions.php` is a file in your theme that gets loaded prior to any templates. If your theme has non-template functionality, such as changing the length of excerpts, adding stylesheets and scripts, etc, this is where that code would go.

Because of the way functions.php is loaded, it can be considered a plugin, as there is no difference between `functions.php` and plugin development. However, there is a difference in how it's loaded.

If your theme registers post types and taxonomies, shortcodes, or widgets, this data is no longer available to the user when they change theme. This is a large problem for data portability, and can cause a persons site to become non-functional or broken.

Post types, taxonomies, shortcodes, and widgets, should be implemented in a separate plugin so that the users data remains portable, and their site is not broken when themes change. To do otherwise is irresponsible.

Loading Stylesheets

- enquing stylesheets properly

Templates and Plugins

- `template_include` filter

Forms

- Forms that submit to a separate standalone PHP file in your theme are bad. An example of how to handle a basic form submission on a page template

Virtual Pages

- For when you need a page/URL that doesn't have an associated post or archive, e.g. a shopping cart or an API endpoint.

Further Reading

- link to template diagram
- interactive template diagram

JavaScript

Javascript is the future of WordPress, but there are a number of things to keep in mind.

While there's a lot of things that should always be done, there are three approaches to using WordPress javascript the right way.

- **The Wrong Way** - Sending AJAX requests to files in your theme or page templates, then including them in your header with a manually coded tag
- **The Old Way** - Using the WP AJAX API for requests
- **The Best Way** - Building your admin UI in Javascript instead of PHP, and powering it with the REST API.

At the time of writing, the REST API and the content endpoints are the future, and admin UIs need to prepare for a fully JS powered admin UI. This means:

- Clean, portable, cachable Data APIs
- Enforced, and simplified built in security in your endpoints
- A standardised system to work in
- More secure interfaces by avoiding the need for escaping with Javascript templating and reactive libraries
- Faster admin screens whose sole job is to bootstrap the JS UI

While information on these are compiled, information is preserved below.

Registering and Enqueueing

WordPress comes with dependency management and enqueueing for JavaScript files. Don't use raw `<script>` tags to embed JavaScript.

JavaScript files should be registered. Registering makes the dependency manager aware of the script. To embed a script onto a page, it must be enqueued.

Let's register and enqueue a script.

```
// Use the wp_enqueue_scripts function for registering and enqueueing scripts on the front end.
add_action( 'wp_enqueue_scripts', 'register_and_enqueue_a_script' );
function register_and_enqueue_a_script() {
    // Register a script with a handle of `my-script`
    //  + that lives inside the theme folder,
    //  + which has a dependency on jQuery,
    //  + where the UNIX timestamp of the last file change gets used as version number
    //    to prevent hardcore caching in browsers - helps with updates and during dev
    //  + which gets loaded in the footer
    wp_register_script(
        'my-script',
        get_template_directory_uri().'/js/functions.js',
        array( 'jquery' ),
        filemtime( get_template_directory().'/js/functions.js' ),
        true
    );
    // Enqueue the script.
    wp_enqueue_script( 'my-script' );
}
```

Scripts should only be enqueued when necessary; wrap conditionals around `wp_enqueue_script()` calls appropriately.

When enqueueing javascript in the admin interface, use the `admin_enqueue_scripts` hook.

When adding scripts to the login screen, use the `login_enqueue_scripts` hook.

Localizing

Localizing a script allows you to pass variables from PHP into JS. This is typically used for internationalization of strings (hence localization), but there are plenty of other uses for this technique.

From a technical side, localizing a script means that there will be a new `<script>` tag added right before your registered script, that contains a *global* JavaScript object with the name you specified during localizing (the 2nd argument). This also means that if you add another script later on, that has this script as dependency, then you will be able to use the global object there as well. WordPress resolves chained dependencies just fine.

Let's localize a script.

```
add_action( 'wp_enqueue_scripts', 'register_localize_and_enqueue_a_script' );
function register_localize_and_enqueue_a_script() {
    wp_register_script(
        'my-script',
        get_template_directory_uri().'/js/functions.js',
        array( 'jquery' ),
        filemtime( get_template_directory().'/js/functions.js' ),
        true
    );
    wp_localize_script(
        'my-script',
        'scriptData',
        // This is the data, which gets sent in localized data to the script.
        array(
            'alertText' => 'Are you sure you want to do this?',
        )
    );
    wp_enqueue_script( 'my-script' );
}
```

In the javascript file, the data is available in the object name specified while localizing.

```
( function( $, plugin ) {
    alert( plugin.alertText );
} )( jQuery, scriptData || {} );
```

Deregister / Dequeueing

Scripts can be deregistered and dequeued via `wp_deregister_script()` and `wp_dequeue_script()` .

AJAX

WordPress offers an easy server-side endpoint for AJAX calls, located in `wp-admin/admin-ajax.php` .

Let's set up a server-side AJAX handler.

```
// Triggered for users that are logged in.
add_action( 'wp_ajax_create_new_post', 'wp_ajax_create_new_post_handler' );
// Triggered for users that are not logged in.
add_action( 'wp_ajax_nopriv_create_new_post', 'wp_ajax_create_new_post_handler' );

function wp_ajax_create_new_post_handler() {
    // This is unfiltered, not validated and non-sanitized data.
    // Prepare everything and trust no input
    $data = $_POST['data'];

    // Do things here.
    // For example: Insert or update a post
    $post_id = wp_insert_post( array(
        'post_title' => $data['title'],
    ) );

    // If everything worked out, pass in any data required for your JS callback.
    // In this example, wp_insert_post() returned the ID of the newly created post
    // This adds an `exit`/`die` by itself, so no need to call it.
    if ( ! is_wp_error( $post_id ) ) {
        wp_send_json_success( array(
            'post_id' => $post_id,
        ) );
    }

    // If something went wrong, the last part will be bypassed and this part can execute:
    wp_send_json_error( array(
        'post_id' => $post_id,
    ) );
}


add_action( 'wp_enqueue_scripts', 'register_localize_and_enqueue_a_script' );
function register_localize_and_enqueue_a_script() {
    wp_register_script(
        'my-script',
        get_template_directory_uri().'/js/functions.js',
        array( 'jquery' ),
        filemtime( get_template_directory().'/js/functions.js' ),
        true
    );
    // Send in localized data to the script.
    wp_localize_script(
        'my-script',
        'scriptData',
        array(
            'ajax_url' => admin_url( 'admin-ajax.php' ),
        )
    );
    wp_enqueue_script( 'my-script' );
}
```

And the accompanying JavaScript:

```
( function( $, plugin ) {
    $( document ).ready( function() {
        $.post(
            // Localized variable, see example below.
            plugin.ajax_url,
            {
                // The action name specified here triggers
                // the corresponding wp_ajax_* and wp_ajax_nopriv_* hooks server-side.
                action : 'create_new_post',
                // Wrap up any data required server-side in an object.
                data   : {
                    title : 'Hello World'
                }
            },
            function( response ) {
                // wp_send_json_success() sets the success property to true.
                if ( response.success ) {
                    // Any data that passed to wp_send_json_success() is available in the data property
                    alert( 'A post was created with an ID of ' + response.data.post_id );

                // wp_send_json_error() sets the success property to false.
                } else {
                    alert( 'There was a problem creating a new post.' );
                }
            }
        );
    } );
} )( jQuery, scriptData || {} );
```

`ajax_url` represents the admin AJAX endpoint, which is automatically defined in admin interface page loads, but not on the front-end.

Let's localize our script to include the admin URL:

```
add_action( 'wp_enqueue_scripts', 'register_localize_and_enqueue_a_script' );
function register_localize_and_enqueue_a_script() {
    wp_register_script( 'my-script', get_template_directory_uri() . '/js/functions.js', array( 'jquery' ) );
    // Send in localized data to the script.
    $data_for_script = array( 'ajax_url' => admin_url( 'admin-ajax.php' ) );
    wp_localize_script( 'my-script', 'scriptData', $data_for_script );
    wp_enqueue_script( 'my-script' );
}
```

The JavaScript side of WP AJAX

There are several ways to go on this. The most common is to use `$.ajax()`. Of course, there are shortcuts available like `$.post()` and `$.getJSON()`.

Here's the default example.

```
/*globals jQuery, $, scriptData */
( function( $, plugin ) {
    "use strict";

    // Alternate solution: jQuery.ajax()
    // One can use $.post(), $.getJSON() as well
    // I prefer defered loading & promises as shown above
    $.ajax( {
         url  : plugin.ajaxurl,
         data : {
            action      : plugin.action,
            _ajax_nonce : plugin._ajax_nonce,
            // WordPress JS-global
            // Only set in admin
            postType     : typenow,
         },
         beforeSend : function( d ) {
             console.log( 'Before send', d );
         }
    } )
        .done( function( response, textStatus, jqXHR ) {
            console.log( 'AJAX done', textStatus, jqXHR, jqXHR.getAllResponseHeaders() );
        } )
        .fail( function( jqXHR, textStatus, errorThrown ) {
            console.log( 'AJAX failed', jqXHR.getAllResponseHeaders(), textStatus, errorThrown );
        } )
        .then( function( jqXHR, textStatus, errorThrown ) {
            console.log( 'AJAX after finished', jqXHR, textStatus, errorThrown );
        } );
} )( jQuery, scriptData || {} );
```

Note that above example uses `_ajax_nonce` to verify the NONCE value, which you will have to set by yourself when localizing the script. Just add `'_ajax_nonce' => wp_create_nonce( "some_value" ),` to your data array. You can then add a referrer check to your PHP callback that looks like `check_ajax_referer( "some_value" )`.

AJAX on click

Actually it's pretty simple to execute an AJAX request when some clicks (or does some other user interaction) on some element. Just wrap up your `$.ajax()` (or similar) call. You can even add a delay like you might be used to.

```
$( '#' + plugin.element_name ).on( 'keyup', function( event ) {
    $.ajax( { ... etc ... } )
        .done( function( ... ) { etc }
        .fail( function( ... ) { etc }

} )
    .delay( 500 );
```

Multiple callbacks for a single AJAX request

You might come into a situation where multiple things have to happen after an AJAX request finished. Gladly jQuery returns an object, where you can attach all of your callbacks.

```
/*globals jQuery, $, scriptData */
( function( $, plugin ) {
    "use strict";

    // Alternate solution: jQuery.ajax()
    // One can use $.post(), $.getJSON() as well
    // I prefer defered loading & promises as shown above
    var request = $.ajax( {
         url  : plugin.ajaxurl,
         data : {
            action      : plugin.action,
            _ajax_nonce : plugin._ajax_nonce,
            // WordPress JS-global
            // Only set in admin
            postType     : typenow,
         },
         beforeSend : function( d ) {
             console.log( 'Before send', d );
         }
    } );

    request.done( function( response, textStatus, jqXHR ) {
        console.log( 'AJAX callback #1 executed' );
    } );

    request.done( function( response, textStatus, jqXHR ) {
        console.log( 'AJAX callback #2 executed' );
    } );

    request.done( function( response, textStatus, jqXHR ) {
        console.log( 'AJAX callback #3 executed' );
    } )
} )( jQuery, scriptData || {} );
```

Chaining callbacks

A common scenario (regarding how often it is needed and how easy it then is to hit the mine trap), is chaining callbacks when an AJAX request finished.

About the problem first:

> AJAX callback (A) executes AJAX Callback (B) doesn't know that it has to wait for (A) You can't see the problem in your local install as (A) is finished too fast.

The interesting question is how to wait until A is finished to then start B and its processing.

The answer is "deferred" loading and "promises", also known as "futures".

Here's an example:

```
( function( $, plugin ) {
    "use strict";

    $.when(
        $.ajax( {
            url :  pluginURl,
            data : { /* ... */ }
        } )
           .done( function( data ) {
                // 2nd call finished
           } )
           .fail( function( reason ) {
               console.info( reason );
           } );
    )
    // Again, you could leverage .done() as well. See jQuery docs.
    .then(
        // Success
        function( response ) {
            // Has been successful
            // In case of more then one request, both have to be successful
        },
        // Fail
        function( resons ) {
            // Has thrown an error
            // in case of multiple errors, it throws the first one
        },
    );
    //.then( /* and so on */ );
} )( jQuery, scriptData || {} );
```

Source: WordPress.StackExchange / Kaiser

JavaScript

Enqueing a script

For the widget form in the admin area

A quick note on how to do it, and a note on running the JS, so that it doesn't get ran on the html used to create new widget forms, only those in the sidebars on the right.

For the frontend

How to enqueue a widgets scripts and styles, but only if the widget is on the page

Events

Running code when:

A New Widget is Added, or Re-ordered

- Make use of the ajaxStop event to process javascript when a widget is added or re-ordered

```
jQuery( document ).ready( function( $ ) {
    function doWidgetStuff() {
        var found = $( '#widgets-right .mywidgetelement' );
        found.each( function( index, value ) {
          // process elements
        } );
    }

    window.counter = 1;

    doWidgetStuff();

    $( document ).ajaxStop( function() {
        doWidgetStuff();
    } );
} );
```

The widget form opens

- Some js to show how to do things when the form opens and closes

Further Reading

- Executing javascript when a widget is added

I18n

When talking about I18n here, we're going to talk about translation strings in user interfaces and on the frontend. For content in multiple languages or language pickers for users, you will need to install a plugin to provide the editing tools for posts and other content types.

At any point, you can manually set the language WordPress uses by user or overriding the `WPLANG` option. Older tutorials will recommend the `WP_LANG` constant, but this has been deprecated

However you will need to make sure the necessary language files are in place in your `wp-content/languages` folder before the change takes full effect.

Translation work falls under the Polyglots group at contributor days. If you're interested in translating WordPress Core, you should read the official translators handbook to find out how

Securing Language Files

Language files have 2 major attack vectors:

- Unescaped translation strings containing javascript tags
- n-plurals

Embedded Security Risks

It's important to use the escaping functions with the translation API to verify that dangerous content isn't inserted. It's possible to do by placing language files for a particular translation domain inside a WordPress install

You can save typing out `echo esc_html( __( '','' ) )` by using the helper functions:

- `esc_html__` instead of `__`
- `esc_attr__` instead of `__`
- etc

There are also an extended set of functions that simplify this further by adding `e` to the function name:

- `esc_html_e`
- `esc_attr_e`

These will output on their own, so an `echo` statement isn't needed.

Translation API Abuse

The `esc_html_e` helper functions are sometimes misused as a susbtitute for `echo esc_html`. Always use the second parameter that sets the translation domain. If you don't it could have unanticipated side effects as your strings are mistranslated:

```
// Good:
echo esc_html('date');
// Great:
esc_html_e( 'date', 'mytheme' );
// Bad:
esc_html_e( 'date' );
```

n-plurals

A relatively unknown part of the translation format is the `n-plurals` field. This determines the way plural forms work in a language for a particular file.

Because of its complexity, and for performance reasons, WordPress loads this field as a string, wraps it in a function, and passes the result to `eval`. Because of this, it's very easy to craft a language file with a primitive PHP shell.

The only way to mitigate this is code review/manual inspection.

Setting the Admin language

On installation WordPress asks you to select a language, but you may want to set different languages for the front end back end. For example a German website ran by an English speaker may want the admin area to be in their native language.

In order to do this, set your sites language to German using the `WP_LANG` constant mentioned earlier, and add this code to set the admin area language to english:

```
add_filter('locale', 'wpse27056_setLocale');
function wpse27056_setLocale($locale) {
    if ( is_admin() ) {
        return 'en_US';
    }

    return $locale;
}
```

Foreign Twitter Embeds

Sometimes oembeds come back in an unexpected foreign language, this is because the service being used is looking at your servers request and tracing it back to its origin to determine it's country. For example, an English website hosted on a German server may result in German twitter embeds.

Further Reading

- Different languages for front and back ends

Multisite

Grabbing Data From Another Blog in a Network

Getting data from another blog on the same multisite install can be done. Some people use SQL commands to do this, but this can be slow, and error prone.

Although it's an inherently expensive operation, you can make use of `switch_to_blog` and `restore_current_blog` to make it easier, while using the standard WordPress APIs.

```
switch_to_blog( $blog_id );
// Do something
restore_current_blog();
```

`restore_current_blog` undos the last call to `switch_to_blog` , but only by one step, the calls are not nestable, so always call `restore_current_blog` before calling `switch_to_blog` again.

Listing Blogs in a Network

Listing blogs in a network is possible, but it's an expensive thing to do.

It can be done using the `get_sites( $args)` function, available since version 4.6 of WordPress. The `get_sites( $args)` function accepts an array or query string of site query parameters specifying the kind of sites you are looking for. For versions 3.7 to 4.6 a similar function `wp_get_sites( $args )` was available.

See the codex entry for `get_sites` for more details.

Domain Mapping

Domain mapping allows a blog on a multisite install to serve from any domain name. This way a blog does not have to be a subdirectory of the main install, or a subdomain. The WordPress Default supports Domain Mapping without Alias. Add the Domain in the blog-settings to the blog of the Network administration area.

Often is it helpful - but not necessary, to set the `COOKIE_DOMAIN` constant to an empty string in your `wp-config.php` :

`define('COOKIE_DOMAIN', '');`

Otherwise WordPress will always set it to your network's `$current_site->domain` , which could cause issues in some situations.

WordPress Core hopes to provide Domain Alias Mapping in the future, but until then you can make use of one of the following plugins:

- Mercator - WordPress multisite domain mapping for the modern era.
- WordPress MU Domain Mapping - Map any blog/site on a WordPressMU or WordPress 3.X network to an external domain.

Testing

It's important that you test your code and your themes, but that takes time. Luckily there are tools and methods of simplifying and automating these things. This chapter is going to cover basic preventative testing, and tools to help catch bugs and things you may have missed.

WP Test & Theme Test Data

- Good for testing content
- It's a content export file
- Contains lots of posts and categories of varying types to test as many possible combinations as possible
- Useful for testing themes and unhandled scenarios such as posts without titles, giant nav menus, or very long tag names.

Theme review tester plugin

- Good for testing theme completion
- A plugin that runs several automated tests on the current theme
- Checks for things the theme review team checks for when submitting themes to wordpress.org
- Includes things such as comment forms, showing tags and categories, displaying author names, etc

Integration vs unit vs behavioural testing

- Good for testing code and as a development methodology
- Automated testing
- Explain the difference between the three
- Mention they're covered in more depth in sub-chapters

Unit Testing

Unit testing tests individual components. Each test runs in isolation, and tests only a single item, such as a function or method. If a unit test involves multiple interacting objects, then you have written an integration test.

For example, if I have this function:

```
function add( $a, $b ) {
  return $a + $b;
}
```

A unit test might check:

- If 5+5 = 10
- That 2+3 is not 7
- That 0+0 does not fail

Tools for Unit Testing

- PHPUnit
- PHPSpec

Helpful Projects and Further Reading

- John P Blochs WP Unit Test Starter project
- WP Mock
- Writing Unit Tests for WordPress

Behaviour Testing

Also known as Behaviour Driven Development, this kind of testing tests the entire stack. For example a behavioral test may start by visiting a webpage, clicking a button, and checking that an expected string was found.

BDD is good for testing business requirements. It generally falls into 2 types, story based testing, and code-based testing. An example of story based testing would be Behat, which uses a human readable format so that clients can read the tests in plain English (with support for other languages included).

A major benefit of these types of tests is that the tests themselves do not need to load the WordPress PHP environment. A test site can be put up on a server, and the tests can be pointed at the test site. Tools such as Behat then run as if they were a user controlling a browser (which is exactly how most Behat Mink tests work). This makes it one of the easiest ways to introduce testing, and the easiest to learn first

Tools for Behaviour Testing

- http://behat.org/Behat
- SpecBDD

Test Driven Development

Explain theory and idea behind TDD

WP_UnitTestCase

Explain the WP_UnitTestCase

Further Reading

- http://taylorlovett.com/2014/07/04/wp_unittestcase-the-hidden-api/

Servers And Deployment

Test Your Changes

Always test changes on a local environment before copying them to your production server (see *Getting Started > Local Development Environment*.)

Make sure the development server has error-reporting turned on so you catch anything that would be invisible on the live site.

Make sure your local environment is as similar to your production server as possible (see also *Migrations*.)

- are you running the same PHP version?
- do you have the same PHP.ini settings?
- do you have the same version of MySQL?
- do you have the same Apache or Nginx version & configuration?
- do you have the same version of WordPress with the same plugins enabled?

Consider using a staging server to help with this.

If you're copying a database from development to production (or vice-versa), you'll need to change the URLs. See the *Migrations* section in this chapter.

Use Version Control

If you use a version control or source code management system such as Git, you'll be able to roll back your changes when (not if) you make a mistake. You can 'push' your changes with a single command and updated files will first be copied to a temporary area before being deployed simultaneously. This avoids the site ever being left in a broken state if you have a slow connection.

A good server host provides SSH access, but if your hosting provider only allows SFTP access, consider using git-ftp, so you can minimise the time it takes to update the site and the chance of forgetting to upload any new files. You'll still benefit from version control locally or if you're working with other developers.

Avoid using file editors on control panels like CPanel.

Built-in Editors

WordPress has theme and plugin editors built into the admin area.

Avoid using them.

- The editor is a simple HTML textarea - you get none of the code highlighting, formatting or syntax checking of an IDE or basic text editor, it might seem quicker but it's also much easier to make mistakes.
- there's no version control, you don't get a list of what you changed and the only protection is your own backups (if you remembered to make any).
- A significant error might break WordPress in such a way that you can no longer access the editor itself.

The theme/plugin editor is also a potential security risk: if someone gains access to an administrator account they can edit sensitive files on the server.

You can turn off file editing completely by adding this line to wp-config.php

```
define( 'DISALLOW_FILE_EDIT', true );
```

WP CLI

WP CLI is a command line tool maintained by numerous experienced WordPress developers and core contributors. It's similar to Drupals Drush.

Deploying a New Install

To download WordPress into a folder on the command line, use this command:

```
wp core download
```

This will download the latest version of WordPress into the current directory. Next you'll need to create your `wp-config.php` :

```
wp core config --dbname=testing --dbuser=wp --dbpass=securepswd
```

Finally, run the install command to set up the database:

```
wp core install --url="example.com" --title="Example Site" --admin_user="exampleadmin" --admin_password="changeme" --
admin_email="example@example.com"
```

You should now have a fresh new WordPress install ready to log in to.

Multisite

If you want to create a multisite install, use the `wp core multisite-convert` command:

```
wp core multisite-convert --title="My New Network" --base="example.com"
```

Importing Content

You may have content you want to pre-add or migrate to your new install. For this WP CLI provides the content import and export commands. These commands accept or create standard wxr files, the same format used by the WordPress export and import plugin in the admin interface.

Use this command to import:

```
wp import content.wxr
```

Use this command to export:

```
wp export
```

Migrations

There are a number of things to take note of when moving sites from server to server, and when changing their URLs.

Since server moves and domain changes are a large topic, we're going to cover only the most important things.

Imports and Exports

When performing imports and exports, there are a lot of pitfalls as your site increases in size. To avoid problems, do the following:

- **Use WP CLI** to run imports and exports. The admin UI is limited by the PHP time limits, if your import or export doesn't finish within the available time, it can fail. Running in a terminal using WP CLI gives you unlimited time to do it
- **Ask the exporter to generate in 5MB chunks**. This reduces the memory requirements of each individual import, and gives a lot more flexibility
- **Disable image resizing**. This speeds up importing of images, letting you manually resize in bulk once the content is imported.

Server Moves

On a new server, the environment may not match the old environment, and so you should look out for:

- Older PHP versions. Using newer PHP features on a server, then moving to an older version could cause your code to Fatal error. Check before hand what version of PHP is used and make sure it's the same or greater.
  - Run `php -v` or use the `phpversion()` command if you don't have access to the server.
  - PHPStorm uses: use the *PHP Language Level* setting to check for errors automatically.
- File system changes. Not every server puts your site at /srv/www, some use /var/www, and you should make sure that any hardcoded paths are changed to match. You can normally substitute `$_SERVER['DOCUMENT_ROOT']` instead.

URL changes

If you're changing your sites URL, you may or may not be moving server. If you do change URL however, it's not enough to change the DNS and expect things to work. WordPress stores data in the database that contains your sites URL.

A new user may decide to use a small SQL command to search for all instances of the old URL, and replace them with the new URL. This will not work.

The reason for this is that some data is stored in serialised PHP data structures. These serialised strings contain the length of the URL, and if your URLs length changes, the data structures are no longer valid. This causes issues when you attempt to load your site.

To get around this, a number of tools are available that can look inside the data structures and modify them correctly. We recommend using WP-CLI's search-replace command, but other solutions exist.

Security

Salts

Your passwords and cookies are stored with salts applied. Salts are strings of data that are kept secret, and hashed together with important data so that it's harder to guess. This way a hacker can't just run through every password and generate a rainbow table of all possible results and brute force every website. Instead they need to generate a new table for every site they target after acquiring the secret salts used. There is an API to provide salts and secret keys at wordpress.org, which you can then copy paste into your `wp-config.php`.

Escaping

When outputting data, you should escape it. For example, if you output a css class, you should use `esc_attr`, otherwise, an attacker could sneak in the value `classname"><script>alert('hello');</script><span` and run arbitrary code on your site.

An important part of escaping however, is to escape as late as possible. If you escape a variable once, then use it 5 times, that variable may be modified at any point between escaping and output, so always escape at the moment of output.

- Sanitise early
- Escape Late
- Escape Often

Nonces

In the days of MySpace, a user could add an image to their profile, and set the `src` tag as `/logout.php`. Any user who visited their profile would be immediatley logged out. This is an example of a CSRF attack or Cross Site Reference attack.

In order to get around this, we use nonces. Nonces are small tokens that can be passed around to validate an action. For example, a form may contain a nonce, which is then checked for when processed. This makes sure that all form submissions came from the form, and not a malicious or unintended script.

@todo: Add notes on how to use nonces effectively

Note: In the United Kingdom, a nonce is a name for a child sex offender, be careful of using the word out of context

The Location of `wp-config`

- You can move it one level up so it's not in a web accessible location

Table prefixes

- Don't use the default wp_
- Notes on automated attacks

User ID 1

- Don't call it 'admin'
- Don't give it administrator priviledges

Roles and Capabilities

- What they are

Removing vs Hiding Settings Pages

- Hiding things with CSS doesn't make it secure
- People have dev tools too
- Automated tools ignore CSS
- how to remove admin menus and change the capabilities needed to do things

Custom Password Reset Code

- Some people write their own password reset facilities. This is bad
- If you really must, make it a forgotten password link, don't make it actually show your password

timthumb.php

- Don't use it
- There's an image API for that
- timthumb was disowned by its creators and is officially no longer supported
- Banned on a number of managed WordPress hosts

SSL

All big clients deserve an SSL certificate. If you're running an e-commerce site, this is especially true, and your entire site should be using SSL for all logged in users.

- A note on public wifi, unsecured wifi, and snooping
- Maybe mention firesheep?

Admin Only SSL

- If your site isn't an ecommerce site, but you have users who visit the backend, their logged in sessions should be sent over an https connection.
- Explain how

Myths

There are a lot of feel good security fixes that float around, that do nothing to help your security, waste your time, and sometimes increase the risk. Here are a few:

Hiding the Admin and Login URLs

- Some people try to change the admin and login URLs in hopes it will fool attackers and automated tools
- WordPress adds in /admin/ and /login/ rewrite rules in the newer versions so moving the files is pointless
- It can break some functionality in code without necessary care
- Trying to go to the admin URLs will redirect you to the changed login URL anyway, and if you fix that then the modal box that shows in the admin screen when your session expires will be broken too

Deactivated Plugins & Themes

- Because of how PHP works, deactivated plugins can still be hit from a users web browser
- Badly written plugins might do things if the right URL is loaded, even if they're not activated. This is especially true of plugins with their own AJAX endpoints that don't use the WP AJAX API.

Recovering From Attacks

- Take and use regular backups
- Download a fresh copy of WordPress and extract it over the top of your existing install to make sure that WP Core is unmodified
- Check your plugins and code against version control

Community

WordCamps

WordCamps are short, 1-2 day conferences that focus on everything WordPress. They are designed to have both a general focus (on blogging, writing content, marketing websites and the business surrounding WordPress) and a technical focus (aimed at developers writing plugins and themes).

There are over 50 WordCamps every year held all over the world in over 40 countries, and are a great way to explore the community. Many WordCamp sessions are placed onto WordPress.tv, and is a great opportunity to help the WordPress community, either by speaking, volunteering, or sponsoring.

You can find out all upcoming WordCamps at WordCamp Central.

Contributor Days

Contributor days, usually held after WordCamps (but can be independent), are events that are set up to help you contribute to WordPress. The benefits to contributing to WordPress are numerous, both for a business looking to get more exposure in and amongst the WordPress community, to lone developers looking to grow their skills working in a team on a massive project.

You do not need to be code proficient to contribute to WordPress. They are looking for a wide range of skills, like support, theme review, as well as accessibility.

Look at your local WordCamp if they are holding a Contributor day. Alternatively, if you're based in the United Kingdom, you can find the latest WordPress Contributor day at http://www.wpcontributorday.com/.

- Local User Groups
- .org Support Forums
- IRC Channels
- WordPress Stack Exchange
- WordPress Slack

Credits

Many people have contributed to WordPress The Right Way, and they've done so using GitHub. You can view the full list of contributors here, and you can fork and submit your own Pull Request to join them!