

Table of Contents

Theme Handbook

Welcome to the WordPress Theme Developer Handbook, your resource for learning all about the exciting world of WordPress themes.

The Theme Developer Handbook is a repository for all things WordPress themes. Whether you're new to WordPress themes, or you're an experienced theme developer, you should be able to find the answer to many of your theme-related questions right here.

1. If you're new to developing WordPress themes, start with section 1, where you can [find out what a theme is](#), learn about [WordPress' license](#), [set up your development environment](#), and [build your first theme](#).
2. Once you're through the introduction, the [Theme Basics](#) section will introduce you to the building blocks of a WordPress theme.
3. The [Theme Functionality](#) section will show you all of the different types of functionality you can make use of in your theme.
4. If you're looking to give flexibility and yet protect your users, head over to the [Customizer](#) and [Security](#) sections
5. If you've got to grips with the basics of themes, check out the [Advanced Theme Topics](#) to learn about child themes, best UI practices, theme testing and more.
6. Once you've got your theme ready for the world, the final section will cover [releasing your theme](#), teaching you some best practices for theme distribution, and for getting it ready for the WordPress.org theme directory.

The WordPress Theme Developer Handbook is created by the WordPress community, for the WordPress community. We are always looking for more contributors; if you're interested stop by the [docs team blog](#) to find out more about getting involved.

Getting Started

Ready to start developing WordPress themes? You're in the right place.

In this opening chapter, we'll take a look at [what WordPress themes are](#) and how they work. Then, we'll discuss [the GPL](#), the software license used for WordPress and all of the themes in the WordPress theme directory.

Next, we'll look at some of the tools you'll need in your WordPress theme development toolkit and teach you how to [setup a local development environment](#).

We'll finish up this chapter by looking at a few solid [WordPress theme examples](#).

Who Should Read This Handbook?

The Theme Developer Handbook is a self-contained resource to help you learn the basics of WordPress theme development. You should read this handbook to accomplish a number of things:

- develop a child theme meant to work with a parent theme;
- create a new theme based on an existing one;
- understand the inner workings of themes in general; and
- develop a theme based on nothing but your imagination.

Your skill level

To get the most out of this handbook, you should have an understanding and some experience with web technologies, such as HTML, CSS, and PHP. You should also be comfortable with setting up and configuring websites using WordPress.

An understanding of how MySQL databases work, as well as server technologies in general, is helpful but not a requirement for developing WordPress themes. Likewise, while JavaScript knowledge can be useful, it is not required.

What this handbook will cover

This handbook provides the basic information required to develop WordPress themes, including in-depth coverage of essential template tags and functions.

WordPress is a vast subject. Covering every possible tag, function, and scenario is beyond the scope of this handbook. If you need help with specific functions, you can find that information in the code reference.

The purpose of this handbook is to give you a solid foundation for WordPress theme development, to provide step-by-step instruction in building basic themes, and to provide tips and resources useful in furthering your skills.

What is a Theme?

A WordPress theme changes the design of your website, often including its layout. Changing your theme changes how your site looks on the front-end, i.e. what a visitor sees when they browse to your site on the web. There are thousands of free WordPress themes in the WordPress.org Theme Directory, though many WordPress sites use custom themes.

What can themes do?

Themes take the content and data stored by WordPress and display it in the browser.

When you create a WordPress theme, you decide how that content looks and is displayed. There are many options available to you when building your theme. For example:

Your theme can have different layouts, such as static or responsive, using one column or two.

Your theme can display content anywhere you want it to be displayed.

Your theme can specify which devices or actions make your content visible.

Your theme can customize its typography and design elements using CSS.

Other design elements like images and videos can be included anywhere in your theme.

WordPress themes are incredibly powerful. But, as with every web design project, a theme is more than color and layout. Good themes improve engagement with your website's content in addition to being beautiful.

What are themes made of?

At their most basic level, WordPress themes are collections of different files that work together to create what you see, as well as how your site behaves.

Required files #

There are only two files absolutely required in a WordPress theme:

index.php – the main template file

style.css – the main style file

Though not required, you may see additional files in a theme's folder including:

PHP files – including template files

Localization files

CSS files

Graphics

JavaScript

Text files – usually license info, readme.txt instructions, and a changelog file

What is the difference between a theme and a plugin? #

It is common to find cross-over between features found in themes and plugins. However, best practices are:

a theme controls the presentation of content; whereas

a plugin is used to control the behavior and features of your WordPress site.

Any theme you create should not add critical functionality. Doing so means that when a user changes their theme, they lose access to that functionality. For example, say you build a theme with a portfolio feature. Users who build their portfolio with your feature will lose it when they change themes.

By moving critical features to plugins, you make it possible for the design of your website to change, while the functionality remains the same.

Remember, some users switch themes often. It is best practice to make sure any functionality your site requires, even if the design changes, is in a separate plugin.

Themes on WordPress.org #

One of the safest places to download WordPress themes is in the WordPress.org Theme Directory. All themes are closely reviewed, and must meet rigorous theme review guidelines to ensure quality and security.

Getting Started

Now you know what a theme is it's time to get started. If you haven't already done so yet, you should set up your local development environment. You can then check out some examples of WordPress themes or, if you can't wait any longer to get started, dive into building your first theme.

WordPress Licensing & the GPL

To develop WordPress themes for the public—either [free](#) or [paid](#)—you need to get acquainted with the [GNU General Public License](#) (GPL) that WordPress uses.

GPL basic freedoms

The spirit of openness and sharing has thrived within the WordPress community because of fundamental principles that form the core of its license. One way to think of the GPL is as a “*Bill of Rights*” for software. The GPL establishes the following four freedoms:

1. Freedom to *run the program* for any purpose.
2. Freedom to *study how the program works* and to change it, so it performs computing as you wish.
3. Freedom to *redistribute copies*, so you can help your neighbor.
4. Freedom to *distribute copies of your modified versions*, giving the community a chance to benefit from your changes.

What is “free” in the context of software?

The ‘free’ in free software, refers to freedom and not price. The Free Software Foundation likes to say “free as in speech, not as in beer.” Free software is software that users can use to do as they wish. It need not be free from cost, although the ones hosted on the [WordPress.org theme directory](#) are.

Free software can come with a price tag. In other words, you can create a GPL theme and sell it for \$50, and it would still be free software. Why? Because the user is free to run, modify, and distribute the software or any modifications of that software.

Keeping it free for all

The freedoms of the GPL don't only apply to the original piece of software; works derived from GPL-licensed software must also adopt the same license, without restrictions or additional terms.

In this sense, the GPL provides the ultimate protection of freedom by making sure that anything that is derived from free software cannot be "locked down" after the fact; it must remain forever free to future experimentation and exploration.

Do I need to license my themes under the GPL?

If you have no plans to distribute your theme then you do not need to adopt the GPL license for your work. The GPL only applies to *distributed* software. If you are not distributing your software – for example, a theme used only by yourself or on your local machine – you do not need to adopt the GPL.

If you wish to submit your creation to the free theme repository on WordPress.org, it must be 100% GPL compliant, including CSS and image files. Because the freedoms spelled out in the GPL are at the heart of WordPress, we encourage developers to distribute their themes with a 100% GPL-compatible license.

Freedom is an important part of developing WordPress themes. If you plan to distribute your theme, it is a good idea to license it fully under the GPL, so others can enjoy the same freedoms that you did when creating it.

Further Reading

To deepen your understanding of WordPress and the GPL:

- [WordPress.org: Themes are GPL, too](#)
- [Matt: Q&A: WordPress & GPL](#)
- [Matt: The Four Freedoms](#)
- [GNU General Public License, version 2](#)
- [GNU General Public License, version 3](#)

Setting up a Development Environment

Why set up a development environment?

Setting up a Development Environment

When developing themes, it is best to do it in an environment identical to the production server which will eventually host your WordPress installation. Your development environment can either be local or remote. Configuring a local environment to work on your WordPress theme is beneficial for several reasons:

- You can build your theme locally without relying on a remote server. This speeds up your development process and allows you to see changes instantly in your browser.
- You do not need an Internet connection to build your theme.
- You can test your theme from a variety of perspectives. This is important, especially if you plan on releasing your theme to a larger audience and want to ensure maximum compatibility.

Your WordPress local development environment

For developing WordPress themes, you need to set up a development environment suited to WordPress. To get started, you will need a local server stack and a text editor. There are a number of options, including:

Local Server Stack

- A local server stack, such as LAMP (**L**inux **A**pache **M**ySQL/**MP**HP) or WAMP (**W**indows **A**pache **M**ySQL/**MP**HP) is a server (much like the server that runs on your web server), which you will configure on your local machine. You can install pre-bundled programs that contain all of these, like [MAMP](#) (for Mac), or [XAMPP](#) (Mac or Windows) to quickly setup your environment.

Virtualized Environment

- A virtualized such created with Vagrant and VirtualBox allows you to create easily reproducible development environments. [Varying Vagrant Vagrants \(VVV\)](#) is a popular Vagrant option which creates a WordPress development environment.

Text Editor

In addition to a local server environment, you also need a text editor to write your code. Your choice of text editor is personal, but remember that a good text editor can speed up your development process. Your text editor can be everything from a basic tool for writing code to a fully integrated development environment (IDE) with tools for debugging and testing. It's worth doing research, and some even include support for WordPress development. Popular choices are Atom, Sublime Text, and PhpStorm.

You can find a [list of tutorials for setting up development environments at the bottom of the page](#).

Supporting older versions of WordPress

It's standard practice for WordPress themes to support *at least two versions back* to ensure a minimum of backward compatibility. For example, if the current version of WordPress is at 4.6, then you should also make sure that your theme works well in versions 4.5 and 4.4 as well.

You can refer to the [WordPress Roadmap](#) page to access older versions of WordPress. Then you can download and install older WordPress versions, creating multiple development sites, each running different WordPress versions for testing.

WP_DEBUG

Configuring debugging is an essential part of WordPress theme development. WordPress provides a number of constants to support your debugging efforts. These includes:

WP_DEBUG

The [WP_DEBUG](#) PHP constant is used to trigger the built-in “debug” mode on your WordPress installation. This allows you to view errors in your theme. To enable it:

1. Open your WordPress installation's *wp-config.php* file
2. Change:

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

to

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

Note:

While normally set to ‘false’ in the `wp-config.php` file, development copies of WordPress—alpha and beta versions of the upcoming release—`WP_DEBUG` is already set to ‘true’ by default.

`WP_DEBUG_DISPLAY` and `WP_DEBUG_LOG`

`WP_DEBUG_LOG` and `WP_DEBUG_DISPLAY` are additional PHP constants which extend `WP_DEBUG`.

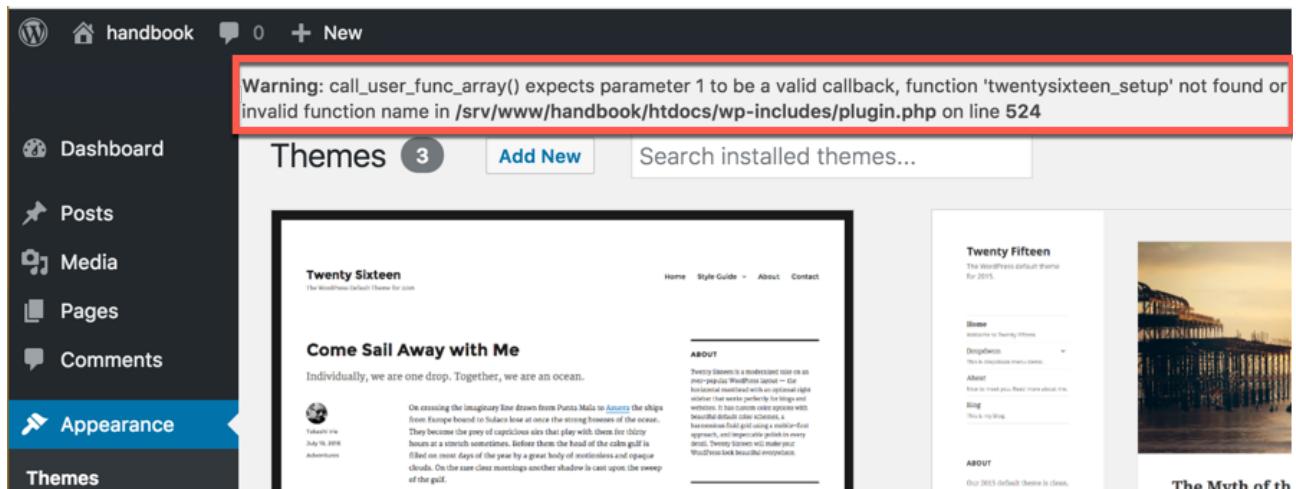
`WP_DEBUG_LOG` is used in conjunction with `WP_DEBUG` to log all error messages to a `debug.log` within your WordPress `/wp-content/` directory. To enable this functionality set `WP_DEBUG_LOG` to true within your `wp-config.php` file.

```
1 define( 'WP_DEBUG_LOG', true );
```

`WP_DEBUG_DISPLAY` is used to control whether debug messages display within the HTML of your theme pages. To display error messages on the screen as they occur, configure this setting to ‘true’ within your `wp-config.php` file.

```
1 define( 'WP_DEBUG_DISPLAY', true );
```

With the `WP_DEBUG` and `WP_DEBUG_DISPLAY` enabled, error messages will display at the top of your site pages.



Note:

Errors will display in the frontend and admin areas of your site. These debug tools are meant for local testing and staging installs, not for live sites.

Other WordPress Development Tools

In addition to `WP_DEBUG`, the following plugins and unit test data sets are an important part of your [development toolset](#) and help you develop better WordPress themes.

Test Data

WordPress.org Theme Unit Test Data

[WordPress.org Theme Unit Test Data](#) is an XML file containing dummy test data that you can upload to test how themes perform with different types and layouts of content.

WordPress.com Theme Unit Test Data

[WordPress.com Theme Unit Test Data](#) is dummy test data that you can upload to a WordPress installation to test your theme, including WordPress.com-specific features.

Plugins

Debug Bar (WordPress plugin)

[Debug Bar](#) adds an admin bar to your WordPress admin providing a central location for debugging.

Query Monitor (WordPress plugin)

[Query Monitor](#) allows debugging of database queries, API request and AJAX called used to generate theme pages and theme functionality.

Log Deprecated Notices (WordPress plugin)

[Log Deprecated Notices](#) logs incorrect function usage and the use of deprecated files and functions in your WordPress theme.

Monster Widgets (WordPress plugin)

[Monster Widget](#) consolidates the core WordPress widgets into a single widget allowing you to test widgets styling and functionality in your theme.

Developer (WordPress plugin)

[Developer](#) helps optimize your development environment by allowing easy installation of tools and plugins that help in troubleshooting and ensuring code quality.

Theme-Check (WordPress plugin)

[Theme-Check](#) tests your theme for compliance with the latest WordPress standards and practices.

WordPress Theme Review Guidelines

In addition to the above development tools, it's a good idea to stay up to date on the WordPress.org Theme Review Team's [Guidelines](#) for theme submission and guidance on meeting [WordPress Coding Standards](#). These guidelines are the “gold standard” for quality theme development and are useful, even if you don't plan on releasing a theme on WordPress.org.

Further Resources

- [Developing WordPress Locally With MAMP](#) (Mac, MAMP)
- [How to Setup a WordPress Development Environment for Windows](#) (Windows, XAMPP)
- [WordPress Theme Review VVV: A Quick Vagrant Setup for Testing Themes](#) (Cross-platform, Vagrant)
- [Setting up your Development Environment](#) (WordPress.com VIP)
- [wptest.io](#) – an exhaustive set of WordPress test data derived from [WordPress' Theme Unit Test](#)

Theme Development Examples

One of the best ways to understand theme coding standards is to find examples of other themes that have been written with these standards in mind.

Default “Twenty” themes

Packaged in every version of WordPress since version 3.0 (and named after the year they were released in), the default themes are some of the best to study how themes are built. This is because they are designed with broad use in mind and fully adhere to WordPress coding standards. You can download and study their theme files, and keep them around as examples to reference while learning how to develop your own themes:

- [Twenty Seventeen](#)
- [Twenty Sixteen \(only packaged in WordPress 4.8\)](#)
- [Twenty Fifteen](#)
- [Twenty Fourteen](#)
- [Twenty Thirteen](#)
- [Twenty Twelve](#)
- [Twenty Eleven](#)
- [Twenty Ten](#)

[Top ↑](#)

The Underscores theme

Unlike the default “Twenties” themes, the `_s` (or Underscores) theme is aimed at developers rather than end-users. It is intended to be a *starter theme* which you can use as a base to speed your development. It has a number of features:

- Well-commented HTML5 templates, including error templates.
- A sample custom header implementation in `inc/custom-header.php`.
- Custom template tags in `inc/template-tags` to keep templates organized and prevent code duplication.
- A number of scripts to improve keyboard navigation—found in `js/keyboard-image-navigation.js`—as well as small screen navigation in `js/navigation.js`.
- Five sample CSS layouts in `/layouts` as well as starter CSS on which to build your design.
- GPL-licensed code.

The features above make Underscores a great theme for a developer wanting to create their own theme. And even if you remove the extras, the base that’s left is a still an excellent example of a well-coded theme, developed with standards and best-practices in mind.

A more detailed overview is available on [Underscores’ website](#).

[Code for Underscores](#) may be found at [github](#).

Other Sources

Additionally, all themes published in the [theme directory](#) are reviewed for standards prior to being published. [Reviewing themes](#) in the directory is a great way to better understand how theme development works and is a good way to get inspiration for your own theme.

Theme Basics

In Chapter 1, you saw an overview of a theme. You also learned how you can start developing a theme.

In this chapter, you’ll begin learning how to build a theme properly. The anatomy of a theme and its moving parts will be broken down and explained. You’ll begin by

understanding the building blocks of a theme by looking at [theme files](#) and [post types](#).

Then you'll learn how to keep your files [organized](#) within your theme.

You'll also look at [The Loop](#), which is responsible for pulling content out of the WordPress database.

Finally, you'll learn more about adding features to your theme by [using theme functions](#), [including CSS & JavaScript](#), leveraging [conditional tags](#) to only show content you need, and working with [default taxonomies and creating your own](#).

Template Files

Template files are used throughout WordPress themes, but first let's learn about the terminology.

Template Terminology

The term "template" is used in different ways when working with WordPress themes:

- Templates files exist within a theme and express how your site is displayed.
- [Page Templates](#) are those that apply *only* to pages to change their look and feel. A page template can be applied to a single page, a page section, or a class of pages.
- [Template Tags](#) are built-in WordPress functions you can use inside a template file to retrieve and display data (such as [the_title\(\)](#) and [the_content\(\)](#)).
- [Template Hierarchy](#) is the logic WordPress uses to decide which theme template file(s) to use, depending on the content being requested.

Template files

WordPress themes are made up of template files. These are PHP files that contain a mixture of HTML, [Template Tags](#), and PHP code.

When you are building your theme, you will use template files to affect the layout and design of different parts of your website. For example, you would use the header.php template to create a header, or the comments.php template to include comments.

When someone visits a page on your website, WordPress loads a template based on the request. The type of content that is displayed in by the template file is determined by the [Post Type](#) associated with the template file. The [Template Hierarchy](#) describes which template file WordPress will load based on the type of request and whether the

template exists in the theme. The server then parses the PHP in the template and returns HTML to the visitor.

The most critical template file is `index.php`, which is the catch-all template if a more-specific template can not be found in the [template hierarchy](#). Although a theme only needs a `index.php` template, typically themes include numerous templates to display different content types and contexts.

Template partials

A template partial is a piece of a template that is included as a part of another template, such as a site header. Template partials can be embedded in multiple templates, simplifying theme creation. Common template partials include:

- `header.php` for generating the site's header
- `footer.php` for generating the footer
- `sidebar.php` for generating the sidebar

While the above template files are special-case in WordPress and apply to just one portion of a page, you can create any number of template partials and include them in other template files.

Common WordPress template files

Below is a list of some basic theme templates and files recognized by WordPress.

`index.php`

The main template file. It is **required** in all themes.

`style.css`

The main stylesheet. It is **required** in all themes and contains the information header for your theme.

`rtl.css`

The right-to-left stylesheet is included automatically if the website language's text direction is right-to-left.

`comments.php`

The comments template.

`front-page.php`

The front page template is always used as the site front page if it exists, regardless of what settings on **Admin > Settings > Reading**.

home.php

The home page template is the front page by default. If you do not set WordPress to use a static front page, this template is used to show latest posts.

header.php

The header template file usually contains your site's document type, meta information, links to stylesheets and scripts, and other data.

singular.php

The singular template is used for posts when `single.php` is not found, or for pages when `page.php` are not found. If `singular.php` is not found, `index.php` is used.

single.php

The single post template is used when a visitor requests a single post.

single-{post-type}.php

The single post template used when a visitor requests a single post from a custom post type. For example, `single-book.php` would be used for displaying single posts from a custom post type named *book*. The `index.php` is used if a specific query template for the custom post type is not present.

archive-{post-type}.php

The archive post type template is used when visitors request a custom post type archive. For example, `archive-books.php` would be used for displaying an archive of posts from the custom post type named *books*. The `archive.php` template file is used if the `archive-{post-type}.php` is not present.

page.php

The page template is used when visitors request individual pages, which are a built-in template.

page-{slug}.php

The page slug template is used when visitors request a specific page, for example one with the “about” slug (page-about.php).

category.php

The category template is used when visitors request posts by category.

tag.php

The tag template is used when visitors request posts by tag.

taxonomy.php

The taxonomy term template is used when a visitor requests a term in a custom taxonomy.

author.php

The author page template is used whenever a visitor loads an author page.

date.php

The date/time template is used when posts are requested by date or time. For example, the pages generated with these slugs:

<http://example.com/blog/2014/>

<http://example.com/blog/2014/05/>

<http://example.com/blog/2014/05/26/>

archive.php

The archive template is used when visitors request posts by category, author, or date. **Note:** this template will be overridden if more specific templates are present like `category.php`, `author.php`, and `date.php`.

search.php

The search results template is used to display a visitor’s search results.

attachment.php

The attachment template is used when viewing a single attachment like an image, pdf, or other media file.

image.php

The image attachment template is a more specific version of `attachment.php` and is used when viewing a single image attachment. If not present, WordPress will use `attachment.php` instead.

404.php

The 404 template is used when WordPress cannot find a post, page, or other content that matches the visitor's request.

Using template files

Within WordPress templates, you can use [Template Tags](#) to display information dynamically, include other template files, or otherwise customize your site.

For example, in your `index.php` you can include other files in your final generated page:

- To include the header, use [get_header\(\)](#)
- To include the sidebar, use [get_sidebar\(\)](#)
- To include the footer, use [get_footer\(\)](#)
- To include the search form, use [get_search_form\(\)](#)
- To include custom theme files, use [get_template_part\(\)](#)

Here is an example of WordPress template tags to *include* specific templates into your page:

```
1 <?php get_sidebar(); ?>
2 <?php get_template_part( 'featured-content' ); ?>
3 <?php get_footer(); ?>
```

There's an entire page on [Template Tags](#) that you can dive into to learn all about them.

Refer to the section [Linking Theme Files & Directories](#) for more information on linking component templates.

Main Stylesheet (style.css)

The `style.css` is a stylesheet (CSS) file required for every WordPress theme. It controls the presentation (visual design and layout) of the website pages.

Location

In order for WordPress to recognize the set of theme template files as a valid theme, the `style.css` file needs to be located in the root directory of your theme, not a subdirectory. For more detailed explanation on how to include the `style.css` file in a theme, see the "Stylesheets" section of [Enqueuing Scripts and Styles](#).

Basic Structure

WordPress uses the header comment section of a style.css to display information about the theme in the Appearance (Themes) dashboard panel.

Example

Here is an example of the header part of style.css.

```
/*
Theme Name: Twenty Seventeen
Theme URI: https://wordpress.org/themes/
twentyseventeen/
Author: the WordPress team
Author URI: https://wordpress.org/
Description: Twenty Seventeen brings your site to life with immersive featured images and subtle animations. With a focus on business sites, it features multiple sections on the front page as well as widgets, navigation and social menus, a logo, and more. Personalize its asymmetrical grid with a custom color scheme and showcase your multimedia content with post formats. Our default theme for 2017 works great in many languages, for any abilities, and on any device.
Version: 1.0
License: GNU General Public License v2 or later
License URI: http://www.gnu.org/licenses/gpl-2.0.html
Text Domain: twentyseventeen
Tags: one-column, two-columns, right-sidebar, flexible-header, accessibility-ready, custom-colors, custom-header, custom-menu, custom-logo, editor-style, featured-images, footer-widgets, post-formats, rtl-language-support, sticky-post, theme-options, threaded-comments, translation-ready
This theme, like WordPress, is licensed under the GPL.
Use it to make something cool, have fun, and share what you've learned with others.
*/
```

Items indicated with (*) are required for a theme in the WordPress Theme Repository.

Note:

WordPress Theme Repository uses the number after “Version” in this file to determine if the theme has a new version available.

- **Theme Name** (*): Name of the theme.
- **Theme URI**: The URL of a public web page where users can find more information about the theme.
- **Author** (*): The name of the individual or organization who developed the theme. Using the Theme Author’s wordpress.org username is recommended.
- **Author URI**: The URL of the authoring individual or organization.
- **Description** (*): A short description of the theme.
- **Version** (*): The version, written in X.X or X.X.X format.
- **License** (*): The license of the theme.
- **License URI** (*): The URL of the theme license.
- **Text Domain** (*): The string used for textdomain for translation.
- **Tags**: Words or phrases that allow users to find the theme using the tag filter. A full list of tags is in the [Theme Review Handbook](#).
- **Domain Path**: Used so that WordPress knows where to find the translation when the theme is disabled. Defaults to `/languages`.

After the required header section, style.css can contain anything a regular CSS file has.

Style.css for a Child Theme

If your theme is a Child Theme, the **Template** line is required in style.css header.

```
/*
Theme Name: My Child Theme
Template: Twenty Seventeen
*/
```

For more information on creating a Child Theme, visit the [Child Themes](#) page.

Post Types

There are many different types of content in WordPress. These content types are normally described as Post Types, which may be a little confusing since it refers to all different types of content in WordPress. For example, a post is a specific Post Type, and so is a page.

Internally, all of the Post Types are stored in the same place — in the wp_posts database table — but are differentiated by a database column called post_type.

In addition to the default Post Types, you can also create Custom Post Types.

The [Template files](#) page briefly mentioned that different Post Types are displayed by different Template files. As the whole purpose of a Template file is to display content a certain way, the Post Types purpose is to categorize what type of content you are dealing with. Generally speaking, certain Post Types are tied to certain template files.

Default Post Types

There are five default Post Types readily available to users or internally used by the WordPress installation:

- Post (Post Type: ‘post’)
- Page (Post Type: ‘page’)
- Attachment (Post Type: ‘attachment’)
- Revision (Post Type: ‘revision’)
- Navigation menu (Post Type: ‘nav_menu_item’)

The Post Types above can be modified and removed by a plugin or theme, but it’s not recommended that you remove built-in functionality for a widely-distributed theme or plugin.

The most common post types you will interact with as a Theme Developer are Post, Page, Attachment, and Custom Post Types. It’s out of the scope of this handbook to flesh out the Revision and Navigation Menu Post Types. However, it is important to note that you will interact with and build the functionality of [navigation menus](#) and that will be detailed later in this handbook.

Post

Posts are used in blogs. They are:

- displayed in reverse sequential order by time, with the newest post first
- have a date and time stamp
- may have the default [taxonomies of categories and tags](#) applied
- are used for creating feeds

The template files that display the Post post type are:

- single.php and single-post.php
- category.php and all its iterations
- tag.php and all its iterations

- taxonomy.php and all its iterations
- archive.php and all its iterations
- author.php and all its iterations
- date.php and all its iterations
- search.php
- home.php
- index.php

Additionally, theme developers can display Post post types in front-page.php if they so desire.

[Read more about Post Template Files.](#)

Page

Pages are a static Post Type, outside of the normal blog stream/feed. Their features are:

- non-time dependent and without a time stamp
- are not organized using the categories and/or tags taxonomies
- can have page templates applied to them
- can be organized in a hierarchical structure — i.e. pages can be parents/children of other pages

The template files that display the Page post type are:

- page.php and all its iterations
- \$custom.php and all its iterations
- front-page.php
- search.php
- index.php

[Read more about Page Template Files.](#)

Attachment

Attachments are commonly used to display images or media in content, and may also be used to link to relevant files. Their features are:

- contain information (such as name or description) about files uploaded through the media upload system
- for images, this includes metadata information stored in the wp_postmeta table (including size, thumbnails, location, etc)

The template files that display the Attachment post type are:

- MIME_type.php
- attachment.php
- single-attachment.php
- single.php
- index.php

[Read more about Attachment Template Files.](#)

Custom Post Types

Using Custom Post Types, you can **create your own post type**. It is not recommended that you place this functionality in your theme. This type of functionality should be placed/created in a plugin. This ensures the portability of your user's content, and that if the theme is changed the content stored in the Custom Post Types won't disappear.

You can learn more about [creating custom post types in the WordPress Plugin Developer Handbook](#).

While you generally won't develop Custom Post Types in your theme, you may want to code ways to display Custom Post Types that were created by a plugin. The following templates can display Custom post types:

- single-{post-type}.php
- archive-{post-type}.php
- search.php
- index.php

Additionally, Theme Developers can display Custom Post Types in any template file, often by using [multiple loops](#).

[Read more about Custom Post Type Templates.](#)

Organizing Theme Files

While WordPress themes technically only require two files (`index.php` and `style.css`), they usually are made up of many files. That means they can quickly become disorganized! This section will show you how to keep your files organized.

Note:

Themes without `header.php` and `footer.php`, with no alternative available, have been deprecated since WordPress 3.0. Your theme may need to include these files as well.

Theme folder and file structure

As mentioned previously, the default Twenty themes are some of the best examples of good theme development. For instance, here is how the [Twenty Seventeen Theme](#) organizes its [file structure](#):

```
1 assets (dir)
2   - css (dir)
3   - images (dir)
4   - js (dir)
5 inc (dir)
6 template-parts (dir)
7   - footer (dir)
8   - header (dir)
9   - navigation (dir)
10  - page (dir)
11  - post (dir)
12 404.php
13 archive.php
14 comments.php
15 footer.php
16 front-page.php
17 functions.php
18 header.php
19 index.php
20 page.php
21 README.txt
22 rtl.css
23 screenshot.png
24 search.php
25 searchform.php
26 sidebar.php
27 single.php
28 style.css
```

You can see that the main theme template files are in the root directory, while JavaScript, CSS, images are placed in assets directory, template-parts are placed in under respective

subdirectory of template-parts and collection of functions related to core functionalities are placed in inc directory.

At this time there are **no required folders within a WordPress theme**. However, WordPress does recognize the following folders by default.

Note:

`style.css` should reside in the root directory of your theme not within the CSS directory.

Languages folder

It's best practice to [internationalize your theme](#) so it can be translated into other languages. Default themes include the `languages` folder, which contains a `.pot` file for translation and any translated `.mo` files. While `languages` is the default name of this folder, you can change the name. If you do so, you must update [`load_theme_textdomain\(\)`](#).

Template Hierarchy

As discussed, [template files](#) are modular, reusable files, used to generate the web pages on your WordPress site. Some template files (such as the header and footer template) are used on all of your site's pages, while others are used only under specific conditions.

This article explains **how WordPress determines which template file(s) to use on individual pages**. If you want to customize an existing WordPress theme it will help you decide which template file needs to be edited.

Tip:

You can also use [Conditional Tags](#) to control which templates are loaded on a specific page.

The Template File Hierarchy

Overview

WordPress uses the [query string](#) to decide which template or set of templates should be used to display the page. The query string is information that is contained in the link to each part of your website. It comes after the initial question mark and may contain a number of parameters separated by ampersands.

Put simply, WordPress searches down through the template hierarchy until it finds a matching template file. To determine which template file to use, WordPress:

1. Matches every query string to a query type to decide which page is being requested (for example, a search page, a category page, etc);
2. Selects the template in the order determined by the template hierarchy;
3. Looks for template files with specific names in the current theme's directory and uses the **first matching template file** as specified by the hierarchy.

With the exception of the basic `index.php` template file, you can choose whether you want to implement a particular template file or not.

If WordPress cannot find a template file with a matching name, it will skip to the next file in the hierarchy. If WordPress cannot find any matching template file, the theme's `index.php` file will be used.

Examples

If your blog is at `http://example.com/blog/` and a visitor clicks on a link to a category page such as `http://example.com/blog/category/your-cat/`,

WordPress looks for a template file in the current theme's directory that matches the category's ID to generate the correct page. More specifically, WordPress follows this procedure:

1. Looks for a template file in the current theme's directory that matches the category's slug. If the category slug is "unicorns," then WordPress looks for a template file named `category-unicorns.php`.
2. If `category-unicorns.php` is missing and the category's ID is 4, WordPress looks for a template file named `category-4.php`.
3. If `category-4.php` is missing, WordPress will look for a generic category template file, `category.php`.
4. If `category.php` does not exist, WordPress will look for a generic archive template, `archive.php`.
5. If `archive.php` is also missing, WordPress will fall back to the main theme template file, `index.php`.

Visual Overview

The following diagram shows which template files are called to generate a WordPress page based on the WordPress template hierarchy.



You can also [interact with this diagram](#).

The Template Hierarchy In Detail

While the template hierarchy is easier to understand as a diagram, the following sections describe the order in which template files are called by WordPress for a number of query types.

Home Page display

By default, WordPress sets your site's home page to display your latest blog posts. This page is called the blog posts index. You can also set your blog posts to display on a separate static page. The template file `home.php` is used to render the blog posts index, whether it is being used as the front page or on separate static page. If `home.php` does not exist, WordPress will use `index.php`.

1. `home.php`
2. `index.php`

Note:

If front-page.php exists, it will override the home.php template.

Front Page display

The front-page.php template file is used to render your site's front page, whether the front page displays the blog posts index (mentioned above) or a static page. The front page template takes precedence over the blog posts index (home.php) template. If the front-page.php file does not exist, WordPress will either use the home.php or page.php files depending on the setup in Settings → Reading. If neither of those files exist, it will use the index.php file.

1. front-page.php – Used for both “**your latest posts**” or “**a static page**” as set in the **front page displays** section of Settings → Reading.
2. home.php – If WordPress cannot find front-page.php and “**your latest posts**” is set in the **front page displays** section, it will look for home.php. Additionally, WordPress will look for this file when the **posts page** is set in the **front page displays** section.
3. page.php – When “front page” is set in the front page displays section.
4. index.php – When “your latest posts” is set in the front page displays section but home.php does not exist or when front page is set but page.php does not exist.

As you can see, there are a lot of rules to what path WordPress takes. Using the chart above is the best way to determine what WordPress will display.

Privacy Policy Page display

The privacy-policy.php template file is used to render your site's Privacy Policy page. The Privacy Policy page template takes precedence over the static page (page.php) template. If the privacy-policy.php file does not exist, WordPress will either use the page.php or singular.php files depending on the available templates. If neither of those files exist, it will use the index.php file.

1. privacy-policy.php – Used for the Privacy Policy page set in the **Change your Privacy Policy page** section of Settings → Privacy.

2. custom template file – The [page template](#) assigned to the page. See `get_page_templates()`.
3. `page-{slug}.php` – If the page slug is `privacy`, WordPress will look to use `page-privacy.php`.
4. `page-{id}.php` – If the page ID is 6, WordPress will look to use `page-6.php`.
5. `page.php`
6. `singular.php`
7. `index.php`

Single Post

The single post template file is used to render a single post. WordPress uses the following path:

1. `single-{post-type}-{slug}.php` – (Since 4.4) First, WordPress looks for a template for the specific post. For example, if [post type](#) is `product` and the post slug is `dmc-12`, WordPress would look for `single-product-dmc-12.php`.
2. `single-{post-type}.php` – If the post type is `product`, WordPress would look for `single-product.php`.
3. `single.php` – WordPress then falls back to `single.php`.
4. `singular.php` – Then it falls back to `singular.php`.
5. `index.php` – Finally, as mentioned above, WordPress ultimately falls back to `index.php`.

Single Page

The template file used to render a static page (page post-type). Note that unlike other post-types, `page` is special to WordPress and uses the following path:

1. custom template file – The [page template](#) assigned to the page. See `get_page_templates()`.
2. `page-{slug}.php` – If the page slug is `recent-news`, WordPress will look to use `page-recent-news.php`.
3. `page-{id}.php` – If the page ID is 6, WordPress will look to use `page-6.php`.
4. `page.php`
5. `singular.php`

6. index.php

Category

Rendering category archive index pages uses the following path in WordPress:

1. category-{slug}.php – If the category's slug is news, WordPress will look for category-news.php.
2. category-{id}.php – If the category's ID is 6, WordPress will look for category-6.php.
3. category.php
4. archive.php
5. index.php

Tag

To display a tag archive index page, WordPress uses the following path:

1. tag-{slug}.php – If the tag's slug is sometag, WordPress will look for tag-sometag.php.
2. tag-{id}.php – If the tag's ID is 6, WordPress will look for tag-6.php.
3. tag.php
4. archive.php
5. index.php

Custom Taxonomies

Custom taxonomies use a slightly different template file path:

1. taxonomy-{taxonomy}-{term}.php – If the taxonomy is sometax, and taxonomy's term is someterm, WordPress will look for taxonomy-sometax-someterm.php. In the case of post formats, the taxonomy is 'post_format' and the terms are 'post-format-{format}'. i.e. taxonomy-post_format-post-format-link.php for the link post format.
2. taxonomy-{taxonomy}.php – If the taxonomy were sometax, WordPress would look for taxonomy-sometax.php.
3. taxonomy.php

4. archive.php
5. index.php

Custom Post Types

[Custom Post Types](#) use the following path to render the appropriate archive index page.

1. archive-{post_type}.php – If the post type is product, WordPress will look for archive-product.php.
2. archive.php
3. index.php

(For rendering a single post type template, refer to the [single post display](#) section above.)

Author display

Based on the above examples, rendering author archive index pages is fairly explanatory:

1. author-{nicename}.php – If the author's nice name is matt, WordPress will look for author-matt.php.
2. author-{id}.php – If the author's ID were 6, WordPress will look for author-6.php.
3. author.php
4. archive.php
5. index.php

Date

Date-based archive index pages are rendered as you would expect:

1. date.php
2. archive.php
3. index.php

Search Result

Search results follow the same pattern as other template types:

1. search.php
2. index.php

404 (Not Found)

Likewise, 404 template files are called in this order:

1. 404.php
2. index.php

Attachment

Rendering an attachment page (attachment post-type) uses the following path:

1. {MIME-type}.php – can be any [MIME type](#) (For example: image.php, video.php, pdf.php). For text/plain, the following path is used (in order):
 1. text-plain.php
 2. plain.php
 3. text.php
2. attachment.php
3. single-attachment-{slug}.php – For example, if the attachment slug is holiday, WordPress would look for single-attachment-holiday.php.
4. single-attachment.php
5. single.php
6. singular.php
7. index.php

Embeds

The embed template file is used to render a post which is being embedded. Since 4.5, WordPress uses the following path:

1. embed-{post-type}-{post_format}.php – First, WordPress looks for a template for the specific post. For example, if its post type is post and it has the audio format, WordPress would look for embed-post-audio.php.
2. embed-{post-type}.php – If the post type is product, WordPress would look for embed-product.php.
3. embed.php – WordPress then falls back to embed.php.
4. Finally, WordPress ultimately falls back to its own wp-includes/theme-compat/embed.php template.

Non-ASCII Character Handling

Since WordPress 4.7, any dynamic part of a template name which includes non-ASCII characters in its name actually supports both the un-encoded and the encoded form, in that order. You can choose which to use.

Here's the page template hierarchy for a page named "Hello World" with an ID of 6:

- page-hello-world-.php
- page-hello-world-%f0%9f%98%80.php
- page-6.php
- page.php
- singular.php

The same behaviour applies to post slugs, term names, and author nicenames.

Filter Hierarchy

The WordPress template system lets you filter the hierarchy. This means that you can insert and change things at specific points of the hierarchy. The filter (located in the [get_query_template\(\)](#) function) uses this filter name: "{\$type}_template" where \$type is the template type.

Here is a list of all available filters in the template hierarchy:

- embed_template
- 404_template
- search_template
- frontpage_template
- home_template
- privacypolicy_template
- taxonomy_template
- attachment_template
- single_template
- page_template
- singular_template
- category_template
- tag_template
- author_template

- date_template
- archive_template
- index_template

Example

For example, let's take the default author hierarchy:

- author-{nicename}.php
- author-{id}.php
- author.php

To add author-{role}.php before author.php, we can manipulate the actual hierarchy using the 'author_template' template type. This allows a request for /author/username where username has the role of editor to display using author-editor.php if present in the current themes directory.

```

1 function author_role_template( $templates = '' ) {
2     $author = get_queried_object();
3     $role = $author->roles[0];
4     if ( ! is_array( $templates ) && ! empty( $templates ) )
5     {
6         $templates = locate_template( array( "author-
$role.php", $templates ), false );
7     } elseif ( empty( $templates ) ) {
8         $templates = locate_template( "author-$role.php",
9 false );
10    } else {
11        $new_template = locate_template( array( "author-
$role.php" ) );
12        if ( ! empty( $new_template ) ) {
13            array_unshift( $templates, $new_template );
14        }
15    }
16    return $templates;
}
add_filter( 'author_template', 'author_role_template' );

```

Template Tags

Template tags are used within themes to **retrieve content from your database**. The content could be anything from a blog title to a complete sidebar. Template tags are the preferred method to pull content into your theme because:

- they can print dynamic content;
- they can be used in multiple theme files; and
- they separate the theme into smaller, more understandable, sections.

What is a Template Tag?

A template tag is simply a piece of code that tells WordPress to get something from the database. It is broken up into three components:

- A PHP code tag
- A WordPress function
- Optional parameters

You can use a template tag to call another theme file or some information from the database.

For example, the template tag [`get_header\(\)`](#) tells WordPress to get the header .php file and include it in the current theme file. Similarly, [`get_footer\(\)`](#) tells WordPress to get the footer.php file.

There are also other kinds of template tags:

- [`the_title\(\)`](#) – tells WordPress to get the title of the page or post from the database and include it.
- [`bloginfo\('name' \)`](#) – tells WordPress to get the blog title out of the database and include it in the template file.

If you look closely at the last example, you will also see that there is a parameter between the parenthesis. Parameters let you do two things:

1. ask for specific pieces of information and
2. format the information in a certain way.

[Parameters are covered extensively below](#), but it's useful to be aware that you can send WordPress-specific instructions for how you want the data presented.

Why Use Template Tags

By encapsulating all of the code for a particular chunk of content, template tags make it very easy to include various pieces of a template in a theme file and also to maintain the theme.

It is far easier to create one header .php file and have all of your theme templates like single.php, page.php, front-page.php, etc. reference that one theme file using [get_header\(\)](#) than copying and pasting the code into each theme file. It also makes maintenance easier. Whenever you make a change in your header .php file, the change is automatically carried over into all of your other theme files.

Another reason to use template tags is to display dynamic data, i.e. data from the database. In your header, you could manually include the title tag, like so:

```
1 <title>My Personal Website</title>
```

However, doing this means manually editing your theme any time you want to change the title of your website. Instead, it's easier to include the [bloginfo\('name' \)](#) template tag, which automatically fetch the site title from the database. Now, you can change the title of your site in WordPress, rather than having to hard code your theme templates.

How to Use Template Tags

Using template tags is very simple. In any template file you can use a template tag by simply printing one line of php code to call the template tag. Printing the header.php file is as simple as:

```
1 get_header();
```

Parameters

Some template tags let you pass parameters. Parameters are extra pieces of information that determine what is retrieved from the database.

For example, the [bloginfo\(\)](#) template tag allows you to give it a parameter telling WordPress the specific piece of information you want. To print the blog name, you just pass along the parameter "name," like so:

```
1 bloginfo( 'name' );
```

To print the version of WordPress that the blog is running on, you would pass a parameter of “version”:

```
1 bloginfo( 'version' );
```

For each template tag, the parameters differ. A list of the parameters and what they can do can be found on specific template tag pages located throughout the [code reference](#).

Using Template Tags Within the Loop

Many template tags work within the [WordPress Loop](#). This means that they are included in the template files as part of the php “loop” that generates the pages users see based upon the instructions inside of the loop.

The WordPress loop begins with:

```
1 if ( have_posts() ) :  
2     while ( have_posts() ) :  
3         the_post();
```

Template tags that work within the loop must be in the middle area, before the ending section of the loop below:

```
1 endwhile;  
2 else :  
3     _e( 'Sorry, no posts matched your criteria.',  
4     'devhub' );  
5 endif;
```

Some of template tags that need to be inside of the loop include

- [the_content\(\)](#)
- [the_excerpt\(\)](#)
- [next_post\(\)](#)
- [previous_post\(\)](#)

The main reason why some functions require the loop is because they require the global post object to be set.

If the template tag you want to use doesn’t have to be within the loop

- [wp_list_cats\(\)](#)
- [wp_list_pages\(\)](#)

then you can put it in any file you'd like, for instance in the sidebar, header, or footer template files.

These are functions that typically do not require the global post object.

The Loop

The Loop is the default mechanism WordPress uses for outputting posts through a theme's [template files](#). How many posts are retrieved is determined by the number of posts to show per page defined in the Reading settings. Within the Loop, WordPress retrieves each post to be displayed on the current page and formats it according to your theme's instructions.

The Loop extracts the data for each post from the WordPress database and inserts the appropriate information in place of each [template tag](#). Any HTML or PHP code in The Loop will be processed **for each post**.

To put it simply, the Loop is true to its name: it loops through each post retrieved for the current page one at a time and performs the action specified in your theme.

You can use the Loop for a number of different things, for example to:

- display post titles and excerpts on your blog's homepage;
- display the content and comments on a single post;
- display the content on an individual page using template tags; and
- display data from [Custom Post Types](#) and Custom Fields.

You can customize the Loop across your template files to display and manipulate different content.

The Loop in Detail

The basic loop is:

```
1 <?php
2 if ( have_posts() ) :
3     while ( have_posts() ) : the_post();
4         // Display post content
5     endwhile;
6 endif;
7 ?>
```

This loop says that when there are posts, loop through and display the posts. Broken down into more detail:

- The [have_posts\(\)](#) function checks whether there are any posts.

- If there are posts, a `while` loop continues to execute as long as the condition in the parenthesis is logically true. As long as `have_posts()` continues to be true, the loop will continue.

Using The Loop

The Loop should be placed in `index.php`, and in any other templates which are used to display post information. Because you do not want to duplicate your header over and over, the loop should always be placed after the call to [get_header\(\)](#). For example:

```

1 <?php
2 get_header();
3 if ( have_posts() ) :
4     while ( have_posts() ) : the_post();
5         // Display post content
6     endwhile;
7 endif;
8 ?>
```

In the above example, the end of the Loop is shown with an `endwhile` and `endif`. The Loop must always begin with the same `if` and `while` statements, as mentioned above and must end with the same `end` statements.

Any [template tags](#) that you wish to apply to all posts must exist between the beginning and ending statements.

Tip:

You can include a custom 404 “not found” message that will be displayed if no posts matching the specified criteria are available. The message must be placed between the `endwhile` and `endif` statements, as seen in examples below.

An extremely simple `index.php` file would look like:

```
1 <?php
2 get_header();
3
4 if ( have_posts() ) :
5     while ( have_posts() ) : the_post();
6         the_content();
7     endwhile;
8 else :
9     _e( 'Sorry, no posts matched your criteria.', 'textdomain' );
10 endif;
11
12 get_sidebar();
13 get_footer();
14 ?>
```

[Expand full source code](#)

[Top ↑](#)

What the Loop Can Display

The Loop can display a number of different elements for each post. For example, some common [template tags](#) used in many themes are:

- [next_post_link\(\)](#) – a link to the post published chronologically *after* the current post
- [previous_post_link\(\)](#) – a link to the post published chronologically *before* the current post
- [the_category\(\)](#) – the category or categories associated with the post or page being viewed
- [the_author\(\)](#) – the author of the post or page
- [the_content\(\)](#) – the main content for a post or page
- [the_excerpt\(\)](#) – the first 55 words of a post's main content followed by an ellipsis (...) or read more link that goes to the full post. You may also use the "Excerpt" field of a post to customize the length of a particular excerpt.
- [the_ID\(\)](#) – the ID for the post or page
- [the_meta\(\)](#) – the custom fields associated with the post or page
- [the_shortlink\(\)](#) – a link to the page or post using the url of the site and the ID of the post or page
- [the_tags\(\)](#) – the tag or tags associated with the post

- `the_title()` – the title of the post or page
- `the_time()` – the time or date for the post or page. This can be customized using standard php date function formatting.

You can also use [conditional tags](#), such as:

- `is_home()` – Returns true if the current page is the homepage
- `is_admin()` – Returns true if inside Administration Screen, false otherwise
- `is_single()` – Returns true if the page is currently displaying a single post
- `is_page()` – Returns true if the page is currently displaying a single page
- `is_page_template()` – Can be used to determine if a page is using a specific template, for example: `is_page_template('about-page.php')`
- `is_category()` – Returns true if page or post has the specified category, for example: `is_category('news')`
- `is_tag()` – Returns true if a page or post has the specified tag
- `is_author()` – Returns true if inside author's archive page
- `is_search()` – Returns true if the current page is a search results page
- `is_404()` – Returns true if the current page does not exist
- `has_excerpt()` – Returns true if the post or page has an excerpt

Examples

Let's take a look at some examples of the Loop in action:

Basic Examples

Blog Archive

Most blogs have a blog archive page, which can show a number of things including the post title, thumbnail, and excerpt. The example below shows a simple loop that checks to see if there are any posts and, if there are, outputs each post's title, thumbnail, and excerpt. If no posts exists, it displays the message in parentheses.

```
1 <?php
2 if ( have_posts() ) :
3     while ( have_posts() ) : the_post();
4         the_title( '<h2>', '</h2>' );
5         the_post_thumbnail();
6         the_excerpt();
7     endwhile;
8 else:
9     _e( 'Sorry, no posts matched your criteria.', 'textdomain' );
10 endif;
11 ?>
```

Individual Post

In WordPress, each post has its own page, which displays the relevant information for that post. Template tags allow you to customize which information you want to display. In the example below, the loop outputs the post's title and content. You could use this example in a post or page template file to display the most basic information about the post. You could also customize this template to add more data to the post, for example the category.

```
1 <?php
2 if ( have_posts() ) :
3     while ( have_posts() ) : the_post();
4         the_title( '<h1>', '</h1>' );
5         the_content();
6     endwhile;
7 else:
8     _e( 'Sorry, no pages matched your criteria.', 'textdomain' );
9 endif;
10 ?>
```

Intermediate Examples

Style Posts from Some Categories Differently

The example below does a couple of things:

- First, it displays each post with its title, time, author, content, and category, similar to the individual post example above.

- Next, it makes it possible for posts with the category ID of “3” to be styled differently, utilizing the `in_category(3)` template tag.

Code comments in this example provide details throughout each stage of the loop:

```
1 <?php
2 // Start the Loop.
3 if ( have_posts() ) :
4     while ( have_posts() ) : the_post();
5         /* * See if the current post is in category 3.
6             * If it is, the div is given the CSS class
7 "post-category-three".
8                 * Otherwise, the div is given the CSS class
9 "post".
10            */
11            if ( in_category( 3 ) ) : ?>
12                <div class="post-category-three">
13                    <?php else : ?>
14                    <div class="post">
15                        <?php endif;
16
17                    // Display the post's title.
18                    the_title( '<h2>', ';' );
19
20                    // Display a link to other posts by this posts
21 author.
22                    printf( __( 'Posted by %s', 'textdomain' ),
23 get_the_author_posts_link() );
24
25                    // Display the post's content in a div.
26                    ?>
27                    <div class="entry">
28                        <?php the_content() ?>
29                    </div>
30
31                    <?php
32                        // Display a comma separated list of the
33 post's categories.
34                        _e( 'Posted in ', 'textdomain' );
35 the_category( ', ' );
36
37                    // closes the first div box with the class of
38 "post" or "post-cat-three"
39                    ?>
40                    </div>
41
42                    <?php
43                        // Stop the Loop, but allow for a "if not posts"
```

Multiple Loops

In some situations, you may need to use more than one loop. For example you may want to display the titles of the posts in a table of content list at the top of the page and then display the content further down the page. Since the query isn't being changed we simply need to rewind the loop when we need to loop through the posts for a second time. For that we will use the function [rewind_posts\(\)](#).

Using [rewind_posts\(\)](#)

You can use [rewind_posts\(\)](#) to loop through the *same* query a second time. This is useful if you want to display the same query twice in different locations on a page.

Here is an example of `rewind_posts()` in use:

```
1 <?php
2 // Start the main loop
3 if ( have_posts() ) :
4     while ( have_posts() ) : the_post();
5         the_title();
6     endwhile;
7 endif;
8
9 // Use rewind_posts() to use the query a second time.
10 rewind_posts();
11
12 // Start a new loop
13 while ( have_posts() ) : the_post();
14     the_content();
15 endwhile;
16 ?>
```

Creating secondary queries and loops

Using two loops with the same query was relatively easy but not always what you will need. Instead, you will often want to create a secondary query to display different content on the template. For example, you might want to display two groups of posts on the same page, but do different things to each group. A common example of this, as shown below, is displaying a single post with a list of posts from the same category below the single post.

```

1 <?php
2 // The main query.
3 if ( have_posts() ) :
4     while ( have_posts() ) : the_post();
5         the_title();
6         the_content();
7     endwhile;
8 else :
9     // When no posts are found, output this text.
10    _e( 'Sorry, no posts matched your criteria.' );
11 endif;
12 wp_reset_postdata();
13
14
15 /*
16 * The secondary query. Note that you can use any
17 category name here. In our example,
18 * we use "example-category".
19 */
20 $secondary_query = new WP_Query( 'category_name=example-
21 category' );
22
23 // The second loop.
24 if ( $secondary_query->have_posts() )
25     echo '<ul>';
26     while ( $secondary_query->have_posts() ) :
27 $secondary_query->the_post();
28         the_title( '<li>', '</li>' );
29     endwhile;
30     echo '</ul>';
31 endif;
32 wp_reset_postdata();
33 ?>

```

As you can see in the example above, we first display a regular loop. Then we define a new variable that uses [WP_Query](#) to query a specific category; in our case, we chose the example-category slug.

Note that the regular loop in the example above has one difference: it calls [wp_reset_postdata\(\)](#) to reset the post data. Before you can use a second loop, you need to reset the post data. There are two ways to do this:

1. By using the [rewind_posts\(\)](#) function; or

2. By creating new query objects.

Resetting multiple loops

It's important when using multiple loops in a template that you reset them. Not doing so can lead to unexpected results due to how data is stored and used within the global \$post variable. There are three main ways to reset the loop depending on the way they are called.

- [wp_reset_postdata\(\)](#)
- [wp_reset_query\(\)](#)
- [rewind_posts\(\)](#)

Using [wp_reset_postdata\(\)](#)

Use [wp_reset_postdata\(\)](#) when you are running custom or multiple loops with WP_Query. This function restores the global \$post variable to the current post in the main query. If you're following best practices, this is the most common function you will use to reset loops.

To properly use this function, place the following code after any loops with WP_Query:

```
1 <?php wp_reset_postdata(); ?>
```

Here is an example of a loop using WP_Query that is reset with [wp_reset_postdata\(\)](#).

```

1 <?php
2 // Example argument that defines three posts per page.
3 $args = array( 'posts_per_page' => 3 );
4
5 // Variable to call WP_Query.
6 $the_query = new WP_Query( $args );
7
8 if ( $the_query->have_posts() ) :
9     // Start the Loop
10    while ( $the_query->have_posts() ) : $the_query-
11    >the_post();
12        the_title();
13        the_excerpt();
14    // End the Loop
15    endwhile;
16 else:
17 // If no posts match this query, output this text.
18     _e( 'Sorry, no posts matched your criteria.', 
19 'textdomain' );
20 endif;
21
wp_reset_postdata();
?>
```

Using [wp_reset_query\(\)](#)

Using [wp_reset_query\(\)](#) restores the [WP_Query](#) and global \$post data to the original main query. You **MUST** use this function to reset your loop if you use [query_posts\(\)](#) within your loop. You can use it after custom loops with [WP_Query](#) because it actually calls [wp_reset_postdata\(\)](#) when it runs. However, it's best practice to use [wp_reset_postdata\(\)](#) with any custom loops involving [WP_Query](#).

Alert:

[query_posts\(\)](#) is *not best practice* and should be avoided if at all possible. Therefore, you shouldn't have much use for [wp_reset_query\(\)](#).

To properly use this function, place the following code after any loops with [query_posts\(\)](#).

1	<?php wp_reset_query(); ?>
---	----------------------------

Theme Functions

The `functions.php` file is where you add unique features to your WordPress theme. It can be used to hook into the core functions of WordPress to make your theme more modular, extensible, and functional.

What is `functions.php`?

The `functions.php` file behaves like a WordPress plugin, adding features and functionality to a WordPress site. You can use it to call WordPress functions and to define your own functions.

Note:

The same result can be produced using either a plugin or `functions.php`. If you are creating new features that should be available no matter what the website looks like, **it is best practice to put them in a plugin.**

There are advantages and tradeoffs to either using a WordPress plugin or using `functions.php`.

A WordPress plugin:

- requires specific, unique header text;
- is stored in `wp-content/plugins`, usually in a subdirectory;
- only executes on page load when activated;
- applies to all themes; and
- should have a single purpose – for example, offer search engine optimization features or help with backups.

Meanwhile, a `functions.php` file:

- requires no unique header text;
- is stored in theme's subdirectory in `wp-content/themes`;
- executes only when in the active theme's directory;
- applies only to that theme (if the theme is changed, the features can no longer be used); and
- can have numerous blocks of code used for many different purposes.

Each theme has its own functions file, but only code in the active theme's `functions.php` is actually run. If your theme already has a functions file, you

can add code to it. If not, you can create a plain-text file named `functions.php` to add to your theme's directory, as explained below.

A [child theme](#) can have its own `functions.php` file. Adding a function to the child functions file is a risk-free way to modify a parent theme. That way, when the parent theme is updated, you don't have to worry about your newly added function disappearing.

Note:

Although the child theme's `functions.php` is loaded by WordPress right before the parent theme's `functions.php`, it does not *override* it. The child theme's `functions.php` can be used to augment or replace the parent theme's functions.

Similarly, `functions.php` is loaded *after any plugin files have loaded*.

With `functions.php` you can:

- Use WordPress hooks. For example, with the `excerpt_length` filter you can change your post excerpt length (from default of 55 words).
- Enable WordPress features with [`add_theme_support\(\)`](#). For example, turn on post thumbnails, post formats, and navigation menus.
- Define functions you wish to reuse in multiple theme template files.

Warning:

If a WordPress plugin calls the same function, or filter, as you do in your `functions.php`, the results can be unexpected, even causing your site to be disabled.

Examples

Below are a number of examples that you can use in your `functions.php` file to support various features. Each of these examples are allowed in your theme if you choose to submit it to the WordPress.org theme directory.

Theme Setup

A number of theme features should be included within a “setup” function that runs initially when your theme is activated. As shown below, each of these features can be added to your `functions.php` file to activate recommended WordPress features.

Note:

It's important to namespace your functions with your theme name. All examples below use `myfirsttheme_` as their namespace, which should be customized based on your theme name.

To create this initial function, start a new function entitled `myfirsttheme_setup()`, like so:

```
1 if ( ! function_exists( 'myfirsttheme_setup' ) ) :  
2 /**  
3 * Sets up theme defaults and registers support for various  
4 WordPress features  
5 *  
6 * It is important to set up these functions before the  
7 init hook so that none of these  
8 * features are lost.  
9 *  
10 * @since MyFirstTheme 1.0  
*/  
function myfirsttheme_setup() {
```

Note: In the above example, the function `myfirsttheme_setup` is started but not closed out. Be sure to close out your functions

Automatic Feed Links

Automatic feed links enables post and comment RSS feeds by default. These feeds will be displayed in `<head>` automatically. They can be called using

[`add_theme_support\(\)`](#).

```
1 add_theme_support( 'automatic-feed-links' );
```

[Top ↑](#)

Navigation Menus

Custom [`navigation menus`](#) allow users to edit and customize menus in the Menus admin panel, giving users a drag-and-drop interface to edit the various menus in their theme.

You can set up multiple menus in `functions.php`. They can be added using [`register_nav_menus\(\)`](#) and inserted into a theme using [`wp_nav_menu\(\)`](#), as discussed [`later in this handbook`](#). If your theme will allow more than one menu, you should use an array. While some themes will not have custom navigation menus, it is recommended that you allow this feature for easy customization.

```
1 register_nav_menus( array(
2     'primary'    => __( 'Primary Menu', 'myfirsttheme' ),
3     'secondary'  => __( 'Secondary Menu', 'myfirsttheme' )
4 ) );
```

Each of the menus you define can be called later using [wp_nav_menu\(\)](#) and using the name assigned (i.e. primary) as the theme_location parameter.

Load Text Domain

Themes can be translated into multiple languages by making the strings in your theme available for translation. To do so, you must use [load_theme_textdomain\(\)](#). For more information on making your theme available for translation, read the [internationalization](#) section.

```
1 load_theme_textdomain( 'myfirsttheme',
    get_template_directory() . '/languages' );
```

Post Thumbnails

[Post thumbnails and featured images](#) allow your users to choose an image to represent their post. Your theme can decide how to display them, depending on its design. For example, you may choose to display a post thumbnail with each post in an archive view. Or, you may want to use a large featured image on your homepage. While not every theme needs featured images, it's recommended that you support post thumbnails and featured images.

```
1 add_theme_support( 'post-thumbnails' );
```

Post Formats

[Post formats](#) allow users to format their posts in different ways. This is useful for allowing bloggers to choose different formats and templates based on the content of the post. `add_theme_support()` is also used for Post Formats. This is **recommended**.

```
1 add_theme_support( 'post-formats', array ( 'aside',
    'gallery', 'quote', 'image', 'video' ) );
```

[Learn more about post formats.](#)

Initial Setup Example

Including all of the above features will give you a functions.php file like the one below.

Code comments have been added for future clarity.

As shown at the bottom of this example, you must add the required

[add_action\(\)](#) statement to ensure the myfirsttheme_setup function is loaded.

```
1 if ( ! function_exists( 'myfirsttheme_setup' ) ) :  
2 /**  
3  * Sets up theme defaults and registers support for  
4  various WordPress features.  
5  *  
6  * Note that this function is hooked into the  
7  after_setup_theme hook, which runs  
8  * before the init hook. The init hook is too late for  
9  some features, such as indicating  
10 * support post thumbnails.  
11 */  
12 function myfirsttheme_setup() {  
13  
14     /**  
15      * Make theme available for translation.  
16      * Translations can be placed in the /languages/  
17      directory.  
18      */  
19     load_theme_textdomain( 'myfirsttheme' ,  
20 get_template_directory() . '/languages' );  
21  
22     /**  
23      * Add default posts and comments RSS feed links to  
24      <head>.  
25      */  
26     add_theme_support( 'automatic-feed-links' );  
27  
28     /**  
29      * Enable support for post thumbnails and featured  
30      images.  
31      */  
32     add_theme_support( 'post-thumbnails' );  
33  
34     /**  
35      * Add support for two custom navigation menus.  
36      */  
37     register_nav_menus( array(  
38         'primary'    => ____( 'Primary Menu' ,  
39 'myfirsttheme' ),  
40         'secondary'  => ____( 'Secondary Menu' , 'myfirsttheme'  
41     )  
42     ) );
```

Content Width

A content width is added to your `functions.php` file to ensure that no content or assets break the container of the site. The content width sets the maximum allowed width for any content added to your site, including uploaded images. In the example below, the content area has a maximum width of 800 pixels. No content will be larger than that.

```
1 if ( ! isset ( $content_width ) )
2     $content_width = 800;
```

Other Features

There are other common features you can include in `functions.php`. Listed below are some of the most common features. Click through and learn more about each of these features.

(Expand this section based on new pages.)

- [Custom Headers](#)
- [Sidebars](#) (widget areas)
- Custom Background (needs link)
- Add Editor Styles (needs link)
- HTML5 (needs link)
- Title tag (needs link)

Your `functions.php` File

If you choose to include all of the functions listed above, this is what your `functions.php` might look like. It has been commented with references to above.

```
1 /**
2  * MyFirstTheme's functions and definitions
3  *
4  * @package MyFirstTheme
5  * @since MyFirstTheme 1.0
6  */
7
8 /**
9  * First, let's set the maximum content width based on the
10 theme's design and stylesheet.
11 * This will limit the width of all uploaded images and
12 embeds.
13 */
14 if ( ! isset( $content_width ) )
15     $content_width = 800; /* pixels */
16
17 if ( ! function_exists( 'myfirsttheme_setup' ) ) :
18 /**
19 * Sets up theme defaults and registers support for
20 various WordPress features.
21 *
22 * Note that this function is hooked into the
23 after_setup_theme hook, which runs
24 * before the init hook. The init hook is too late for
25 some features, such as indicating
26 * support post thumbnails.
27 */
28 function myfirsttheme_setup() {
29
30 /**
31 * Make theme available for translation.
32 * Translations can be placed in the /languages/
33 directory.
34 */
35     load_theme_textdomain( 'myfirsttheme',
36 get_template_directory() . '/languages' );
37
38 /**
39 * Add default posts and comments RSS feed links to
40 <head>.
41 */
42     add_theme_support( 'automatic-feed-links' );
43
```

Linking Theme Files & Directories

Linking to Core Theme Files

As you've learned, WordPress themes are built from a number of different template files.

At the very least this will usually include a `sidebar.php`, `header.php` and `footer.php`. These are called using [Template Tags](#), for example:

- `get_header();`
- `get_footer();`
- `get_sidebar();`

You can create custom versions of these files can be called as well by naming the file `sidebar-{your_custom_template}.php`, `header-{your_custom_template}.php` and `footer-{your_custom_template}.php`. You can then use Template Tags with the custom template name as the only parameter, like this:

```
1 get_header( 'your_custom_template' );
2 get_footer( 'your_custom_template' );
3 get_sidebar( 'your_custom_template' );
```

WordPress creates pages by assembling various files. Aside from the standard files for the header, footer and sidebar, you can create custom template files and call them at any location in the page using [get_template_part\(\)](#). To create a custom template file in your theme give the file an appropriate name and use the same custom template system as with the header, sidebar and footer files:

```
1 slug-template.php
```

For example, if you would like to create a custom template to handle your post content you could create a template file called `content.php` and then add a specific content layout for product content by extending the file name to `content-product.php`. You would then load this template file in your theme like this:

```
1 get_template_part( 'content', 'product' );
```

If you want to add more organization to your templates, you can place them in their own directories within your theme directory. For example, suppose you add a couple more

content templates for *profiles* and *locations*, and group them in their own directory called `content-templates`.

The theme hierarchy for your theme called `my-theme` might look like the following.

`style.css` and `page.php` are included for context.

- themes
- my-theme
- content-templates
- content-location.php
- content-product.php
- content-profile.php
- style.css

To include your content templates, prepend the directory names to the `slug` argument like this:

```
1 get_template_part( 'content-templates/content', 'location'
2 );
3 get_template_part( 'content-templates/content',
4   'product' );
5 get_template_part( 'content-templates/content',
6   'profile' );
```

Linking to Theme Directories

To link to the theme's directory, you can use the following function:

- `get_theme_file_uri()`

If you are not using a child theme, this function will return the full URI to your theme's main folder. You can use this to reference sub-folders and files in your theme like this:

```
1 echo get_theme_file_uri( 'images/logo.png' );
```

If you are using a child theme then this function will return the URI of the file in your child theme if it exists. If the file cannot be found in your child theme, the function will return the URI of the file in the parent theme. This is particularly important to keep in mind when distributing a theme or in any other case where a child theme may or may not be active.

To access the path to a file in your theme's directories, you can use the following function:

- `get_theme_file_path()`

Like `get_theme_file_uri()`, this will access the path of the file in the child theme if it exists. If the file cannot be found in the child theme, the function will access the path to the file in the parent theme.

In a child theme, you can link to a file URI or path in the parent theme's directories using the following functions:

- `get_parent_theme_file_uri()`;
- `get_parent_theme_file_path()`;

As with `get_theme_file_uri()`, you can reference sub-folders and files like this:

```
1 echo get_parent_theme_file_uri( 'images/logo.png' );
2 //or
3 echo get_parent_theme_file_path( 'images/logo.png' );
```

Take care when referencing files that may not be present, as these functions will return the URI or file path whether the file exists or not. If the file is missing, these functions will return a broken link.

Note:

The functions `get_theme_file_uri()`, `get_theme_file_path()`, `get_parent_theme_file_uri()`, `get_parent_theme_file_path()` were introduced in WordPress 4.7.

For previous WordPress versions, use `get_template_directory_uri()`, `get_template_directory()`, `get_stylesheet_directory_uri()`, `get_stylesheet_directory()`.

Take note that the newer 4.7 functions run the older functions anyway as part of the checking process so it makes sense to use the newer functions when possible.

Dynamic Linking in Templates

Regardless of your permalink settings, you can link to a page or post dynamically by referring to its unique numerical ID (seen in several pages in the admin interface) with

```
1 <a href="php echo get_permalink($ID); ?">This is a
link</a>
```

This is a convenient way to create page menus as you can later change page slugs without breaking links, as IDs will stay the same. However, this might increase database queries.

Including CSS & JavaScript

When you're creating your theme, you may want to create additional stylesheets or JavaScript files. However, remember that a WordPress website will not just have your theme active, it will also be using many different plugins. So that everything works harmoniously, it's important that theme and plugins load scripts and stylesheets using the standard WordPress method. This will ensure the site remains efficient and that there are no incompatibility issues.

Adding scripts and styles to WordPress is a fairly simple process. Essentially, you will create a function that will enqueue all of your scripts and styles. When enqueueing a script or stylesheet, WordPress creates a handle and path to find your file and any dependencies it may have (like jQuery) and then you will use a hook that will insert your scripts and stylesheets.

Enqueuing Scripts and Styles

The proper way to add scripts and styles to your theme is to enqueue them in the `functions.php` files. The `style.css` file is required in all themes, but it may be necessary to add other files to extend the functionality of your theme.

Tip:

WordPress includes a number of JavaScript files as part of the software package, including commonly used libraries such as jQuery. Before adding your own JavaScript, [check to see if you can make use of an included library](#).

The basics are:

1. Enqueue the script or style using `wp_enqueue_script()` or `wp_enqueue_style()`

Stylesheets

Your CSS stylesheets are used to customize the presentation of your theme. A stylesheet is also the file where information about your theme is stored. For this reason, the `style.css` file is **required in every theme**.

Rather than loading the stylesheet in your header `.php` file, you should load it in using `wp_enqueue_style`. In order to load your main stylesheet, you can enqueue it in `functions.php`

To enqueue `style.css`

```
1 wp_enqueue_style( 'style', get_stylesheet_uri() );
```

This will look for a stylesheet named “style” and load it.

The basic function for enqueueing a style is:

```
1 wp_enqueue_style( $handle, $src, $deps, $ver, $media );
```

You can include these parameters:

- **\$handle** is simply the name of the stylesheet.
- **\$src** is where it is located. The rest of the parameters are optional.
- **\$deps** refers to whether or not this stylesheet is dependent on another stylesheet. If this is set, this stylesheet will not be loaded unless its dependent stylesheet is loaded first.
- **\$ver** sets the version number.
- **\$media** can specify which type of media to load this stylesheet in, such as ‘all’, ‘screen’, ‘print’ or ‘handheld.’

So if you wanted to load a stylesheet named “slider.css” in a folder named “CSS” in your theme’s root directory, you would use:

```
1 wp_enqueue_style( 'slider', get_template_directory_uri() .  
'css/slider.css', false, '1.1', 'all' );
```

Scripts

Any additional JavaScript files required by a theme should be loaded using `wp_enqueue_script`. This ensures proper loading and caching, and allows the use of conditional tags to target specific pages. These are **optional**.

`wp_enqueue_script` uses a similar syntax to `wp_enqueue_style`.

Enqueue your script:

```
1 wp_enqueue_script( $handle, $src, $deps, $ver,  
$in_footer);
```

- **\$handle** is the name for the script.
- **\$src** defines where the script is located.
- **\$deps** is an array that can handle any script that your new script depends on, such as jQuery.
- **\$ver** lets you list a version number.

- `$in_footer` is a boolean parameter (true/false) that allows you to place your scripts in the footer of your HTML document rather than in the header, so that it does not delay the loading of the DOM tree.

Your enqueue function may look like this:

```
1 wp_enqueue_script( 'script', get_template_directory_uri() .  
  '/js/script.js', array ( 'jquery' ), 1.1, true);
```

The Comment Reply Script

WordPress comments have quite a bit of functionality in them right out of the box, including threaded comments and enhanced comment forms. In order for comments to work properly, they require some JavaScript. However, since there are certain options that need to be defined within this JavaScript, the comment-reply script should be added to every theme that uses comments.

The proper way to include comment reply is to use conditional tags to check if certain conditions exist, so that the script isn't being loaded unnecessarily. For instance, you can only load scripts on single post pages using `is_singular`, and check to make sure that "Enable threaded comments" is selected by the user. So you can set up a function like:

```
1 if ( is_singular() && comments_open() &&  
2 get_option( 'thread_comments' ) ) {  
3   wp_enqueue_script( 'comment-reply' );  
}
```

If comments are enabled by the user, and we are on a post page, then the comment reply script will be loaded. Otherwise, it will not.

Combining Enqueue Functions

It is best to combine all enqueued scripts and styles into a single function, and then call them using the `wp_enqueue_scripts` action. This function and action should be located somewhere below the initial setup (performed above).

```

1 function add_theme_scripts() {
2     wp_enqueue_style( 'style', get_stylesheet_uri() );
3
4     wp_enqueue_style( 'slider', get_template_directory_uri()
5 . '/css/slider.css', array(), '1.1', 'all' );
6
7     wp_enqueue_script( 'script',
8 get_template_directory_uri() . '/js/script.js', array
9 ( 'jquery' ), 1.1, true );
1
0     if ( is_singular() && comments_open() &&
1 get_option( 'thread_comments' ) ) {
1         wp_enqueue_script( 'comment-reply' );
1     }
2 }
add_action( 'wp_enqueue_scripts', 'add_theme_scripts' );

```

[Top ↑](#)

Default Scripts Included and Registered by WordPress

By default, WordPress includes many popular scripts commonly used by web developers, as well as the scripts used by WordPress itself. Some of them are listed in the table below:

Script Name	Handle	Needed Dependency *
Image Cropper	Image cropper (not used in core, see jcrop)	
Jcrop	jcrop	
SWFObject	swfobject	
SWFUpload	swfupload	
SWFUpload Degrade	swfupload-degrade	
SWFUpload Queue	swfupload-queue	

SWFUpload Handlers	swfupload-handlers	
jQuery	jquery	json2 (for AJAX calls)
jQuery Form	jquery-form	jquery
jQuery Color	jquery-color	jquery
jQuery Masonry	jquery-masonry	jquery
jQuery UI Core	jquery-ui-core	jquery
jQuery UI Widget	jquery-ui-widget	jquery
jQuery UI Mouse	jquery-ui-mouse	jquery
jQuery UI Accordion	jquery-ui-accordion	jquery
jQuery UI Autocomplete	jquery-ui-autocomplete	jquery
jQuery UI Slider	jquery-ui-slider	jquery
jQuery UI Progressbar	jquery-ui-progressbar	jquery
jQuery UI Tabs	jquery-ui-tabs	jquery
jQuery UI Sortable	jquery-ui-sortable	jquery
jQuery UI Draggable	jquery-ui-draggable	jquery
jQuery UI Droppable	jquery-ui-droppable	jquery
jQuery UI Selectable	jquery-ui-selectable	jquery

jQuery UI Position	jquery-ui-position	jquery
jQuery UI Datepicker	jquery-ui-datepicker	jquery
jQuery UI Tooltips	jquery-ui-tooltip	jquery
jQuery UI Resizable	jquery-ui-resizable	jquery
jQuery UI Dialog	jquery-ui-dialog	jquery
jQuery UI Button	jquery-ui-button	jquery
jQuery UI Effects	jquery-effects-core	jquery
jQuery UI Effects – Blind	jquery-effects-blind	jquery-effects-core
jQuery UI Effects – Bounce	jquery-effects-bounce	jquery-effects-core
jQuery UI Effects – Clip	jquery-effects-clip	jquery-effects-core
jQuery UI Effects – Drop	jquery-effects-drop	jquery-effects-core
jQuery UI Effects – Explode	jquery-effects-explode	jquery-effects-core
jQuery UI Effects – Fade	jquery-effects-fade	jquery-effects-core
jQuery UI Effects – Fold	jquery-effects-fold	jquery-effects-core
jQuery UI Effects – Highlight	jquery-effects-highlight	jquery-effects-core

jQuery UI Effects – Pulsate	jquery-effects-pulsate	jquery-effects-core
jQuery UI Effects – Scale	jquery-effects-scale	jquery-effects-core
jQuery UI Effects – Shake	jquery-effects-shake	jquery-effects-core
jQuery UI Effects – Slide	jquery-effects-slide	jquery-effects-core
jQuery UI Effects – Transfer	jquery-effects-transfer	jquery-effects-core
MediaElement.js (WP 3.6+)	wp-mediaelement	jquery
jQuery Schedule	schedule	jquery
jQuery Suggest	suggest	jquery
ThickBox	thickbox	
jQuery HoverIntent	hoverIntent	jquery
jQuery Hotkeys	jquery-hotkeys	jquery
Simple AJAX Code-Kit	sack	
QuickTags	quicktags	
Iris (Colour picker)	iris	jquery
Farbtastic (deprecated)	farbtastic	jquery

<u>ColorPicker (deprecated)</u>	colorpicker	jquery
<u>Tiny MCE</u>	tiny_mce	
Autosave	autosave	
WordPress AJAX Response	wp-ajax-response	
List Manipulation	wp-lists	
WP Common	common	
WP Editor	editorremov	
WP Editor Functions	editor-functions	
AJAX Cat	ajaxcat	
Admin Categories	admin-categories	
Admin Tags	admin-tags	
Admin custom fields	admin-custom-fields	
Password Strength Meter	password-strength-meter	
Admin Comments	admin-comments	
Admin Users	admin-users	
Admin Forms	admin-forms	
XFN	xfn	

Upload	upload
PostBox	postbox
Slug	slug
Post	post
Page	page
Link	link
Comment	comment
Threaded Comments	comment-reply
Admin Gallery	admin-gallery
Media Upload	media-upload
Admin widgets	admin-widgets
Word Count	word-count
Theme Preview	theme-preview
JSON for JS	json2
Plupload Core	plupload
Plupload All Runtimes	plupload-all
Plupload HTML4	plupload-html4
Plupload HTML5	plupload-html5

Plupload Flash	plupload-flash
Plupload Silverlight	plupload-silverlight
Underscore.js	underscore
Backbone.js	backbone

Removed from Core			
Script Name	Handle	Removed Version	Replaced With
Scriptaculous Root	scriptaculous-root	WP 3.5	Google Version
Scriptaculous Builder	scriptaculous-builder	WP 3.5	Google Version
Scriptaculous Drag & Drop	scriptaculous-dragdrop	WP 3.5	Google Version
Scriptaculous Effects	scriptaculous-effects	WP 3.5	Google Version
Scriptaculous Slider	scriptaculous-slider	WP 3.5	Google Version
Scriptaculous Sound	scriptaculous-sound	WP 3.5	Google Version
Scriptaculous Controls	scriptaculous-controls	WP 3.5	Google Version
Scriptaculous	scriptaculous	WP 3.5	Google Version

Prototype Framework	prototype	WP 3.5	Google Version
-------------------------------------	-----------	--------	----------------

The list is far from complete. You can find a full list of included files in [wp-includes/script-loader.php](#).

Conditional Tags

Conditional Tags can be used in your Template Files to alter the display of content depending on the conditions that the current page matches. They tell WordPress what code to display under specific conditions. Conditional Tags usually work with PHP [if](#) / [else](#) Conditional Statements.

The code begins by checking to see **if** a statement is true or false. If the statement is found to be true, the first set of code is executed. If it's false, the first set of code is skipped, and the second set of code (after the **else**) is executed instead.

For example, you could ask if a user is logged in, and then provide a different greeting depending on the result.

```

1 if ( is_user_logged_in() ):
2     echo 'Welcome, registered user!';
3 else:
4     echo 'Welcome, visitor!';
5 endif;

```

Note the close relation these tags have to [WordPress Template Hierarchy](#). [to be added, or removed?]

Where to Use Conditional Tags

For a Conditional Tag to modify your data, the information must already have been retrieved from your database, i.e. the query must have already run. If you use a Conditional Tag before there is data, there'll be nothing to ask the if/else statement about. It's important to note that WordPress loads **functions.php** before the query is run, so if you simply include a Conditional Tag in that file, it won't work.

Two ways to implement Conditional Tags:

- place it in a [Template File](#)
- create a function out of it in **functions.php** that hooks into an action/filter that triggers at a later point

The Conditions For

Listed below are the conditions under which each of the following **conditional statements proves to be true**. Tags which can accept parameters are noted.

The Main Page

is_home()

This condition returns true when the main blog page is being displayed, usually in standard reverse chronological order. If your home page has been set to a Static Page instead, then this will only prove true on the page which you set as the “Posts page” in Settings > Reading.

The Front Page

is_front_page()

This condition returns true when the front page of the site is displayed, regardless of whether it is set to show posts or a static page.

Returns true when:

1. the main blog page is being displayed **and**
2. the Settings > Reading -> *Front page displays* option is set to *Your latest posts*

OR

1. when Settings > Reading -> *Front page displays* is set to *A static page* **and**
2. the Front Page value is the current Page being displayed.

The Administration Panels

is_admin()

This condition returns true when the Dashboard or the administration panels are being displayed.

A Single Post Page

is_single()

Returns true when any single Post (or attachment, or custom Post Type) is being displayed. This condition returns false if you are on a page.

```
is_single( '17' )
```

[is_single\(\)](#) can also check for certain posts by ID and other parameters. The above example proves true when Post 17 is being displayed as a single Post.

```
is_single( 'Irish Stew' )
```

Parameters include Post titles, as well. In this case, it proves true when the Post with title “Irish Stew” is being displayed as a single Post.

```
is_single( 'beef-stew' )
```

Proves true when the Post with Post Slug “beef-stew” is being displayed as a single Post.

```
is_single( array( 17, 'beef-stew', 'Irish Stew' ) )
```

Returns true when the single post being displayed is either post ID 17, or the post_name is “beef-stew”, or the post_title is “Irish Stew”.

```
is_single( array( 17, 19, 1, 11 ) )
```

Returns true when the single post being displayed is either post ID = 17, post ID = 19, post ID = 1 or post ID = 11.

```
is_single( array( 'beef-stew', 'pea-soup', 'chilli' ) )
```

Returns true when the single post being displayed is either the post_name “beef-stew”, post_name “pea-soup” or post_name “chilli”.

```
is_single( array( 'Beef Stew', 'Pea Soup', 'Chilli' ) )
```

Returns true when the single post being displayed is either the post_title is “Beef Stew”, post_title is “Pea Soup” or post_title is “Chilli”.

Note: This function does not distinguish between the post ID, post title, or post name. A post named “17” would be displayed if a post ID of 17 was requested. Presumably the same holds for a post with the slug “17”.

A Single Post, Page, or Attachment

[is_singular\(\)](#)

Returns true for any is_single, is_page, and is_attachment. It does allow testing for post types.

A Sticky Post

[is_sticky\(\)](#)

Returns true if the “Stick this post to the front page” check box has been checked for the current post. In this example, no post ID argument is given, so the post ID for the Loop post is used.

`is_sticky('17')`

Returns true when Post 17 is considered a sticky post.

A Post Type

[get_post_type\(\)](#)

You can test to see if the current post is of a certain type by including [get_post_type\(\)](#) in your conditional. It's not really a conditional tag, but it returns the [registered post type](#) of the current post.

`if ('book' == get_post_type()) ...`

[post_type_exists\(\)](#)

Returns true if a given post type is a registered post type. This does not test if a post is a certain post_type. Note: This function replaces a function called is_post_type which existed briefly in 3.0 development.

A Post Type is Hierarchical

[is_post_type_hierarchical\(\\$post_type\)](#)

Returns true if this \$post_type has been set with hierarchical support when registered.

`is_post_type_hierarchical('book')`

Returns true if the book post type was registered as having support for hierarchical.

A Post Type Archive

[is_post_type_archive\(\)](#)

Returns true on any post type archive.

`is_post_type_archive($post_type)`

Returns true if on a post type archive page that matches \$post_type (can be a single post type or an array of post types).

To turn on post type archives, use 'has_archive' => true, when [registering the post type](#).

A Comments Popup

[is_comments_popup\(\)](#)

When in Comments Popup window.

Any Page Containing Posts

comments_open()

When comments are allowed for the current Post being processed in the [WordPress Loop](#).

pings_open()

When pings are allowed for the current Post being processed in the WordPress Loop.

A “PAGE” Page

This section refers to WordPress [Pages](#), not any generic webpage from your blog, or in other words to the built in *post_type* ‘page’.

is_page()

When any Page is being displayed.

`is_page('42')`

When Page 42 (ID) is being displayed.

`is_page('About Me And Joe')`

When the Page with a *post_title* of “About Me And Joe” is being displayed.

`is_page('about-me')`

When the Page with a *post_name* (slug) of “about-me” is being displayed.

`is_page(array(42, 'about-me', 'About Me And Joe'))`

Returns true when the Pages displayed is either *post ID* = 42, or *post_name* is “about-me”, or *post_title* is “About Me And Joe”.

`is_page(array(42, 54, 6))`

Returns true when the Pages displayed is either *post ID* = 42, or *post ID* = 54, or *post ID* = 6.

Testing for Paginated Pages

You can use this code to check whether you’re on the nth page in a Post or Page that has been divided into pages using the `<!--nextpage-->`

QuickTag. This can be useful, for example, if you wish to display meta-data only on the first page of a post divided into several pages.

Example 1

```
1 <?php
2     $paged = $wp_query->get( 'page' );
3     if ( ! $paged || $paged < 2 ) {
4         // This is not a paginated page (or it's simply the
5         first page of a paginated page/post) } else {
6         // This is a paginated page.
7     } ?>
```

Example 2

```
1 <?php $paged = get_query_var( 'page' ) ?
2 get_query_var( 'page' ) : false;
3     if ( $paged == false ) {
4         // This is not a paginated page (or it's simply the
5         first page of a paginated page/post) } else {
6         // This is a paginated page.
7     }
8 ?>
```

Testing for Sub-Pages

There is no `is_subpage()` function, but you can test this with a little code:

Snippet 1

```
1 <?php global $post; // if outside the loop
2     if ( is_page() && $post->post_parent ) {
3         // This is a subpage
4     } else {
5         // This is not a subpage
6     }
7 ?>
```

You can create your own `is_subpage()` function using the code in Snippet 1. Add it to your `functions.php` file. It tests for a parent page in the same way as Snippet 1, but will return the ID of the page parent if there is one, or false if there isn't.

Snippet 2

```

1 function is_subpage() {
2     global $post;                                // load
3 details about this page
4
5     if ( is_page() && $post->post_parent ) {    // test to
6 see if the page has a parent
7         return $post->post_parent;                // return
8 the ID of the parent post
9
10    } else {                                     // there is
no parent so ...
11        return false;                           // ... the
answer to the question is false
12    }
13}

```

It is advisable to use a function like that in Snippet 2, rather than using the simple test like Snippet 1, if you plan to test for sub-pages frequently.

To test if the parent of a page is a specific page, for instance “About” (page id pid 2 by default), we can use the tests in Snippet 3. These tests check to see if we are looking at the page in question, as well as if we are looking at any child pages. This is useful for setting variables specific to different sections of a web site, so a different banner image, or a different heading.

Snippet 3

```

1 <?php if ( is_page( 'about' ) || '2' == $post->post_parent
2 ) {
3     // the page is "About", or the parent of the page is
4 "About"
5     $bannerimg = 'about.jpg';
6 } elseif ( is_page( 'learning' ) || '56' == $post-
7 >post_parent ) {
8     $bannerimg = 'teaching.jpg';
9 } elseif ( is_page( 'admissions' ) || '15' == $post-
10 >post_parent ) {
11     $bannerimg = 'admissions.jpg';
} else {
    $bannerimg = 'home.jpg'; // just in case we are at an
unclassified page, perhaps the home page
}
?>

```

Snippet 4 is a function that allows you to carry out the tests above more easily. This function will return true if we are looking at the page in question (so “About”) or one of its sub pages (so a page with a parent with ID “2”).

Snippet 4

```
1 function is_tree( $pid ) {      // $pid = The ID of the
2   page we're looking for pages underneath
3   global $post;                  // load details about this
4   page
5
6   if ( is_page($pid) )
7     return true;                // we're at the page or at
8   a sub page
9
10  $anc = get_post_ancestors( $post->ID );
11  foreach ( $anc as $ancestor ) {
12    if( is_page() && $ancestor == $pid ) {
13      return true;
14    }
15  }
16
17  return false; // we aren't at the page, and the page
18  is not an ancestor
19 }
```

Add Snippet 4 to your functions.php file, and call `is_tree(‘id’)` to see if the current page is the page, or is a sub page of the page. In Snippet 3, `is_tree(‘2’)` would replace “`is_page(‘about’) || ‘2’ == $post->post_parent`” inside the first if tag.

Note that if you have more than one level of pages the parent page is the one directly above and not the one at the very top of the hierarchy.

Is a Page Template

Allows you to determine whether or not you are in a page template or if a specific page template is being used.

`is_page_template()`

Is a Page Template being used?

`is_page_template(‘about.php’)`

Is Page Template ‘about’ being used? Note that unlike other conditionals, if you want to specify a particular Page Template, you need to use the filename, such as about.php or my_page_template.php.

Note: if the file is in a subdirectory you must include this as well. Meaning that this should be the filepath in relation to your theme as well as the filename, for example ‘page-templates/about.php’.

A Category Page

is_category()

When a Category archive page is being displayed.

is_category(‘9’)

When the archive page for Category 9 is being displayed.

is_category(‘Stinky Cheeses’)

When the archive page for the Category with Name “Stinky Cheeses” is being displayed.

is_category(‘blue-cheese’)

When the archive page for the Category with Category Slug “blue-cheese” is being displayed.

is_category(array(9, ‘blue-cheese’, ‘Stinky Cheeses’))

Returns true when the category of posts being displayed is either term_ID 9, or slug “blue-cheese”, or name “Stinky Cheeses”.

in_category(‘5’)

Returns true if the current post is in the specified category id.

in_category(array(1, 2, 3))

Returns true if the current post is in either category 1, 2, or 3.

! in_category(array(4, 5, 6))

Returns true if the current post is NOT in either category 4, 5, or 6. Note the ! at the beginning.

Note: Be sure to check your spelling when testing. There’s a big difference between “is” or “in”.

See also is_archive() and Category Templates.

A Tag Page

is_tag()

When any Tag archive page is being displayed.

`is_tag('mild')`

When the archive page for tag with the slug of ‘mild’ is being displayed.

`is_tag(array('sharp', 'mild', 'extreme'))`

Returns true when the tag archive being displayed has a slug of either “sharp”, “mild”, or “extreme”.

[has_tag\(\)](#)

When the current post has a tag. Must be used inside The Loop.

`has_tag('mild')`

When the current post has the tag ‘mild’.

`has_tag(array('sharp', 'mild', 'extreme'))`

When the current post has any of the tags in the array.

See also [is_archive\(\)](#) and [Tag Templates](#).

A Taxonomy Page

[is_tax\(\)](#)

When any Taxonomy archive page is being displayed.

`is_tax('flavor')`

When a Taxonomy archive page for the flavor taxonomy is being displayed.

`is_tax('flavor', 'mild')`

When the archive page for the flavor taxonomy with the slug of ‘mild’ is being displayed.

`is_tax('flavor', array('sharp', 'mild', 'extreme'))`

Returns true when the flavor taxonomy archive being displayed has a slug of either “sharp”, “mild”, or “extreme”.

[has_term\(\)](#)

Check if the current post has any of given terms. The first parameter should be an empty string. It expects a taxonomy slug/name as a second parameter.

`has_term('green', 'color')`

When the current post has the term ‘green’ from taxonomy ‘color’.

`has_term(array('green', 'orange', 'blue'), 'color')`

When the current post has any of the terms in the array.

See also [is_archive\(\)](#).

A Registered Taxonomy

[taxonomy_exists\(\)](#)

When a particular taxonomy is registered via [register_taxonomy\(\)](#). Formerly [is_taxonomy\(\)](#), which was deprecated in Version 3.0

An Author Page

[is_author\(\)](#)

When any Author page is being displayed.

`is_author('4')`

When the archive page for Author number (ID) 4 is being displayed.

`is_author('Vivian')`

When the archive page for the Author with Nickname “Vivian” is being displayed.

`is_author('john-jones')`

When the archive page for the Author with Nicename “john-jones” is being displayed.

`is_author(array(4, 'john-jones', 'Vivian'))`

When the archive page for the author is either user ID 4, or user_nicename “john-jones”, or nickname “Vivian”.

See also [is_archive\(\)](#) and [Author Templates](#).

A Multi-author Site

[is_multi_author\(\)](#)

When more than one author has published posts for a site. Available with Version 3.2.

A Date Page

[is_date\(\)](#)

When any date-based archive page is being displayed (i.e. a monthly, yearly, daily or time-based archive).

[is_year\(\)](#)

When a yearly archive is being displayed.

[is_month\(\)](#)

When a monthly archive is being displayed.

[is_day\(\)](#)

When a daily archive is being displayed.

[is_time\(\)](#)

When an hourly, “minutely”, or “secondly” archive is being displayed.

is_new_day()

If today is a new day according to post date. Should be used inside the loop.

Any Archive Page

is_archive()

When any type of Archive page is being displayed. Category, Tag, Author and Date based pages are all types of Archives.

A Search Result Page

is_search()

When a search result page archive is being displayed.

A 404 Not Found Page

is_404()

When a page displays after an “HTTP 404: Not Found” error occurs.

An Attachment

is_attachment()

When an attachment document to a post or Page is being displayed. An attachment is an image or other file uploaded through the post editor’s upload utility. Attachments can be displayed on their own ‘page’ or template.

A Single Page, Single Post or Attachment

is_singular()

When any of the following return true: `is_single()`, `is_page()` or `is_attachment()`.

`is_singular(‘book’)`

True when viewing a post of the Custom Post Types book.

`is_singular(array(‘newspaper’, ‘book’))`

True when viewing a post of the Custom Post Types newspaper or book.

A Syndication

is_feed()

When the site requested is a Syndication. This tag is not typically used by users; it is used internally by WordPress and is available for Plugin Developers.

A Trackback

is_trackback()

When the site requested is WordPress' hook into its Trackback engine. This tag is not typically used by users; it is used internally by WordPress and is available for Plugin Developers.

A Preview

is_preview()

When a single post being displayed is viewed in Draft mode.

Has An Excerpt

has_excerpt()

When the current post has an excerpt

has_excerpt(42)

When the post 42 (ID) has an excerpt.

```
1 <?php // Get $post if you're inside a function global
2 $post;
3     if ( empty( $post->post_excerpt ) ) {
4         // This post has no excerpt
5     } else {
6         // This post has excerpt
7     }
?>
```

Other use

When you need to hide the auto displayed excerpt and only display your post's excerpts.

```
1 <?php
2     if ( ! has_excerpt() ) {
3         echo '';
4     } else {
5         the_excerpt();
6     }
7 ?>
```

Replace auto excerpt for a text or code.

```
1 <?php if ( ! has_excerpt() ) {
2     // your text or code
3 } ?>
```

Has A Nav Menu Assigned

has_nav_menu()

Whether a registered nav menu location has a menu assigned

Returns: assigned(true) or not(false)

Inside The Loop

in_the_loop()

Check to see if you are “inside the loop”. Useful for plugin authors, this conditional returns as true when you are inside the loop.

Is Sidebar Active

is_active_sidebar()

Check to see if a given sidebar is active (in use). Returns true if the sidebar (identified by name, id, or number) is in use, otherwise the function returns false.

Part of a Network (Multisite)

is_multisite()

Check to see whether the current site is in a WordPress MultiSite install.

Main Site (Multisite)

is_main_site()

Determines if a site is the main site in a network.

Admin of a Network (Multisite)

is_super_admin()

Determines if a user is a network (super) admin.

An Active Plugin

is_plugin_active()

Checks if a plugin is activated.

A Child Theme

is_child_theme()

Checks whether a child theme is in use.

Theme supports a feature

current_theme_supports()

Checks if various theme features exist.

Working Examples

Here are working samples to demonstrate how to use these conditional tags.

Single Post

This example shows how to use `is_single()` to display something specific only when viewing a single post page:

```
1 if ( is_single() ) {  
2     echo 'This is just one of many fabulous entries in the  
3     ' . single_cat_title() . ' category!';  
4 }  
5 }
```

Another example of how to use Conditional Tags in the Loop. Choose to display content or excerpt in index.php when this is a display single post or the home page.

```
1 if ( is_home() || is_single() ) {  
2     the_content();  
3 }  
4 else {  
5     the_excerpt();  
6 }  
7  
8 }  
9  
10 }
```

When you need display a code or element, in a place that is NOT the home page.

```
1 <?php if ( ! is_home() ) {  
2     // Insert your markup ...  
3 } ?>
```

Check for Multiple Conditionals

You can use [PHP operators](#) to evaluate multiple conditionals in a single if statement.

This is handy if you need to check whether combinations of conditionals evaluate to true or false.

```
1 // Check to see if any of 2 conditionals are met  
2 if ( is_single() || is_page() ) {  
3     // If it's a single post or a single page, do something  
4     special  
5 }  
6  
7 if ( is_archive() && ! is_category( 'nachos' ) ) {  
8     // If it's an archive page for any category EXCEPT  
9     nachos, do something special  
10 }
```

```
1 // Check to see if 3 conditionals are met
2 if ( $query->is_main_query() &&
3 is_post_type_archive( 'products' ) && ! is_admin() ) {
4
5 // If it's the main query on a custom post type archive
6 for Products
7 // And if we're not in the WordPress admin, then do
8 something special
9
10 }
11 if ( is_post_type_archive( 'movies' ) || is_tax( 'genre' )
12 || is_tax( 'actor' ) ) {
    // If it's a custom post type archive for Movies
    // Or it's a taxonomy archive for Genre
    // Or it's a taxonomy archive for Actor, do something
    special
}
```

Date Based Differences

If someone browses our site by date, let's distinguish posts in different years by using different colors:

```

1 <?php // this starts The Loop
2 if ( have_posts() ) : while ( have_posts() ) :
3 the_post(); ?>
4 <h2 id="post-<?php the_ID(); ?>">
5 <a href="<?php the_permalink() ?>" rel="bookmark"><?php
6 the_title(); ?></a></h2>
7
8 <small><?php the_time('F jS, Y') ?> by <?php the_author()
9 ?></small>
10
11 <?php
12 // are we showing a date-based archive?
13 if ( is_date() ) {
14 if ( date( 'Y' ) != get_the_date( 'Y' ) ) {
15 // this post was written in a previous year
16 // so let's style the content using the "oldentry" class
17 echo '<div class="oldentry">';
18 } else {
19 echo '<div class="entry">';
20 }
21 } else {
22 echo '<div class="entry">';
23 }

the_content( 'Read the rest of this entry »' );
?></div>
```

Variable Sidebar Content

This example will display different content in your sidebar based on what page the reader is currently viewing.

```

1 <div id="sidebar">
2 <?php // let's generate info appropriate to the page being
3 displayed
4 if ( is_home() ) {
5     // we're on the home page, so let's show a list of all
6     top-level categories
7     wp_list_categories( 'optionall=0&sort_column=name&
8 amp;list=1&children=0' );
9 } elseif ( is_category() ) {
10    // we're looking at a single category view, so let's show
11    _all_ the categories
12    wp_list_categories( 'optionall=1&sort_column=name&
13 amp;list=1&children=1&hierarchical=1' )
14 } elseif ( is_single() ) {
15    // we're looking at a single page, so let's not show
16    anything in the sidebar
17 } elseif ( is_page() ) {
18    // we're looking at a static page. Which one?
19    if ( is_page( 'About' ) ) {
20        // our about page.
21        echo "This is my about page!";
22    } elseif ( is_page( 'Colophon' ) ) {
23        echo "This is my colophon page, running on WordPress " .
24        bloginfo( 'version' ) . "";
25    } else {
26        // catch-all for other pages
27        echo "Vote for Pedro!";
28    }
29 } else {
30     // catch-all for everything else (archives, searches,
31     404s, etc)
32     echo "Pedro offers you his protection.";
33 } // That's all, folks!
34 ?>
35 </div>

```

Helpful 404 Page

The Creating an Error 404 Page article [NEED Link to this?] has an example of using the PHP conditional function `isset()` in the Writing Friendly Messages section.

In the theme's footer.php file

At times queries performed in other templates such as sidebar.php may corrupt certain conditional tags. For instance, in header.php a conditional tag works properly but doesn't work in a theme's footer.php. The trick is to put wp_reset_query before the conditional test in the footer. For example:

```
1 <?php wp_reset_query();  
2 if ( is_page( '2' ) ) {  
3 echo 'This is page 2!';  
4 }  
5 ?>
```

Conditional Tags Index

- [comments_open](#)
- [has_tag](#)
- [has_term](#)
- [in_category](#)
- [is_404](#)
- [is_admin](#)
- [is_archive](#)
- [is_attachment](#)
- [is_author](#)
- [is_category](#)
- [is_child_theme](#)
- [is_comments_popup](#)
- [is_date](#)
- [is_day](#)
- [is_feed](#)
- [is_front_page](#)
- [is_home](#)
- [is_month](#)
- [is_multi_author](#)
- [is_multisite](#)
- [is_main_site](#)
- [is_page](#)
- [is_page_template](#)
- [is_paged](#)
- [is_preview](#)
- [is rtl](#)

- [is_search](#)
- [is_single](#)
- [is_singular](#)
- [is_sticky](#)
- [is_super_admin](#)
- [is_tag](#)
- [is_tax](#)
- [is_time](#)
- [is_trackback](#)
- [is_year](#)
- [pings_open](#)

Function Reference

- Function: [comments_open\(\)](#)
- Function: [is_404\(\)](#)
- Function: [is_admin\(\)](#)
- Function: [is_admin_bar_showing\(\)](#)
- Function: [is_archive\(\)](#)
- Function: [is_attachment\(\)](#)
- Function: [is_author\(\)](#)
- Function: [is_category\(\)](#)
- Function: [is_comments_popup\(\)](#)
- Function: [is_date\(\)](#)
- Function: [is_day\(\)](#)
- Function: [is_feed\(\)](#)
- Function: [is_front_page\(\)](#)
- Function: [is_home\(\)](#)
- Function: [is_local_attachment\(\)](#)
- Function: [is_main_query](#)
- Function: [is_multi_author](#)
- Function: [is_month\(\)](#)
- Function: [is_new_day\(\)](#)
- Function: [is_page\(\)](#)
- Function: [is_page_template\(\)](#)
- Function: [is_paged\(\)](#)
- Function: [is_plugin_active\(\)](#)
- Function: [is_plugin_active_for_network\(\)](#)
- Function: [is_plugin_inactive\(\)](#)
- Function: [is_plugin_page\(\)](#)
- Function: [is_post_type_archive\(\)](#)
- Function: [is_preview\(\)](#)

- Function: `is_search()`
- Function: `is_single()`
- Function: `is_singular()`
- Function: `is_sticky()`
- Function: `is_tag()`
- Function: `is_tax()`
- Function: `is_taxonomy_hierarchical()`
- Function: `is_time()`
- Function: `is_trackback()`
- Function: `is_year()`
- Function: `in_category()`
- Function: `in_the_loop()`
- Function: `is_active_sidebar()`
- Function: `is_active_widget()`
- Function: `is_blog_installed()`
- Function: `is rtl()`
- Function: `is_dynamic_sidebar()`
- Function: `is_user_logged_in()`
- Function: `has_excerpt()`
- Function: `has_post_thumbnail()`
- Function: `has_tag()`
- Function: `pings_open()`
- Function: `email_exists()`
- Function: `post_type_exists()`
- Function: `taxonomy_exists()`
- Function: `term_exists()`
- Function: `username_exists()`
- Function: `wp_attachment_is_image()`
- Function: `wp_script_is()`

Categories, Tags, & Custom Taxonomies

Categories, tags, and taxonomies are all related and can be easily confused.

We'll use the example of building a theme for a recipe website to help break down categories, tags, and taxonomies.

In our recipe website, the **categories** would be Breakfast, Lunch, Dinner, Appetizers, Soups, Salads, Sides, and Desserts. All recipes will fit within those categories, but users

might want to search for something specific like chocolate desserts or ginger chicken dinners.

Chocolate, ginger, and chicken are all examples of **tags**. They are another level of specificity that provides meaning to the user.

Lastly, there are taxonomies. In reality, categories and tags are examples of default taxonomies which simply are a way to organize content. Taxonomies are the method of classifying content and data in WordPress. When you use a taxonomy you're grouping similar things together. The taxonomy refers to the sum of those groups. As with [Post Types](#), there are a number of default taxonomies, and you can also create your own. Recipes are normally organized by category and tag, but there are some other helpful ways to break the recipes down to be more user friendly. For example, the recipe website might want an easy way to display recipes by cook time. A custom taxonomy of cook time with 0-30 min, 30-min to an hour, 1 to 2 hours, 2+ hours would be a great breakdown. Additionally, cook method such as grill, oven, stove, refrigerator, etc would be another example of a custom taxonomy that would be relevant for the site. There could also be a custom taxonomy for how spicy the recipe is and then a rating from 1-5 on spiciness.

Default Taxonomies

The default taxonomies in WordPress are:

- categories: a hierarchical taxonomy that organizes content in the *post* Post Type
- tags: a non-hierarchical taxonomy that organizes content in the *post* Post Type
- post formats: a method for creating formats for your posts. You can learn more about these on the [Post Formats](#) page.

Terms

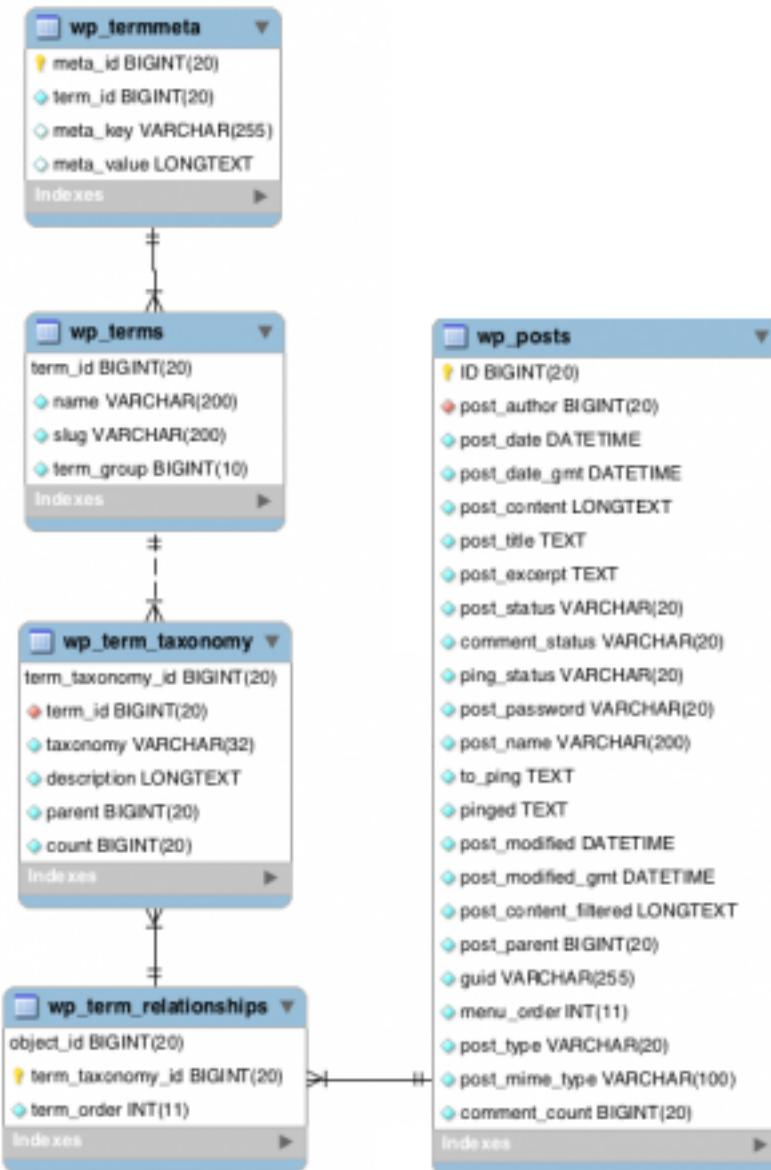
Terms are items within your taxonomy. So, for example, if you have the *Animal* taxonomy you would have the terms, dogs, cats, and sheep. Terms can be created via the WordPress admin, or you can use the [wp_insert_term\(\)](#) function.

Database Schema

Taxonomies and terms are stored in the following database tables:

- wp_terms – stores all of the terms
- wp_term_taxonomy – places the term in a taxonomy

- wp_term_relationships – relates the taxonomy to an object (for example, *category* to *post*)



Templates

WordPress offers several different hierarchies of templates for categories, tags, or custom taxonomies. More details on their structure and usage may be found on the [Taxonomy Templates](#) page.

Custom Taxonomies

It is possible to create new taxonomies in WordPress. You may, for example, want to create an *author* taxonomy on a book review website, or an *actor* taxonomy on a film site.

As with custom post type **it is recommended that you put this functionality in a plugin**. This ensures that when the user changes their website's design, their content is preserved in the plugin.

You can read more about creating custom taxonomies in the [Plugin Developer Handbook](#).

Template Files Section

We introduced [template files](#) earlier in the handbook. This next section is going to break up the different template files through the [post types](#) that they display and provide a more in depth explanation of the purpose for these template files. You'll also get a handle on some practical use cases for the different template files.

You'll start with [post template files](#) that deal with displaying the Post post type. Then you'll look at [page template files](#) that display the Page post type and drill down into [specific page templates](#) that WordPress has built in to its functionality. Next you'll learn about [attachment template files](#) which display the attachment post type. You'll also learn about how to display custom post types that are built by a plugin in the [custom post type templates](#). Lastly, you'll touch on some [partial and miscellaneous template files](#) which are important to include but don't necessarily display post types.

Note:

It's important to understand WordPress doesn't really break down template files down into specific types of template files such as post template files or attachment template files. Especially since many template files, such as `index.php` or `search.php` can display many different post types. To help you reference specific template files this handbook is categorizing them based on the post types they can display.

Post Template Files

There are many [template files](#) that WordPress uses to display the Post [post type](#). Any content dealing with a blog or its posts are within the Post post type.

Index.php

`index.php` will display Post post types if there is no other template file in place. As stated in many places, every theme must have an `index.php` file to be valid. Many

basic themes can get away with just using the `index.php` to display their Post post types, but the use cases given above would justify creating other template files. Often you will want unique content structure or layout depending on what is being displayed. There are many templates you can use to customize content structure based on the context within the site. The two most notable post template files are `home.php` and `single.php` which display a feed of posts and a single post respectively.

Home.php

When a static front page is used and the site has a page defined for the blog list the `home.php` file is used for the designated blog list page. Use of this template is encouraged over creating a custom page template because blog pagination on a custom page template will not work properly. If there is no `home.php` in the theme `index.php` will be used instead.

Single.php

It's good sense to build as simply as possible in your template structure and not make more templates unless you have real need for them. Therefore, most theme developers don't create a `single-post.php` file because `single.php` is specific enough. For the most part, all themes should have a `single.php`. Below is an example of a `single.php` file from the theme Twenty Fifteen.

```
1 <?php
2 /**
3  * The template for displaying all single posts and
4  * attachments
5  *
6  * @package WordPress
7  * @subpackage Twenty_Fifteen
8  * @since Twenty Fifteen 1.0
9  */
1
0 get_header(); ?>
1
1     <div id="primary" class="content-area">
1         <main id="main" class="site-main" role="main">
2
1             <?php
3             // Start the loop.
4             while ( have_posts() ) : the_post();
5
6                 /*
7                  * Include the post format-specific template
8                  * for the content. If you want to
9                  * use this in a child theme, then include a
10                 file called content-___.php
11                 * (where ___ is the post format) and that
12                 will be used instead.
13                 */
14
15                 get_template_part( 'content',
16                 get_post_format() );
17
18
19                 // If comments are open or we have at least
20                 one comment, load up the comment template.
21                 if ( comments_open() ||
22                 get_comments_number() ) :
23                     comments_template();
24                 endif;
25
26
27                 // Previous/next post navigation.
28                 the_post_navigation( array(
29                     'next_text' => '<span class="meta-nav"'
30                     aria-hidden="true">' . __( 'Next', 'twentyfifteen' ) . '</
31                     span> ' .
32
33                     '<span class="screen-reader-text">' .
```

In the code example above you can see the header is pulled in with `get_header()` then there are two html tags. Next `the Loop` starts and the `template tag` `get_template_part('content', get_post_format())`; pulls in the appropriate content by determining the post type with `get_post_format()`. Next, `comments` are pulled in with the template tag `comments_template()`. Then there is some `pagination`. Lastly, the content divs are closed and then footer is pulled in with `get_footer()`.

Singular.php

WordPress Version 4.3 added `singular.php` that comes in the hierarchy after `single.php` for posts, `page.php` for pages, and the variations of each. This template follows the rules of `is_singular()` and is used for a single post, regardless of post type. Themes that used the same code for both of those files (or included one in the other) can now simplify down to the one template.

Archive.php

Unless a developer includes meta data with permalinks in their templates, the `archive.php` will not be used. Meta data is information tied to the post. For example the date something was posted on, the author, and any `categories, tags, or taxonomies` used for the post are all examples of meta data. When a visitor to a website clicks on the meta data, the `archive.php` will render any posts associated with that piece of meta data. For example, if a visitor clicks on the name of an author, the `archive.php` will display all posts by that author.

Commonly, the title of the page being displayed by `archive.php` will be the name of the meta data the user clicked on. So if the user clicked on the Author's name, the page name displaying all the other author's posts will be the Author's name and frequently there might be an additional description about the meta data. Here is a code example

from Twenty Fifteen on their `archive.php` file. This snippet is the only piece of code that makes the `archive.php` file different from a `home.php` or `index.php` file.

```
1 <header class="page-header">
2     <?php
3         the_archive_title( '
4
5             <h1 class="page-title">', '</h1>
6
7             '
8         );
9         the_archive_description( '
10
11             <div class="taxonomy-description">', '</div>
12
13             '
14         );
15     ?>
16 </header>
17
18
19 <!-- .page-header -->
```

Author.php and Date.php

`Author.php` and `date.php` are more specific archive type files. If you need a refresher check out where they fit within the [template hierarchy](#). Generally, `archive.php` will suffice for most themes' needs and you won't need to create these templates.

Author.php

If you are building a theme designed for multiple authors, it might make sense to build an `author.php` template. In the `author.php` template you could provide more information about an author, their gravatar, pull in their social media sites, and then all posts written by them. This would be a step up from relying just on the `archive.php` file.

Additionally, you can build specific `author.php` files for individual author's by using their author ID or nicename. For example, say John Doe is the head author for a site with many guest authors. You may want all the guest authors' information to display with `author.php` but you might build a specific author page with more information for John Doe by creating `author-johndoe.php` or `author-3.php` if his author ID is 3.

Date.php

Similarly, if you are building a theme directed at magazine or news websites, a `date.php` file might make sense to build as these websites frequently organize their articles and posts by date or issue. Additionally, you could build a `day.php`, `month.php`, or `year.php` if you found enough justification for it.

Category.php, Tag.php, and Taxonomy.php

If you need a refresher on what [categories, tags, & taxonomies](#) are you can look at their page. Often you won't need to build out these template files. However, in an example of building a theme for food bloggers, there are some use cases for building these specific templates. In a food blogger website, the categories could be Great Restaurants, Beautiful Food, Ethnic Cuisine, and Recipes.

You might want most of your blog posts to display the same way except for any blogs that are categorized as recipes, because all recipes have ingredients and instructions sections. Therefore, you may want to build a `category-recipe.php` file to display your recipe blog posts in a grid view with some of the important details about the recipe visible.

Additionally, perhaps chocolate is a really important tag for the theme you're building. It might make sense to build a `tag-chocolate.php` file so that you can display a specialized banner image of chocolate.

Search.php

Most themes have a search.php file so it is clear to users that their query went through. It is common to have some sort of header identifying the query results such as this snippet found in twenty fifteen's theme.

```
1 <header class="page-header">
2
3
4 <h1 class="page-title"><?php printf( __(
5   'Search Results
6   for: %s',
7   'twentyfifteen' ),
8   get_search_query() );
9 ?></h1>
10
11 </header>
12
13
14 <!-- .page-header -->
```

This code snippet pulls in the query that was searched with `get_search_query()`. Often `search.php` will only pull in the excerpt instead of the full content since the user is trying to determine if the article or page fits their search.

Page Templates

Page templates are a specific type of [template file](#) that can be applied to a specific page or groups of pages.

Note:

As of WordPress 4.7 page templates support all post types. For more details how to set a page template to specific post types [see example below](#).

Since a page template is a specific type of template file, here are some distinguishing features of page templates:

- Page templates are used to change the look and feel of a page.
- A page template can be applied to a single page, a page section, or a class of pages.
- Page templates generally have a high level of specificity, targeting an individual page or group of pages. For example, a page template named `page-`

`about.php` is more specific than the template files `page.php` or `index.php` as it will only affect a page with the slug of “about.”

- If a page template has a template name, WordPress users editing the page have control over what template will be used to render the page.

Uses for Page Templates

Page templates display your site’s dynamic content on a page, e.g., posts, news updates, calendar events, media files, etc. You may decide that you want your homepage to look a specific way, that is quite different to other parts of your site. Or, you may want to display a featured image that links to a post on one part of the page, have a list of latest posts elsewhere, and use a custom navigation. You can use page templates to achieve these things.

This section shows you how to build page templates that can be selected by your users through their admin screens.

For example, you can build page templates for:

- full-width, one-column
- two-column with a sidebar on the right
- two-column with a sidebar on the left
- three-column

Page Templates within the Template Hierarchy

When a person browses to your website, WordPress selects which template to use for rendering that page. As we learned earlier in the [Template Hierarchy](#), WordPress looks for template files in the following order:

1. **Page Template** — If the page has a custom template assigned, WordPress looks for that file and, if found, uses it.
2. **page-{slug}.php** — If no custom template has been assigned, WordPress looks for and uses a specialized template that contains the page’s slug.
3. **page-{id}.php** — If a specialized template that includes the page’s slug is not found, WordPress looks for and uses a specialized template named with the page’s ID.
4. **page.php** — If a specialized template that includes the page’s ID is not found, WordPress looks for and uses the theme’s default page template.

5. `singular.php` – If `page.php` is not found, WordPress looks for and uses the theme's template used for a single post, regardless of post type.

6. `index.php` – If no specific page templates are assigned or found, WordPress defaults back to using the theme's index file to render pages.

Alert:

There is also a WordPress-defined template named `paged.php`. It is *not* used for the page post-type but rather for displaying multiple pages of archives.

Page Templates Purpose & User Control

If you plan on making a custom page template for your theme, you should decide a couple of things before proceeding:

- Whether the page template will be for one specific page or for any page; and
- What type of user control you want available for the template.

Every page template that has a template name can be selected by a user when they create or edit a page. The list of available templates can be found at **Pages > Add New > Attributes > Template**. Therefore, a WordPress user can choose any page template with a template name, which might not be your intention.

For example, if you want to have a specific template for your “About” page, it might not be appropriate to name that page template “About Template” as it would be globally available to all pages (i.e. the user could apply it to any page). Instead, [create a single use template](#) and WordPress will render the page with the appropriate template, whenever a user visits the “About” page.

Conversely, many themes include the ability to choose how many columns a page will have. Each of these options is a page template that is available globally. To give your WordPress users this global option, you will need to create page templates for each option and give each a template name.

Dictating whether a template is for global use vs. singular use is achieved by the way the file is named and whether or not it has a specific comment.

Note:

Sometimes it is appropriate to have a template globally available even if it appears to be a single use case. When you're creating themes for release, it can be hard to predict what a user will name their pages. Portfolio pages are a great example as not every WordPress

user will name their portfolio the same thing or have the same page ID and yet they may want to use that template.

File Organization of Page Templates

As discussed in [Organizing Theme Files](#), WordPress recognizes the subfolder **page-templates**. Therefore, it's a good idea to store your global page templates in this folder to help keep them organized.

Alert:

A specialized page template file (those created for only one time use) **cannot** be in a sub-folder, nor, if using a [Child Theme](#), in the Parent Theme's folder.

Creating Custom Page Templates for Global Use

Sometimes you'll want a template that can be used globally by any page, or by multiple pages. Some developers will group their templates with a filename prefix, such as `page_two-columns.php`

Alert:

Important! Do not use `page-` as a prefix, as WordPress will interpret the file as a specialized template, meant to apply to only *one* page on your site.

For information on theme file-naming conventions and filenames you cannot use, see [reserved theme filenames](#).

Tip:

A quick, safe method for creating a new page template is to make a copy of `page.php` and give the new file a distinct filename. That way, you start off with the HTML structure of your other pages and you can edit the new file as needed.

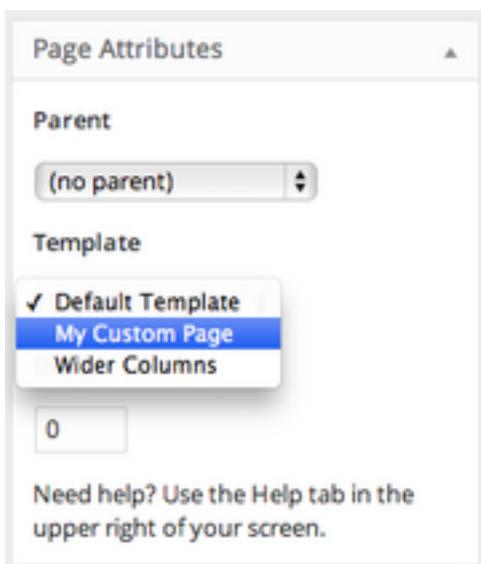
To create a global template, write an opening PHP comment at the top of the file that states the template's name.

```
1 <?php /* Template Name: Example Template */ ?>
```

It's a good idea to choose a name that describes what the template does as the name is visible to WordPress users when they are editing the page. For example, you could name your template Homepage, Blog, or Portfolio.

This example from the TwentyFourteen theme creates a page template called Full Width Page:

```
1 <?php
2 /**
3 * Template Name: Full Width Page
4 *
5 * @package WordPress
6 * @subpackage Twenty_Fourteen
7 * @since Twenty Fourteen 1.0
8 */
```



Once you upload the file to your theme's folder (e.g., page-templates), go to the **Page > Edit** screen in your admin dashboard.

On the right hand side under attributes you'll see template. This is where users are able to access your global page templates.

Tip:

The select list has a maximum width of 250px, so longer names may be cut off.

Creating a Custom Page Template for One Specific Page

As mentioned in the [Template Hierarchy](#) page, you can create a template for a specific page. To create a template for one specific page, copy your existing `page.php` file and rename it with your page's slug or ID:

1. `page-{slug}.php`
2. `page-{ID}.php`

For example: Your About page has a slug of ‘about’ and an ID of 6. If your active theme’s folder has a file named `page-about.php` or `page-6.php`, then WordPress will automatically find and use that file to render the About page.

To be used, specialized page templates must be in your theme’s folder (i.e. `/wp-content/themes/my-theme-name/`).

Creating page templates for specific post types

By default, a custom page template will be available to the “page” post type.

To create a page template to specific post types, add a line under the template name with the post types you would like the template to support.

Example:

```
1 <?php
2 /*
3 Template Name: Full-width layout
4 Template Post Type: post, page, event
5 */
6 // Page code here...
```

Alert:

This ability to add page templates to post types other than “page” post type is supported only from WordPress 4.7

When at least one template exists for a post type, the ‘Post Attributes’ meta box will be displayed in the back end, without the need to add post type support for ‘page-attributes’ or anything else. The ‘Post Attributes’ label can be customized per post type using the ‘attributes’ label when registering a post type.

Backward Compatibility:

Let's say you want to publicly release a theme with support for post type templates.

WordPress versions before 4.7 will ignore the Template Post Type header and show the template in the list of page templates, even though it only works for regular posts. To prevent that, you can hook into the theme_page_templates filter to exclude it from the list.

Here's an example:

```
1 /**
2 * Hides the custom post template for pages on
3 WordPress 4.6 and older
4 *
5 * @param array $post_templates Array of page
6 templates. Keys are filenames, values are
7 translated names.
8 * @return array Filtered array of page templates.
9 */
10 function
11 makewp_exclude_page_templates( $post_templates ) {
12 if ( version_compare( $GLOBALS['wp_version'],
13 '4.7', '<' ) ) {
14 unset( $post_templates['templates/my-full-width-
15 post-template.php'] );
16 }
17 return $post_templates;
18 }
19 add_filter( 'theme_page_templates',
20 'makewp_exclude_page_templates' );
```

That way you can support custom post type templates in WordPress 4.7 and beyond while maintaining full backward compatibility.

Note that theme_page_templates is actually a dynamic theme_{\$post_type}_templates filter. The dynamic portion of the hook name, \$post_type, refers to the post type supported by the templates. E.g. you can hook into theme_product_templates to filter the list of templates for the product post type.

Using Conditional Tags in Page Templates

You can make smaller, page-specific changes with [Conditional Tags](#) in your theme's `page.php` file. For instance, the below example code loads the file `header-home.php` for your front page, but loads another file (`header-about.php`) for your About page, and then applies the default `header.php` for all other pages.

```
1 if ( is_front_page() ) :  
2     get_header( 'home' );  
3 elseif ( is_page( 'About' ) ) :  
4     get_header( 'about' );  
5 else:  
6     get_header();  
7 endif;
```

[You can learn more about Conditional Tags here.](#)

Identifying a Page Template

If your template uses the [`body_class\(\)`](#) function, WordPress will print classes in the `body` tag for the post type class name (`page`), the page's ID (`page-id-{ID}`), and the page template used. For the default `page.php`, the class name generated is `page-template-default`:

```
1 <body class="page page-id-6 page-template-default">
```

Note:

A specialized template (`page-{slug}.php` or `page-{ID}.php`) also gets the `page-template-default` class rather than its own body class.

When using a custom page template, the class `page-template` will print, along with a class naming the specific template. For example, if your custom page template file is named as follows:

```
1 <?php  
2 /* Template Name: My Custom Page */  
3 ?gt;
```

Then the rendered HTML generated will be as follows:

```
1 <body class="page page-id-6 page-template page-  
template-my-custom-page-php">
```

Notice the `page-template-my-custom-page-php` class that is applied to the body tag.

Page Template Functions

These built-in WordPress functions and methods can help you work with page templates:

- [`get_page_template\(\)`](#) returns the path of the page template used to render the page.
- [`wp_get_theme\(\)->get_page_templates\(\)`](#) returns all custom page templates available to the currently active theme (`get_page_templates()` is a method of the [`WP_Theme`](#) class).
- [`is_page_template\(\)`](#) returns true or false depending on whether a custom page template was used to render the page.
- [`get_page_template_slug\(\)`](#) returns the value of custom field `_wp_page_template` (null when the value is empty or “default”). If a page has been assigned a custom template, the filename of that template is stored as the value of a [custom field](#) named '`_wp_page_template`' (in the [`wp_postmeta`](#) database table). (Custom fields starting with an underscore do not display in the edit screen’s custom fields module.)

Attachment Template Files

Attachments are a special post type that holds information about a file uploaded through the WordPress media upload system, such as its description and name, which can display through several **post type – attachment** template files.

For images, as an example, the attachment post type links to metadata information, about the size of the images, the thumbnails generated, the location of the image files, the HTML alt text, and even information obtained from EXIF data embedded in the images.

Tip:

Utilizing attachment templates to gain additional metadata information for uploads, help with SEO efforts.

As shown in the [template hierarchy](#), you are able to display your attachments through several template files in order of fallback:

1. MIME_type.php and a subtype.php
2. It can be any MIME type (For example: `image.php`, `video.php`, `application.php`). For `text/plain`, the following path is used (in order):
 1. `text_plain.php`
 2. `plain.php`
 3. `text.php`
3. `attachment.php`
4. `single-attachment.php`
5. `single.php`
6. `singular.php`
7. `index.php`

MIME_type.php

Attachments are served by template files based on their mime-type. As an example, if your attachment is an image, your can customize how they display through the creation of an `image.php` template file. All images with the `post_mime_type` of `image/*` will render though your `image.php` template file.

Attachments also support the use of a mime subtype .php file. To continue with the image example, you can further customize your theme to support not only an **image.php** file but a **jpg.php** subtype file.

Attachment.php

An attachment page (**attachment.php**) is a single post page with the post type of attachment, generated through the creation of an attachment.php. Just like a [single post page](#), which is dedicated to your article, the attachment page provides a dedicated page in attachments in your theme.

Creation of attachment page is as simple as creating an attachment.php file. The attachment.php file then contains code similar to the single.php post page.

```
1 <div class="entry-attachment">
2     <?php $image_size =
3         apply_filters( 'wporg_attachment_size', 'large' );
4             echo wp_get_attachment_image( get_the_ID(),
5         $image_size ); ?>
6
7         <?php if ( has_excerpt() ) : ?>
8
9             <div class="entry-caption">
10                 <?php the_excerpt(); ?>
11             </div><!-- .entry-caption -->
12         <?php endif; ?>
13     </div><!-- .entry-attachment -->
```

Function Reference

- [get_attachment_template\(\)](#) : Retrieve path of attachment template in current or parent template.

Custom Post Type Template Files

The WordPress theme system supports custom [templates](#) for custom post types. Custom templates for the single display of posts belonging to custom post types have been

supported since WordPress [Version 3.0](#) and the support for custom templates for archive displays was added in [Version 3.1](#).

Custom Post Type – Template Hierarchy

WordPress will work through the [template hierarchy](#) and use the template file it comes across first. So if you want to create a custom template for your `acme_product` custom post type, a good place to start is by copying the `single.php` file, saving it as `single-acme_product.php` and editing that.

However if you don't want to create custom template files, WordPress will use the files already present in your theme, which would be `archive.php` and `single.php` and `index.php` files.

Single posts and their archives can be displayed using the `single.php` and `archive.php` template files respectively,

- single posts of a custom post type will use `single-{post_type}.php`
- and their archives will use `archive-{post_type}.php`
- and if you don't have this post type archive page you can pass **BLOG_URL?**
`post_type={post_type}`

where `{post_type}` is the `$post_type` argument of the [register_post_type\(\)](#) function.

So for the above example, you could create `single-acme_product.php` and `archive-acme_product.php` template files for single product posts and their archives.

Alternatively, you can use the `is_post_type_archive()` function in any template file to check if the query shows an archive page of a given post types(s), and the `post_type_archive_title()` to display the post type title.

Custom Post Type templates

- `single-{post-type}.php`

- The single post template used when a visitor requests a single post from a custom post type. For example, `single-acme_product.php` would be used for displaying single posts from a custom post type named `acme_product`.
- `archive-{post-type}.php`
- The archive post type template is used when visitors request a custom post type archive. For example, `archive-acme_product.php` would be used for displaying an archive of posts from the custom post type named `acme_product`. The `archive.php` template file is used if the `archive-{post-type}.php` is not present.
- `search.php`
- The search results template is used to display a visitor's search results. To include search results from your custom post type, [refer to the code sample below](#).
- `index.php`
- The `index.php` is used if a specific query template (`single-{post-type}.php`, `single.php`, `archive-{post-type}.php`, `archive.php`, `search.php`) for the custom post type is not present.

Function Reference

- [`register_post_type\(\)`](#) : Registers a post type.
- [`is_post_type_archive\(\)`](#) : Checks if the query for an existing post type archive page.
- [`post_type_archive_title\(\)`](#) : Display or retrieve title for a post type archive.

Partial and Miscellaneous Template Files

Introduction

Not all template files generate the entire content that will be rendered by the browser.

Some template files are pulled in by other template files such as `Comments.php`,

`header.php`, `footer.php`, `sidebar.php` and `content-{$slug}.php`. You'll walk through each of these template files to get an understanding of the purpose and how to build them.

Header.php

The `header.php` file does exactly what you would expect. It contains all the code that the browser will render for the header. This is a partial template file because unless a different template file calls the [template tag](#), `get_header()`, the browser will not render the contents of this file.

Often sites have the same header regardless of the page or post you're on. However, some sites have slight variations such as a secondary navigation or different banner image depending on the page. Your `header.php` file can handle all these variations if you use [conditional tags](#).

Almost all themes have a `header.php` file as the functionality and maintainability of this file pretty much demands its creation.

Below is an example of a `header.php` found in the twenty fifteen theme.

Some of the code may look a little daunting at first, but if we break it down, it becomes simple enough. After the opening comment, the `head` is created. The template tag `wp_head()` pulls in all of our styles and any scripts that would appear in the head rather than the footer that we enqueue in our `functions.php` file.

Next the `body` is opened and a mix of HTML and PHP are present. You can see some conditional tags in the site branding div that tweak a little bit of what is shown based on the page the user is on. Then the site navigation is pulled in. Lastly, the main site-content div is opened which will be closed most likely in the `footer.php` file.

One important template tag to note is `body_class()` found in the opening `body` tag. This is a super handy tag that makes styling your theme a lot easier by giving your body classes depending on the template file and other settings being used.

```
<!DOCTYPE html>
<html <?php language_attributes(); ?> class="no-js">
<head>
    <meta charset="<?php bloginfo( 'charset' ); ?>">
        <meta name="viewport" content="width=device-width">
            <link rel="profile" href="http://gmpg.org/xfn/11">
                <link rel="pingback" href="<?php bloginfo( 'pingback_url' ); ?>">
                    <!--[if lt IE 9]>
                        <script src="<?php echo esc_url( get_template_directory_uri() ); ?>/js/html5.js"></script>
                    <![endif]-->
                <?php wp_head(); ?>
</head>

<body <?php body_class(); ?>>

<div id="page" class="hfeed site">
    <a class="skip-link screen-reader-text" href="#content"><?php _e( 'Skip to content', 'twentyfifteen' ); ?></a>

<div id="sidebar" class="sidebar">

<header id="masthead" class="site-header" role="banner">

<div class="site-branding">
    <?php if ( is_front_page() && is_home() ) : ?>
```

```
<h1 class="site-title"><a href="<?php echo  
esc_url( home_url( '/' ) ); ?>" rel="home"><?php  
bloginfo( 'name' ); ?></a></h1>  
  
  
<?php else : ?>  
  
  
<a href="<?php echo esc_url( home_url( '/' ) ); ?>"  
rel="home"><?php bloginfo( 'name' ); ?></a>  
  
                                <?php endif; $description =  
get_bloginfo( 'description', 'display' ); if  
( $description || is_customize_preview() ) : ?>  
  
  
<?php echo $description; ?>  
  
                                <?php endif; ?>  
                                <button class="secondary-toggle"><?php  
_e( 'Menu and widgets', 'twentyfifteen' ); ?></button>  
                                </div>  
  
  
<!-- .site-branding -->  
    </header>  
  
  
<!-- .site-header -->  
  
    <?php get_sidebar(); ?>  
    </div>  
  
  
<!-- .sidebar -->  
  
  
<div id="content" class="site-content">
```

Footer.php

Much like the `header.php` file the `footer.php` is a very common template file that most themes utilize. The code in the `footer.php` file will not be rendered unless another template file pulls in the `footer.php` with `get_footer()` [template tag](#).

Similarly to headers, you can make variations of footers using [conditional tags](#). Often developers will put [widgetized areas](#) in the footer so that the end user can easily drop and drag different content types into the footer.

Here is an example of a `footer.php` file from the Twenty Fifteen theme.

```
1 </div>
2
3
4 <!-- .site-content -->
5
6
7
8 <footer id="colophon" class="site-footer"
9 role="contentinfo">
1
0
1 <div class="site-info">
1           <?php /** * Fires before the Twenty Fifteen
1 footer text for footer customization. * * @since Twenty
2 Fifteen 1.0 */ do_action( 'twentyfifteen_credits' ); ?>
1           <a href="<?php echo esc_url( __( 'https://
3 wordpress.org/' ), 'twentyfifteen' ) ); ?>"><?php printf( __(
1 'Proudly powered by %s', 'twentyfifteen' ),
4 'WordPress' ); ?></a>
1           </div>
5
1
6 <!-- .site-info -->
1           </footer>
7
1
8 <!-- .site-footer -->
1
9 </div>
2
0
2 <!-- .site -->
1
2 <?php wp_footer(); ?>
2
2 </body>
3 </html>
2
4
2
5
2
6
```

404.php

When users try to visit a page on your website that doesn't exist they'll be directed to your `index.php` page unless you've created a `404.php` template. It's a good idea to have some sort of message that explains the page is missing or no longer available. Creating this template helps keep the visual aspects of your theme consistent and provides your end users with helpful information.

Here is an example of a `404.php` template file from the twenty fifteen theme.

```
1 get_header(); ?>
2
3
4
5 <div id="primary" class="content-area">
6     <main id="main" class="site-main" role="main">
7
8
9
10 <section class="error-404 not-found">
11
12
13 <header class="page-header">
14
15
16 <h1 class="page-title"><?php _e( 'Oops! That page
17 can&rsquo;t be found.', 'twentyfifteen' ); ?></h1>
18
19
20         </header>
21
22
23 <!-- .page-header -->
24
25
26
27 <div class="page-content">
28
29
30 <?php _e( 'It looks like nothing was found at this
31 location. Maybe try a search?', 'twentyfifteen' ); ?>
32
33
34             <?php get_search_form(); ?>
35         </div>
36
37
38 <!-- .page-content -->
39         </section>
40
41
42 <!-- .error-404 -->
43
```

Comments.php

The `Comments.php` file handles exactly what you would expect, comments. This is a partial template that is pulled into other template files to display comments that users leave on a page or post. Several different pages and posts show comments so it makes sense to have one file that can be pulled in when needed.

The code involved in creating comments is expanded upon on the [comment template page](#).

Sidebar.php

A lot of themes utilize sidebars to display widgets. For a sidebar to work in a theme it must be registered and then a template file for the sidebar must be created. You'll learn more about [registering sidebars](#) in a later chapter. Sidebar files often have conditional statements and the `is_active_sidebar('sidebar-name')` function in them to ensure a widget is in use within the sidebar so that empty HTML isn't added to a page unnecessarily.

Here is an example of a sidebar template file from the twenty fifteen theme. Notice at the bottom the sidebar is pulled in using `<?php dynamic_sidebar('sidebar-1'); >`. Now whatever, widgets are put

into that sidebar will display on the page that is using this pulling in this template.

```

1 if ( has_nav_menu( 'primary' ) || has_nav_menu( 'social' )
2 || is_active_sidebar( 'sidebar-1' ) ) : >
3
4
5
6
7 <div id="secondary" class="secondary">
8
9     <?php if ( is_active_sidebar( 'sidebar-1' ) ) : >
10
11
12
13
14 <div id="widget-area" class="widget-area"
15 role="complementary">
16     <?php dynamic_sidebar( 'sidebar-1' ); >
17     </div>
18
19
20     <!-- .widget-area -->
21     <?php endif; >
22
23 </div>
24
25
26 <!-- .secondary -->
27
28     <?php endif; >

```

Content-{\$slug}.php

Many theme developers break their template files into small bite sized pieces. They'll often put wrappers and page architecture elements in template files like `page.php`, `home.php`, `comments.php` etc but then they put the code displaying the content of those pages in another template file. Enter `content-{$slug}.php`: common examples would be `content-page.php`, `content-post.php`, `content-portfolio.php`, `content-none.php`. These are not file

names that WordPress recognizes and will interpret a certain way, rather they are a common approach to display specific types of content.

For example, often on blog posts you want to display the author's name, the date of the post, and possibly the category of the post. You'd also likely have links to previous and next posts. That information wouldn't be appropriate on a regular page. Similarly on a portfolio page you would likely have a featured image or gallery you would want to display in a way differently than say a blog post's or page's featured images.

You'll want to use the `get_template_part()` [template tag](#) to pull in the content-
{\$slug}.php file. To pull in your content-page.php file you would call
`get_template_part('content', 'page');`

Here is twenty fifteen's example of a `content-page.php` template file.

```
1 <article id="post-<?php the_ID(); ?>" <?php
2 post_class(); ?>
3     <?php // Post thumbnail.
4 twentyfifteen_post_thumbnail(); ?>
5
6
7
8 <header class="entry-header">
9     <?php the_title( '
10
11     ' ); ?>
12 </header>
13
14
15
16 <!-- .entry-header -->
17
18
19
20 <div class="entry-content">
21     <?php the_content(); ?>
22     <?php wp_link_pages( array( 'before' => '
23
24         <div class="page-links"><span class="page-links-title">' .
25             __( 'Pages:', 'twentyfifteen' ) . '</span>',
26             'after'           => '</div>'
27
28         ,
29             'link_before'  => '<span>',
30             'link_after'   => '</span>',
31             'pagelink'     => '<span class="screen-
32 reader-text">' . __( 'Page', 'twentyfifteen' ) . ' </
33 span>%',
34             'separator'    => '<span class="screen-
35 reader-text">, </span>',
36         ) );
37
38     ?>
39 </div>
40
41
42
```

Comment Template

Press displays comments in your theme based on the settings and code in the `comments.php` file within your WordPress theme.

Simple comments loop

```
1 //Get only the approved comments
2 $args = array(
3     'status' => 'approve'
4 );
5
6 // The comment Query
7 $comments_query = new WP_Comment_Query;
8 $comments = $comments_query->query( $args );
9
10 // Comment Loop
11 if ( $comments ) {
12     foreach ( $comments as $comment ) {
13         echo '<p>' . $comment->comment_content . '</p>';
14     }
15 } else {
16     echo 'No comments found.';
17 }
```

The `comments.php` template contains all the logic needed to pull comments out of the database and display them in your theme.

Before we explore the template file you'll want to know how to pull in the partial template file on the appropriate pages such as `single.php`. You'll wrap the comment [template tag](#) in a conditional statement so `comments.php` is only pulled in if it makes sense to do.

```
1 // If comments are open or we have at least one comment,
2 load up the comment template.
3 if ( comments_open() || get_comments_number() ) :
4     comments_template();
5 endif;
```

ONE THOUGHT ON “HOME”



★ **admin**

SEPTEMBER 12, 2013 AT 8:44 PM  EDIT

Hello

COMMENTS ARE CLOSED.

Another Comments.php Example

Here's an example of the `Comments.php` template included with the Twenty Thirteen theme:

```
1 <?php
2 /**
3  * The template for displaying Comments.
4  *
5  * The area of the page that contains comments and the
6  * comment form.
7  *
8  * @package WordPress
9  * @subpackage Twenty_Thirteen
10 * @since Twenty Thirteen 1.0
11 */
12
13 /**
14  * If the current post is protected by a password and the
15  * visitor has not yet
16  * entered the password we will return early without
17  * loading the comments.
18 */
19 if ( post_password_required() )
20     return;
21 ?>
22
23 <div id="comments" class="comments-area">
24
25     <?php if ( have_comments() ) : ?>
26         <h2 class="comments-title">
27             <?php
28                 printf( _nx( 'One thought on "%2$s"', '%1$s
29 thoughts on "%2$s"', get_comments_number(), 'comments
30 title', 'twentythirteen' ),
31                         number_format_i18n( get_comments_number()
32 () ), '<span>' . get_the_title() . '</span>' );
33             ?>
34         </h2>
35
36         <ol class="comment-list">
37             <?php
38                 wp_list_comments( array(
39                     'style'      => 'ol',
40                     'short_ping' => true,
41                     'avatar_size' => 74,
42                 ) );
43             ?>
44     
```

Breaking down the comments.php

The above `Comments.php` can be broken down to the below parts for better understanding.

1. [Template Header](#)
2. [Comments Title](#)
3. [Comment Listing](#)
4. [Comment Pagination](#)
5. [Comments are closed message.](#)
6. [The End](#)

Template Header

This template begins by identifying the template.

```
1 <?php
2 /**
3  * The template for displaying Comments.
4  *
5  * The area of the page that contains comments and the
6  * comment form.
7  *
8  * @package WordPress
9  * @subpackage Twenty_Thirteen
10 * @since Twenty Thirteen 1.0
11 */
```

Next, there's a test to see if the post is password protected and, if so, it stops processing the template.

```
1 /*
2  * If the current post is protected by a password and the
3  * visitor has not yet
4  * entered the password we will return early without
5  * loading the comments.
6  */
7 if ( post_password_required() )
8     return;
9 ?>
```

Finally, there's a test to see if there are comments associated with this post.

```
1 <div id="comments" class="comments-area">
2     <?php if ( have_comments() ) : ?>
```

Comments Title

Prints out the header that appears above the comments.

Note:

Uses the [_nx\(\)](#) translation function so other developers can provide alternative language translations.

```
1 <h2 class="comments-title">
2     <?php
3         printf( _nx( 'One thought on "%2$s"', '%1$s
4 thoughts on "%2$s"', get_comments_number(), 'comments
5 title', 'twentythirteen' ),
6             number_format_i18n( get_comments_number()
7             ), '<span>' . get_the_title() . '</span>' );
8         ?>
9     </h2>
```

Comment Listing

The following snippet creates an ordered listing of comments using the [wp_list_comments\(\)](#) function.

```
1 <ol class="comment-list">
2     <?php
3         wp_list_comments( array(
4             'style'      => 'ol',
5             'short_ping' => true,
6             'avatar_size' => 74,
7         ) );
8     ?>
9     </ol><!-- .comment-list -->
```

Comment Pagination

Checks to see if there are enough comments to merit adding comment navigation and, if so, create comment navigation.

```

1 <?php
2     // Are there comments to navigate through?
3     if ( get_comment_pages_count() > 1 &&
4         get_option( 'page_comments' ) ) :
5     ?>
6     <nav class="navigation comment-navigation"
7         role="navigation">
8         <h3 class="screen-reader-text section-heading"><?php
9             _e( 'Comment navigation', 'twentythirteen' ); ?></h3>
10            <div class="nav-previous"><?php
11                previous_comments_link( __( '&larr; Older Comments',
12                    'twentythirteen' ) ); ?></div>
13            <div class="nav-next"><?php
14                next_comments_link( __( 'Newer Comments &rarr;',
15                    'twentythirteen' ) ); ?></div>
16        </nav><!-- .comment-navigation -->
17        <?php endif; // Check for comment navigation ?>

```

Comments are closed message

If comments aren't open, displays a line indicating that they're closed.

```

1 <?php if ( ! comments_open() &&
2     get_comments_number() ) : ?>
3     <p class="no-comments"><?php _e( 'Comments are closed.', 'twentythirteen' ); ?></p>
4     <?php endif; ?>

```

The End

This section ends the comments loop, includes the comment form, and closes the comment wrapper.

```

1 <?php endif; // have_comments() ?>
2
3     <?php comment_form(); ?>
4
5 </div><!-- #comments -->

```

Comments Pagination

If you have a lot of comments (which makes your page long), then there are a number of potential benefits to paginating your comments. Pagination helps improve page load speed, especially on mobile devices.

Enabling comments pagination is done in two steps.

1. Enable paged comments within WordPress by going to *Settings > Discussion* , and checking the box “*Break comments into pages*” . You can enter any number for the “*top level comments per page*”.
2. Open your **Comments.php** template file and add the following line where you want the comment pagination to appear.

```
1 <div class="pagination">
2     <?php paginate_comments_links(); ?>
3 </div>
```

Alternative Comment Template

On some occasions you may want display your comments differently within your theme. For this you would build an alternate file (ex. short-comments.php) and call it as follows:

```
1 <?php comments_template( '/short-comments.php' ); ?>
```

The path to the file used for an alternative comments template should be relative to the current theme root directory, and include any subfolders. So if the custom comments template is in a folder inside the theme, it may look like this when called:

```
1 <?php comments_template( '/custom-templates/alternative-
comments.php' ); ?>
```

Function Reference

- [wp_list_comments\(\)](#) : Displays all comments for a post or Page based on a variety of parameters including ones set in the administration area.
- [comment_form\(\)](#) : This tag outputs a complete commenting form for use within a template.
- [comments_template\(\)](#) : Load the comment template specified in first argument
- [paginate_comments_links\(\)](#) : Create pagination links for the comments on the current post.

- [`get_comments\(\)`](#) : Retrieve the comments with possible use of arguments
- [`get_approved_comments\(\)`](#) : Retrieve the approved comments for post id provided.

Functions reference for retrieving comments meta

- [`get_comment_link\(\)`](#)
- [`get_comment_author\(\)`](#)
- [`get_comment_date\(\)`](#)
- [`get_comment_time\(\)`](#)
- [`get_comment_text\(\)`](#)

Taxonomy Templates

When a visitor clicks on a hyperlink to category, tag or custom taxonomy, WordPress displays a page of posts in reverse chronological order filtered by that taxonomy.

By default, this page is generated using the *index.php* template file. You can create optional template files to override and refine the *index.php* template files. This section explains how to use and create such templates.

Taxonomy Template Hierarchy

WordPress display posts in the order determined by the [Template Hierarchy](#).

The *category.php*, *tag.php*, and *taxonomy.php* templates allow posts **filtered** by taxonomy to be treated differently from **unfiltered** posts or posts **filtered by a different taxonomy**. (Note: post refers to any post type – posts, pages, custom post types, etc.).

These files let you target specific taxonomies or specific taxonomy terms. For example:

- `taxonomy-{taxonomy}-{term}.php`
- `taxonomy-{taxonomy}.php`
- `tag-{slug}.php`
- `tag-{id}.php`
- `category-{slug}.php`
- `category-{ID}.php`

So you could format all posts in an animal taxonomy named *news* on a page that looks different from posts filtered in other categories.

The *archive.php* template provides the most general form of control, providing a layout for all archives; that is, a page that displays a list of posts.

Category

For categories, WordPress looks for the *category-{slug}.php* file. If it doesn't exist, WordPress then looks for a file for the next hierarchical level, *category-{ID}.php*, and so on. If WordPress fails to find any specialized templates or an *archive.php* template file, it reverts to the default behavior, using *index.php*.

The category hierarchy is listed below:

1. *category-{slug}.php*: For example, if the category's slug is named "news," WordPress would look for a file named *category-news.php*.
2. *category-{ID}.php*: For example, if the category's ID is "6", WordPress would look for a file named *category-6.php*.
3. *category.php*
4. *archive.php*
5. *index.php*

Tag

For tags, WordPress looks for the *tag-{slug}.php* file. If it doesn't exist, WordPress then looks for a file for the next hierarchical level, *tag-{ID}.php*, and so on. If WordPress fails to find any specialized templates or an *archive.php* template file, it will revert to the default behavior, using *index.php*.

The tag hierarchy is listed below:

1. *tag-{slug}.php*: For example, if the tag's slug is named "sometag," WordPress would look for a file named *tag-sometag.php*.
2. *tag-{id}.php*: For example, if the tag's ID were "6," WordPress would look for a file named *tag-6.php*.
3. *tag.php*
4. *archive.php*

5. index.php

Custom Taxonomy

A custom taxonomy hierarchy works similarly to the categories and tags hierarchies described above. WordPress looks for the *taxonomy-{taxonomy}-{term}.php* file. If it doesn't exist, WordPress then looks for a file for the next hierarchical level, *taxonomy-{taxonomy}.php*, and so on. If WordPress fails to find any specialized templates or an *archive.php* template file, it will revert to the default behavior, using *index.php*.

The hierarchy for a custom taxonomy is listed below:

1. *taxonomy-{taxonomy}-{term}.php*: For example, if the taxonomy is named "sometax," and the taxonomy's term is "someterm," WordPress would look for a file named *taxonomy-sometax-someterm.php*.
2. *taxonomy-{taxonomy}.php*: For example, if the taxonomy is named "sometax," WordPress would look for a file named *taxonomy-sometax.php*
3. *taxonomy.php*
4. *archive.php*
5. *index.php*

Creating Taxonomy Template Files

Now you've decided that you need to create custom designs for content based on taxonomies, where do you start?

Rather than starting from a blank file, it is good practice to **copy the next file in the hierarchy**, if it exists. If you've already created an *archive.php*, make a copy called *category.php* and modify that to suit your design needs. If you don't have an *archive.php* file, use a copy of your theme's *index.php* as a starting point.

Follow the same procedure if you are creating any taxonomy template file. Use a copy of your *archive.php*, *category.php*, *tag.php*, or *index.php* as a starting point.

Examples

Now that you've selected the template file in your theme's directory that you need to modify, let's look at some examples.

Adding Text to Category Pages

Static Text Above Posts

Suppose you want some static text displayed before the list of posts on your category page(s). "Static" is text that remains the same, no matter which posts are displayed below, and no matter which category is displayed.

Open your file and above [The Loop](#) section of your Template file, insert the following code:

```
1 <p>
2 This is some text that will display at the top of the
3 Category page.
</p>
```

This text will only display on an archive page displaying posts in that category.

Different Text on Some Category Pages

What if you want to display different text based on the category page that the visitor is using? You could add default text to the main *category.php* file, and create special *category-{slug}.php* files each with their own version of the text, but this would create lots of files in your theme. Instead, you can use [conditional tags](#).

Again, this code would be added before the loop:

```
1 <?php if (is_category('Category A')) : ?>
2     <p>This is the text to describe category A</p>
3 <?php elseif (is_category('Category B')) : ?>
4     <p>This is the text to describe category B</p>
5 <?php else : ?>
6     <p>This is some generic text to describe all other
7 category pages,
8 I could be left blank</p>
<?php endif; ?>
```

This code does the following:

1. Checks to see if the visitor has requested Category A. If yes, it displays the first piece of text.
2. Checks for category B if the user didn't request category A. If yes, it displays the second piece of text.
3. Displays the default text, if neither was requested.

Display Text only on First Page of Archive

If you have more posts than fits on one page of your archive, the category splits into multiple pages. Perhaps you want to display static text, if the user is on the first page of the results.

To do this, use a PHP if statement that looks at the value of the \$paged WordPress variable.

Put the following above The Loop:

```
1 <?php if ( $paged < 2 ) : ?>
2     <p>Text for first page of Category archive.</p>
3 <?php else : ?>
4 <?php endif; ?>
```

This code asks whether the page displayed is the first page of the archive. If it is, the text for the first page is displayed. Otherwise, the text for the subsequent pages is displayed.

Modify How Posts are Displayed

Excerpts vs. Full Posts

You can choose whether to display full posts or just excerpts. By displaying excerpts, you shorten the length of your archive page.

Open your file and find the loop. Look for:

```
1 the_content()
```

And replace it with:

```
1 the_excerpt()
```

And if your theme is displaying excerpts but you want to display the full content, replace `the_excerpt` with `the_content`.

404 Pages

A 404 page is important to add into your theme in case a user stumbles upon a page that doesn't exist or hasn't been created yet. It is also important that your 404 page gives your visitors a way to arrive at the right place.

These next few steps will help you build a useful 404 page for your WordPress theme.

Creating the 404.php file

As the `404.php` file is used on a page not found error, the first step is to create a file named `404.php` and add it to your WordPress theme folder.

Tip:

Often times a great starting point for your `404.php` is your `index.php`

Add a file header

To use your theme's `header.php` file, open your `404.php` file. At the top, add a comment describing what the file is and make sure you include your theme's header.

For example:

```
1  /**
2  * The template for displaying 404 pages (Not Found)
3  */
4 get_header();
```

Adding the body to your 404 Page

To make the 404 page functional you have to add the body content to your template:

Example:

```

1 <div id="primary" class="content-area">
2   <div id="content" class="site-content" role="main">
3
4     <header class="page-header">
5       <h1 class="page-title"><?php _e( 'Not
6 Found', 'twentythirteen' ); ?></h1>
7     </header>
8
9     <div class="page-wrapper">
10      <div class="page-content">
11        <h2><?php _e( 'This is somewhat
12 embarrassing, isn't it?', 'twentythirteen' ); ?></h2>
13        <p><?php _e( 'It looks like nothing was
14 found at this location. Maybe try a search?', 'twentythirteen' ); ?></p>
15
16          <?php get_search_form(); ?>
17        </div><!-- .page-content -->
18      </div><!-- .page-wrapper -->
19
20    </div><!-- #content -->
21  </div><!-- #primary -->
22
23
24
25
26
27
28

```

This example is from the Twenty Thirteen theme that comes with WordPress. This adds some text and a search form in the 404 page.

Note:

You can add your own text to make your 404 page look better.

Adding the footer and sidebar

The final step when creating a 404 page is to add the footer. You may also add a sidebar if you want.

Example:

```

1 get_sidebar();
2 get_footer();

```

Now you have a functional 404 page with a search form and some text in case a user stumbles upon the wrong page. Once you make sure the `404.php` has been uploaded, you can test your 404 page. Just type a URL address for your website that doesn't exist. You can try something like `http://example.com/fred.php`. This is sure to result in a 404 error unless you actually have a PHP file called `fred.php`.

Theme Functionality

Now that you're familiar with the basics of theme development, it's time to start digging deeper into the world of themes. This section covers all of the basic theme functionality that you'll need to consider while creating your theme:

- [Custom headers](#) – learn how to give your users the flexibility of adding their own header image
- [Sidebars](#) – add widgetized areas to your theme where users can add widgets
- [Widgets](#) – create re-usable PHP objects that can be added to a sidebar
- [Navigation menus](#) – register and display menus within your theme
- [Taxonomy Templates](#) – create templates for your taxonomy archive pages
- [Pagination](#) – work with WordPress' built-in pagination
- [Comments](#) – customize the comments template
- [Media](#) – work with WordPress core's media capabilities
- [Featured Images & Post Thumbnails](#) – work with and customize post thumbnails
- [Post Formats](#) – create different formats for displaying users' posts
- [Internationalization](#) – learn how to prepare your theme for translation into different languages
- [Localization](#) – learn how to translate a WordPress theme
- [Accessibility](#) – learn about accessibility best practices to ensure everyone can use your theme
- [Administration Menus](#) – add menu items to the WordPress administration
- [404 Pages](#) – create a custom 404 page

If you're new to themes it's worth working your way through each section, but you can also jump into a section if you're looking for something specific.

Administration Menus

Warning:

This is no longer the recommended way to work with theme options. The [Customizer API](#) is the recommended way to give more control and flexibility to your users

Theme authors might need to provide a settings screen, so users can customize how their theme is used or works. The best way to do this is to create an administration menu item that allows the user to access that settings screen from all the administration screens.

Function Reference

Menu Pages

[add_menu_page\(\)](#)
[add_object_page\(\)](#)
[add_utility_page\(\)](#)
[remove_menu_page\(\)](#)

Sub-menu Pages

[add_submenu_page\(\)](#)
[remove_submenu_page\(\)](#)

WordPress Administration Menus

[add_dashboard_page\(\)](#)
[add_posts_page\(\)](#)
[add_media_page\(\)](#)
[add_links_page\(\)](#)
[add_pages_page\(\)](#)
[add_comments_page\(\)](#)
[add_theme_page\(\)](#)
[add_plugins_page\(\)](#)
[add_users_page\(\)](#)
[add_management_page\(\)](#)
[add_options_page\(\)](#)

Every Plot Needs a Hook

To add an administration menu, there are three things you need to do:

1. Create a function that contains the menu-building code.
2. Register the above function using the `admin_menu` action hook – or `network_admin_menu`, if you are adding a menu for the Network.
3. Create the HTML output for the screen, displayed when the menu item is clicked.

Most developers overlook the second step in this list. You can't simply call the menu code. You need to put it **inside a function**, and then register this function.

Here is a simple example of these three steps described. This will add a sub-level menu item under the Settings top-level menu. When selected, that menu item will show a very basic screen.

```
1 <?php
2 /** Step 2 (from text above). */
3 add_action( 'admin_menu', 'my_menu' );
4
5 /** Step 1. */
6 function my_menu() {
7     add_options_page(
8         'My Options',
9         'My Menu',
10        'manage_options',
11        'my-unique-identifier',
12        'my_options'
13    );
14 }
15
16 /** Step 3. */
17 function my_options() {
18     if ( !current_user_can( 'manage_options' ) ) {
19         wp_die( __( 'You do not have sufficient
20 permissions to access this page.' ) );
21     }
22     echo 'Here is where I output the HTML for my screen.';
23     echo '</div><pre>';
24 }
```

In this example, the function `my_menu()` adds a new item to the Settings administration menu via the `add_options_page\(\)` function.

Note: Note that the `add_action\(\)` call in Step 2 registers the `my_menu()` function under the `admin_menu` hook. **Without that, `add_action()` call, a PHP error for undefined function will be thrown.** Finally, the `add_options_page\(\)` call refers to the `my_options()` function which contains the actual page to be displayed (and PHP code to be processed) when someone clicks the menu item.

These steps are described in more detail in the sections below. Remember to enclose the creation of the menu and the page in functions, and use the `admin_menu` [hook](#) to get the whole process started at the right time.

Determining Location for New Menus

Before creating a new menu, first decide if the menu should be a **top-level** menu, or a **sub-level** menu item. A top-level menu displays as new section in the administration menus and contains sub-level menu items. This means a sub-level menu item is a member of an existing top-level menu.

It is rare that a theme would require the creation of a new top-level menu. If the theme introduces an entirely new concept to WordPress and needs many screens to do it, then that theme may warrant a new top-level menu. Adding a top-level menu should only be considered if you really need multiple, related screens to make WordPress do something it was not originally designed to accomplish. Examples of new top-level menus might include job management or conference management. Please note that, with the native `post type` registration, WordPress automatically creates top-level menus to manage this kind of features.

If the creation of a top-level menu is not necessary, you need to decide under what top-level menu to place your new sub-level menu item. As a point of reference, several themes add sub-level menu items underneath existing WordPress top-level menus. Use this guide of the WordPress top-level menus to determine the correct location for your sub-level menu item:

- Dashboard – information central for your site and include the Updates option for updating WordPress core, plugins, and themes.
- Posts – displays tools for writing posts (time oriented content).

- Media – uploading and managing your pictures, videos, and audio.
- Links – manage references to other blogs and sites of interest.
- Pages – displays tools for writing your static content called pages.
- Comments – controlling and regulation reader to responses to posts.
- Appearance – displays controls for manipulation of theme/style files, sidebars, etc.
- Plugins – displays controls dealing with plugin management, not configuration options for a plugin itself.
- Users – displays controls for user management.
- Tools – manage the export, import, and even backup of blog data.
- Settings – displays plugin options that only administrators should view.
- Network Admin – displays plugin options that are set on the Network. Instead of `admin_menu`, you should use `network_admin_menu` (see also [Create A Network](#))

Top-Level Menus

If you have decided your theme requires a brand-new top-level menu, the first thing you'll need to do is to create one with the `add_menu_page()` function. *Note:* skip to [Sub-Level Menus](#) if you don't need a top-level menu.

Parameter values:

- `page_title` – the text to be displayed in the title tags of the page when the menu is selected.
- `menu_title` – the on-screen name text for the menu.
- `capability` – the capability required for this menu to be displayed to the user.

When using the Settings API to handle your form, you should use `manage_options` here as the user won't be able to save options without it.

User levels are deprecated and should not be used here.

- `menu_slug` – the slug name to refer to this menu by (should be unique for this menu). Prior to Version 3.0 this was called the file (or handle) parameter. If the function parameter is omitted, the `menu_slug` should be the PHP file that handles the display of the menu page content.
- `function` – the function that displays the page content for the menu page.

- `icon_url` – the URL to the icon to be used for this menu. This parameter is optional.
- `position` – the position in the menu order this menu should appear. By default, if this parameter is omitted, the menu will appear at the bottom of the menu structure. To see the current menu positions, use `print_r($GLOBALS['menu'])` after the menu has loaded.
- Sub-Level Menus – once your top-level menu is defined, or you have chosen to use an existing WordPress top-level menu, you are ready to define one or more sub-level menu items using the `add_submenu_page()` function.

Sub-Level Menus

If you want your new menu item to be a sub-menu item, you can create it using the `add_submenu_page()` function instead.

Parameter values:

- `parent_slug` – the slug name for the parent menu, or the file name of a standard WordPress admin file that supplies the top-level menu in which you want to insert your sub-menu, or your plugin file if this sub-menu is going into a custom top-level menu. Examples:
 - Dashboard – `add_submenu_page('index.php', ...)`
 - Posts – `add_submenu_page('edit.php', ...)`
 - Media – `add_submenu_page('upload.php', ...)`
 - Links – `add_submenu_page('link-manager.php', ...)`
 - Pages – `add_submenu_page('edit.php?post_type=page', ...)`
 - Comments – `add_submenu_page('edit-comments.php', ...)`
 - Custom Post Types – `add_submenu_page('edit.php?post_type=your_post_type', ...)`
 - Appearance – `add_submenu_page('themes.php', ...)`
 - Plugins – `add_submenu_page('plugins.php', ...)`
 - Users – `add_submenu_page('users.php', ...)`

- Tools – add_submenu_page('tools.php', ...)
- Settings – add_submenu_page('options-general.php', ...)
- **page_title** – text that will go into the HTML page title for the page when the submenu is active.
- **menu_title** – the text to be displayed in the title tags of the page when the menu is selected.
- **capability** – the capability required for this menu to be displayed to the user.
User levels are deprecated and should not be used here.
- **menu_slug** – for existing WordPress menus, the PHP file that handles the display of the menu page content. For sub-menus of a custom top-level menu, a unique identifier for this sub-menu page.
- **function** – the function that displays the page content for the menu page.
Technically, as in the **add_menu_page** function, the function parameter is optional, but if it is not supplied, then WordPress will basically assume that including the PHP file will generate the administration screen, without calling a function.

Using Wrapper Functions

Since most sub-level menus belong under the Settings, Tools, or Appearance menus, WordPress supplies wrapper functions that make adding a sub-level menu items to these top-level menus easier. Note that the function names may not match the names seen in the admin UI as they have changed over time:

Dashboard

```
1 <?php add_dashboard_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Posts

```
1 <?php add_posts_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Media

```
1 <?php add_media_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Links

```
1 <?php add_links_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Pages

```
1 <?php add_pages_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Comments

```
1 <?php add_comments_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Appearance

```
1 <?php add_theme_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Plugins

```
1 <?php add_plugins_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Users

```
1 <?php add_users_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Tools

```
1 <?php add_management_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Settings

```
1 <?php add_options_page( $page_title, $menu_title,  
    $capability, $menu_slug, $function); ?>
```

Also see [Theme Options](#) for the currently recommended way of creating options via the

Customizer API.

Example

Here's a quick example illustrating how to insert a top-level menu page and a sub-menu page, where the title on the sub-menu page is different from the top-level page. In this example, `register_my_theme_more_settings_menu` is the name of the function that displays the first sub-menu page:

```
1 function register_my_theme_settings_menu() {
2     add_menu_page(
3         "My Theme's Settings",
4         "My Theme",
5         "manage_options",
6         "my-theme-settings-menu"
7     );
8 }
9
10 function register_my_theme_more_settings_menu() {
11     add_submenu_page(
12         "my-themes-settings-menu",
13         "More Settings for My Theme",
14         "More Settings",
15         "manage_options",
16         "my-theme-more-settings-menu"
17     );
18 }
19
20 add_action( "admin_menu",
21             "register_my_theme_settings_menu");
22 add_action( "admin_menu",
23             "register_my_theme_more_settings_menu");
```

Here's an example of adding an option page under a custom post type menu block (see also [here](#)):

CODE

Inserting the Pages

Here is an example of how to insert multiple menus into various places:

```
1 <?php
2 // Hook for adding admin menus
3 add_action('admin_menu', 'mt_add_pages');
4
5 // Action function for hook above
6 function mt_add_pages() {
7 // Add a new submenu under Settings:
8 add_options_page__(__('Test Settings','menu-test'), __('Test
9 Settings','menu-test'), 'manage_options', 'testsettings',
1 'mt_settings_page');
0
1 // Add a new submenu under Tools:
1 add_management_page( __('Test Tools','menu-test'), __('Test
1 Tools','menu-test'), 'manage_options', 'testtools',
2 'mt_tools_page');
1
3 // Add a new top-level menu (ill-advised):
1 add_menu_page(__('Test Toplevel','menu-test'), __('Test
4 Top-level','menu-test'), 'manage_options', 'mt-top-level-
1 handle', 'mt_toplevel_page' );
5
1 // Add a submenu to the custom top-level menu:
6 add_submenu_page('mt-top-level-handle', __('Test Sub-
1 Level','menu-test'), __('Test Sub-Level','menu-test'),
7 'manage_options', 'sub-page', 'mt_sublevel_page');
1
8 // Add a second submenu to the custom top-level menu:
1 add_submenu_page('mt-top-level-handle', __('Test Sub-Level
9 2','menu-test'), __('Test Sub-Level 2','menu-test'),
2 'manage_options', 'sub-page2', 'mt_sublevel_page2');
0 }
2
1 // mt_settings_page() displays the page content for the
2 Test settings sub-menu
2 function mt_settings_page() {
2     echo "</pre>
3         <h2>" . __('Test Settings', 'menu-test') . "</h2>
2         <pre>
4         ";
2 }
5
2 // mt_tools_page() displays the page content for the Test
6 Tools sub-menu
```

Sample Menu Page

Note: See the [Settings API](#) for information on creating settings pages.

The previous example contains several dummy functions, such as

`mt_settings_page()`, as placeholders for actual page content. Let's expand on them. What if you wanted to create an option called `mt_favorite_color` that allows the site owner to type in their favorite color via a Settings page? The `mt_options_page()` function will need to output a data entry form on the screen, as well as to process the entered data.

Here is a function that does this:

```
1 // mt_settings_page() displays the page content for the
2 Test settings sub-menu
3 function mt_settings_page() {
4     //must check that the user has the required capability
5     if (!current_user_can('manage_options'))
6     {
7         wp_die( __('You do not have sufficient permissions
8 to access this page.') );
9     }
10
11     // Variables for the field and option names
12     $opt_name = 'mt_favorite_color';
13     $hidden_field_name = 'mt_submit_hidden';
14     $data_field_name = 'mt_favorite_color';
15
16     // Read in existing option value from database
17     $opt_val = get_option( $opt_name );
18
19     // See if the user has posted us some information
20     // If they did, this hidden field will be set to 'Y'
21     if( isset($_POST[ $hidden_field_name ]) &&
22 $_POST[ $hidden_field_name ] == 'Y' ) {
23         // Read their posted value
24         $opt_val = $_POST[ $data_field_name ];
25
26         // Save the posted value in the database
27         update_option( $opt_name, $opt_val );
28
29         // Put a "Settings updated" message on the screen
30     ?>
31     <div class="updated"></div>
32
33     <div class="wrap">
34     <?php
35     // Header
36     echo "<h2>" . __('Menu Test Settings', 'menu-test') .
37     "</h2>";
38
39     // Settings form
40     ?>
41     <form action="" method="post" name="form1"></form>
42     <?php _e("Favorite Color:", 'menu-test') ; ?>
```

A few notes:

- The WordPress functions such as `add_menu_page()` and `add_submenu_page()` take a capability which will be used to determine whether the top-level or sub-level menu is displayed.
- The function which is hooked in to handle the output of the page must check that the user has the required capability as well.
- The WordPress administration functions take care of validating the user login, so you don't have to worry about it in your function.
- The function example above has been internationalized — see the [I18n for WordPress Developers](#) for more details.
- The function processes any entered data before putting the data entry form on the screen, so that the new values will be shown in the form (rather than the values from the database).
- You don't have to worry about this working the first time, because the WordPress `update_option` function will automatically add an option to the database if it doesn't already exist.
- These admin-menu-adding procedures are parsed every single time you navigate to a page in Admin. So if you are writing a theme which has no options page, but add one later, you can just add it using the instructions above and re-upload, and tweak until you're happy with it. In other words, menus are not “permanently added” or put into a database upon activating a theme. They're parsed on the fly, so you can add or subtract menu items at will, re-upload, and the change will be reflected right away.

Page Hook Suffix

Every function that adds a new administration menu – `add_menu_page()`, `add_submenu_page()` and its specialized versions such as `add_options_page()` – returns a special value called **Page Hook Suffix**. It can be used later as a hook to which an action called only on that particular page can be registered.

One such action hook is `load-{page_hook}`, where `{page_hook}` is the value returned by one of these `add_*_page()` functions. This hook is called when the particular page is loaded. In the example below, it is used to display the “Theme is not configured” notice on all admin pages except for plugin’s options page:

```
<?php
add_action( 'admin_menu', 'my_menu' );

// Here you can check if plugin is configured (e.g.
// check if some option is set). If not, add new hook.
// In this example hook is always added.
add_action( 'admin_notices', 'my_admin_notices' );

function my_menu() {
    // Add the new admin menu and page and save the
    // returned hook suffix
    $hook_suffix = add_options_page('My Options', 'My
    Theme', 'manage_options', 'my-unique-identifier',
    'my_options');
    // Use the hook suffix to compose the hook and
    // register an action executed when plugin's options
    // page is loaded
    add_action( 'load-' . $hook_suffix ,
    'my_load_function' );
}

function my_load_function() {
    // Current admin page is the options page for our
    // plugin, so do not display the notice
    // (remove the action responsible for this)
    remove_action( 'admin_notices',
    'my_admin_notices' );
}

function my_admin_notices() {
    echo '<pre>
<div class="updated fade" id="notice">
```

```
    My Plugin is not configured yet. Please do it
now.</div>
    </pre>';
}

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

    echo '</pre>
<div class="wrap">';
    echo 'Here is where the form would go if I
actually had options.';
    echo '</div>
<pre>
';
}
?>
```

Custom Headers

Custom Header

[Custom headers](#) allow site owners to upload their own “title” image to their site, which can be placed at the top of certain pages. These can be customized and cropped by the user through a visual editor in the **Appearance > Header** section of the admin panel. You may also place text beneath or on top of the header. To support fluid layouts and responsive design, these headers may also be flexible. Headers are placed into a theme using `get_custom_header()`, but they must first be added to your `functions.php` file using `add_theme_support()`. Custom headers are **optional**.

To set up a basic, flexible, custom header with text you would include the following code:

```
1 function themename_custom_header_setup() {
2     $args = array(
3         'default-image'      =>
4             get_template_directory_uri() . 'img/default-image.jpg',
5         'default-text-color' => '000',
6         'width'              => 1000,
7         'height'             => 250,
8         'flex-width'         => true,
9         'flex-height'        => true,
10    )
11    add_theme_support( 'custom-header', $args );
12 }
```

```
add_action( 'after_setup_theme',
'themename_custom_header_setup' );
```

The [after_setup_theme](#) hook is used so that custom headers are registered after the theme is loaded.

What are Custom Headers?

When you enable Custom Headers in your theme, users can change their header image using the WordPress theme Customizer. This gives users more control and flexibility over the look of their site.

Add Custom Header Support to your Theme

To enable Custom Headers in your theme, add the following to your `functions.php` file:

```
1 add_theme_support( 'custom-header' );
```

When enabling Custom Headers, you can configure several other options by passing along arguments to the `add_theme_support()` function.

You can pass specific configuration options to the `add_theme_support` function using an array:

```

1 function themename_custom_header_setup() {
2     $defaults = array(
3         // Default Header Image to display
4         'default-image'          =>
5         get_template_directory_uri() . '/images/headers/
6         default.jpg',
7         // Display the header text along with the image
8         'header-text'            => false,
9         // Header text color default
10        'default-text-color'    => '000',
11        // Header image width (in pixels)
12        'width'                 => 1000,
13        // Header image height (in pixels)
14        'height'                => 198,
15        // Header image random rotation default
16        'random-default'        => false,
17        // Enable upload of image file in admin
18        'uploads'                => false,
19        // function to be called in theme head section
20        'wp-head-callback'      => 'wphead_cb',
21        // function to be called in preview page head
22    section
23        'admin-head-callback'    => 'adminhead_cb',
24        // function to produce preview markup in the admin
25    screen
26        'admin-preview-callback' => 'adminpreview_cb',
27    );
28 }
29 add_action( 'after_setup_theme',
30 'themename_custom_header_setup' );

```

Flexible Header Image

If flex-height or flex-width are not included in the array, height and width will be fixed sizes. If flex-height and flex-width are included, height and width will be used as suggested dimensions instead.

Header text

By default, the user will have the option of whether or not to display header text over the image. There is no option to force the header text on the user, but if you want to remove

the header text completely, you can set ‘header-text’ to ‘false’ in the arguments. This will remove the header text and the option to toggle it.

Set a custom header image

When the user first installs your theme, you can include a default header that will be selected before they choose their own header. This allows users to set up your theme more quickly and use your default image until they’re ready to upload their own.

Set a default header image 980px width and 60px height:

```
1 $header_info = array(
2     'width'          => 980,
3     'height'         => 60,
4     'default-image'  => get_template_directory_uri() . '/
5 images/sunset.jpg',
6 );
7 add_theme_support( 'custom-header', $header_info );
8
9 $header_images = array(
10    'sunset' => array(
11        'url'           =>
12        get_template_directory_uri() . '/images/sunset.jpg',
13        'thumbnail_url' =>
14        get_template_directory_uri() . '/images/
15 sunset_thumbnail.jpg',
16        'description'   => 'Sunset',
17    ),
18    'flower' => array(
19        'url'           =>
20        get_template_directory_uri() . '/images/flower.jpg',
21        'thumbnail_url' =>
22        get_template_directory_uri() . '/images/
23 flower_thumbnail.jpg',
24        'description'   => 'Flower',
25    ),
26);
27 register_default_headers( $header_images );
```

X

Saved



Customizing Header Image

While you can crop images to your liking after clicking Add new image, your theme recommends a header size of 980 × 60 pixels.

Current header



[Hide image](#)

[Add new image](#)

Suggested



My First Site

Welcome

Contents

Welcome to your site!

Do not forget to call [register_default_headers\(\)](#) to register a default image. In this example, `sunset.jpg` is the default image and `flower.jpg` is an alternative selection in Customizer.

From Administration Screen, Click **Appearance > Header** to display Header Image menu in Customizer. Notice that width and height specified in [add_theme_support\(\)](#) is displayed as recommended size, and `flower.jpg` is shown as selectable option.

Use flexible headers

By default, the user will have to crop any images they upload to fit in the width and height you specify. However, you can let users upload images of any height and width by specifying ‘flex-width’ and ‘flex-height’ as true. This will let the user skip the cropping step when they upload a new photo.

Set flexible headers:

```
1 $args = array(
2     'flex-width'      => true,
3     'width'          => 980,
4     'flex-height'    => true,
5     'height'         => 200,
6     'default-image'  => get_template_directory_uri() . '/
7 images/header.jpg',
8 );
add_theme_support( 'custom-header', $args );
```

update your `header.php` file to:

```
1
2     <div id="site-header">
3         <a href="<?php echo esc_url( home_url( '/' ) ); ?>" 
4 rel="home">
5             width ); ?>" height="<?php
7 echo absint( get_custom_header()->height ); ?>" alt="<?php
echo esc_attr( get_bloginfo( 'name', 'display' ) ); ?>">
8         </a>
9     </div>
10    <?php endif; ?>
```

Backwards Compatibility

Custom Headers are supported in WordPress 3.4 and above. If you'd like your theme to support WordPress installations that are older than 3.4, you can use the following code instead of `add_theme_support('custom-header');`

```
1 global $wp_version;
2 if ( version_compare( $wp_version, '3.4', '>=' ) ) :
3     add_theme_support( 'custom-header' );
4 else :
5     add_custom_image_header( $wp_head_callback,
6 $admin_head_callback );
7 endif;
```

Function Reference

- [header_image\(\)](#) Display header image URL.
- [get_header_image\(\)](#) Retrieve header image for custom header.
- [get_custom_header\(\)](#) Get the header image data.
- [get_random_header_image\(\)](#) Retrieve header image for custom header.
- [add_theme_support\(\)](#) Registers theme support for a given feature.
- [register_default_headers\(\)](#) Registers a selection of default headers to be displayed by the Customizer.

Custom Logo

What is a Custom Logo?

Using a custom logo allows site owners to upload an image for their website, which can be placed at the top of their website. It can be uploaded from **Appearance > Header**, in your admin panel. The custom logo support should be added first to your theme using `add_theme_support()`, and then be called in your theme using `the_custom_logo()`. A custom logo is **optional**, but theme authors should use this function if they include a logo to their theme.

Adding Custom Logo support to your Theme

To enable the use of a custom logo in your theme, add the following to your `functions.php` file:

```
1 add_theme_support( 'custom-logo' );
```

When enabling custom logo support, you can configure five parameters by passing along arguments to the `add_theme_support()` function using an array:

```
1 function themename_custom_logo_setup() {
2     $defaults = array(
3         'height'      => 100,
4         'width'       => 400,
5         'flex-height' => true,
6         'flex-width'  => true,
7         'header-text' => array( 'site-title', 'site-description' )
8     );
9 };
10 add_theme_support( 'custom-logo', $defaults );
11 }
add_action( 'after_setup_theme',
    'themename_custom_logo_setup' );
```

The `after_setup_theme` hook is used so that the custom logo support is registered after the theme has loaded.

- height

- Expected logo height in pixels. A custom logo can also use built-in image sizes, such as `thumbnail`, or register a custom size using [`add_image_size\(\)`](#).
- `width`

Expected logo width in pixels. A custom logo can also use built-in image sizes, such as `thumbnail`, or register a custom size using [`add_image_size\(\)`](#).
- `flex-height`
- Whether to allow for a flexible height.
- `flex-width`

Whether to allow for a flexible width.
- `header-text`

Classes(s) of elements to hide. It can pass an array of class names here for all elements constituting header text that could be replaced by a logo.

Displaying the custom logo in your theme

A custom logo can be displayed in the theme using `the_custom_logo()` function. But it's recommended to wrap the code in a `function_exists()` call to maintain backward compatibility with the older versions of WordPress, like this:

```
1 if ( function_exists( 'the_custom_logo' ) ) {  
2     the_custom_logo();  
3 }
```

Generally logos are added to the `header.php` file of the theme, but it can be elsewhere as well.

If you want to get your current logo URL (or use your own markup) instead of the default markup, you can use the following code:

```
1 $custom_logo_id = get_theme_mod( 'custom_logo' );
2 $logo = wp_get_attachment_image_src( $custom_logo_id ,
3 'full' );
4 if ( has_custom_logo() ) {
5     echo '' . get_bloginfo( 'name' ) . '</h1>';
9 }
```

Custom logo template tags

To manage displaying a custom logo in the front-end, these three template tags can be used:

- [get_custom_logo\(\)](#) – Returns markup for a custom logo.
- [the_custom_logo\(\)](#) – Displays markup for a custom logo.
- [has_custom_logo\(\)](#) – Returns a boolean true/false, whether a custom logo is set or not.

Post Formats

A Post Format is used by a theme for presenting posts in a certain format and style. The Post Formats feature provides a standardized list of formats available to all themes that support the feature. A theme may not support every format on the list; in such a case, it is good form to make this known to users.

A theme cannot introduce formats not on the standardized list, even through plugins. This standardization ensures compatibility between themes and a way for external tools to use the feature in a consistent fashion.

In short, with a theme that supports Post Formats, a blogger can change how a post looks by choosing a Post Format.

Using **Asides** as an example, in the past, a category called Asides was created, and posts were assigned that category, and then displayed differently based on styling rules from [post_class\(\)](#) or from [in_category\('asides'\)](#).

With **Post Formats**, the new approach allows a theme to add support for a Post Format (e.g. [add_theme_support\('post-formats', array\('aside'\)\)](#)), and then the post format can be selected in the Publish meta box when saving the post. A function call of

`get_post_format($post->ID)` can be used to determine the format, and `post_class()` will also create the “format-asides” class, for pure-css styling.

Supported Formats

The following Post Formats are available, if supported by the theme. Note that while actual post content won't change, the theme can display a post differently based on the format chosen. How posts are displayed is entirely up to the theme, but here are some general guidelines on typical uses for the different Post Formats.

- **aside** – Typically styled without a title. Similar to a Facebook note update.
- **gallery** – A gallery of images. Post will likely contain a gallery shortcode and will have image attachments.
- **link** – A link to another site. Themes may wish to use the first `` tag in the post content as the external link for that post. An alternative approach could be if the post consists only of a URL, then that will be the URL and the title (`post_title`) will be the name attached to the anchor for it.
- **image** – A single image. The first `` tag in the post could be considered the image. Alternatively, if the post consists only of a URL, that will be the image URL and the title of the post (`post_title`) will be the title attribute for the image.
- **quote** – A quotation. Probably will contain a blockquote holding the quote content. Alternatively, the quote may be just the content, with the source/author being the title.
- **status** – A short status update, similar to a Twitter status update.
- **video** – A single video. The first `<video />` tag or object/embed in the post content could be considered the video. Alternatively, if the post consists only of a URL, that will be the video URL. May also contain the video as an attachment to the post, if video support is enabled on the blog (like via a plugin).
- **audio** – An audio file. Could be used for Podcasting.
- **chat** – A chat transcript, like so:

```
John: foo
Mary: bar
John: foo 2
```

Note:

When writing or editing a Post, “Standard” designates that no Post Format is specified. Also if an invalid format is specified, “Standard” (no format) is applied by default.

Function Reference

Main Functions

- [set_post_format\(\)](#)
- [get_post_format\(\)](#)
- [has_post_format\(\)](#)

Other Functions

- [get_post_format_link\(\)](#)
- [get_post_format_string\(\)](#)

Adding Theme Support

Themes need to use [add_theme_support\(\)](#) in the *functions.php* file to tell WordPress which post formats to support by passing an array of formats like so:

```
1 function themename_post_formats_setup() {  
2     add_theme_support( 'post-formats', array( 'aside',  
3     'gallery' ) );  
4 }  
add_action( 'after_setup_theme',  
    'themename_post_formats_setup' );
```

The [after_setup_theme](#) hook is used so that the post formats support is registered after the theme has loaded.

Adding Post Type Support

Post Types need to use [add_post_type_support\(\)](#) in the *functions.php* file to tell WordPress which post formats to support:

```
1 function themename_custom_post_formats_setup() {
2     // add post-formats to post_type 'page'
3     add_post_type_support( 'page', 'post-formats' );
4
5     // add post-formats to post_type 'my_custom_post_type'
6     add_post_type_support( 'my_custom_post_type', 'post-
7     formats' );
8 }
add_action( 'init',
    'themename_custom_post_formats_setup' );
```

Or in the function [register_post_type\(\)](#), add ‘post-formats’ in ‘supports’ parameter array:

```
1 $args = array(
2     ...
3     'supports' => array('title', 'editor', 'author', 'post-
4     formats')
5 );
register_post_type('book', $args);
```

[add_post_type_support](#) should be hooked to [init](#) hook, as custom post types may not have been registered at [after_setup_theme](#).

Using Formats

In the theme, use [get_post_format\(\)](#) to check the format for a post, and change its presentation accordingly. Note that posts with the default format will return a value of FALSE. Alternatively, use the [has_post_format\(\) conditional tag](#):

```
1 if ( has_post_format( 'video' ) ) {
2     echo 'this is the video format';
3 }
```

Suggested Styling

An alternate approach to formats is through styling rules. Themes should use the [post_class\(\)](#) function in the wrapper code that surrounds the post to add dynamic styling classes. Post formats will cause extra classes to be added in this manner, using the “format-foo” name.

For example, one could hide post titles from status format posts by putting this in your theme’s stylesheet:

```
.format-status .post-title {  
    display:none;  
}
```

Each of the formats lend themselves to a certain type of “style”, as dictated by modern usage. It is well to keep in mind the intended usage for each format when applying styles. For example, the aside, link, and status formats are simple, short, and minor. These will typically be displayed without title or author information. The aside could contain perhaps a paragraph or two, while the link would be only a sentence with a link to a URL in it. Both the link and aside might have a link to the single post page (using [the permalink\(\)](#)) and would thus allow comments, but the status format very likely would not have such a link. An image post, on the other hand, would typically just contain a single image, with or without a caption/text to go along with it. An audio/video post would be the same but with audio/video added in. Any of these three could use either plugins or standard [Embeds](#) to display their content. Titles and authorship might not be displayed for them either, as the content could be self-explanatory.

The quote format is especially well suited to posting a simple quote from a person with no extra information. If you were to put the quote into the post content alone, and put the quoted person’s name into the title of the post, then you could style the post so as to display [the content\(\)](#) by itself but restyled into a blockquote format, and use [the title\(\)](#) to display the quoted person’s name as the byline.

A chat in particular will probably tend towards a monospaced type display, in many cases. With some styling on the .format-chat, you can make it display the content of the post using a monospaced font, perhaps inside a gray background div or similar, thus distinguishing it visually as a chat session.

Formats in a Child Theme

[Child Themes](#) inherit the post formats defined by the parent theme. Calling [add_theme_support\(\)](#) for post formats in a child theme must be done at a later priority than that of the parent theme and will **override** the existing list, not add to it.

```
1 add_action( 'after_setup_theme', 'childtheme_formats',  
2 11 );  
3 function childtheme_formats(){  
4     add_theme_support( 'post-formats', array( 'aside',  
5 'gallery', 'link' ) );  
6 }
```

Calling `remove_theme_support('post-formats')` will remove it all together.

Sticky Posts

A Sticky Post is the post will be placed at the top of the front page of posts. This feature is only available for the built-in post type post and not for custom post types.

How to stick a post

1. Go to Administration Screen > Posts > Add New or Edit
2. In the right side menu, Click Edit link of Visibility option in Publish group
3. Click Stick this post to the front page option

The screenshot shows the 'Publish' tab of the WordPress post editor. At the top right is a 'Preview Changes' button. Below it, under 'Status', is 'Published' with an 'Edit' link. Under 'Visibility', 'Public' is selected (indicated by a radio button) and 'Sticky' is checked (also indicated by a radio button). A red box highlights the 'Stick this post to the front page' checkbox, which is also checked. Other visibility options shown are 'Password protected' and 'Private'. At the bottom left are 'OK' and 'Cancel' buttons. At the bottom right are 'Move to Trash' and a blue 'Update' button. Below the visibility section, there are links for 'Revisions: 4 Browse' and 'Published on: Nov 26, 2016 @ 14:42 Edit'.

Display Sticky Posts

Show Sticky Posts

Display just the first sticky post. At least one post must be designated as a “sticky post” or else the loop will display all posts:

```
1 $sticky = get_option( 'sticky_posts' );
2 $query = new WP_Query( 'p=' . $sticky[0] );
```

Display just the first sticky post, if none return the last post published:

```
1 $args = array(
2     'posts_per_page' => 1,
3     'post__in' => get_option( 'sticky_posts' ),
4     'ignore_sticky_posts' => 1
5 );
6 $query = new WP_Query( $args );
```

Display just the first sticky post, if none return nothing:

```
1 $sticky = get_option( 'sticky_posts' );
2 $args = array(
3     'posts_per_page' => 1,
4     'post__in' => $sticky,
5     'ignore_sticky_posts' => 1
6 );
7 $query = new WP_Query( $args );
8 if ( isset( $sticky[0] ) ) {
9     // insert here your stuff...
10 }
```

Don't Show Sticky Posts

Exclude all sticky posts from the query:

```
1 $query = new WP_Query( array( 'post__not_in' =>
get_option( 'sticky_posts' ) ) );
```

Exclude sticky posts from a category. Return ALL posts within the category, but don't show sticky posts at the top. The ‘sticky posts’ will still show in their natural position (e.g. by date):

```
1 $query = new  
WP_Query( 'ignore_sticky_posts=1&posts_per_page=3&cat=6' )  
;
```

Exclude sticky posts from a category. Return posts within the category, but exclude sticky posts completely, and adhere to paging rules:

```
1 $paged = get_query_var( 'paged' ) ? get_query_var( 'paged' )  
2 : 1;  
3 $sticky = get_option( 'sticky_posts' );  
4 $args = array(  
5     'cat' => 3,  
6     'ignore_sticky_posts' => 1,  
7     'post__not_in' => $sticky,  
8     'paged' => $paged  
9 );  
$query = new WP_Query( $args );
```

Note:

Use `get_query_var('page')` if you want this query to work in a Page template that you've set as your static front page.

```
1 <?php  
2 /* Get all Sticky Posts */  
3 $sticky = get_option( 'sticky_posts' );  
4  
5 /* Sort Sticky Posts, newest at the top */  
6 rsort( $sticky );  
7  
8 /* Get top 5 Sticky Posts */  
9 $sticky = array_slice( $sticky, 0, 5 );  
10  
11 /* Query Sticky Posts */  
12 $query = new WP_Query( array( 'post__in' => $sticky,  
13 'ignore_sticky_posts' => 1 ) );  
?>
```

Style Sticky Posts

To help theme authors perform simpler styling, the `post_class()` function is used to add `class="..."` to DIV, just add:

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

The `post_class()` outputs the class="whatever" piece for that div. This includes several different classes of value: post, hentry (for hAtom microformat pages), category-X (where X is the slug of every category the post is in), and tag-X (similar, but with tags). It also adds "sticky" for posts marked as Sticky Posts.

```
1 .sticky { color:red; }
```

Note:

The "sticky" class is only added for sticky posts on the first page of the home page (`is_home()` is true and `is_paged()` is false)

Sidebars

What are Sidebars

A sidebar is any widgetized area of your theme. Widget areas are places in your theme where users can add their own widgets. You do not need to include a sidebar in your theme, but including a sidebar means users can add content to the widget areas through the Customizer or the Widgets Admin Panel.

Widgets can be employed for a variety of purposes, ranging from listing recent posts to conducting live chats.

Tip:

The name "sidebars" comes from a time when widget areas were normally created in a long strip to the lefthand or righthand side of a blog. Today, sidebars have evolved beyond their original name. They can be included anywhere on your website. Think of a sidebar as **any area that contains widgets**.

Registering a Sidebar

To use sidebars, you must register them in `functions.php`.

To begin, `register_sidebar()` has several parameters that should always be defined regardless of whether they are marked as optional. These include x, y, and z.

- **name** – your name for the sidebar. This is the name users will see in the Widgets panel.

- **id** – must be lowercase. You will call this in your theme using the `dynamic_sidebar` function.
- **description** – A description of the sidebar. This will also be shown in the admin Widgets panel.
- **class** – The CSS class name to assign to the widget's HTML.
- **before_widget** – HTML that is placed before every widget.
- **after_widget** – HTML that is placed after every widget. Should be used to close tags from `before_widget`.
- **before_title** – HTML that is placed before the title of each widget, such as a header tag.
- **after_title** – HTML that is placed after every title. Should be used to close tags from `before_title`.

To register a sidebar we use `register_sidebar` and the `widgets_init` function.

```
function themename_widgets_init() {
    register_sidebar( array(
        'name'          => __( 'Primary Sidebar',
'theme_name' ),
        'id'            => 'sidebar-1',
        'before_widget' => '<aside id="%1$s" class="widget
%2$s">',
        'after_widget'  => '</aside>',
        'before_title'  => '<h3 class="widget-title">',
        'after_title'   => '</h3>',
    ) );
}

register_sidebar( array(
    'name'          => __( 'Secondary Sidebar',
'theme_name' ),
    'id'            => 'sidebar-2',
    'before_widget' => '<ul><li id="%1$s" class="widget
%2$s">',
    'after_widget'  => '</li></ul>',
    'before_title'  => '<h3 class="widget-title">',
    'after_title'   => '</h3>',
) );
}
```

Registering a sidebar tells WordPress that you're creating a new widget area in **Appearance > Widgets** that users can drag their widgets to. There are two functions for registering sidebars:

- register_sidebar()
- register_sidebars()

The first lets you register one sidebar and the second lets you register multiple sidebars.

Tip:

It is recommended that you register sidebars individually as it allows you to give unique and descriptive names to each sidebar.

Examples

For widget areas in header and footer, it makes sense to name them "Header Widget Area" and "Footer Widget Area", rather than "Sidebar 1" and "Sidebar 2" (which is the default). This provides a useful description of where the sidebar is located.

The following code, added to `functions.php` registers a sidebar:

```
add_action( 'widgets_init', 'my_register_sidebars' );
function my_register_sidebars() {
    /* Register the 'primary' sidebar. */
    register_sidebar(
        array(
            'id'          => 'primary',
            'name'        => __( 'Primary Sidebar' ),
            'description' => __( 'A short description of
the sidebar.' ),
            'before_widget' => '<div id="%1$s" class="widget
%2$s">',
            'after_widget'  => '</div>',
            'before_title'  => '<h3 class="widget-title">',
            'after_title'   => '</h3>',
        )
    );
    /* Repeat register_sidebar() code for additional
sidebars. */
}
```

The code does the following:

- `register_sidebar` – tells WordPress that you're registering a sidebar

- `'name' => __('Primary Widget Area', 'mytheme')`, -
is the widget area's name that will appear in Appearance > Widgets
- `'id' => 'sidebar-1'` – assigns an ID to the sidebar. WordPress uses
'id' to assign widgets to a specific sidebar.
- `before_widget/after_widget` – a wrapper element for widgets
assigned to the sidebar. The "%1\$s" and "%2\$s" should always be left in `id` and
`class` respectively so that plugins can make use of them. By default, WordPress
sets these as list items but in the above example they have been altered to `div`.
- `before_title/after_title` – the wrapper elements for the widget's
title. By default, WordPress sets it to `h2` but using `h3` makes it more semantic.
Once your sidebar is registered, you can display it in your theme.

Displaying Sidebars in your Theme

Now that your sidebars are registered, you will want to display them in your theme. To do this, there are two steps:

1. Create the `sidebar.php` template file and display the sidebar using the `dynamic_sidebar` function
2. Load in your theme using the `get_sidebar` function

Create a Sidebar Template File

A sidebar template contains the code for your sidebar. WordPress recognizes the file `sidebar.php` and any template file with the name `sidebar-{name}.php`.

This means that you can organize your files with each sidebar in its own template file.

Example:

1. Create `sidebar-primary.php`
2. Add the following code:

```
<div id="sidebar-primary" class="sidebar">
    <?php dynamic_sidebar( 'primary' ); ?>
</div>
```

Note that `dynamic_sidebar` takes a single parameter of `$index`, which can be either the sidebar's name or id.

Load your Sidebar

To load your sidebar in your theme, use the `get_sidebar` function. This should be inserted into the template file where you want the sidebar to display. To load the default `sidebar.php` use the following:

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

To display the Primary sidebar, pass the `$name` parameter to the function:

```
1 <?php get_sidebar( 'primary' ); ?>
```

Customizing your Sidebar

There are lots of ways that you can customize your sidebars. Here are some examples:

Display Default Sidebar Content

You may wish to display content if the user hasn't added any widgets to the sidebar yet. To do this, you use the `is_sidebar_active()` function to check to see if the sidebar has any widgets. This accepts the `$index` parameter which should be the ID of the sidebar that you wish to check.

This code checks to see if the sidebar is active, if not it displays some content:

```
<div id="sidebar-primary" class="sidebar">
    <?php if ( is_active_sidebar( 'primary' ) ) : ?>
        <?php dynamic_sidebar( 'primary' ); ?>
    <?php else : ?>
        <!-- Time to add some widgets! -->
    <?php endif; ?>
</div>
```

Display Default Widgets

You may want your sidebar to be populated with some widgets by default. For example, display the Search, Archive, and Meta Widgets. To do this you would use:

```
<div id="primary" class="sidebar">
    <?php do_action( 'before_sidebar' ); ?>
    <?php if ( ! dynamic_sidebar( 'sidebar-primary' ) ) : ?>
        <aside id="search" class="widget widget_search">
            <?php get_search_form(); ?>
        </aside>
        <aside id="archives" class="widget">
            <h3 class="widget-title"><?php _e( 'Archives', 'shape' ); ?></h3>
            <ul>
                <?php wp_get_archives( array( 'type' => 'monthly' ) ); ?>
            </ul>
        </aside>
        <aside id="meta" class="widget">
            <h3 class="widget-title"><?php _e( 'Meta', 'shape' ); ?></h3>
            <ul>
                <?php wp_register(); ?>
                <li><?php wp_loginout(); ?></li>
                <?php wp_meta(); ?>
            </ul>
        </aside>
    <?php endif; ?>
</div>
```

Widgets

A widget adds content and features to a widget area (also called a [sidebar](#)). Widget areas provide a way for users to customize their site. A widget area can appear on multiple pages or on only one page. Your theme might have just one widget area or many of them. Some examples of ways you can use widget areas are:

- Lay out a homepage using widgets. This allows site owners to decide what should appear in each section of their homepage.
- Create a footer that users can customize with their own content.
- Add a customizable sidebar to a blog.

A widget is a PHP object that outputs some HTML. The same kind of widget can be used multiple times on the same page (e.g. the Text Widget). Widgets can save data in the database (in the options table).

When you create a new kind of widget, it will appear in the user's Administration Screens at **Appearance > Widgets**. The user can add the widget to a widget area and customize the widget settings from the WordPress admin.

Built-in versus stand-alone widgets

A set of widgets is included with the default WordPress installation. In addition to these standard widgets, extra widgets can be included by themes or plugins. An advantage of widgets built into themes or plugins is to provide extra features and increase the number of widgets.

A disadvantage is that if a theme is changed or a plugin is deactivated, the plugin or theme widget's functionality will be lost. However, the data and preferences from the widget will be saved to the options table and restored if the theme or plugin is reactivated.

If you include a widget with your theme, it can only be used if that theme is active. If the user decides to change their theme they will lose access to that widget. However, if the widget is included with a plugin, the user can change their theme without losing access to the widget functionality.

Anatomy of a Widget

Visually, a widget comprises two areas:

1. Title Area
2. Widget Options

For example, here is the layout of the built-in text widget in the admin and on the front-end:

Text

Title: 1

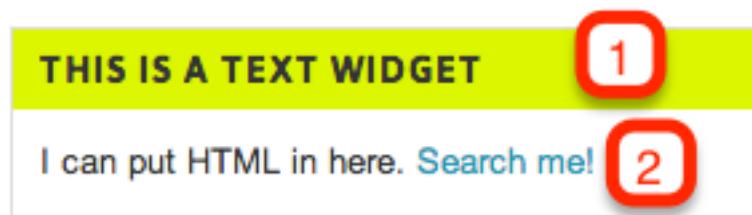
I can put HTML in here. 2

```
<a href="http://google.com">Search me!</a>
```

Automatically add paragraphs

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

A widget form in the admin area.



A widget as it appears to a site visitor.

HTML output for this widget looks like this:

```
1 <div id="text-7" class="widget widget_text">
2 <!-- Widget Container -->
3     <div class="widget-wrap">
4         <h4 class="widgettitle">This is a text widget</
5 h4>
6         <!-- Title -->
7         <div class="textwidget">
8             <!-- Content of Widget -->
9                 I can put HTML in here. <a href="Search me!>
11             </div>
12         </div>
13     </div>
14 </div>
```

Each widget has its own way of outputting HTML that is relevant to the data being displayed. The wrapper tags for the widget are defined by the widget area in which it is being displayed.

The PHP code necessary to create a widget like the built-in text widget looks like this:

```
1<?php
2
3class My_Widget extends WP_Widget {
4
5    function __construct() {
6
7        parent::__construct(
8            'my-text', // Base ID
9            'My Text' // Name
10);
11
12    add_action( 'widgets_init', function() {
13        register_widget( 'My_Widget' );
14    });
15
16}
17
18public $args = array(
19    'before_title' => '<h4 class="widgettitle">',
20    'after_title'  => '</h4>',
21    'before_widget' => '<div class="widget-wrap">',
22    'after_widget'  => '</div></div>'
23);
24
25public function widget( $args, $instance ) {
26
27    echo $args['before_widget'];
28
29    if ( ! empty( $instance['title'] ) ) {
30        echo $args['before_title'] .
31apply_filters( 'widget_title', $instance['title'] ) .
32$args['after_title'];
33    }
34
35    echo '<div class="textwidget">';
36
37    echo esc_html__( $instance['text'], 'text_domain' );
38
39    echo '</div>';
40
41    echo $args['after_widget'];
42
43}
```

The code above will be explained in detail later in the article.

Developing Widgets

To create and display a widget, you need to do the following:

1. Create your widget's class by extending the standard [WP_Widget](#) class and some of its functions.
2. Register your widget so that it's made available in the **Widgets** screen.
3. Make sure that your theme has at least one [widget area](#) in which to add the widgets.

Your Widget Class

The [WP_Widget](#) class is located in [wp-includes/class-wp-widget.php](#)

```
1 <?php
2
3 class My_Widget extends WP_Widget {
4
5     public function __construct() {
6         // actual widget processes
7     }
8
9     public function widget( $args, $instance ) {
10        // outputs the content of the widget
11    }
12
13    public function form( $instance ) {
14        // outputs the options form in the admin
15    }
16
17    public function update( $new_instance,
18 $old_instance ) {
19        // processes widget options to be saved
20    }
21
22 ?>
```

The documentation for each of these functions can be found in the widget class code:

1. construct: Set up your widget with a description, name, and display width in your admin.

2. widget: Process the widget options and display the HTML on your page. The `$args` parameter provides the HTML you can use to display the widget title class and widget content class.

3. form: Display the form that will be used to set the options for your widget. If your widget doesn't have any options, you can skip this function (although it is still best practice to include it even if it's empty).

4. update: Save the widget options to the database. If your widget doesn't have any options, you can skip this function (although it is still best practice to include it even if it's empty).

Registering a Widget

The `register_widget()` function is used to register a widget.

Call this function using the `widgets_init` hook:

```
1 <?php
2 add_action( 'widgets_init', 'wpdocs_register_widgets' );
3
4 function wpdocs_register_widgets() {
5     register_widget( 'My_Widget' );
6 }
7 ?>
```

The HTML that wraps the widget, as well as the class for the title and widget content, is specified at the time you register the widget area using `register_sidebar()`.

Example Text Widget

To build the text widget from the example at the beginning of this article. You will start by setting up a widget class that extends the `WP_Widget` class.

In the class constructor you will call the parent constructor and pass it your widget's base ID and name. Also in the class constructor you will hook into the `widgets_init` action to register your widget.

Next you will declare the arguments you will use when creating your widget. There are four arguments you must define, `before_title`, `after_title`, `before_widget`, and `after_widget`. These arguments will define the code that wraps your widgets title and the widget itself.

After defining your arguments, you will define the `widgets` function. This function takes two parameters, the `$args` array from before, and the `$instance` of the widget, and is the function that will process options from the form and display the HTML for the widget on the front-end of your site. In the example above the widget function simply outputs the widget title, while passing it through the `widget_title` filter. It then outputs a simple widget wrapper and the content of the widget's text field. As outlined in the example, you can access the options from the widget that are stored in the `$instance`.

Next you will define the `form` function. This function takes one parameter, `$instance`, and outputs the form that the user uses to create the widget in the Widgets admin screen. In the example above, the function starts by defining the `$title` and `$text` variables and setting them to the previously entered values, if those values exist. Then it outputs a simple form with a text field for the title and a textarea for the text content.

Lastly you will define the `update` function. This function takes two parameters, `$new_instance` and `$old_instance`, and is responsible for updating your widgets with new options when they are submitted. Here you simply define `$instance` as an empty array. You then set the title and text keys to the `$new_instance` values if they exist. You then return `$instance`.

Finally, when all of the above is defined, you instantiate your new widget class and test your work.

Sample Widget

```
<?php
```

```
/**
```

```
* Adds Foo_Widget widget.
*/
class Foo_Widget extends WP_Widget {

    /**
     * Register widget with WordPress.
     */
    public function __construct() {
        parent::__construct(
            'foo_widget', // Base ID
            'Foo_Widget', // Name
            array( 'description' => __( 'A Foo
Widget', 'text_domain' ), ) // Args
        );
    }

    /**
     * Front-end display of widget.
     *
     * @see WP_Widget::widget()
     *
     * @param array $args      Widget arguments.
     * @param array $instance Saved values from
database.
     */
    public function widget( $args, $instance ) {
        extract( $args );
        $title = apply_filters( 'widget_title',
$instance['title'] );

        echo $before_widget;
        if ( ! empty( $title ) ) {
            echo $before_title . $title .
$title;
        }
        echo __( 'Hello, World!', 'text_domain' );
        echo $after_widget;
    }

    /**

```

```

 * Back-end widget form.
 *
 * @see WP_Widget::form()
 *
 * @param array $instance Previously saved values
from database.
 */
public function form( $instance ) {
    if ( isset( $instance[ 'title' ] ) ) {
        $title = $instance[ 'title' ];
    }
    else {
        $title = __( 'New title',
'text_domain' );
    }
    ?>
    <p>
        <label for=<?php echo $this->get_field_name( 'title' ); ?>><?php _e( 'Title:' );
?></label>
        <input class="widefat" id=<?php echo $this->get_field_id( 'title' ); ?>" name=<?php echo $this->get_field_name( 'title' ); ?>" type="text"
value=<?php echo esc_attr( $title ); ?>" />
    </p>
<?php
}

/**
 * Sanitize widget form values as they are saved.
 *
 * @see WP_Widget::update()
 *
 * @param array $new_instance Values just sent to
be saved.
 * @param array $old_instance Previously saved
values from database.
 *
 * @return array Updated safe values to be saved.
*/

```

```

    public function update( $new_instance,
$new_instance ) {
    $instance = array();
    $instance[ 'title' ] = ( !
empty( $new_instance[ 'title' ] ) ) ?
strip_tags( $new_instance[ 'title' ] ) : '';
}

    return $instance;
}

} // class Foo_Widget

?>

```

This sample widget can then be registered in the widgets_init hook:

```

1 <?php
2 // Register Foo_Widget widget
3 add_action( 'widgets_init', 'register_foo' );
4
5 function register_foo() {
6     register_widget( 'Foo_Widget' );
7 }
8 ?>

```

Example with a Namespace

If you use PHP 5.3 with namespaces you should call the constructor directly as in the following example:

```

1 <?php
2 namespace a\b\c;
3
4 class My_Widget_Class extends \WP_Widget {
5     function __construct() {
6         parent::__construct( 'baseID', 'name' );
7     }
8     // ... rest of the functions
9 }
10 ?>

```

and call the register widget with:

```
1 <?php
2 // Register Foo_Widget widget
3 add_action( 'widgets_init', 'register_my_widget' );
4
5 function register_my_widget() {
6     register_widget( 'a\b\c\My_Widget_Class' );
7 }
8 ?>
```

See [this answer at stack exchange](#) for more detail.

Special considerations

If you want to use a widget inside another template file, rather than in a sidebar, you can use [`the_widget\(\)`](#) to display it programmatically. The function accepts widget class names. You pass the widget class name to the function like this:

```
1 <?php the_title(); ?>
2
3 <div class="content">
4     <?php the_content(); ?>
5 </div>
6
7 <div class="widget-section">
8     <?php the_widget( 'My_Widget_Class' ); ?>
9 </div>
```

You may want to use this approach if you need to use a widget in a specific area on a page, such as displaying a list of events next to a form in a section on the front page of your site or displaying an email capture form on a mega-menu alongside your navigation.

Navigation Menus

Navigation Menus are customizable menus in your theme. They allow users to add Pages, Posts, Categories, and URLs to the menu. To create a navigation menu you'll need to register it, and then display the menu in the appropriate location in your theme.

Register Menus

In your theme's functions.php, you need to register your menu(s). This sets the name that will appear at **Appearance -> Menus**.

First of all, you will use [register_nav_menus\(\)](#) to register the menu.

In this example, two locations are added to the "Manage Locations" tab: "Header Menu" and "Extra Menu".

```
1 function register_my_menus() {  
2     register_nav_menus(  
3         array(  
4             'header-menu' => ___('Header Menu'),  
5             'extra-menu' => ___('Extra Menu')  
6         )  
7     );  
8 }  
9 add_action('init', 'register_my_menus');
```

Display Menus

Once you've registered your menus, you need to use [wp_nav_menu\(\)](#) to tell your theme where to display them. For example, add the following code to your header.php file to display the header-menu that was registered above.

```
1 wp_nav_menu( array( 'theme_location' => 'header-menu' ) );
```

Note:

A full list of parameters can be found in the [wp_nav_menu\(\)](#) page in the function reference

Repeat this process for any additional menus you want to display in your theme.

Optionally, you can add a container class which allows you to style the menu with CSS.

```
1 wp_nav_menu(  
2     array(  
3         'theme_location' => 'extra-menu',  
4         'container_class' => 'my_extra_menu_class'  
5     )  
6 );
```

Note:

A full list of CSS Classes can be found in the [wp_nav_menu\(\)](#) page in the function reference. You can use these to style your menus.

Display Additional Contents

Below is a simplified version of the Twenty Seventeen footer social menu, which displays `span` elements before and after the menu item label text.

```
1 wp_nav_menu(  
2     array(  
3         'menu' => 'primary',  
4         'link_before' => '<span class="screen-reader-text">',  
5         'link_after' => '</span>',  
6     )  
7 );
```

The output will display as...

```
1 <div class="menu-social-container">  
2     <ul id="menu-social">  
3         <li id="menu-item-1">  
4             <a href="http://twitter.com/"><span class="screen-  
5 reader-text">Twitter</span>  
6             </a>  
7         </li>  
8     </ul>  
9 </div>
```

Note:

To display text between the `` and `<a>` elements for each menu item, use `before` and `after` parameters.

Define Callback

By default, WordPress displays the first non-empty menu when the specified menu or location is not found, or generates a Page menu when there is no custom menu selected.

To prevent this, use the `theme_location` and `fallback_cb` parameters.

```
1 wp_nav_menu(  
2   array(  
3     'menu' => 'primary',  
4     // do not fall back to first non-empty menu  
5     'theme_location' => '__no_such_location',  
6     // do not fall back to wp_page_menu()  
7     'fallback_cb' => false  
8   )  
9 );
```

Pagination

Pagination allows your user to *page* back and forth through multiple pages of content.

WordPress can use pagination when:

- Viewing lists of posts when more posts exist than can fit on one page, or
- Breaking up longer posts by manually by using the following tag.

●	● <! --nextpage-->	1	.	
---	--------------------	---	---	--

Using Pagination to Navigate Post Lists

The most common use for pagination in WordPress sites is to break up long lists of posts into separate pages. Whether you're viewing a category, archive, or default index page for a blog or site, WordPress only shows 10 posts per page by default. Users can change the number of posts that appear on each page on the Reading screen: **Admin > Settings > Reading**.

Loop with Pagination

This simplified example shows where you can add pagination functions for the main loop. Add the functions just before or after the loop.

```

1 <?php if ( have_posts() ) : ?>
2
3     <!-- Add the pagination functions here. -->
4
5     <!-- Start of the main loop. -->
6     <?php while ( have_posts() ) : the_post(); ?>
7
8     <!-- the rest of your theme's main loop -->
9
10    <?php endwhile; ?>
11    <!-- End of the main loop -->
12
13    <!-- Add the pagination functions here. -->
14
15
16 <div class="nav-previous alignleft"><?php
17 next_posts_link( 'Older posts' ); ?></div>
18
19
20
21 <div class="nav-next alignright"><?php
22 previous_posts_link( 'Newer posts' ); ?></div>
23
24
25 <?php else : ?>
26
27 <?php _e('Sorry, no posts matched your criteria.');?>
28
29 <?php endif; ?>
```

Methods for displaying pagination links

Note:

When using any of these pagination functions outside the template file with the loop that is being paginated, you must call the global variable \$wp_query.

```

1 function your_themes_pagination(){
2     global $wp_query;
3     echo paginate_links();
4 }
```

WordPress has numerous functions for displaying links to other pages in your loop. Some of these functions are only used in very specific contexts. You would use a different

function on a single post page then you would on a archive page. The following section covers archive template pagination functions. The section after that cover single post pagination.

Simple Pagination

posts_nav_link

One of the simplest methods is [posts_nav_link\(\)](#). Simply place the function in your template after your loop. This generates both links to the next page of posts and previous page of posts where applicable. This function is ideal for themes that have simple pagination requirements.

```
1 posts_nav_link();
```

next_posts_link & prev_posts_link

When building a theme, use [next_posts_link\(\)](#) and [prev_posts_link\(\)](#). to have control over where the previous and next posts page link appears.

```
1 next_posts_link();
2 previous_posts_link();
```

If you need to pass the pagination links to a PHP variable, you can use

[get_next_posts_link\(\)](#) and [get_previous_posts_link\(\)](#).

```
1 $next_posts = get_next_posts_link();
2 $prev_posts = get_previous_posts_link();
```

Numerical Pagination

When you have many pages of content it is a better experience to display a list of page numbers so the user can click on any one of the page links rather than having to repeatedly click next or previous posts. WordPress provides several functions for automatically displaying a numerical pagination list.

For WordPress 4.1+

If you want more robust pagination options, you can use [the_posts_pagination\(\)](#) for WordPress 4.1 and higher. This will output a set of page numbers with links to previous and next pages of posts.

```
1 the_posts_pagination();
```

For WordPress prior to 4.1

If you want your pagination to support older versions of WordPress, you must use [paginate_links\(\)](#).

```
1 echo paginate_links();
```

Pagination Between Single Posts

All of the previous functions should be used on index and archive pages. When you are viewing a single blog post, you must use [prev_post_link](#) and [next_post_link](#). Place the following functions below the loop on your single.php.

```
1 previous_post_link();
2 next_post_link();
```

Pagination within a post

WordPress gives you a tag that can be placed in post content to enable pagination for that post.

```
1 <!--nextpage-->
```

If you use that tag in the content, you need to ensure that the [wp_link_pages](#) function is placed in your single.php template within the loop.

```
1 <?php if ( have_posts() ) : ?>
2
3     <!-- Start of the main loop. -->
4     <?php while ( have_posts() ) : the_post(); ?>
5
6         <?php the_content(); ?>
7
8         <?php wp_link_pages(); ?>
9
10    <?php endwhile; ?>
11
12 <?php endif; ?>
```

Media

WordPress enables theme developers to customize the look, feel, and functionality of the platform's core media capabilities.

General

In WordPress you can upload, store, and display a variety of media such as image, video and audio files. Media can be uploaded via the **Media > Add New** in the [Administration Screen](#), or Add Media button on the Post/Page Editor.

If a media file is uploaded within the edit screen, it will be automatically attached to the current post being created or edited. If it is uploaded via the Media's Add New Screen or the Media Library Screen, it will be unattached, but may become attached to a post when it is inserted into a post later on.

Retrieving attachment ID or image ID

To retrieve the attachment ID, use [get_posts\(\)](#) or [get_children\(\)](#) function.

This example retrieves the all attachments of the current post and getting all metadata of attachment by specifying the ID.

```
1 // Insert into the Loop
2 $args = array(
3     'post_parent'      => get_the_ID(),
4     'post_type'        => 'attachment',
5 );
6 $attachments = get_posts( $args );
7 if ( $attachments ) {
8     foreach ( $attachments as $attachment ) {
9         $meta_data =
10        wp_get_attachment_metadata( $attachment->ID, false );
11    }
12 }
```

If you want to retrieve images from the post ID only, specify post_mime_type as image.

```
1 $args = array(
2     'post_parent'      => get_the_ID(),
3     'post_type'        => 'attachment',
4     'post_mime_type'   => 'image',
5 );
```

References

- [get_posts\(\)](#)

- get_children()
- wp_get_attachment_metadata()

Compatible media formats

In the Media Library, you can upload any file (with the network administrator's unfiltered_upload) and not just images or videos but text files, office documents or even binary files. Single site administrators do not have the unfiltered_upload capability by default and requires that definition to be set for the capability to kick in. Audio and Video files are processed by the internal library **MediaElement.js**.

- Supported Audio format
- Supported Video format

Cannot retrieve attachment

When you cannot get your attached media by get_posts() or get_children() function, confirm your media is really attached to the post.

From the Administration Screen, Click **Media > Library** to open the Media Library and confirm the value in "Uploaded to" column of the media.

Audio

Audio

You can directly embed audio files and play them back using a simple shortcode **[audio]**. Supported file types are mp3, ogg, wma, m4a and wav.

Audio shortcode

Following shortcode displays audio player that loads music.mp3 file:

```
1 [audio src="music.mp3"]
```

To use the shortcode from template file, use `do_shortcode` function. When music.mp3 file was stored in (theme_directory)/sounds directory, insert following code into your template file:

```
1 $music_file = get_template_directory_uri() . "/sounds/
2 music.mp3";
echo do_shortcode('[audio mp3=' . $music_file . ']');
```

The shortcode creates the audio player as shown in the screenshot below.



Loop and Autoplay

The following basic options are supported:

loop

Allows for the looping of media.

- “off” – Do not loop the media. Default.
- “on” – Media will loop to beginning when finished and automatically continue playing.

autoplay

Causes the media to automatically play as soon as the media file is ready.

- 0 – Do not automatically play the media. Default.
- 1 – Media will play as soon as it is ready.

The following example starts playing music immediately after the page load and loops.

```
1 echo do_shortcode('[audio mp3=' . $music_file . ' loop =
"on" autoplay = 1]');
```

Styling

If you want to change the look & feel of audio player, you can do so by targeting the default class name of “wp-audio-shortcode”. If you insert following code into your style.css, half width of audio player will be displayed.

```
1 .wp-audio-shortcode {
2   width: 50%;
3 }
```

Supported Audio format

- mp3
- ogg
- wma
- m4a
- wav

References

For more technical details such as the internal library that enables this function, refer to

- <https://make.wordpress.org/core/2013/04/08/audio-video-support-in-core/>.
- [Audio Shortcode](#)
- [Function do_shortcode\(\)](#)

Images

Images

Note:

This section describes the handling of images in the Media Library. If you want to display the image file located within your theme directory, just specify the location with the img tag, and style it with CSS.

```

```

Getting img code

To display the image in the Media Library, use [wp_get_attachment_image\(\)](#) function.

```
1 echo wp_get_attachment_image( $attachment->ID, 'thumbnail' );
```

You will get the following HTML output with the selected thumbnail size

```

```

You can specify other size such as ‘full’ for original image or ‘medium’ and ‘large’ for the sizes set at **Settings > Media** in the [Administration Screen](#), or any pair of width and height as array. You’re also free to set custom size strings with [add_image_size\(\)](#);

```
1 echo wp_get_attachment_image( $attachment->ID, Array(640, 480) );
```

Getting URL of image

If you want to get the URL of the image, use

[wp_get_attachment_image_src\(\)](#). It returns an array (URL, width, height, is_intermediate), or `false`, if no image is available.

```
1 $image_attributes =
2 wp_get_attachment_image_src( $attachment->ID );
3 if ( $image_attributes ) : ?>
4     <img src=<?php echo $image_attributes[0]; ?>
      width=<?php echo $image_attributes[1]; ?>" height=<?php
      echo $image_attributes[2]; ?>" />
    <?php endif; ?>
```

Alignments

When adding the image in your site, you can specify the image alignment as right, left, center or none. WordPress core automatically adds CSS classes to align the image:

- alignright
- alignleft
- aligncenter
- alignnone

This is the sample output when center align si chosen

```
<img class="aligncenter size-full wp-image-131" src= ... />
```

In order to take advantage of these CSS classes for alignment and text wrapping, your theme must include the styles in a stylesheet such as the [main stylesheet file](#). You can use the `style.css` bundled with official themes such as Twenty Seventeen for reference.

Caption

If a Caption was specified to image in the Media Library, HTML `img` element was enclosed by the shortcode [caption] and [/caption].

```
<div class="mceTemp"><dl id="attachment_133" class="wp-caption aligncenter" style="width: 1210px"><dt class="wp-caption-dt"></dt><dd class="wp-caption-dd">Sun set over the sea</dd></dl></div>
```

And, it will be rendered as in HTML as the figure tag:

```
<figure id="attachment_133" style="width: 1200px" class="wp-caption aligncenter">
  

  <figcaption class="wp-caption-text">Sun set over the sea</figcaption>

</figure>
```

Similar to alignments, your theme must include following styles.

- `wp-caption`
- `wp-caption-text`

References

- [wp_get_attachment_image\(\)](#)
- [wp_get_attachment_image_src\(\)](#)
- [Styling Images in Posts and Pages](#)
- [CSS \(Codex\)](#)

Galleries

Galleries

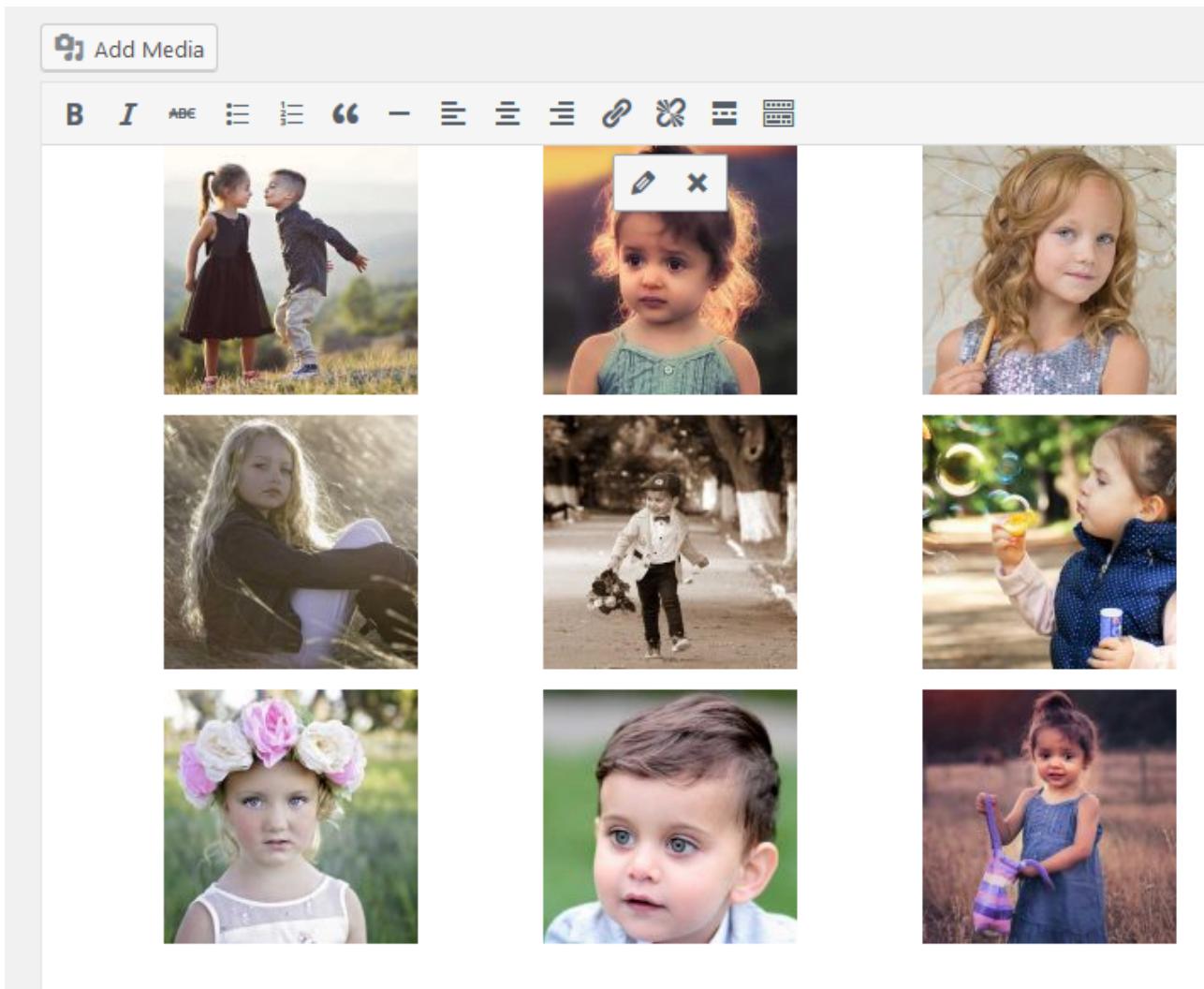


Image galleries are the best way to showcase your pictures on your WordPress sites.

WordPress bundles the **Create Gallery** feature by default in the media uploader which allows you to create a simple gallery.

Note: Before adding a gallery, you must have images in your media library. Otherwise, you need to upload the images into the library and can proceed on gallery creation.

Gallery shortcode

The **Gallery** feature allows you to add one or more image galleries to your posts and pages using a simple Shortcode.

The basic form of gallery shortcode is:

```
1 [gallery]
```

Tip:

If you use the [gallery] shortcode without using the `ids` argument in your post or page, only images that are “attached” to that post or page will be displayed.

If you need to add multiple images with ID's, use the following sample shortcode

```
1 [gallery ids="10, 205, 552, 607"]  
2 //Note: 10, 205, 552 and 607 are the IDs of respected  
image.
```

Tip:

NOTE: find the proper IDs of the images for the gallery. Go to Media library and click on the respected image and ID will appear on the URL.

To use the shortcode from the template file, use the `do_shortcode()` function. Insert the following code into your template file:

```
1 echo do_shortcode( [gallery] );
```

If you need to use the shortcode with IDs, insert the following code in your template file:

```
1 echo do_shortcode( [gallery ids="10, 205, 552, 607"] );
```

Usage

There are may options that may be specified using the below syntax:

```
1 [gallery option1="value1" option2="value2"]
```

If you want to print the gallery directly on the template file, use ``do_shortcode()`` function like below:

```
1 <?php echo do_shortcode(' [gallery option1="value1"] '); ?>
```

If you need to filter the shortcodes, the following example gives you some tips

```
1 // Note: 'the_content' filter is used to filter the
2 content of the
3 // post after it is retrieved from the database and before
4 it is
5 // printed to the screen
6 <?php $gallery_shortcode = '[gallery id="" .
intval( $post->post_parent ) . '"];
    print apply_filters( 'the_content', $gallery_shortcode
);
?>
```

Supported Options

Gallery Shortcodes supports the basic options which are listed below:

Orderby

'orderby' specifies the order the thumbnails show up. The default order is 'menu_order'.

- menu_order: You can reorder the images in the Gallery tab of the Add Media popup
- title: Order by the title of the image in the Media Library
- post_date: Sort by date/time
- rand: Order randomly
- ID: Specify the post ID

Order

order specify the sort order used to display thumbnail; ASC or DESC. For Example, to sort by ID and DESC:

```
1 [gallery order="DESC" orderby="ID"]
```

If you need to print it on template file, use the [do_shortcode\(\)](#) function;

```
1
```

columns

The Columns options specify the number of columns in the gallery. The default value is 3.

If you want to increase the number of column in the galley, use the following shortcode.

```
1 [gallery columns="4"]
```

If you need to print it on your template file, use the [do_shortcode\(\)](#) function;

```
1 <?php echo do_shortcode(' [gallery columns="4"] '); ?>
```

IDs

The IDs option on the gallery shortcode loads images with specific post IDs.

If you want to display the attached image with the specific post ID, follow the following code example.

```
1 // Note: remove each space between brackets and 'gallery'  
2 and brackets and `123``.  
3 //Here "123" stands for the post IDs. If you want to  
4 display more than  
//one ID, separate the IDs by a comma `,`.  
[ gallery id="123" ]
```

Use ‘do_shortcode’ function to print the gallery with IDs on template files like below:

```
1 // Note: remove each space between brackets and 'gallery'  
2 and brackets and `123``.  
<?php echo do_shortcode(' [ gallery id="123" ] '); ?>
```

Size

Size determines the image size to use for the thumbnail display. Valid values include “thumbnail”, “medium”, “large”, “full” and any other additional image size that was registered with [add_image_size\(\)](#). The default value is “thumbnail”. The size of the images for “thumbnail”, “medium” and “large” can be configured in WordPress admin panel under Settings > Media.

For example, to display a gallery of medium sized images:

```
1 [gallery size="medium"]
```

Some advanced options are also available on Gallery shortcodes.

itemtag

The name of the HTML tag used to enclose each item in the gallery. The default is “dl”.

icontag

The name of the HTMLtag used to enclose each thumbnail icon in the gallery. The default is “dt”.

captiontag

The name of the HTML tag used to enclose each caption. The default is “dd”.

You are allowed to change the defaults.

```
1 [gallery itemtag="div" icontag="span" captiontag="p"]
```

Link

Specify where you want the image to link. The default value links to the attachment’s [permalink](#). Options:

- file – Link directly to image file
- none – No link

Example:

```
1 [gallery link="file"]
```

Include

Include allows you to insert an “array” of comma separated attachment IDs to show only the images from these attachments.

```
1 [gallery include="23,39,45"]
```

Exclude

Exclude callows you to insert an “array” of comma separated attachment IDs to not show the images from these attachments. Please note that include and exclude cannot be used together.

```
1 [gallery exclude="21,32,43"]
```

References

For more technical details take a reference from below links

- [Gallery Shortcode](#)
- [Function do_shortcode\(\)](#)

Video

Video

The WordPress video feature allows you to embed video files and play them back using a simple shortcode **[video]**. Supported file types are mp4, m4v, webm, ogv, wmv and flv.

Video shortcode

Following shortcode displays video player that loads pepper.mp4 file:

```
1 [video src="pepper.mp4"]
```

To use the shortcode in the template file, use the [do_shortcode\(\)](#) function. If the video file is stored in your theme directory, get the file url directly using [get_template_directory_uri\(\)](#) or [get_stylesheet_uri\(\)](#)

```
1 $video_file = get_template_directory_uri() . "/videos/
2 pepper.mp4";
echo do_shortcode('[video mp4=' . $video_file . ']');
```

The following video player will be loaded.

Loop and Autoplay

The shortcode video has the same option with audio. Refer to the related section for the [loop and autoplay](#) options.

The following example starts playing the video immediately after the page load and loops.

```
1 echo do_shortcode('[video mp4=' . $video_file . ' loop =
"on" autoplay = 1]');
```

Initial image and Styling

The following basic options are supported:

Poster

Defines image to show as placeholder before the media plays.

The following same code takes `album_cover.jpg` stored in (theme directory)/images folder as the initial image:

```
1 echo do_shortcode('[video mp4=' . $video_file . ' poster =  
' . get_template_directory_uri() . '/images/  
album_cover.jpg']);
```

Height

Defines height of the media. Value is automatically detected on file upload. When you omit this option, the media file height is used.

Width

Defines width of the media. Value is automatically detected on file upload. When you omit this option, the media file width is used.

Tip:

The theme's `content_width` sets the maximum width.

The following example will load the audio player with 320 pixels width and 240 pixels height:

```
1 echo do_shortcode('[video mp4=' . $video_file . ' width =  
320 height = 240']);
```

Styling

If you want to change look & feel of video player from stylesheet, you can target the class name of "wp-video-shortcode". If you want to show the audio player like above in 320 x 240 size, insert following code into your stylesheet.

```
1 .wp-video-shortcode {  
2   width: 320px;  
3   height: 240px;  
4 }
```

Supported Video format

- mp4
- m4v

- webm
- ogv
- wmv
- flv

References

For more technical details such as internal library that enables this function, refer to

- <https://make.wordpress.org/core/2013/04/08/audio-video-support-in-core/>.
- [Video Shortcode](#)
- [Function do_shortcode\(\)](#)

Featured Images & Post Thumbnails

Featured images (also sometimes called Post Thumbnails) are images that represent an individual Post, Page, or Custom Post Type. When you create your Theme, you can output the featured image in a number of different ways, on your archive page, in your header, or above a post, for example.

Enabling Support for Featured Image

Themes must declare support for the Featured Image function before the Featured Image interface will appear on the Edit screen. Support is declared by putting the following in your theme's [functions.php](#) file:

```
1 add_theme_support( 'post-thumbnails' );
```

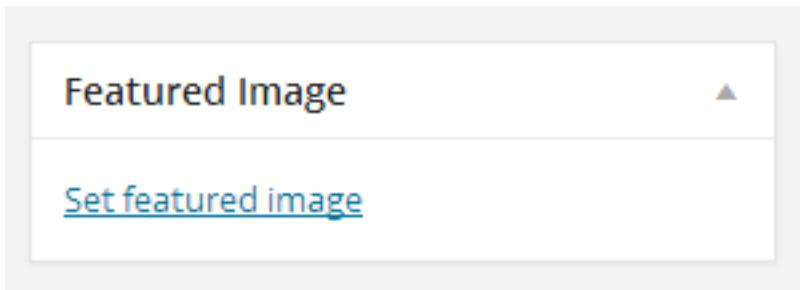
Note:

To enable Featured Image only for specific post types, see [add_theme_support\(\)](#)

Setting a Featured Image

Once you add support for Featured Images, the Featured Image meta box will be visible on the appropriate content item's Edit screens. If a user is unable to see it, they can enable it in their screen options.

By default, the Featured Image meta box is displayed in the sidebar of the Edit Post and Edit Page screens.



Function Reference

[add_image_size\(\)](#) – Register a new image size...

[set_post_thumbnail_size\(\)](#) – Registers an image size for the post thumbnail...

[has_post_thumbnail\(\)](#) – Check if post has an image attached...

[the_post_thumbnail\(\)](#) – Display Post Thumbnail...

[get_the_post_thumbnail\(\)](#) – Retrieve Post Thumbnail...

[get_post_thumbnail_id\(\)](#) – Retrieve Post Thumbnail ID...

Image Sizes

The default image sizes of WordPress are “Thumbnail”, “Medium”, “Large” and “Full Size” (the original size of the image you uploaded). These image sizes can be configured in the WordPress Administration Media panel under >**Settings > Media**. You can also define your own image size by passing an array with your image dimensions:

```
1 the_post_thumbnail(); // Without parameter -> Thumbnail
2 the_post_thumbnail( 'thumbnail' ); // Thumbnail (default
3 150px x 150px max)
4 the_post_thumbnail( 'medium' ); // Medium resolution
5 (default 300px x 300px max)
6 the_post_thumbnail( 'medium_large' ); // Medium-large
7 resolution (default 768px x no height limit max)
the_post_thumbnail( 'large' ); // Large resolution (default
1024px x 1024px max)
the_post_thumbnail( 'full' ); // Original image resolution
(unmodified)
the_post_thumbnail( array( 100, 100 ) ); // Other
resolutions (height, width)
```

Add Custom Featured Image Sizes

In addition to defining image sizes individually using

```
1 the_post_thumbnail( array( , ) );
```

you can create custom featured image sizes in your theme's functions file that can then be called in your theme's template files.

```
1 add_image_size( 'category-thumb', 300, 9999 ); // 300
pixels wide (and unlimited height)
```

Here is an example of how to create custom Featured Image sizes in your theme's `functions.php` file.

```
1 if ( function_exists( 'add_theme_support' ) ) {
2     add_theme_support( 'post-thumbnails' );
3     set_post_thumbnail_size( 150, 150, true ); // default
4     Featured Image dimensions (cropped)
5
6     // additional image sizes
7     // delete the next line if you do not need additional
8     image sizes
9     add_image_size( 'category-thumb', 300, 9999 ); // 300
pixels wide (and unlimited height)
}
```

Set the Featured Image Output Size

To be used in the current Theme's functions.php file.

You can use `set_post_thumbnail_size()` to set the default Featured Image size by resizing the image proportionally (that is, without distorting it):

```
1 set_post_thumbnail_size( 50, 50 ); // 50 pixels wide by 50 pixels tall, resize mode
```

Set the default Featured Image size by cropping the image (either from the sides, or from the top and bottom):

```
1 set_post_thumbnail_size( 50, 50, true ); // 50 pixels wide by 50 pixels tall, crop mode
```

Styling Featured Images

Featured Images are given a class “wp-post-image”. They also get a class depending on the size of the thumbnail being displayed. You can style the output with these CSS selectors:

```
1 img.wp-post-image
2 img.attachment-thumbnail
3 img.attachment-medium
4 img.attachment-large
5 img.attachment-full
```

You can also give Featured Images their own classes by using the attribute parameter in [the_post_thumbnail\(\)](#).

Display the Featured Image with a class “alignleft”:

```
1 the_post_thumbnail( 'thumbnail', array( 'class' =>
'alignleft' ) );
```

Default Usage

```
1 // check if the post or page has a Featured Image assigned
2 to it.
3 if ( has_post_thumbnail() ) {
4     the_post_thumbnail();
5 }
```

Note:

To return the Featured Image for use in your PHP code instead of displaying it, use: `get_the_post_thumbnail()`

```
1 // check for a Featured Image and then assign it to a PHP
2 variable for later use
3 if ( has_post_thumbnail() ) {
4     $featured_image = get_the_post_thumbnail();
}
```

Linking to Post Permalink or Larger Image

Alert:

Don't use these two examples together in the same Theme.

Example 1. To link Post Thumbnails to the Post Permalink in a specific loop, use the following within your Theme's template files:

```
<?php if ( has_post_thumbnail() ) : ?>
    <a href="php the_permalink(); ?&gt;" alt="<?php
the_title_attribute(); ?&gt;"&gt;
        &lt;?php the_post_thumbnail(); ?&gt;
    &lt;/a&gt;
&lt;?php endif; ?&gt;</pre
```

Example 2. To link **all Post Thumbnails** on your website to the Post Permalink, put this in the current Theme's `functions.php` file:

```
1 add_filter( 'post_thumbnail_html', 'my_post_image_html',
2 10, 3 );
3
4 function my_post_image_html( $html, $post_id,
5 $post_image_id ) {
6
7     $html = '<a href="' . get_permalink( $post_id ) . '">' .
8 $html . '</a>';
9     return $html;
10 }
```

This example links to the “large” Post Thumbnail image size and must be used within The Loop.

```
1 if ( has_post_thumbnail() ) {  
2     $large_image_url =  
3     wp_get_attachment_image_src( get_post_thumbnail_id(),  
4     'large');  
5     echo '<a href="' . $large_image_url[0] . '">';  
6     the_post_thumbnail('thumbnail');  
7     echo '</a>';  
8 }
```

Source File

- [wp-includes/post-thumbnail-template.php](#)

External Resources

- [Everything you need to know about WordPress 2.9's post image feature](#)
- [The Ultimative Guide For the_post_thumbnail In WordPress 2.9](#)
- [New in WordPress 2.9: Post Thumbnail Images](#)

Internationalization

What is internationalization?

Internationalization is the process of developing your theme, so it can easily be translated into other languages. Internationalization is often abbreviated as **i18n** (because there are 18 letters between the letters i and n).

Why is internationalization important?

WordPress is used all over the world, in countries where English is not the main language. The strings in the WordPress plugins need to be coded in a special way so that can be easily translated into other languages. As a developer, you may not be able to provide localizations for all your users; however, a translator can successfully localize the theme without needing to modify the source code itself.

How to internationalize your theme?

For the text in the theme to be able to be translated easily the text should not be hardcoded in the theme but be passed as an argument through one of the localization functions in WordPress.

The following example could not be translated unless the translator modified the source code which is not very efficient.

```
1 <h1>Settings Page</h1>
```

By passing the string through a localization function it can be easily parsed to be translated.

```
1 <h1><?php _e( 'Settings Page' ); ?></h1>
```

WordPress uses [gettext](#) libraries to be able to add the translations in PHP. In WordPress you should use the WordPress localization functions instead of the native PHP gettext-compliant translation functions.

Text Domain

The text domain is the second argument that is used in the internationalization functions. The text domain is a unique identifier, allowing WordPress to distinguish between all of the loaded translations. The text domain is only needed to be defined for themes and plugins.

Themes that are hosted on WordPress.org the text domain must match the slug of your theme URL (wordpress.org/themes/<slug>). This is needed so that the translations from translate.wordpress.org work correctly.

The text domain name must use dashes and not underscores and be lowercase. For example, if the theme's name My Theme is defined in the `style.css` or it is contained in a folder called `my-theme` the text domain should be `my-theme`.

The text domain is used in three different places:

1. In the `style.css` theme header
2. As an argument in the localization functions
3. As an argument when loading the translations using
`load_theme_textdomain()` or
`load_child_theme_textdomain()`

style.css theme header

The text domain is added to the `style.css` header so that the theme meta-data like the description can be translated even when the theme is not enabled. The text domain should be same as the one used when [loading the text domain](#).

Example:

```
1 /*  
2 * Theme Name: My Theme  
3 * Author: Theme Author  
4 * Text Domain: my-theme  
5 */
```

Domain Path

The domain path is needed when the translations are saved in a directory other than `languages`. This is so that WordPress knows where to find the translation when the theme is not activated. For example, if .mo files are located in the `languages` folder then Domain Path will be `/languages` and must be written with the first slash. Defaults to the `languages` folder in the theme.

Example:

```
1 /*  
2 * Theme Name: My Theme  
3 * Author: Theme Author  
4 * Text Domain: my-theme  
5 * Domain Path: /languages  
6 */
```

Add text domain to strings

The text domain should be added as an argument to all of the localization functions for the translations to work correctly.

Example 1:

```
1 __( 'Post' )
```

should become

```
1 __( 'Post', 'my-theme' )
```

Example 2:

```
1 _e( 'Post' )
```

should become

```
1 _e( 'Post', 'my-theme' )
```

Example 3:

```
1 _n( 'One post', '%s posts', $count )
```

should become

```
1 _n( 'One post', '%s posts', $count, 'my-theme' )
```

Warning:

The text domain should be passed as a string to the localization functions instead of a variable. It allows parsing tools to differentiate between text domains. Example of what not to do:

```
1 __( 'Translate me.' , $text_domain );
```

Loading Translations

The translations in WordPress are saved in .po and .mo files which need to be loaded. They can be loaded by using the functions [load_theme_textdomain\(\)](#) or [load_child_theme_textdomain\(\)](#). This loads {locale}.mo from your theme's base directory or {text-domain}-{locale}.mo from the WordPress theme language folder in /wp-content/languages/themes/.

Note:

As of version 4.6 WordPress automatically checks the language directory in wp-content for translations from [translate.wordpress.org](#). This means that plugins that are translated via translate.wordpress.org do not require [load_plugin_textdomain\(\)](#) anymore.

If you don't want to add a `load_plugin_textdomain()` call to your plugin you should set the `Requires at least:` field in your `readme.txt` to 4.6.

To find out more about the different language and country codes, see [the list of languages](#).

Watch Out

- Name your MO file as `{locale}.mo` (e.g. `de_DE.po` & `de_DE.mo`) if adding the translation to the theme folder.
- Name your MO file as `{text-domain}-{locale}.mo` (e.g `my-theme-de_DE.po` & `my-theme-de_DE.mo`) if you are adding the translation to the WordPress theme language folder.

Example:

```
1 function my_theme_load_theme_textdomain() {  
2     load_theme_textdomain( 'my-theme',  
3 get_template_directory() . '/languages' );  
4 }  
add_action( 'after_setup_theme',  
    'my_theme_load_theme_textdomain' );
```

This function should ideally be run within the themes' `function.php`.

Language Packs

If you're interested in language packs and how the import to [translate.wordpress.org](#) is working, please read the [Meta Handbook page about Translations](#).

Internationalizing your theme

Now that your translations are loaded, you can start writing every string in your theme with Internationalization functions.

Check the [Internationalization](#) page on the [Common APIs Handbook](#) for more information and best practices.

Localization

What is localization?

Localization describes the subsequent process of translating an internationalized theme.

Localization is abbreviated as `l10n` (because there are 10 letters between the l and the n.)

Please refer to the [Localization](#) page on the [Common APIs handbook](#) to learn how to Localize your theme.

Accessibility

A WordPress theme should generate pages that everyone can use — including those who cannot see or use a mouse. The default WordPress theme generates content in a fairly accessible manner but, as a theme developer, you need to maintain these accessibility standards in your own theme. Although web accessibility can be a complex subject, it boils down to only four principles — that content must be:

Perceivable

Content must be available to all — no matter what user agent is employed or what senses the user lacks.

Operable

Users must be able to move around and operate the final site effectively — irrespective of whether they use a mouse or not.

Understandable

The content should be presented in a manner that supports understanding — including supporting the construction of a mental model of the site for screen reader users.

Similarly, the site's operation (navigation menus, links, forms etc.) should be easily understandable. Building a theme that incorporates known user behaviours (such as underlining links within the main content area) helps in this respect.

Robust

Content must be equally available across a wide range of user agents. Disabled users may employ a range of hardware and software solutions (commonly referred to as “assistive technology”) to allow them to access the Web – including screen reading & voice recognition software; braille readers and switches (single input devices).

A theme that is designed with these four principles in mind should facilitate the creation of an accessible web site.

Headings

This shouldn't need saying but headings are more than just big, bold, text. They are an essential way of breaking content down into logical sub-sections and may be relied upon by screen reader users. The JAWS screen reader (for example) can automatically create a list of headings from any given page. This allows its users to "scan" the page content in a similar way that a sighted person might scroll quickly down a page.

So it is important that heading tags are used logically rather than for any presentational or search engine optimisation (SEO) effect. A page containing twenty **H1** headings might make for good, theoretical, SEO but it pretty much destroys the actual intended use of heading tags — to break a complex page down into sub-sections.

Obviously, the use of heading markup will vary from template to template but do try to keep the intended usage of headings in mind when building your theme's files. One way to check your heading structure is to examine a page using the "View Document Outline" tool in [Firefox's Web Developer Toolbar](#) under its Information menu.

Images

When possible, decorative images should be included using CSS. Where images are being added to your template markup, ensure that they incorporate an appropriate **alt** attribute. Decorative images within a theme might include:

- A banner or header image used alongside header text
- Images accompanying navigation text links

Non-decorative images within a theme might include:

- A banner or header image that replaces header text
- Images used in place of text for navigation

Alt text

- decorative images (null alt)
- Non-decorative images (appropriate alt – with at least 1 example)

To test whether your images within your template markup are informative or purely decorative, use a [simple alt text decision tree](#) to check whether images are using the alt attribute appropriately.

Skip Links

Skip links provide a mechanism that enables users to navigate directly to content or navigation on entering any given page. For example, where content is uppermost in the generated page markup with the menu markup lower down the page, a skip link might allow a user to “skip to navigation”. A “skip to content” link would be used in situations where the main navigation menu is uppermost in the page markup.

In designs with multiple menus and content area, multiple skip links can be used — eg:

- Skip to main navigation
- Skip to secondary navigation
- Skip to footer

These links may be positioned off screen initially using an appropriate CSS technique but should remain available to screen reader users and be visible on focus for sighted keyboard navigators.

Link Text

Link text should describe the resource that it links to – even when the text is read out of context. Some assistive software scans a page for links and presents them to the user as a simple list. In these situations, all the links will be read out of context. So it is important the text used in a link is descriptive. Bare urls should **never** be used as links.

Avoid repetitive non-contextual text strings such as multiple “read more” links. Use something like:

```
1 <?php the_content( the_title('', '', false) . __('Continue  
reading', 'theme_text_domain') ); ?>
```

Or, if you are generating the link outside of `the_content()`, try something like:

```
1 <?php printf( __('%1$s%2$s%3$s - read more',  
'theme_text_domain'), '<span>', get_the_title(), '</  
span>'); ?>
```

Note that, in each case, the unique part of the link (ie the post title) is being rendered first. This will further enhance accessibility in situations where links are read out of context. However, it is acceptable to hide the article title in a ‘read more’ link, as long as the method of hiding leaves the text available to screen readers — such as using absolute positioning or the CSS `text-indent` property to move it off screen. In fact, such an approach may have positive benefits for other user groups by reducing visible screen clutter.

Link Highlighting

With the exception of Opera, the default focus highlighting in most modern browsers is pretty useless. This means that sighted keyboard navigators can quickly become “lost” once they tab into a page as they can no longer easily distinguish which link has the current focus.

Providing good `:focus` and `:active` link highlighting (both in navigation menus & elsewhere) is a very simple solution to this issue. In terms of “return on investment”, it greatly enhances sighted keyboard navigation with minimal effort on your part. It’s also very easy to test. Just put your mouse to one side and try tabbing around pages in your theme. If you get lost easily, so will others. Try replicating your current `:hover` styling and see if that helps.

Link Underlining

Generally speaking, links should be underlined if they are outside navigation menus. Using color alone to distinguish links is insufficient as not everyone can perceive color. Underlining link means that users do not have to “mouse scrub” a page or play “guess which text is a link”. Also consider removing the underline as part of your hover/active/focus styling to, again, ensure that you are not relying on color alone.

Forms

Form Markup

Screen reader software may automatically toggle from a reading mode into an interactive, forms, mode as soon as a `<form>` tag is encountered. In this forms mode, the software

may not render text that is not explicitly associated with a form control, so any text that uses plain `<p></p>` tags may be ignored. In order to extract all important information, screen reader users may need to make two passes at the form to extract all of the visual information — once in forms mode and then again in a reading mode. This represents a significant accessibility barrier.

Therefore, all forms — including the theme's comment form — should ensure that all content within the `<form></form>` tags is explicitly associated to a form control via the `<label>` tag, its `for` attribute and an `id` attribute within the relevant `input` tag. Avoid the use of plain text (e.g. inside `<p></p>` tags) within the `form` block.

To date, no issue has been reported when associating more than one `label` with a single form control, so this “multiple labels” approach can be used, if required.

On Form Submission

Post-submission responses — including any error messages — should always be perceivable. If possible, error messages should be generated at the top of the post-submission page so the user is immediately aware of any issues. Error messages should also make sense when read out of context.

Single Input Forms

Forms that only have a single input (such as a standard search form) may position the associated input label off screen in a manner that ensure the label text is available to screen reader users.

Title Attributes

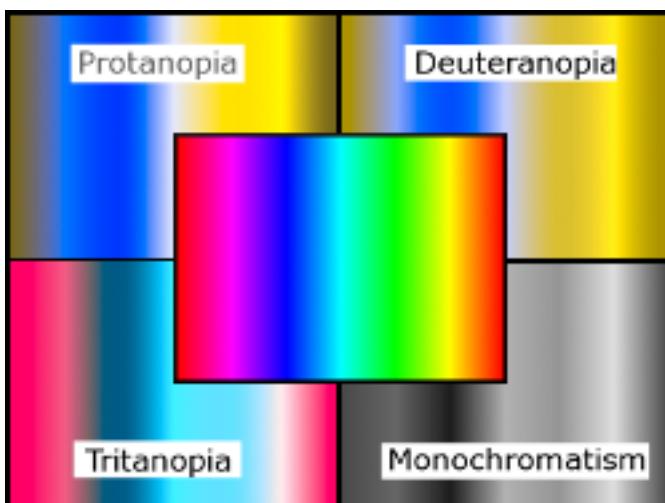
Do not rely on the `title` attribute to convey information. This attribute may not be available to all users (e.g. voice recognition (VR) software users). If information is important enough to be added, it's important enough to be added as clear text so that everyone can see it.

Readability

Sometimes, the simplest thing you can do is to create a more readable page. Reading from a screen is much harder than reading a printed page. Crowded text, lots of images and too much information makes a page very difficult to read. Some general accessible design tips include:

- White space is your friend. Use it as a tool to reduce distraction.
- The CSS line-height property can increase the readability of paragraph text.
- The CSS letter-spacing property can increase the readability of headings and larger text.
- Position navigation elements in logical places and do not allow them to intrude into a page's main content area.
- Make sure all fonts are large enough to be readable.
- If you are using custom fonts, embed them in your theme so that you are not relying on a 3rd party site for font delivery. If you really do need a 3rd party site for fonts, check that the text is still readable if the fonts are not available.

Color



Approximately 10% of all internet users have problems seeing colors, especially those suffering from color blindness.

The commonest form of color blindness affects the red/green spectrum. An affected user may perceive red, orange, yellow, and green as a single color with the rest of the color spectrum being perceived as blue, gradually changing to purple. In rarer forms of the condition, blue and pink colours may predominate or the sufferer may not be able to perceive any colors at all — with all colors reduced to shades of grey.

Whenever possible, use a [color blindness simulator](#) to check your design palette and avoid using color alone to distinguish important elements.

Contrasts

Visitors with a visual impairment may prefer higher contrast pages whilst those with reading difficulties may need a lower contrast. Try to strike a reasonable balance and avoid extremes. Remember that contrast levels apply to images too when they contain text or are used in place of text.

Whenever possible, use a [color contrast tool](#) to check foreground/background contrast ratios across your theme. The minimum contrast ratio that you should aim for is 4.5:1. However, do try to avoid very contrast ratios as these may cause pixelation problems for those using screen magnifiers.

jQuery & JavaScript

You can still use jQuery in an accessible theme. Just don't rely on it for any primary functionality. Test by viewing the theme's output whilst JavaScript is disabled within the browser. Does the site still function effectively?

When implementing jQuery slideshows etc. check that they can be navigated by keyboard alone.

Using ARIA does offer some benefits in terms of future-proofing but, again, should not be relied upon at the present time. It is also worth noting that incorporating ARIA attributes will cause markup validation failures, but it is more valuable to use ARIA than to aim for 100% validation.

In all things script-related, aim for progressive enhancement and graceful degradation.

Validation

Validate your theme's markup and CSS. Validation still represents the best way of ensuring that your theme's pages are displayed to their best advantage across a whole range of web software. Most people would describe communication via a web site as "author -> viewer" but in reality, it's actually "author -> machine(server) -> machine (browser) -> viewer". Since the core of this is machine-machine communication, it makes sense to follow the relevant specifications (wherever possible) to maximise effective communication — irrespective of the final user agent being used to view a site.

That said, validation parsers are just that – dumb software. ARIA, for example, will cause validation failures but still represents the best way forward in creating an accessible theme. Always use your best judgement and, if in doubt, ask...

Accessibility Testing Tools

A word of warning here. Unlike markup & CSS validators that operate using a strict binary “Right/Wrong” approach, accessibility validators have to try and audit complex scenarios. Consequently, their reports can include false positives and/or negatives. Again, do not be afraid to use your best judgement in these situations or ask around. There are plenty of accessibility-related resources — including forums — that can help.

Cross-Browser Testing

By all means, develop your theme in your preferred browser, but do remember to check its output in the current versions of Internet Explorer, Firefox, Chrome, Opera, and Safari. Those using Mac’s built-in screen reader, VoiceOver, may not have an option to use anything other than Safari. Users of the voice recognition software, Dragon Naturally Speaking, may rely on Internet Explorer to effectively navigate web pages. Opera is a popular browser choice amongst disabled users due to its many, inbuilt, accessibility features.

In most cases, disabled users are perfectly capable of optimising their hardware & software to suit their specific needs. Your job — as a theme developer — is to ensure that page content is never scrambled, hidden or lost when viewed across a range of modern web browsers. Obviously, not all browsers will display a given page in exactly the same way but that is rarely an accessibility issue. Most modern browsers, however, can display valid pages in an effective and pleasing manner. Don’t aim for pixel perfection. That way lies madness! Instead aim for an attractive, effective, navigable site in each of your test browsers.

Things to Avoid

Spawning New Windows or Tabs

Spawning new windows breaks the browser ‘Back’ button leaving some sighted keyboard navigator stranded without any means of returning to the original page. For that reason,

please avoid links and other elements that open a new window or tab. If you really, *really*, have to spawn a new window or tab, place a warning in clear text (preferably as part of the link or control text) so that users can make an informed choice.

Autoplay & Animations

Avoid animated content if at all possible. In 1997, a cartoon on television in Japan sent over 700 children to the hospital, including about 500 who had seizures ([citation](#)). And do not play any sounds without the user's express permission.

Autoplay issues apply to:

- Sound — creates problems for screen reader users who may find their software's output drowned out by the site's audio.
- Animations (Flash or .gif images) — can trigger photo-epileptic seizures in some situations.
- Slideshows — may create issues for screen reader users (who are presented with continuously changing content) and sighted keyboard navigators (who may be unable to move past the slideshow).
- Other special effects — effects such as falling snow have been known to trigger photo-epileptic seizures.

If you absolutely must have moving or audio content that (1) starts automatically, (2) lasts more than five seconds, and (3) is presented in parallel with other content, ensure that there is an easily operable mechanism that pauses, stops, or hides it. Also ensure that the visual animation does not flash more than three times in any one second period.

Tabindexing

The `tabindex` attribute (with the exception of negative `tabindex` in specific circumstances) is to be avoided at all costs. As a theme developer, you are not the best person to determine where any one person wants to move to next on a site. Only the user can make that decision, so do not try to hijack their browser. As long as the natural tab order within a page is logical and can be easily perceived, most users are perfectly capable of sorting out their own navigational needs — thank you very much.

Accesskeys

Again — a nice idea in theory — but in reality, often a complete disaster when implemented. Apart from the fact that users have no information as to which shortcuts do what on a site, using anything other than numeric keys risks hijacking shortcut keys within the user's own browsing software.

Make WordPress Accessible

[Make WordPress Accessible](#) is the official blog for the WordPress accessibility group – dedicated to improving accessibility in core WordPress and related projects. Our aim is to provide accessibility suggestions, feedback and assistance to WordPress core, theme and plugin developers.

Anyone can join in the discussions. You can also follow discussions via email or subscribe to feeds for both posts and comments. We would also be more than happy if you [joined us](#) on a more formal basis.

Resources

General

- [W3C Web Accessibility Initiative ‘How To Meet WCAG2.0’ Quick Reference](#)
- [Make WordPress Accessible](#)
- [Accessites.org](#) — general articles on web accessible design.
- [Accessify Forum](#)
- [Evaluation, Repair, and Transformation Tools for Web Content Accessibility](#)

Contrast & Color Testing

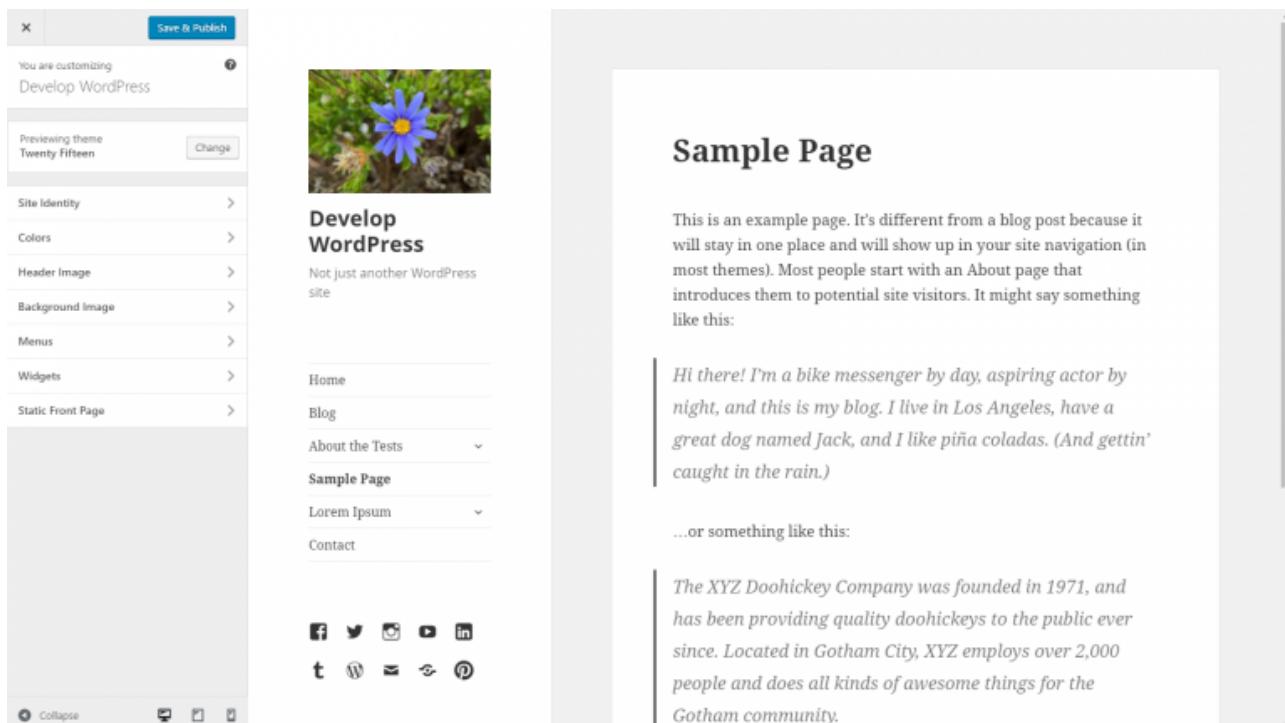
- [Vischeck](#) — Online color blindness simulator
- [Contrast Analyser for Windows and Mac](#) — free desktop tool with color blindness simulators.
- [Sim Daltonism](#) — a color blindness simulator for Mac OS X.
- [Alternative Color Contrast Analyzer](#) — provides a warning for high contrasts.

Toolbars

- [WAVE](#) — Firefox accessibility evaluation tool.
- [Firefox Accessibility Extension](#) — check the use of structural markup in a page.
- [Web Developer Toolbar for Firefox](#) — adds various web developer tools to the browser.

Theme Options – The Customize API

The Customize API (Customizer) is a framework for live-previewing any change to WordPress. It provides a unified interface for users to customize various aspects of their theme and their site, from colors and layouts to widgets, menus, and more. Themes and plugins alike can add options to the Customizer. The Customizer is the canonical way to add options to your theme.



The customizer as it appears in WordPress 4.6 with the Twenty Fifteen theme.

Customizer options can be granted to users with different capabilities on a granular basis, so while most options are visible only to administrators by default, other users may

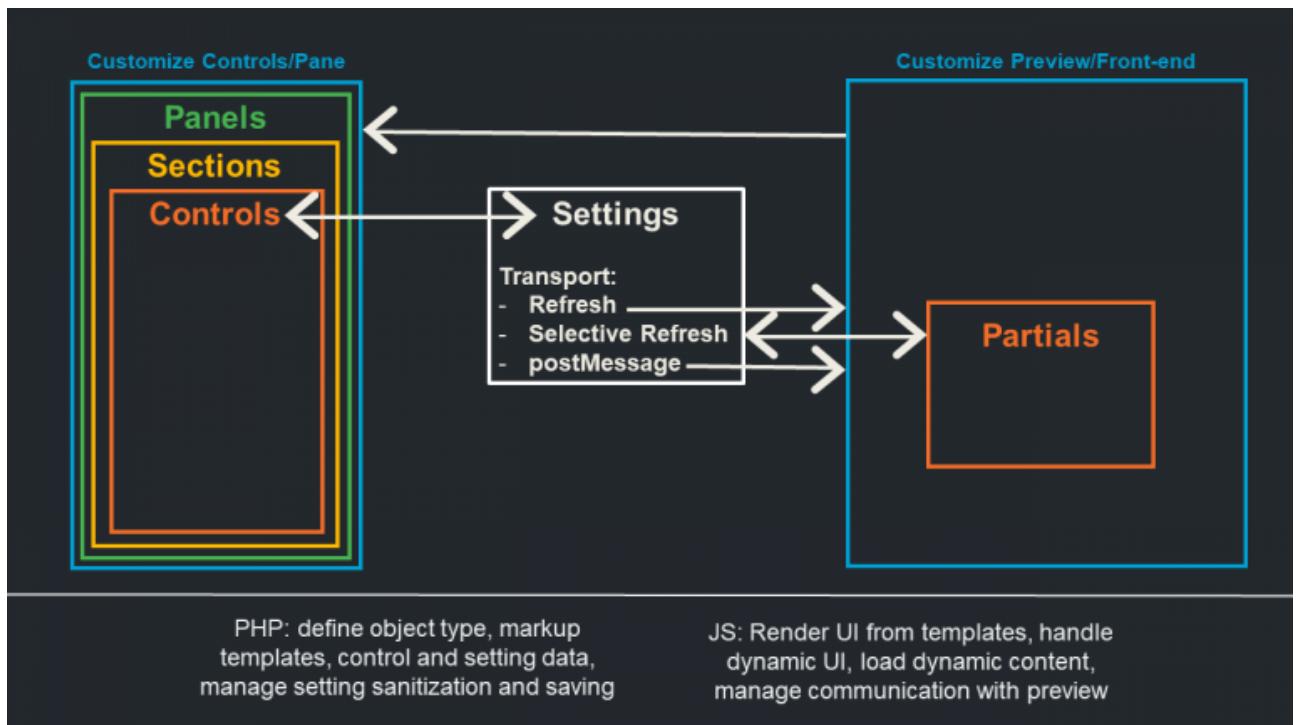
access certain options if you want them to be able to. Different parts of the Customizer can also be contextual to whether they're relevant to the front-end context that the user is previewing. For example, the core widgets functionality only shows widget areas that are displayed on the current page; other widget areas are shown when the user navigates to a page that includes them within the Customizer preview window.

This section contains an overview of the Customize API, including code examples and discussion of best practices. For more details, it is strongly recommended that developers study the core customizer code (all core files containing "customize"). This is considered the canonical, official documentation for the Customize API outside of the inline documentation within the core code.

Customizer Objects

The Customize API is object-oriented. There are four main types of Customizer objects: Panels, Sections, Settings, and Controls. Settings associate UI elements (controls) with settings that are saved in the database. Sections are UI containers for controls, to improve their organization. Panels are containers for sections, allowing multiple sections to be grouped together.

Each Customizer object is represented by a PHP class, and all of the objects are managed by the Customize Manager object, [WP_Customize_Manager](#).



To add, remove, or modify any Customizer object, and to access the Customizer Manager, use the [customize_register](#) hook:

```
1 function themeslug_customize_register( $wp_customize ) {  
2     // Do stuff with $wp_customize, the WP_Customize_Manager  
3     // object.  
4 }  
5 add_action( 'customize_register',  
6             'themeslug_customize_register' );
```

The Customizer Manager provides add_, get_, and remove_ methods for each Customizer object type; each works with an id. The get_ methods allow for direct modification of parameters specified when adding a control.

```
1 add_action( 'customize_register', 'my_customize_register' );  
2 function my_customize_register( $wp_customize ) {  
3     $wp_customize->add_panel();  
4     $wp_customize->get_panel();  
5     $wp_customize->remove_panel();  
6  
7     $wp_customize->add_section();  
8     $wp_customize->get_section();  
9     $wp_customize->remove_section();  
10  
11    $wp_customize->add_setting();  
12    $wp_customize->get_setting();  
13    $wp_customize->remove_setting();  
14  
15    $wp_customize->add_control();  
16    $wp_customize->get_control();  
17    $wp_customize->remove_control();  
18 }
```

Note: themes generally should not modify core sections and panels with the get methods, since themes should not modify core, theme-agnostic functionality. Plugins are encouraged to use these functions where necessary. Themes should not “reorganize” customizer sections that aren’t added by the theme.

Settings

Settings handle live-previewing, saving, and sanitization of your customizer objects. Each setting is managed by a control object. There are several parameters available when adding a new setting:

```
1 $wp_customize->add_setting( 'setting_id', array(
2   'type' => 'theme_mod', // or 'option'
3   'capability' => 'edit_theme_options',
4   'theme_supports' => '', // Rarely needed.
5   'default' => '',
6   'transport' => 'refresh', // or postMessage
7   'sanitize_callback' => '',
8   'sanitize_js_callback' => '', // Basically to_json.
9 ); );
```

Alert:

Important: Do *not* use a setting ID that looks like `widget_*`, `sidebars_widgets[*]`, `nav_menu[*]`, or `nav_menu_item[*]`. These setting ID patterns are reserved for widget instances, sidebars, nav menus, and nav menu items respectively. If you need to use “widget” in your setting ID, use it as a suffix instead of a prefix, for example “`homepage_widget`”.

There are two primary types of settings: options and theme modifications. Options are stored directly in the `wp_options` table of the WordPress database and apply to the site regardless of the active theme. Themes should rarely if ever add settings of the option type. Theme mods, on the other hand, are specific to a particular theme. Most theme options should be theme_mods. For example, a custom CSS plugin could register a custom theme css setting as a theme_mod, allowing each theme to have a unique set of CSS rules without losing the CSS when switching themes then switching back.

Theme_mods vs. Options

```
$wp_customize->add_setting(
    'custom_theme_css' , array(
        'type'      => 'theme_mod',
    ) );

$wp_customize->add_setting(
    'custom_plugin_css' , array(
        'type'      => 'option',
    ) );
```

The screenshot shows a user interface for managing custom CSS. On the left, there's a sidebar with a 'Custom CSS' tab. Under 'Custom Theme CSS', it says 'Theme-specific: safely sticks with each theme when switching themes.' followed by a code snippet: '.site-content article { border: 18px solid #00f; }'. Under 'Custom Plugin CSS', it says 'Theme-agnostic: persists across theme changes.' followed by a code snippet: '.henry .mejs-controls .mejs-time-rail .mejs-time-current { background: #6bf; }'.

* This is not pseudo-code, this is it. Copied and pasted from the Modular Custom CSS Plugin.

Theme_mod vs. option setting type example.

It is usually most important to set the default value of the setting as well as its sanitization callback, which will ensure that no unsafe data is stored in the database. Typical theme usage:

```
1 $wp_customize->add_setting( 'accent_color', array(
2     'default' => '#f72525',
3     'sanitize_callback' => 'sanitize_hex_color',
4 ) );
```

Typical plugin usage:

```
1 $wp_customize->add_setting( 'myplugin_options[color]' ,
2 array(
3     'type' => 'option',
4     'capability' => 'manage_options',
5     'default' => '#ff2525',
6     'sanitize_callback' => 'sanitize_hex_color',
) );
```

Note that the Customizer can handle options stored as keyed arrays for settings using the option type. This allows multiple settings to be stored in a single option that isn't a theme mod. To retrieve and use the values of your Customizer options, use

[get_theme_mod\(\)](#) and [get_option\(\)](#) with the setting id:

```

1 function my_custom_css_output() {
2     echo '<style type="text/css" id="custom-theme-css">' .
3         get_theme_mod( 'custom_theme_css', '' ) . '</style>';
4     echo '<style type="text/css" id="custom-plugin-css">' .
5         get_option( 'custom_plugin_css', '' ) . '</style>';
6 }
7 add_action( 'wp_head', 'my_custom_css_output' );

```

Note that the second argument for `get_theme_mod()` and `get_option()` is the default value, which should match the default you set when adding the setting.

Controls

Controls are the primary Customizer object for creating UI. Specifically, every control must be associated with a setting, and that setting will save user-entered data from the control to the database (in addition to displaying it in the live-preview and sanitizing it). Controls can be added by the Customizer Manager and provide a robust set of UI options with minimal effort:

```

1 $wp_customize->add_control( 'setting_id', array(
2     'type' => 'date',
3     'priority' => 10, // Within the section.
4     'section' => 'colors', // Required, core or custom.
5     'label' => __( 'Date' ),
6     'description' => __( 'This is a date control with a red
7 border.' ),
8     'input_attrs' => array(
9         'class' => 'my-custom-class-for-js',
10        'style' => 'border: 1px solid #900',
11        'placeholder' => __( 'mm/dd/yyyy' ),
12    ),
13    'active_callback' => 'is_front_page',
) );

```

The most important parameter when adding a control is its type — this determines what type of UI the Customizer will display. Core provides several built-in control types:

- <input> elements with any allowed type (see below)
- checkbox
- textarea
- radio (pass a keyed array of values => labels to the `choices` argument)
- select (pass a keyed array of values => labels to the `choices` argument)

- `dropdown-pages` (use the `allow_addition` argument to allow users to add new pages from the control)

For any input type supported by the html `input` element, simply pass the type attribute value to the type parameter when adding the control. This allows support for control types such as `text`, `hidden`, `number`, `range`, `url`, `tel`, `email`, `search`, `time`, `date`, `datetime`, and `week`, pending browser support.

Controls must be added to a section before they will be displayed (and sections must contain controls to be displayed). This is done by specifying the section parameter when adding the control. Here is an example for adding a basic textarea control:

```

1 $wp_customize->add_control( 'custom_theme_css', array(
2   'label' => __( 'Custom Theme CSS' ),
3   'type' => 'textarea',
4   'section' => 'custom_css',
5 ) );

```

And here's an example of a basic range (slider) control. Note that most browsers will not display the numeric value of this control because the range input type is designed for settings where the exact value is unimportant. If the numeric value is important, consider using the number type. The `input_attrs` parameter will map a keyed array of attributes => values to attributes on the input element, and can be used for purposes ranging from placeholder text to `data-` JavaScript-referenced data in custom scripts.

For number and range controls, it allows us to set the minimum, maximum, and step values.

```

1 $wp_customize->add_control( 'setting_id', array(
2   'type' => 'range',
3   'section' => 'title_tagline',
4   'label' => __( 'Range' ),
5   'description' => __( 'This is the range control
6 description.' ),
7   'input_attrs' => array(
8     'min' => 0,
9     'max' => 10,
10    'step' => 2,
11  ),
12);

```

Core Custom Controls

If none of the basic control types suit your needs, you can easily create and add custom controls. Custom controls are explained more fully later in this post, but they are essentially subclasses of the base [WP_Customize_Control](#) object that allow any arbitrary html markup and functionality that you might need. Core features several built-in custom controls that allow developers to implement rich JavaScript-driven features with ease. A color picker control can be added as follows:

```
1 $wp_customize->add_control( new
2 WP_Customize_Color_Control( $wp_customize,
3 'color_control', array(
4   'label' => __( 'Accent Color', 'theme_textdomain' ),
5   'section' => 'media',
6 ) ) );
```

WordPress 4.1 and 4.2 also added support for any type of multimedia content, with the Media control. The media control implements the native WordPress media manager, allowing users to select files from their library or upload new ones. By specifying the `mime_type` parameter when adding the control, you can instruct the media library show to a specific type such as images or audio:

```
1 $wp_customize->add_control( new
2 WP_Customize_Media_Control( $wp_customize,
3 'image_control', array(
4   'label' => __( 'Featured Home Page Image',
5   'theme_textdomain' ),
6   'section' => 'media',
7   'mime_type' => 'image',
8 ) ) );
1 $wp_customize->add_control( new
2 WP_Customize_Media_Control( $wp_customize,
3 'audio_control', array(
4   'label' => __( 'Featured Home Page Recording',
5   'theme_textdomain' ),
6   'section' => 'media',
7   'mime_type' => 'audio',
8 ) ) );
```

Note that settings associated with `WP_Customize_Media_Control` save the associated attachment ID, while all other media-related controls (children of `WP_Customize_Upload_Control`) save the media file URL to the setting.

[More information](#) is available on Make WordPress Core.

Additionally, WordPress 4.3 introduced the

`WP_Customize_Cropped_Image_Control`, which provides an interface for cropping an image after selecting it. This is useful for instances where a particular aspect ratio is needed.

Sections

Sections are UI containers for Customizer controls. While you can add custom controls to the core sections, if you have more than a few options you may want to add one or more custom sections. Use the [`add_section`](#) method of the [`WP_Customize_Manager`](#) object to add a new section:

```
1 $wp_customize->add_section( 'custom_css', array(
2   'title' => __( 'Custom CSS' ),
3   'description' => __( 'Add custom CSS here' ),
4   'panel' => '', // Not typically needed.
5   'priority' => 160,
6   'capability' => 'edit_theme_options',
7   'theme_supports' => '', // Rarely needed.
8 ); );
```

You only need to include fields that you want to override the default values of. For example, the default priority (order of appearance) is typically acceptable, and most sections shouldn't require descriptive text if your options are self-explanatory. If you do want to change the location of your custom section, the priorities of the core sections are below:

Title	ID	Priority (Order)
Site Title & Tagline	title_tagline	20
Colors	colors	40

Header Image	header_image	60
Background Image	background_image	80
Menus (Panel)	nav_menus	100
Widgets (Panel)	widgets	110
Static Front Page	static_front_page	120
<i>default</i>		160
Additional CSS	custom_css	200

In most cases, sections can be added with only one or two parameters being specified. Here's an example for adding a section for options that pertain to a theme's footer:

```

1 // Add a footer/copyright information section.
2 $wp_customize->add_section( 'footer' , array(
3   'title' => __( 'Footer', 'themename' ),
4   'priority' => 105, // Before Widgets.
5 ) );

```

Panels

The Customizer Panels API was introduced in WordPress 4.0, and allows developers to create an additional layer of hierarchy beyond controls and sections. More than simply grouping sections of controls, panels are designed to provide distinct contexts for the Customizer, such as Customizing Widgets, Menus, or perhaps in the future, editing posts. There is an important technical distinction between the section and panel objects.

Themes should not register their own panels in most cases. Sections do *not* need to be nested under a panel, and each section should generally contain multiple controls. Controls should also be added to the Sections that core provides, such as adding color options to the colors Section. Also make sure that your options are as streamlined and efficient as possible; see the [WordPress philosophy](#). Panels are designed as contexts for entire features such as Widgets, Menus, or Posts, not as wrappers for generic sections. If

you absolutely must use Panels, you'll find that the API is nearly identical to that for Sections:

```
1 $wp_customize->add_panel( 'menus', array(
2   'title' => __( 'Menus' ),
3   'description' => $description, // Include html tags such
4   as <p>.
5   'priority' => 160, // Mixed with top-level-section
6   hierarchy.
7 );
8 $wp_customize->add_section( $section_id , array(
9   'title' => $menu->name,
10  'panel' => 'menus',
11 );
```

Panels must contain at least one Section, which must contain at least one Control, to be displayed. As you can see in the above example, Sections can be added to Panels similarly to how Controls are added to Sections. However, unlike with controls, if the Panel parameter is empty when registering a Section, it will be displayed on the main, top-level Customizer context, as most sections should not be contained with a panel.

Custom Controls, Sections, and Panels

Custom Controls, Sections, and Panels can be easily created by subclassing the PHP objects associated with each Customizer

object: [WP_Customize_Control](#), [WP_Customize_Section](#),

and [WP_Customize_Panel](#) (this can also be done for

[WP_Customize_Setting](#), but custom settings are typically better implemented using custom setting types, as outlined in the next section). Here's an example for a basic custom control:

```
1 class WP_New_Menu_Customize_Control extends
2 WP_Customize_Control {
3     public $type = 'new_menu';
4     /**
5      * Render the control's content.
6      */
7     public function render_content() {
8         ?>
9             <button class="button button-primary" id="create-new-
10 menu-submit" tabindex="0"><?php _e( 'Create Menu' ); ?></
11 button>
12         <?php
13     }
14 }
```

By subclassing the base control class, you can override any functionality with custom functionality or use the core functionality depending on your needs. The most common function to override is `render_content()` as it allows you to create custom UI from scratch with HTML. Custom Controls should be used with caution, however, as they may introduce UI that is inconsistent with the surrounding core UI and cause confusion for users. Custom Customizer objects can be added similarly to how default controls, sections, and panels are added:

```
1 $wp_customize->add_control(
2     new WP_Customize_Color_Control(
3         $wp_customize, // WP_Customize_Manager
4         'accent_color', // Setting id
5         array( // Args, including any custom ones.
6             'label' => __( 'Accent Color' ),
7             'section' => 'colors',
8         )
9     )
10 );
```

Parameters passed when adding controls are mapped to class variables within the control class, so you can add and use custom ones where certain parts of your custom object are different across different instances.

When creating custom Controls, Sections, or Panels, it is strongly recommended to reference the core code, in order to fully understand the available functionality that can be overridden. Core also includes examples of custom objects of each type. This can be found in [wp-includes/class-wp-customize-control.php](#), [wp-includes/class-wp-customize-](#)

[section.php](#), and [wp-includes/class-wp-customize-panel.php](#). There is also a JavaScript API for each Customizer object type, which can be extended with custom objects; see the Customizer JavaScript API section for more details.

Customizer UI Standards

Custom customizer controls, sections, and panels should match core UI practices wherever possible. This includes relying on standards from wp-admin, such as using the `.button` and `.button-primary` classes, for example. There are also a few standards specific to the customizer (as of WordPress 4.7):

- White background colors are used *only* to indicate navigation and actionable items (such as inputs)
- The general `#eee` background color provides visual contrast against the white elements
- `1px #ddd` borders separate navigational elements from background margins and from each other
- `15px` of spacing is provided between elements where visual separation is desired
- `4px` borders are used on one side of a navigation element to show hover or focus, with a color of `#0073aa`
- Customizer text uses `color: #555d66`, with `#0073aa` for hover and focus states on navigation elements

Custom Setting Types

By default, the Customizer supports saving settings as options or theme modifications. But this behavior can be easily overwritten to manually save and preview settings outside of the `wp_options` table of the WordPress database, or to apply other custom handling. To get started, specify a type other than `option` or `theme_mod` when adding your setting (you can use pretty much any string):

```
1 $wp_customize->add_setting( $nav_menu_setting_id, array(
2     'type' => 'nav_menu',
3     'default' => $item_ids,
4 ) );
```

The setting will no longer be saved or previewed when its value is changed in the associated control. Now, you can use the `CUSTOMIZE_UPDATE_SETTING->TYPE` and `CUSTOMIZE_PREVIEW_SETTING->TYPE` actions to implement custom saving and previewing functionality. Here is an example for saving a menu item's order property from the Menu Customizer project (the value of the setting is an ordered array of menu ids):

```
1 function menu_customizer_update_nav_menu( $value,
2   $setting ) {
3   $menu_id = str_replace( 'nav_menu_', '', $setting-
4   >id );
5   // ...
6   $i = 0;
7   foreach( $value as $item_id ) { // $value is ordered
8     menu_customizer_update_menu_item_order( $menu_id,
9     $item_id, $i );
10    $i++;
11  }
12  add_action( 'customize_update_nav_menu',
13   'menu_customizer_update_nav_menu', 10, 2 );
```

And here is how the same plugin implements previewing for nav menu items (note that this example requires PHP 5.3 or higher):

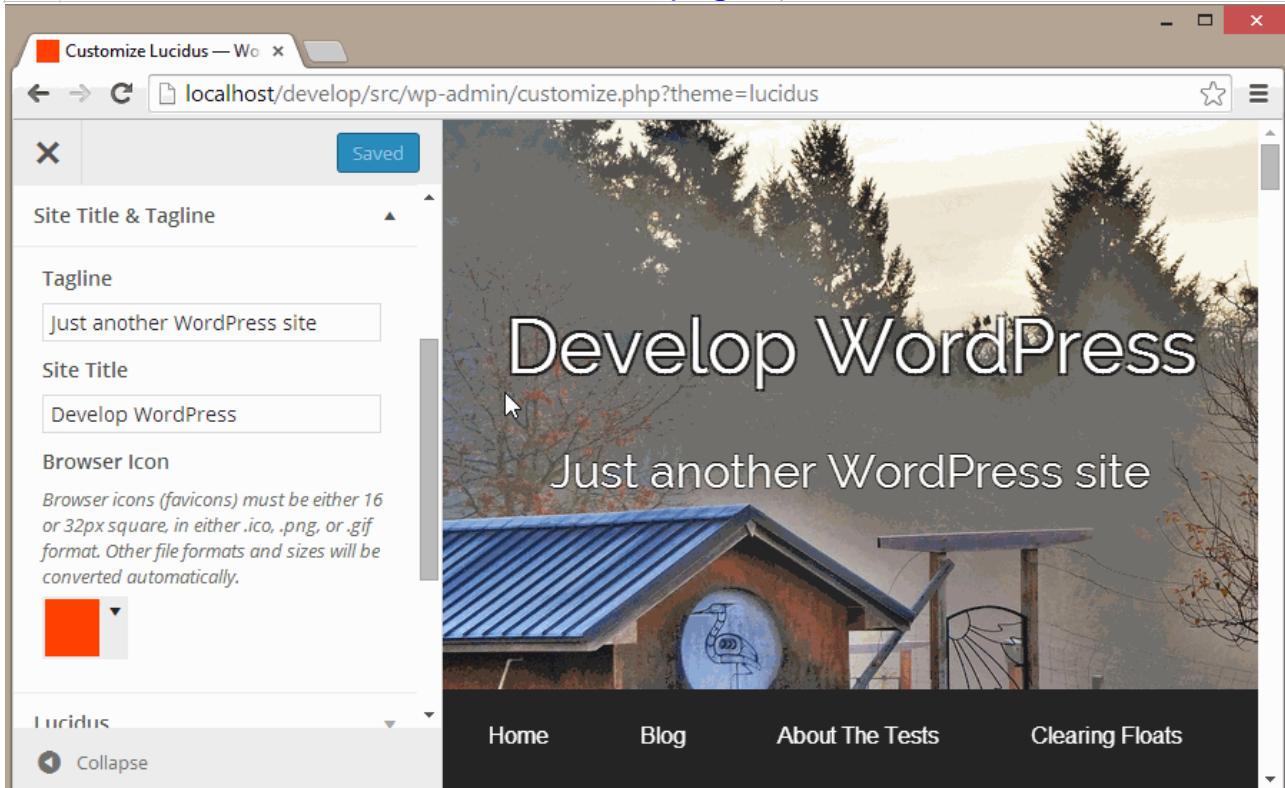
```
1 function menu_customizer_preview_nav_menu( $setting ) {
2     $menu_id = str_replace( 'nav_menu_', '', $setting->id );
3     add_filter( 'wp_get_nav_menu_items', function( $items,
4 $menu, $args ) use ( $menu_id, $setting ) {
5         $preview_menu_id = $menu->term_id;
6         if ( $menu_id == $preview_menu_id ) {
7             $new_ids = $setting->post_value();
8             foreach ( $new_ids as $item_id ) {
9                 $item = wp_setup_nav_menu_item( $item );
10                $item->menu_order = $i;
11                $new_items[] = $item;
12                $i++;
13            }
14            return $new_items;
15        } else {
16            return $items;
17        }
18    }, 10, 3 );
19 }
add_action( 'customize_preview_nav_menu',
'menu_customizer_preview_nav_menu', 10, 2 );
```

Tools for Improved User Experience

Contextual Controls, Sections, and Panels

WordPress 4.0 and 4.1 also added support for making parts of the Customizer UI be visible or hidden depending on the part of the site that the user was previewing within the Customizer preview window. A simple contextual control example would be that your theme only displays the header image and the site tagline on the front page. This is a perfect use case for the Customizer Manager's get_ methods, as we can modify the core controls for these settings directly to make them contextual to the front page:

```
1 // Hide core sections/controls when they aren't used on
2 // the current page.
3 $wp_customize->get_section( 'header_image' )-
>active_callback = 'is_front_page';
$wp_customize->get_control( 'blogdescription' )-
>active_callback = 'is_front_page';
```



In this contextual control example, the theme only displays the site tagline on the front page, so the corresponding field in the Customizer is hidden when the user navigates to a different page within the preview window.

The `active_callback` parameter for Panels, Sections, and Controls takes a callback function name, either core or custom. This parameter can also be set when registering the object for objects that you add. Here's an example from the Twenty Fourteen Theme:

```
1 $wp_customize->add_section( 'featured_content', array(
2     'title'      => __( 'Featured Content' ,
3     'twentyfourteen' ),
4     'description' => //...
5     'priority'    => 130,
6     'active_callback' => 'is_front_page',
) );
```

In the previous example, `is_front_page` is used directly. But for more complex logic, such as checking if the current view is a page (or even a specific page, by id), custom functions can be used (see [#30251](#) for details on why this is needed). If you don't need to support PHP 5.2, this can be done inline:

```
1 'active_callback' => function () { return is_page(); }
```

PHP 5.2 support is as simple as creating a named function and referencing it with the `active_callback` parameter:

```
1 //...
2 'active_callback' => 'prefix_return_is_page';
3 //...
4 function prefix_return_is_page() {
5     return is_page();
6 }
```

Within Custom Controls, Sections, and Panels, there is also an option to override the `active_callback` function directly within the custom Customizer object class:

```
1 class WP_Customize_Greeting_Control extends
2 WP_Customize_Control {
3     // ...
4     function active_callback() {
5         return is_front_page();
6     }
7 }
```

Finally, there is a filter that can be used to override all other `active_callback` behavior:

```
1 // Hide all controls without a description when previewing
2 single posts.
3 function title_tagline_control_filter( $active, $control )
4 {
5     if ( '' === $control->description ) {
6         $active = is_singular();
7     }
8     return $active;
}
add_filter( 'customize_control_active',
'title_tagline_control_filter', 10, 2 );
```

Note that the `active_callback` API works exactly the same for all of the

Customizer object types (Controls, [Sections](#), and [Panels](#)). As an added bonus, sections will automatically be hidden if all of the controls within them are contextually hidden, and the same works for panels.

Selective Refresh: Fast, Accurate Updates

Introduced in WordPress 4.5, Selective Refresh updates in the Customizer “preview” only refresh areas whose associate settings are changed. By only updating the elements that have changed, it’s much faster and less disruptive than a full-iframed refresh. Some other benefits, as noted in [Selective Refresh In The Customizer](#), are:

- Don’t Repeat Yourself (DRY) logic
- Accurate preview update
- Association between parts of the preview and associated settings and controls, along with visible edit shortcuts [as of WordPress 4.7](#)

The logic in pure-JavaScript `postMessage` updates is duplicated. The JavaScript in the Customizer must mirror the PHP that produces the markup, or take shortcuts to approximate it. But Selective Refresh is DRY as there’s no duplication of JavaScript and PHP. An Ajax request retrieves the new markup for the preview.

And because of this Ajax call, the refresh is **accurate**. It uses the filters that can alter the markup. It shows the same result that appears on the front end.

Additionally, selective refresh partials provide an association between areas of the preview and their corresponding settings. The customizer leverages this relationship to provide visible edit shortcuts that help users find controls associated with a particular part

of their site. In the future the partials API could expand to facilitate editing settings directly within the preview and to include a structured JS API for previewing settings with partials. For these reasons, all settings are strongly recommended to leverage selective refresh transport for improved user experience, with the option of providing additional JavaScript-based transport to further enhance setting previewing.

Registering Partial

Setting previews need to opt-in to use Selective Refresh by registering the necessary partials. In this example, largely taken from the them [Twenty Sixteen](#), Selective Refresh is added for the `blogdescription` setting by adding a partial with the same name.

```
1 function
2 foo_theme_customize_register( WP_Customize_Manager
3 $wp_customize ) {
4     $wp_customize->selective_refresh-
5     >add_partial( 'blogdescription', array(
6         'selector' => '.site-description',
7         'container_inclusive' => false,
8         'render_callback' => function() {
9             bloginfo( 'description' );
10        },
11    );
12 }
13 add_action( 'customize_register',
14 'foo_theme_customize_register' );
```

If the `settings` argument is not supplied, it defaults to be the same as the partial ID, in the same way as the settings for controls default to the control ID. Here are some of the key arguments for partials:

Variable	Type	Description
<code>settings</code>	array	Setting IDs associated with the partial.
<code>selector</code>	string	Targets the element(s) in the page markup to be refreshed.

container_inclusive	boolean	If true, a refresh replaces the entire container. Otherwise, it only replaces the container's children. Defaults to false.
render_callback	function	Produces the markup to be rendered on refresh.
fallback_refresh	bool	Whether or not a full page refresh should occur if the partial is not found in the document.

Selective Refresh JavaScript Events

These fire on wp.customize.selectiveRefresh:

- partial-content-rendered
- When the placement is rendered. As mentioned, JavaScript-driven widgets can rebuild on this event.
- render-partials-response
- When data is returned, after a request for partial rendering. The server filters this data with 'customize_render_partials_response'.
- partial-content-moved
- When a widget has moved in its sidebar. As shown above, JavaScript-driven widgets can refresh on this event.
- widget-updated
- When the **WidgetPartial** is refreshed with its `renderContent` method.
- sidebar-updated
- When a sidebar has a widget that's refreshed or updated. Or when a sidebar's widgets are sorted, using `reflowWidgets()`.

Widgets: Opting-In To Selective Refresh

Both themes and widgets **need to opt-in** to use Selective Refresh. All core widgets and themes have already enabled this.

Theme Support In Sidebars

In order to allow partial refreshes of widgets in a theme's sidebars:

```
1 add_theme_support( 'customize-selective-refresh-
  widgets' );
```

Important: Selective refresh for widgets requires that the theme include a `before_widget/after_widget` wrapper element around each widget that contains the widget's ID. Such wrappers are the default when you `register_sidebar()`. For example:

```
1 function example_widgets_init() {
2     register_sidebar(
3         array(
4             'name'          => esc_html__( 'Sidebar',
5 'example' ),
6             'id'            => 'sidebar-1',
7             'description'   => esc_html__( 'Add widgets
8 here.', 'example' ),
9             'before_widget' => '<section id="%1$s"
10 class="widget %2$s">', // <= Key for selective refresh.
11             'after_widget'  => '</section>',
12             'before_title'  => '<h2 class="widget-
13 title">',
14             'after_title'   => '</h2>',
15         )
16     );
17 }
18 add_action( 'widgets_init', 'example_widgets_init' );
```

Widget Support

Even if a theme supports Selective Refresh, widgets also have to opt-in. All core widgets have already enabled it. Here's an example widget adding support for Selective Refresh:

```

1 class Foo_Widget extends WP_Widget {
2
3     public function __construct() {
4         parent::__construct(
5             'foo',
6             __('Example', 'bar-plugin'),
7             array(
8                 'description' => __('An example widget'),
9                 'bar-plugin' ),
10                'customize_selective_refresh' => true,
11            )
12        );
13
14        if ( is_active_widget( false, false, $this->id_base ) || is_customize_preview() ) {
15            add_action( 'wp_enqueue_scripts', array( $this, 'enqueue_scripts' ) );
16        }
17    }
18
19 ...

```

Line 9 above enables Selective Refresh:

```
1 'customize_selective_refresh' => true,
```

Line 13 above ensures the widget's stylesheet always appears in Customizer sessions.

Adding the widget won't cause a full-page refresh to retrieve the styling:

```

1 if ( is_active_widget( false, false, $this->id_base ) || 
2 is_customize_preview() ) {
3     add_action( 'wp_enqueue_scripts', array( $this,
4     'enqueue_scripts' ) );
5 }

```

See [Implementing Selective Refresh Support for Widgets](#).

JavaScript-Driven Widget Support

Widgets that rely on JavaScript for their markup will need additional steps, as shown in [Implementing Selective Refresh Support for Widgets](#):

1. Enqueue any JavaScript files based on `is_customize_preview()`, as shown above for stylesheets.

2. Add a handler for the `partial-content-rendered` event, and refresh the widget as needed:

```
1 wp.customize.selectiveRefresh.bind( 'partial-content-
2 rendered', function( placement ) {
3     // logic to refresh
} );
```

3. If the widget includes an iframe, add a handler to refresh the partial:

```
1 wp.customize.selectiveRefresh.bind( 'partial-content-
2 moved', function( placement ) {
3     // logic to refresh, perhaps conditionally
} );
```

Using PostMessage For Improved Setting Previewing

The Customizer automatically handles previewing all settings out-of-the-box. This is done by silently reloading the entire preview window, with settings being filtered by PHP in that Ajax call. While this works just fine, it can be very slow since the entire front-end must be reloaded for every single setting change. Selective Refresh improves this experience by refreshing only the elements that have changed, but due to the Ajax call, there is still a delay in seeing the changes in the preview.

To further improve the user experience, the Customizer offers an API for managing setting changes directly in JavaScript, allowing for truly-live previewing. The below images show a comparison of a Custom CSS option that leverages this technology, called `postMessage`, versus the standard refresh option:

Manage Themes < Dev □ X

localhost/develop/src/wp-admin/customize.php?theme=twentytwelve

Save & Publish

Custom CSS

Custom Theme CSS

Theme-specific; safely sticks with each theme when switching themes.

```
.site-content article {  
    border: |  
}
```

Custom Plugin CSS

Theme-agnostic; persists across theme changes.

```
.hentry .mejs-controls .mejs-time-rail .mejs-time-current {  
    background: #6bf;  
}
```

Leave a reply

Embedded audio player:

00:00 00:00

Embedded playlist:

 "Etude #1"
Etudes for Cello Choir
NICK HALSEY
00:24 00:48

1. "Etude #1" — NICK HALSEY	0:48
2. "Etude #2" — NICK HALSEY	0:35
3. "Etude #3" — NICK HALSEY	0:53

This entry was posted in [Uncategorized](#) on [July 28, 2014](#) by [Nick Halsey](#). [Edit](#)

Collapse

Custom CSS setting in the Customizer with the postMessage setting transport.

The screenshot shows the WordPress Customizer interface. On the left, the 'Custom CSS' panel is open, displaying two sections: 'Custom Theme CSS' and 'Custom Plugin CSS'. 'Custom Theme CSS' contains the rule `.site-content article { border: 1px solid #ccc; }`. 'Custom Plugin CSS' contains the rule `.hentry .mejs-controls .mejs-time-rail .mejs-time-current { background: #6bf; }`. On the right, the 'Audio' panel is active, showing an 'Embedded audio player' with a progress bar and volume control, and an 'Embedded playlist' for 'Etude #1' by Nick Halsey, featuring three tracks: 'Etude #1' (0:48), 'Etude #2' (0:35), and 'Etude #3' (0:53). Below the preview area, a note states: 'This entry was posted in Uncategorized on July 28, 2014 by Nick Halsey. Edit'

Custom CSS setting in the Customizer with the default refresh setting transport.

To use `postMessage`, first set the transport parameter to `postMessage` when adding your setting. Many themes also modify core settings such as the title and tagline to leverage `postMessage` by modifying the `transport` property of those settings:

```
1 $wp_customize->get_setting( 'blogname' )->transport  
2 = 'postMessage';  
$wp_customize->get_setting( 'blogdescription' )->transport  
= 'postMessage';
```

Once the setting's transport is set to `postMessage`, the setting will no longer trigger a refresh of the preview when its value changes. To implement the JavaScript to update the setting within the preview of the front-end, first create and enqueue a JavaScript file:

```
1 function my_preview_js() {
2     wp_enqueue_script( 'custom_css_preview', 'path/to/
3 file.js', array( 'customize-preview', 'jquery' ) );
4 }
5 add_action( 'customize_preview_init', 'my_preview_js' );
```

Your JavaScript file should look something like this:

```
1 ( function( $ ) {
2     wp.customize( 'setting_id', function( value ) {
3         value.bind( function( to ) {
4             $( '#custom-theme-css' ).html( to );
5         } );
6     } );
7     wp.customize( 'custom_plugin_css', function( value ) {
8         value.bind( function( to ) {
9             $( '#custom-plugin-css' ).html( to );
10        } );
11    } );
12 } )( jQuery );
```

Note that you don't necessarily need to be great with JavaScript to use postMessage – most of the code is boilerplate. The types of settings that benefit most from postMessage transport require simple JS changes such as using jQuery's .html() or .text() methods, or swapping out a class on the <body> or another element to trigger a different set of CSS rules. Doing this, or simplifying the instant preview logic with fully-accurate changes updating with selective refresh, the user experience can be fast without duplicating all of the PHP logic in JS.

Setting Validation

WordPress 4.6 includes new APIs related to validation of Customizer setting values. The Customizer has had sanitization of setting values since it was introduced. Sanitization involves coercing a value into something safe to persist to the database: common examples are converting a value into an integer or stripping tags from some text input. As such, sanitization is a *lossy* operation. With the addition of setting validation:

1. All modified settings are validated up-front before any of them are saved.
2. If any setting is invalid, the Customizer save request is rejected: a save thus becomes *transactional* with all the settings left dirty to try saving again. (The Customizer [transactions proposal](#) is closely related to setting validation here.)

3. Validation error messages are displayed to the user, prompting them to fix their mistake and try again.

Sanitization and validation are also both part of the REST API infrastructure via

`WP_REST_Request::sanitize_params()` and

`WP_REST_Request::validate_params()`, respectively. A setting's value goes through validation before it goes through sanitization.

For more information on the validation behavior, and additional code examples, see [the feature announcement post](#).

Validating Settings in PHP

Just as you can supply a `sanitize_callback` when registering a setting, you can also supply a `validate_callback` arg:

```
$wp_customize->add_setting( 'established_year', array(
    'sanitize_callback' => 'absint',
    'validate_callback' => 'validate_established_year'
) );
function validate_established_year( $validity, $value ) {
    $value = intval( $value );
    if ( empty( $value ) || ! is_numeric( $value ) ) {
        $validity->add( 'required', __( 'You must supply a
valid year.' ) );
    } elseif ( $value < 1900 ) {
        $validity->add( 'year_too_small', __( 'Year is too
old.' ) );
    } elseif ( $value > gdate( 'Y' ) ) {
        $validity->add( 'year_too_big', __( 'Year is too
new.' ) );
    }
    return $validity;
}
```

Just as supplying a `sanitize_callback` arg adds a filter for `customize_sanitize_{$setting_id}`, so too supplying a `validate_callback` arg will add a filter for `customize_validate_{$setting_id}`. Assuming that the

`WP_Customize_Setting` instances apply filters on these in their `validate` methods, you can add this filter if you need to add validation for settings that have been previously added.

The `validate_callback` and any

`customize_validate_{$setting_id}` filter callbacks take a `WP_Error` instance as its first argument (which initially is empty of any errors added), followed by the `$value` being sanitized, and lastly the `WP_Customize_Setting` instance that is being validated.

Custom setting classes can also override the `validate` method of the setting class directly.

Client-side Validation

If you have a setting that is previewed purely via JavaScript (and the `postMessage` transport *without* selective refresh), you should also add client-side validation. Otherwise, any validation errors will persist until a full refresh happens or a save is attempted. Client-side validation must not take the place of server-side validation, since malicious users could bypass the client-side validation to save an invalid value if corresponding server-side validation is not in place.

There is a `validate` method available on the `wp.customize.Settings` class (actually, the `wp.customize.Value` base class). Its name is a bit misleading, as it actually behaves very similarly to the `WP_Customize_Setting::sanitize()` PHP method, but it can be used to both sanitize and validate a value in JS. Note that this JS runs in the context of the Customizer *pane* not the preview, so any such JS should have `CUSTOMIZE-CONTROLS` as a dependency (not `customize-preview`) and enqueued during the `customize_controls_enqueue_scripts` action. Some example JS validation:

```
wp.customize( 'established_year', function ( setting ) {
    setting.validate = function ( value ) {
        var code, notification;
        var year = parseInt( value, 10 );

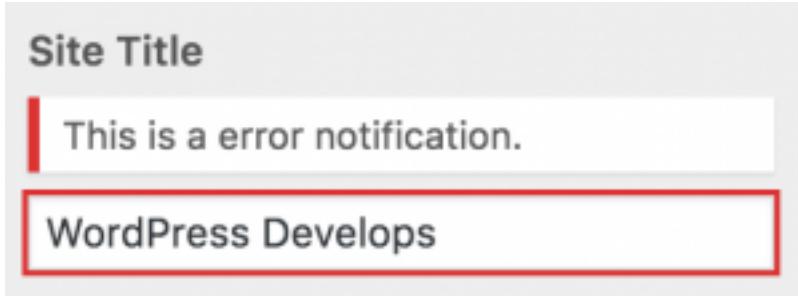
        code = 'required';
        if ( isNaN( year ) ) {
            notification = new
wp.customize.Notification( code, {message:
myPlugin.l10n.yearRequired} );
            setting.notifications.add( code, notification );
        } else {
            setting.notifications.remove( code );
        }

        code = 'year_too_small';
        if ( year < 1900 ) {
            notification = new
wp.customize.Notification( code, {message:
myPlugin.l10n.yearTooSmall} );
            setting.notifications.add( code, notification );
        } else {
            setting.notifications.remove( code );
        }

        code = 'year_too_big';
        if ( year > new Date().getFullYear() ) {
            notification = new
wp.customize.Notification( code, {message:
myPlugin.l10n.yearTooBig} );
            setting.notifications.add( code, notification );
        } else {
            setting.notifications.remove( code );
        }

        return value;
    };
} );
```

Notifications



Notifications provide user feedback, typically based on the value of a control's setting. An error notification is added to a setting's `notifications` collection when a setting's validation routine returns a `WP_Error` instance. Each error added to a PHP `WP_Error` instance is represented as a `wp.customize.Notification` in JavaScript:

- A `WP_Error`'s `code` is available as `notification.code` in JS.
- A `WP_Error`'s `message` is available as `notification.message` in JS. Note that if there are multiple messages added to a given error code in PHP they will be concatenated into a single message in JS.
- A `WP_Error`'s `data` is available as `notification.data` in JS. This is useful to pass additional error context from the server to the client.

Any time that a `WP_Error` is returned from a validation routine on the server it will result in a `wp.customize.Notification` being created with a `type` property of "error".

While setting non-error notifications from PHP is not currently supported (see #37281), you can also add non-error notifications with JS as follows:

```
wp.customize( 'blogname', function( setting ) {
    setting.bind( function( value ) {
        var code = 'long_title';
        if ( value.length > 20 ) {
            setting.notifications.add( code, new
wp.customize.Notification(
            code,
            {
                type: 'warning',
                message: 'This theme prefers title with
max 20 chars.'
            }
        );
    } else {
        setting.notifications.remove( code );
    }
} );
} );
```

You can also supply “info” as a notification’s `type`. The default `type` is “error”. Custom types may also be supplied, and the notifications can be styled with CSS selector matching `notice.notice-foo` where “foo” is the type supplied. A control may also override the default behavior for how notifications are rendered by overriding the `wp.customize.Control.renderNotifications` method.

The Customizer JavaScript API

In WordPress 4.1, newly-expanded JavaScript APIs were introduced for all customizer objects. The entire JavaScript API is currently located in a single file, [wp-admin/js/customize-controls.js](#), which contains models for all objects, core custom controls, and more.

Preview JS and Controls JS

The customizer app is currently split into two distinct areas: the customizer controls “pane” and the customize preview. The preview is currently in an iframe, meaning that all

JS runs either in the controls pane or in the preview. The postMessage API is used to communicate between the preview and the controls.

Most themes only implement JavaScript in the customize preview, and use it to implement instant previewing of settings via postMessage. However, JS on the controls side can be used for many things, such as dynamically showing and hiding controls based on the values of other settings, changing the previewed URL, focusing parts of the preview, and more. Here's an example from core of controls-side JS that interacts with the preview, in this case changing the previewed URL when the page for posts changes:

```
1 // Change the previewed URL to the selected page when
2 changing the page_for_posts.
3
4 wp.customize( 'page_for_posts', function( setting ) {
5     setting.bind( function( pageId ) {
6         pageId = parseInt( pageId, 10 );
7         if ( pageId > 0 ) {
8             api.previewUrl.set( api.settings.url.home + '?'
9             page_id=' + pageId );
10            }
11        });
12    });
13});
```

Similar logic can be used to **activate** UI objects based on the value of a setting. The Twenty Seventeen theme includes some useful examples for leveraging the customize JS API for improved user experience. Note that there is one JS file for the controls pane, named `CUSTOMIZE-CONTROLS.js` and one file for the customize preview, named `customize-preview.js`. For clarity, all themes and plugins are recommended to follow this naming convention, even if customize JS is only provided in the controls or preview but not both.

The rest of this page is dedicated primarily to the controls-side JS API that was built-out in WordPress 4.1.

Models for Controls, Sections, and Panels

As in PHP, each Customizer object type has a corresponding object in JavaScript. There are `wp.customize.Control`, `wp.customize.Panel`, and `wp.customize.Section` models, as well

as `wp.customize.panel`, `wp.customize.section`, and `wp.customize.control` collections (yes, they are singular) that store all control instances. You can iterate over panels, sections, and controls via:

```
1 wp.customize.panel.each( function ( panel ) { /* ...  
2 */ } );  
3 wp.customize.section.each( function ( section ) { /* ...  
4 */ } );  
 wp.customize.control.each( function ( control ) { /* ...  
 */ } );
```

Relating Controls, Sections, and Panels together

When registering a new control in PHP, you pass in the parent section ID:

```
1 $wp_customize->add_control( 'blogname', array(  
2   'label' => __( 'Site Title' ),  
3   'section' => 'title_tagline',)  
4 );
```

In the JavaScript API, a control's section can be obtained predictably:

```
1 id = wp.customize.control( 'blogname' ).section(); //  
 returns title_tagline by default
```

To get the section object from the ID, look up the section by the ID as normal:

`wp.customize.section(id)`.

You can move a control to another section using this `section` state as well, here moving it to the Navigation section:

```
1 wp.customize.control( 'blogname' ).section( 'nav' );
```

Likewise, you can get a section's panel ID in the same way:

```
1 id = wp.customize.section( 'sidebar-widgets-  
 sidebar-1' ).panel(); // returns widgets by default
```

You can go the other way as well, to get the children of panels and sections:

```
1 sections =  
 wp.customize.panel( 'widgets' ).sections(); controls =  
 wp.customize.section( 'title_tagline' ).controls();
```

You can use these to move all controls from one section to another:

```
1  _.each( wp.customize.section( 'title_tagline' ).controls()
2  , function ( control ) {
3      control.section( 'nav' )
4      ;
5  );
```

Contextual Panels, Sections, and Controls

Control, Panel, and Section instances have an `active` state (a `wp.customize.Value` instance). When the `active` state changes, the panel, section, and control instances invoke their respective `onChangeActive` method, which by default slides the container element up and down, if `false` and `true` respectively. There are also `activate()` and `deactivate()` methods now for manipulating this `active` state, for panels, sections, and controls. The primary purpose of these states is to show or hide the object without removing it entirely from the Customizer.

```
1  wp.customize.section( 'nav' ).deactivate(); // slide up
2  wp.customize.section( 'nav' ).activate({ duration:
3      1000 }); // slide down slowly
4  wp.customize.section( 'colors' ).deactivate({ duration:
5      0 }); // hide immediately
6  wp.customize.section( 'nav' ).deactivate({ completeCallback:
7      function () {
8          wp.customize.section( 'colors' ).activate(); // show
9              after nav hides completely
10         }
11     });
12 );
```

Note that manually changing the `active` state would only stick until the preview refreshes or loads another URL. The `activate()/deactivate()` methods are designed to follow the pattern of the new `expanded` state.

Focusing UI Objects

Building upon the `expand()`/`collapse()` methods for panels, sections, and controls, these models also support a `focus()` method which not only expands all of the necessary elements, but also scrolls the target container into view and puts the browser focus on the first focusable element in the container. For instance, to expand the “Static Front Page” section and focus on select dropdown for the “Front page”:

```
1 wp.customize.control( 'page_on_front' ).focus()
```

The focus functionality is used to implement [autofocus](#): deep-linking to panels, sections, and controls inside of the customizer. Consider these URLs:

- [.../wp-admin/customize.php?autofocus\[panel\]=widgets](#)
- [.../wp-admin/customize.php?autofocus\[section\]=colors](#)
- [.../wp-admin/customize.php?autofocus\[control\]=blogname](#)

This is used in WordPress core to [add a link](#) on the widgets admin page to link directly to the widgets panel within the customizer, as well as to connect visible edit shortcuts in the customize preview with the associated controls in the customize pane.

Priorities

When registering a panel, section, or control in PHP, you can supply a `priority` parameter. This value is stored in a `WP_Customize_Value` instance for each respective `Panel`, `Section`, and `Control` instance. For example, you can obtain the priority for the widgets panel via:

```
1 priority = wp.customize.panel( 'widgets' ).priority(); //  
    returns 110 by default
```

You can then dynamically change the priority and the Customizer UI will automatically re-arrange to reflect the new priorities:

```
1 wp.customize.panel( 'widgets' ).priority( 1 ); // move  
    Widgets to the top
```

Custom Controls, Panels, and Sections

When working with custom Customizer objects in JS, it is usually easiest to examine the custom objects in WordPress core to understand the code structure. See [wp-admin/js/customize-controls.js](#), particularly the wp.customize.Panel|Section|Control models. Note several examples in the core code, particularly in the media controls, which build on each others' functionality through object hierarchy.

JavaScript/Underscore.js-Rendered Custom Controls

WordPress 4.1 also added support for rendering JavaScript-heavy and/or high-quantity controls entirely with JavaScript. This allows for more dynamic behavior, especially related to dynamically-added controls. The core Color and Media controls currently leverage this API, and all core Controls will eventually use it in the future. The PHP-based control API is not going away, but in the future most controls will likely use the new API since it provides a faster experience for users and developers. Similar APIs for JS-templated Sections and Panels were introduced WordPress 4.3; however, there remain some gaps in the ease of dynamically-creating objects in JS as of WordPress 4.7, see [#30741](#).

Registered Control Types

In order to introduce a concept of having one template for multiple Customizer controls of the same type, we needed to introduce a way to register a type of control with the Customize Manager. Previously, custom control objects were only encountered when custom controls were added using

[WP_Customize_Manager::add_control\(\)](#). But detecting added control types to render one template per type wouldn't allow new controls to be created dynamically if no other instances of that type were

loaded. [WP_Customize_Manager::register_control_type\(\)](#) solves this:

```
1 add_action( 'customize_register',
2   'prefix_customize_register' );
3 function prefix_customize_register( $wp_customize ) {
4   // Define a custom control class,
5   WP_Customize_Custom_Control.
6   // Register the class so that its JS template is
7   // available in the Customizer.
8   $wp_customize-
9   >register_control_type( 'WP_Customize_Custom_Control' );
}
```

All registered control types have their templates printed to the customizer by [WP_Customize_Manager::print_control_templates\(\)](#).

Sending PHP Control Data to JavaScript

While Customizer control data has always been passed to the control JS models, and this has always been able to be extended, you're much more likely to need to send data down when working with JS templates. Anything that you would want access to in [render_content\(\)](#) in PHP will need to be exported to JavaScript to be accessible in your control template. [WP_Customize_Control](#) exports the following control class variables by default:

- type
- label
- description
- active (boolean state)

You can add additional parameters specific to your custom control by overriding [WP_Customize_Control::to_json\(\)](#) in your custom control subclass. In most cases, you'll want to call the parent class' [to_json\(\)](#) method also, to ensure that all core variables are exported as well. Here's an example from the core color control:

```
1 public function to_json() {
2     parent::to_json();
3     $this->json['statuses'] = $this->statuses;
4     $this->json['defaultValue'] = $this->setting->default;
5 }
```

JS/Underscore Templating

Once you've registered your custom control class as a control type and exported any custom class variables, you can create the template that will render the control UI. You'll override [WP_Customize_Control::content_template\(\)](#) (empty by default) as a replacement for

[WP_Customize_Control::render_content\(\)](#). Render content is still

called, so be sure to override it with an empty function in your subclass as well.

Underscore-style custom control templates are very similar to PHP. As more and more of WordPress core becomes JavaScript-driven, these templates are becoming increasingly more common. Some sample template code in core can be found in [media](#), [revisions](#), the [theme browser](#), and even [in the Twenty Fifteen theme](#), where a JS template is used to both save the color scheme data and instantly preview color scheme changes in the Customizer. The best way to learn how these templates work is to study similar code in core and, accordingly, here is a brief example:

```
1 class WP_Customize_Color_Control extends
2 WP_Customize_Control {
3     public $type = 'color';
4 // ...
5 /**
6     * Render a JS template for the content of the color
7     picker control.
8 */
9 public function content_template() {
10     ?>
11     <# var defaultValue = '';
12     if ( data.defaultValue ) {
13         if ( '#' !== data.defaultValue.substring( 0, 1 ) ) {
14             defaultValue = '#' + data.defaultValue;
15         } else {
16             defaultValue = data.defaultValue;
17         }
18         defaultValue = ' data-default-color=' + defaultValue;
19 // Quotes added automatically.
20     } #>
21     <label>
22         <# if ( data.label ) { #>
23             <span class="customize-control-
24 title">{{{ data.label }}}</span>
25             <# } #>
26             <# if ( data.description ) { #>
27                 <span class="description customize-control-
28 description">{{{ data.description }}}</span>
29                 <# } #>
30                 <div class="customize-control-content">
31                     <input class="color-picker-hex" type="text"
32 maxlength="7" placeholder="<?php esc_attr_e( 'Hex Value' );
33 ?>" {{ defaultValue }} />
34                 </div>
35             </label>
36             <?php
37         }
38     }
39
40
41
42
43
44
45
46
```

In the above template for the core custom color control, you can see that after the closing PHP tag, we have a JS template. notation is used around statements to be evaluated – in most cases, this is used for conditionals. All of the control instance data that we exported to JS is stored in the `data` object, and we can print a variable using double (escaped) or triple (unescape) brace notation `{}{ }{}`. As I said before, the best way to get the hang of writing controls like this is to read through existing examples.

[WP_Customize_Upload_Control](#) was recently [updated to leverage this API](#)

as well, integrating nicely with the way the media manager is implemented, and squeezing a ton of functionality out of a minimal amount of code. If you want some really good practice, try converting some of the other core controls to use this API – and submit patches to core too, of course!

Putting the pieces together

Here's a summary of what's needed to leverage the new API in a custom customizer control subclass:

1. Make your `render_content()` function empty (but it does need to exist to override the default one).
2. Create a new function, `content_template()`, and put the old contents of `render_content()` there.
3. Add any custom class variables that are needed for the control to be exported to the JavaScript in the browser (the JSON data) by modifying the `to_json()` function (see [WP_Customize_Color_Control](#) for an example).
4. Convert the PHP from `render_content()` into a JS template, using around JS statements to evaluate and `{}{ }{}` around variables to print. PHP class variables are available in the data object; for example, the label can be printed with `{}{ data.label }{}`.
5. **Register the custom control class/type.** This critical step tells the Customizer to print the template for this control. This is distinct from just printing templates for all controls that were added because the ideas are that many instances of this control

type could be rendered from one template, and that any registered control types would be available for dynamic control-creation in the future. Just do something like \$wp_customize-

```
>register_control_type( 'WP\_Customize\_Color\_Control' );.
```

The PHP-only parts of the Customize API are still fully supported and perfectly fine to use. But given long term goals for making the customizer more flexible for doing things like switching themes in the customizer without a pageload, it is strongly encouraged to use JS/Underscore templates for all custom customizer objects where feasible.

Advanced Usage

The customize API is actively developed; this page contains additional more advanced topics. Additional discussion of advanced topics can be found by searching the archives for the [#core-customize](#) channel in [Slack](#).

Allow Non-administrators to Access the Customizer

Customizer access is controlled by the customize meta capability (mapped to edit_theme_options by default), which is assigned only to administrators by default. This allows for wider use of the Customizer's extensive capability-access options, which are built into panels, sections, and settings. Additionally, this makes it possible to allow non-administrators to use the customizer for, for example, customizing posts. This change is an important step toward expanding the scope of the Customizer beyond themes.

```
1 function
2 allow_users_who_can_edit_posts_to_customize( $caps, $cap,
3 $user_id ) {
4     $required_cap = 'edit_posts';
5     if ( 'customize' === $cap && user_can( $user_id,
6 $required_cap ) ) {
7         $caps = array( $required_cap );
8     }
9     return $caps;
10 }
11 add_filter( 'map_meta_cap',
12 'allow_users_who_can_edit_posts_to_customize', 10, 3 );
```

Note that it is currently necessary to manually add links to the Customizer in the admin menu, admin bar, or elsewhere if you are granting the customize meta capability to non-administrator users.

Theme Security

Introduction

The WordPress development team takes security seriously. With so much of the web relying on the integrity of the platform, security is a necessity. While the core developers have a dedicated team focused on securing the core platform, as a theme developer you are quite aware that there is potentially much that is outside the core which can be vulnerable. Because WordPress provides so much power and flexibility, plugins and themes are key points of weakness.

Developing a Security Mindset

When developing your themes, it is important to consider security as you add functionality. Use the following principles as you begin your theme development efforts:

- **Don't trust any data.** Don't trust user input, third-party APIs, or data in your database without verification. Protection of your WordPress themes begins with ensuring the data entering and leaving your theme is as intended. Always make sure to validate on input and sanitize (escape) data before use or on output.

- **Rely on the WordPress API.** Many core WordPress functions provide the build in functionality of validating and sanitizing data. Rely on the WordPress provided functions when possible.
- **Keep your themes up to date.** As technology evolves, so does the potential for new security holes in your themes. Stay vigilant maintaining your code and updating your themes as required.

This chapter provides some background on writing secure themes with [data validation](#) and [sanitization/escaping](#) techniques, [using nonces](#) to generate secure tokens, and a review of the [common security attacks](#), and how you can harden your theme against them.

Additional Resources

- A Guide to Writing WordPress Themes – WordPress.org Blog Series
 - [A Guide to Writing Secure Themes – Part 1: Introduction](#)
 - [A Guide to Writing Secure Themes – Part 2: Validation](#)
 - [A Guide to Writing Secure Themes – Part 3: Sanitization](#)
 - [A Guide to Writing Secure Themes – Part 4: Securing Post Meta](#)
- [Sucuri Security Blog](#)

Data Sanitization/Escaping

Sanitization: Securing Input

Sanitization is the process of cleaning or filtering your input data. Whether the data is from a user or an API or web service, you use sanitizing when you don't know what to expect or you don't want to be strict with [data validation](#).

The easiest way to sanitize data is with built-in WordPress functions.

The `sanitize_*` series of helper functions provide an effective way to ensure you're ending up with safe data, and they require minimal effort on your part:

- [sanitize_email\(\)](#)
- [sanitize_file_name\(\)](#)
- [sanitize_html_class\(\)](#)
- [sanitize_key\(\)](#)
- [sanitize_meta\(\)](#)
- [sanitize_mime_type\(\)](#)

- [sanitize_option\(\)](#)
- [sanitize_sql_orderby\(\)](#)
- [sanitize_text_field\(\)](#)
- [sanitize_title\(\)](#)
- [sanitize_title_for_query\(\)](#)
- [sanitize_title_with_dashes\(\)](#)
- [sanitize_user\(\)](#)
- [esc_url_raw\(\)](#)
- [wp_filter_post_kses\(\)](#)
- [wp_filter_nohtml_kses\(\)](#)

Tip:

Any time you're accepting potentially unsafe data, it is important to validate or sanitize it.

Example -Simple Input Field

Let's say we have an input field named title.

```
1 <input id="title" type="text" name="title">
```

You can sanitize the input data with the [sanitize_text_field\(\)](#) function:

```
1 $title = sanitize_text_field( $_POST['title'] );
2 update_post_meta( $post->ID, 'title', $title );
```

Behind the scenes, [sanitize_text_field\(\)](#) does the following:

- Checks for invalid UTF-8
- Converts single less-than characters (<) to entity
- Strips all tags
- Removes line breaks, tabs and extra white space
- Strips octets

Tip:

Remember, rely on the WordPress API and its help functions to assist with securing your themes.

Escaping: Securing Output

Whenever you're outputting data make sure to properly **escape it**.

Escaping is the process of securing output by stripping out unwanted data, like malformed HTML or script tags, preventing this data from being seen as code.

Escaping helps secure your data prior to rendering it for the end user and prevents XSS (Cross-site scripting) attacks.

Note:

[Cross-site scripting \(XSS\)](#) is a type of computer security vulnerability typically found in web applications. XSS enables attackers to inject client-side scripts into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy.

WordPress has a few helper functions you can use for most common scenarios.

- [`esc_html\(\)`](#) – Use this function anytime an HTML element encloses a section of data being displayed.

●	●	<?php echo esc_html(\$title); ?>	.	
1				

- [`esc_url\(\)`](#) – Use this function on all URLs, including those in the `SRC` and `href` attributes of an HTML element.

●	●		.	
1				

- [`esc_js\(\)`](#) – Use this function for inline Javascript.

●	●	<a href="#" onclick="<?php echo esc_js(\$custom_js); ?>">Click me	.	
1				

- [`esc_attr\(\)`](#) – Use this function on everything else that's printed into an HTML element's attribute.

●	●	<ul class="<?php echo esc_attr(\$stored_class); ?>"> 	.	
1				

- [`esc_textarea\(\)`](#) – encodes text for use inside a textarea element.

●	●	<textarea><?php echo esc_textarea(\$text); ?></textarea>	.	
1				

Tip:

Output escaping should occur as late as possible.

Escaping with Localization

Rather than using `echo` to output data, it's common to use the WordPress localization functions, such as [`_e\(\)`](#) or [`_\(\)`](#).

These functions simply wrap a localization function inside an escaping function:

```
1 esc_html_e( 'Hello World', 'text_domain' );
2 // same as
3 echo esc_html( __( 'Hello World', 'text_domain' ) );
```

These helper functions combine localization and escaping:

- [esc_html\(\)](#)
- [esc_html_e\(\)](#)
- [esc_html_x\(\)](#)
- [esc_attr\(\)](#)
- [esc_attr_e\(\)](#)
- [esc_attr_x\(\)](#)

Custom Escaping

In the case that you need to escape your output in a specific way, the function [wp_kses\(\)](#) (pronounced “kisses”) will come in handy. For example, there are instances when you want HTML elements or attributes to display in your output.

This function makes sure that only the specified HTML elements, attributes, and attribute values will occur in your output, and normalizes HTML entities.

```
1 $allowed_html = [
2     'a'      => [
3         'href'  => ___,
4         'title' => ___,
5     ],
6     'br'      => ___,
7     'em'      => ___,
8     'strong' => ___,
9 ];
10 echo wp_kses( $custom_content, $allowed_html );
```

[wp_kses_post\(\)](#) is a wrapper function for wp_kses where `$allowed_html` is a set of rules used by post content.

```
1 echo wp_kses_post( $post_content );
```

Database Escaping

All data in SQL queries must be SQL-escaped before the SQL query is executed to prevent against [SQL injection attacks](#). WordPress provides helper classes to assist with escaping SQL queries `$wpdb`.

Selecting Data

The escaped SQL query (`$sql` in this example) can then be used with one of the methods:

- `\$wpdb->get_row\(\$sql\)`
- `\$wpdb->get_var\(\$sql\)`
- `\$wpdb->get_results\(\$sql\)`
- `\$wpdb->get_col\(\$sql\)`
- `\$wpdb->query\(\$sql\)`

Inserting and Updating Data

- `\$wpdb->update\(\)`
- `\$wpdb->insert\(\)`

Like Statements

- `\$wpdb->prepare`

Data Validation

Data validation is the process of analyzing the data against a predefined pattern (or patterns) with a definitive result: valid or invalid.

Usually this applies to data coming from external sources such as user input and calls to web services via API.

Simple examples of data validation:

- Check that required fields have not been left blank
- Check that an entered phone number only contains numbers and punctuation
- Check that an entered postal code is a valid postal code
- Check that a quantity field is greater than 0

Data validation should be performed as early as possible. That means validating the data before performing any actions.

Note:

Validation can be performed by using JavaScript on the front end and by using PHP on the back end.

Validating the Data

There are at least three ways: built-in PHP functions, core WordPress functions, and custom functions you write.

Built-in PHP functions

Basic validation is doable using many built-in PHP functions, including these:

- [`isset\(\)`](#) and [`empty\(\)`](#) for checking whether a variable exists and isn't blank
- [`mb_strlen\(\)`](#) or [`strlen\(\)`](#) for checking that a string has the expected number of characters
- [`preg_match\(\)`](#), [`strpos\(\)`](#) for checking for occurrences of certain strings in other strings
- [`count\(\)`](#) for checking how many items are in an array
- [`in_array\(\)`](#) for checking whether something exists in an array

Core WordPress functions

WordPress provides many useful functions that help validate different kinds of data. Here are several examples:

- [`is_email\(\)`](#) will validate whether an email address is valid.
- [`term_exists\(\)`](#) checks whether a tag, category, or other taxonomy term exists.
- [`username_exists\(\)`](#) checks if username exists.
- [`validate_file\(\)`](#) will validate that an entered file path is a real path (but not whether the file exists).

Check the the list of [conditional tags](#) for more functions like these.

Search for functions with names like these: `*_exists()`, `*_validate()`, and `is_*``()`. Not all of these are validation functions, but many are helpful.

Custom PHP and JavaScript functions

You can write your own PHP and JavaScript functions and include them in your plugin. When writing a validation function, you'll want to name it like a question (examples: `is_phone`, `is_available`, `is_us_zipcode`).

The function should return a boolean, either true or false, depending on whether the data is valid or not. This will allow using the function as a condition.

Example 1

Let's say you have an U.S. zip code input field that a user submits.

```
1 <input id="wporg_zip_code" type="text" maxlength="10"  
name="wporg_zip_code">
```

The text field allows up to 10 characters of input with no limitations on the types of characters that can be used. Users could enter something valid like `1234567890` or something invalid (and evil) like `eval()`.

The `maxlength` attribute on our `input` field is only enforced by the browser, so you still need to validate the length of the input on the server. If you don't, an attacker could alter the `maxlength` value.

By using validation we can ensure we're accepting only valid zip codes.

First you need to write a function to validate a U.S. zip codes:

```
1 <?php function is_us_zip_code($zip_code) { // scenario 1:  
2 empty if (empty($zip_code)) { return false; } // scenario  
3 2: more than 10 characters if (strlen(trim($zip_code)) >  
4 10) {  
5     return false;  
6 }  
7  
8     // scenario 3: incorrect format  
9     if (!preg_match('/^\d{5}(\-\d{4})?$/ ', $zip_code)) {  
10         return false;  
11     }  
12  
13     // passed successfully  
14     return true;  
15 }
```

When processing the form, your code should check the `wporg_zip_code` field and perform the action based on the result:

```
1 if ( isset( $_POST['wporg_zip_code'] ) &&  
2 is_us_zip_code( $_POST['wporg_zip_code'] ) ) {  
3     // your action  
4 }
```

Example 2

Say you're going to query the database for some posts, and you want to give the user the ability to sort the query results.

This example code checks an incoming sort key (stored in the “orderby” input parameter) for validity by comparing it against an array of allowed sort keys using the built-in PHP function [in_array](#). This prevents the user from passing in malicious data and potentially compromising the website.

Before checking the incoming sort key against the array, the key is passed into the built-in WordPress function [sanitize_key](#). This function ensures, among other things, that the key is in lowercase ([in_array](#) performs a *case-sensitive* search).

Passing “true” into the third parameter of [in_array](#) enables strict type checking, which tells the function to not only compare values but value [types](#) as well. This allows the code to be certain that the incoming sort key is a string and not some other data type.

```
1 <?php
2 $allowed_keys = ['author', 'post_author', 'date',
3 'post_date'];
4
5 $orderby = sanitize_key( $_POST['orderby'] );
6
7 if ( in_array( $orderby, $allowed_keys, true ) ) {
8     // modify the query to sort by the orderby key
9 }
```

Using Nonces

WordPress nonces are one-time use security tokens generated by WordPress to help protect URLs and forms from misuse.

If your theme allows users to submit data; be it in the Admin or the front-end; nonces can be used to **verify a user intends to perform an action**, and is instrumental in protecting against [Cross-Site Request Forgery\(CSRF\)](#).

An example is a WordPress site in which authorized users are allowed to upload videos. As an authorized user uploading videos is an intentional action and permitted. However, in a CSRF, a hacker can hijack (forge) the use of an authorized user and perform a fraudulent submission.

The one-time use hash generated by a nonce, prevents this type of forged attacks from being successful by validating the upload request is done by the current logged in user. Nonces are unique only to the current user’s session, so if an attempt is made to log in or out any nonces on the page become invalid.

Creating a Nonce

- [wp_nonce_url\(\)](#) – Adding a nonce to an URL.
- [wp_nonce_field\(\)](#) – Adding a nonce to a form.
- [wp_create_nonce\(\)](#) – Using a nonce in a custom way; useful for processing AJAX requests.

Verifying a Nonce

- [`check_admin_referer\(\)`](#) – To verify a nonce that was passed in a URL or a form in an admin screen.
- [`check_ajax_referer\(\)`](#) – Checks the nonce (but not the referrer), and if the check fails then by default it terminates script execution.
- [`wp_verify_nonce\(\)`](#) – To verify a nonce passed in some other context.

Example

In this example, we have a basic submission form.

Create theNonce

To secure your form with a nonce, create a hidden nonce field using [`wp_nonce_field\(\)`](#) function:

```
1 <form method="post">
2     <!-- some inputs here ... -->
3     <?php wp_nonce_field( 'name_of_my_action',
4     'name_of_nonce_field' ); ?>
5 </form>
```

Verify theNonce

In our example we first check if the nonce field is set since we do not want to run anything if the form has not been submitted. If the form has been submitted we use the nonce field value function. If nonce verification is successful the form will process.

```
1 if (
2     ! isset( $_POST['name_of_nonce_field'] )
3     || ! wp_verify_nonce( $_POST['name_of_nonce_field'],
4     'name_of_my_action' )
5 ) {
6
7     print 'Sorry, your nonce did not verify.';
8     exit;
9
10 } else {
11
12     // process form data
13 }
```

In this example the basic nonce process:

1. Generates a nonce with the [`wp_nonce_field\(\)`](#) function.

2. The nonce is submitted with the form submission.
3. The nonce is verified for validity using the [wp_verify_nonce\(\)](#) function. If not verified the request exits with an error message.

Common Vulnerabilities

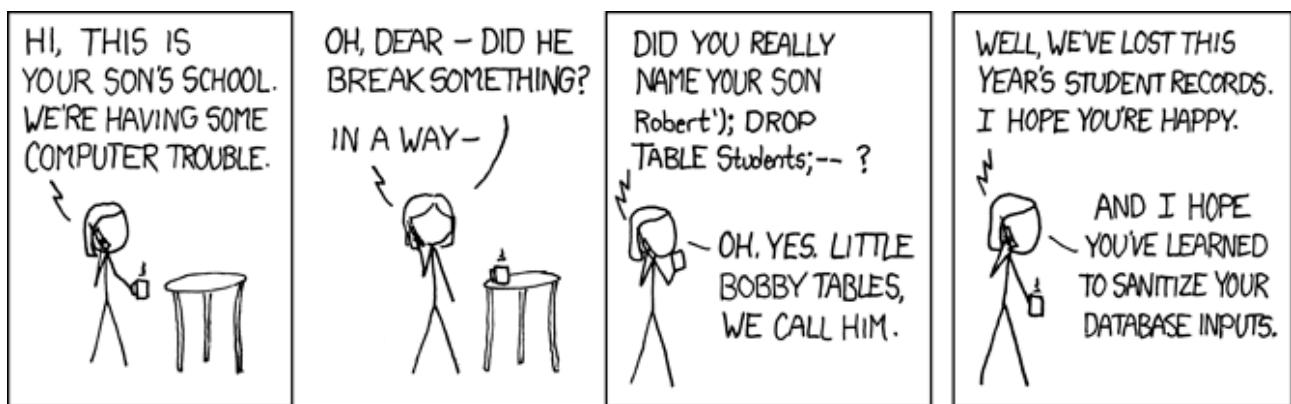
Security is an ever-changing landscape, and vulnerabilities evolve over time. The following is a discussion of common vulnerabilities you should protect against, and the techniques for protecting your theme from exploitation.

Types of Vulnerabilities

SQL Injection

What it is:

SQL injection happens when values being inputted are not properly sanitized allowing for any SQL commands in the inputted data to potentially be executed. To prevent this, the WordPress API is extensive, offering functions like `add_post_meta()`; instead of you needing to adding the post meta manually via SQL (`INSERT INTO wp_postmeta...`).



xkcd [Exploits of a Mom](#)

The first rule for hardening your theme against SQL injection is: When there's a WordPress function, use it.

But sometimes you need to do complex queries, that have not been accounted for in the API. If this is the case, always use the [\\$wpdb functions](#). These were built specifically to protect your database.

All data in SQL queries must be SQL-escaped before the SQL query is executed to prevent against SQL injection attacks. The best function to use for SQL-escaping is `$wpdb->prepare()` which supports both a [sprintf\(\)](#)-like and [vsprintf\(\)](#)-like syntax.

```
1 $wpdb->get_var( $wpdb->prepare(
2     "SELECT something FROM table WHERE foo = %s and status =
3     %d",
4     $name, // an unescaped string (function will do the
5     sanitization for you)
    $status // an untrusted integer (function will do the
    sanitization for you)
) );
```

Cross Site Scripting (XSS)

Cross Site Scripting (XSS) happens when a nefarious party injects JavaScript into a web page.

Avoid XSS vulnerabilities by escaping output, stripping out unwanted data. As a theme's primary responsibility is outputting content, a theme should [escape dynamic content](#) with the proper function depending on the type of the content.

An example of one of the escaping functions is escaping URL from a user profile.

```
1 
```

Content that has HTML entities within can be sanitized to allow only specified HTML elements.

```
1 $allowed_html = array(
2     'a' => array(
3         'href' => array(),
4         'title' => array()
5     ),
6     'br' => array(),
7     'em' => array(),
8     'strong' => array(),
9 );
10
11 echo wp_kses( $custom_content, $allowed_html );
```

Cross-site Request Forgery (CSRF)

Cross-site request forgery or CSRF (pronounced sea-surf) is when a nefarious party tricks a user into performing an unwanted action within a web application they are authenticated in. For example, a phishing email might contain a link to a page that would delete a user's account in the WordPress admin.

If your theme includes any HTML or HTTP-based form submissions, use a [nonce](#) to guarantee a user intends to perform an action.

```
1 <form method="post">
2     <!-- some inputs here ... -->
3     <?php wp_nonce_field( 'name_of_my_action',
4     'name_of_nonce_field' ); ?>
5 </form>
```

Staying Current

It is important to stay current on potential security holes. The following resources provide a good starting point:

- [WordPress Security Whitepaper](#)
- [WordPress Security Release](#)
- [Open Web Application Security Project \(OWASP\) Top 10](#)

Theme Privacy

In WordPress 4.9.6, several tools were introduced to help sites meet the requirements of the new European Union's new GDPR (General Data Protection Regulation) laws. This article details what theme authors need to know about compatibility with the features. Theme authors should test their themes to confirm that there are no design conflicts between the features and their themes detailed below.

Privacy Policy Pages

WordPress 4.9.6 introduced the ability to select a page as a privacy policy for a site in the Settings > Privacy section of the admin area. For new sites, a privacy policy template page will automatically be created in draft status.

To link to the selected page in plugins and themes, three template tags have been added:

- `get_privacy_policy_url()` – Retrieves the URL to the privacy policy page.
- `the_privacy_policy_link()` – Displays the privacy policy link with formatting, when applicable.
- `get_the_privacy_policy_link()` – Returns the privacy policy link with formatting, when applicable.

Example

The following example will display the privacy policy link surrounded by a

```
1 if ( function_exists( 'the_privacy_policy_link' ) ) {  
2     the_privacy_policy_link( '<div>', '</div>' );  
3 }
```

Commenter Cookie Opt-Ins

When a logged out user comments on a post, they are asked for their name, email, and website. This information is stored locally in the commenter's browser for two purposes:

1. When they leave another comment on the site, their name, email, and website will be pre-populated into the respective fields.
2. If their comment is held for moderation, they can return to that post and remove the comment before it is approved.

The information stored in this cookie is for convenience and is not essential. Therefore, the user needs to be given the choice to opt in or opt out of the storage of this data. For this reason, a checkbox has been added to the comment form that allows commenters to opt-in to storing this data in the cookie. This checkbox will be unchecked by default, as opt-in is an action the user must explicitly approve. The new checkbox field is automatically added to comment forms displayed using the [comment_form\(\)](#) function inside a p.comment-form-cookies-consent element. While most themes will not require any action, it is recommended that you double check that the input and label does not require CSS adjustments in custom themes.

Bundled Themes

All 8 currently supported bundled themes (Twenty Ten-Twenty Seventeen) have been updated to support these changes. Site footers will display a link to the site's privacy policy when one has been selected, and the commenter cookie opt-in field has been styled.

Child themes built on top of bundled themes should be checked to see if any adjustments are necessary for the privacy policy link in the footer.

Advanced Theme Topics

In this section, we'll look at some advanced theme topics.

We'll start with the use of [Child Themes](#) for modifying aspects of the theme.

We'll also look at [User Interface \(UI\) Best Practices](#) and [JavaScript Best Practices](#). This is followed by performing checks of the theme using [Theme Unit Tests](#) and [validating the theme's markup](#).

Finally, we'll look at ensuring your theme works well with [Plugin API Hooks](#).

Child Themes

A child theme allows you to change small aspects of your site's appearance yet still preserve your theme's look and functionality. To understand how child themes work it is first important to understand the relationship between parent and child themes.

What is a Parent Theme?

A parent theme is a complete theme which includes all of the [required WordPress template files](#) and assets for the theme to work. All themes – excluding child themes – are considered parent themes.

What is a Child Theme?

As indicated in the overview, a child theme inherits the look and feel of the parent theme and all of its functions, but can be used to make modifications to any part of the theme. In this way, customizations are kept separate from the parent theme's files. Using a child theme lets you upgrade the parent theme without affecting the customizations you've made to your site.

Child themes:

- make your modifications portable and replicable;
- keep customization separate from parent theme functions;
- allow parent themes to be updated without destroying your modifications;
- allow you to take advantage of the effort and testing put into parent theme;
- save on development time since you are not recreating the wheel; and
- are a *great way* to start learning about theme development.

Note:

If you are making extensive customizations – beyond styles and a few theme files – creating a parent theme might be a better option than a child theme. Creating a parent theme allows you to avoid issues with deprecated code in the future. This needs to be decided on a case-by-case basis.

How to Create a Child Theme

1. Create a child theme folder

First, create a new folder in your themes directory, located at `wp-content/themes`.

The directory needs a name. It's best practice to give a child theme the same name as the parent, but with `-child` appended to the end. For example, if you were making a child theme of `twentyfifteen`, then the directory would be named `twentyfifteen-child`.

2. Create a stylesheet: style.css

Next, you'll need to create a stylesheet file named `Style.css`, which will contain all of the CSS rules and declarations that control the look of your theme. Your stylesheet must contain the below required header comment at the very top of the file. This tells WordPress basic info about the theme, including the fact that it is a child theme with a particular parent.

```
1  /*
2   Theme Name:      Twenty Fifteen Child
3   Theme URI:     http://example.com/twenty-fifteen-
4   child/
5   Description:    Twenty Fifteen Child Theme
6   Author:          John Doe
7   Author URI:     http://example.com
8   Template:        twentyfifteen
9   Version:         1.0.0
10  License:         GNU General Public License v2 or
11  later
12  License URI:    http://www.gnu.org/licenses/
13  gpl-2.0.html
14  Tags:            light, dark, two-columns, right-
15  sidebar, responsive-layout, accessibility-ready
16  Text Domain:    twentyfifteenchild
17 */
```

The following information is required:

- **Theme Name** – needs to be unique to your theme
- **Template** – the name of the parent theme directory. The parent theme in our example is the Twenty Fifteen theme, so the Template will be `twentyfifteen`.
You may be working with a different theme, so adjust accordingly.

Add remaining information as applicable. The only required child theme file is `style.css`, but `functions.php` is necessary to enqueue styles correctly (below).

3. Enqueue stylesheet

The final step is to enqueue the parent and child theme stylesheets.

Note:

In the past, the common method was to import the parent theme stylesheet using `@import`; this is no longer the recommended practice, as it increases the amount of time it takes style sheets to load. Plus it is possible for the parent stylesheet to get included twice.

The recommended way of enqueueing the parent theme stylesheet currently is to add a `wp_enqueue_scripts` action and use [`wp_enqueue_style\(\)`](#) in your child theme's `functions.php`.

You will therefore need to create a `functions.php` in your child theme directory.

The first line of your child theme's `functions.php` will be an opening PHP tag (`<?php`), after which you can enqueue your parent and child theme stylesheets. The following example function will only work if your Parent Theme uses only one main `style.css` to hold all of the css. If your child theme has more than one `.css` file (eg. `ie.css`, `style.css`, `main.css`) then you will have to make sure to maintain all of the Parent Theme dependencies.

```
1 <?php
2 add_action( 'wp_enqueue_scripts',
3   'my_theme_enqueue_styles' );
4 function my_theme_enqueue_styles() {
5   wp_enqueue_style( 'parent-style',
6     get_template_directory_uri() . '/style.css' );
7 }
```

If your child theme `style.css` contains actual CSS code (as it normally does), you will need to enqueue it as well. Setting 'parent-style' as a dependency will ensure that the child theme stylesheet loads after it. Including the child theme version number ensures that you can bust cache also for the child theme. The complete (recommended) example becomes:

```
1 <?php
2 add_action( 'wp_enqueue_scripts',
3 'my_theme_enqueue_styles' );
4 function my_theme_enqueue_styles() {
5
6     $parent_style = 'parent-style'; // This is
7 'twentyfifteen-style' for the Twenty Fifteen theme.
8
9     wp_enqueue_style( $parent_style,
10 get_template_directory_uri() . '/style.css' );
11     wp_enqueue_style( 'child-style',
12         get_stylesheet_directory_uri() . '/style.css',
13         array( $parent_style ),
14         wp_get_theme()->get('Version')
15     );
16 }
```

where parent-style is the same `$handle` used in the parent theme when it registers its stylesheet.

Activate child theme

Your child theme is now ready for activation. Log in to your site's Administration Screen, and go to **Administration Screen > Appearance > Themes**. You should see your child theme listed and ready for activation. (If your WordPress installation is multi-site enabled, then you may need to switch to your network Administration Screen to enable the theme (within the Network Admin Themes Screen tab). You can then switch back to your site-specific WordPress Administration Screen to activate your child theme.)

Note:

You may need to re-save your menu from **Appearance > Menus** and theme options (including background and header images) after activating the child theme.

Adding Template Files

Other than the `functions.php` file (as noted above), any file you add to your child theme will overwrite the same file in the parent theme.

In most cases, it's best to create a copy of the template files you want to change from the parent theme, then make your modifications to the copied files, leaving the parent files

unchanged. For example, if you wanted to change the code of the parent theme's `header.php` file, you would copy the file to your child theme folder and customize it there.

Tip:

There are several plugins which allow you to detect what specific template is being used on the page at which you are looking.

- [What The File](#)
- [What Template File Am I Viewing?](#)
- [Debug Bar](#)

You can also include files in the child theme that are not included in the parent theme. For instance, you might want to create a more specific template than is found in your parent theme, such as a template for a specific page or category archive (e.g. `page-3.php` would load for a Page with the ID of 3).

See the [Template Hierarchy](#) page for more information about how WordPress determines which template to use.

Using functions.php

Unlike `style.css`, the `functions.php` of a child theme does not override its counterpart from the parent. Instead, it is **loaded in addition to the parent's `functions.php`**. (Specifically, it is loaded right before the parent's file.)

In that way, the `functions.php` of a child theme provides a smart, trouble-free method of modifying the functionality of a parent theme. Say that you want to add a PHP function to your theme. The fastest way would be to open its `functions.php` file and put the function there. But that's not smart: The next time your theme is updated, your function will disappear. But there is an alternative way which is the smart way: you can create a child theme, add a `functions.php` file in it, and add your function to that file. The function will do the exact same job from there too, with the advantage that it will not be affected by future updates of the parent theme. Do not copy the full content of `functions.php` of the parent theme into `functions.php` in the child theme.

The structure of `functions.php` is simple: An opening PHP tag at the top, and below it, your bits of PHP. In it you can put as many or as few functions as you wish. The

example below shows an elementary `functions.php` file that does one simple thing: Adds a favicon link to the head element of HTML pages.

```
1 <?php // Opening PHP tag - nothing should be before this,  
2 not even whitespace  
3  
4 // Custom Function to Include  
5 function my_favicon_link() {  
6     echo <link rel="shortcut icon" type="image/x-icon"  
7 href="/favicon.ico" />' . "\n";  
}  
add_action( 'wp_head', 'my_favicon_link' );
```

Tip:

The fact that a child theme's `functions.php` is loaded first means that you can make the user functions of your theme pluggable —that is, replaceable by a child theme— by declaring them conditionally.

```
if ( ! function_exists( 'theme_special_nav' ) ) {  
    function theme_special_nav() {  
        // Do something.  
    }  
}
```

In that way, a child theme can replace a PHP function of the parent by simply declaring it beforehand.

For more information about what to include in your child theme's `functions.php` file, read through the [Theme Functions](#) page.

Referencing or Including Other Files

When you need to include files that reside within your child theme's directory structure, you will need to use [get_stylesheet_directory\(\)](#). Since the `style.css` is in the root of your child theme's subdirectory, [get_stylesheet_directory\(\)](#) points to your child theme's directory (not the parent theme's directory). To reference the parent theme directory, you would use [get_template_directory\(\)](#) instead.

Below is an example illustrating how to use [get_stylesheet_directory\(\)](#) when referencing a file stored within the child theme directory:

```
1 <?php require_once( get_stylesheet_directory() . '/  
my_included_file.php' ); ?>
```

Meanwhile, this example uses [get_stylesheet_directory_uri\(\)](#) to display an image that is stored within the `/images` folder in the child theme directory.

```
1 
```

Unlike `get_stylesheet_directory()`, which returns a file path, `get_stylesheet_directory_uri()` returns a URL, which is useful for front-end assets.

Enqueueing Styles and Scripts

Scripts and styles should each be enqueued with their own function, and then those should be wrapped in an action. For more information, read the page on [Including CSS and JavaScript](#).

WordPress won't automatically load the stylesheet for your child theme on the front-end.

Below is an example of using the [wp_enqueue_scripts\(\)](#) action hook to call a function that enqueues the child theme's stylesheet.

```
1 <?php  
2 add_action( 'wp_enqueue_scripts',  
3   'my_plugin_add_stylesheet' );  
4 function my_plugin_add_stylesheet() {  
5   wp_enqueue_style( 'my-style',  
6     get_stylesheet_directory_uri() . '/style.css', false,  
7     '1.0', 'all' );  
8 }
```

Special Considerations

Post Formats

A child theme inherits [post formats](#) as defined by the parent theme. But when creating child themes, be aware that using [add_theme_support\('post-formats'\)](#) will **override** the formats as defined by the parent theme, not add to it.

RTL Support

To support RTL languages, add a `rtl.css` file to your child theme, containing:

```
1  /*
2  Theme Name: Twenty Fifteen Child
3  Template: twentyfifteen
4  */
```

Even if the parent theme does not have an `rtl.css` file, it's recommended to add the `rtl.css` file to your child theme. WordPress will auto-load the `rtl.css` file only if `is_rtl()` is true.

Internationalization

Child themes can be prepared for translation into other languages by using the WordPress [Internationalization API](#). There are special considerations regarding internationalization of child themes.

To internationalize a child theme follow these steps:

1. Add a languages directory.

- For example: twentyfifteen-child/languages/

2. Add language files.

- Your filenames have to be `he_IL.po` & `he_IL.mo` (depending on your language), unlike plugin files which are `domain-he_IL.xx`.

3. Load a textdomain

- Use [`load_child_theme_textdomain\(\)`](#) in `functions.php` during the `after_setup_theme` action.
- The text domain defined in [`load_child_theme_textdomain\(\)`](#) should be used to translate all strings in the child theme.

4. Use GetText functions to add i18n support for your strings.

Example: textdomain

```
1 <?php
2 /**
3  * Set up My Child Theme's textdomain.
4  *
5  * Declare textdomain for this child theme.
6  * Translations can be added to the /languages/
7  directory.
8 */
9 function twentyfifteenchild_theme_setup() {
10     load_child_theme_textdomain( 'twentyfifteenchild',
11 get_stylesheet_directory() . '/languages' );
12 }
13 add_action( 'after_setup_theme',
14 'twentyfifteenchild_theme_setup' );
```

At this point, strings in the child theme are ready for translation. To ensure they are properly internationalized for translation, each string needs to have the `twentyfifteenchild` textdomain.

Example: gettext functions

Here is an example of echoing the phrase “Code is Poetry”:

```
1 <?php
2 esc_html_e( 'Code is Poetry', 'twentyfifteenchild' );
3 ?>
```

The text domain defined in [load_child_theme_textdomain\(\)](#) should be used to translate all strings in the child theme. In the event that a template file from the parent theme has been included, the textdomain should be changed from the one defined in the parent theme to the one defined by the child theme.

UI Best Practices

Logo Homepage Link

The logo at the top each page should send the user to the homepage of your site. Assuming your logo is in your theme directory, this is how to display it in the `header.php` template file.

```
1 <a href="php echo home_url(); ?&gt;"&gt;&lt;img src="<?php echo<br/get_stylesheet_directory_uri(); ?>/logo.png" alt="Home  
Page" /></a>
```

Descriptive Anchor Text

The anchor text is the visible text for a hyperlink. Good link text should give the reader an idea of the action that will take place when clicking it.

A bad example:

The best way to learn WordPress is to start using it.
To Download WordPress, [click here](#).

A better example:

[Download WordPress](#) and start using it. That's the best way to learn.

Style Links with Underlines

By default, browsers underline links to let the user know what is clickable. Some designers use CSS to turn off underlines for hyperlinks. This causes usability and accessibility problems, as it makes it more difficult to identify hyperlinks from the surrounding text.

Different Link Colors

Color is another visual cue that text is clickable. Styling hyperlinks with a different color than the surrounding text makes them easier to distinguish.

Hyperlinks are one of the few HTML features that have state. The two most important states are *visited* and *unvisited*.

Having different colors for these two states helps users identify the pages they've visited before. A good trick for taking the guess work out of visited links is to color them 10%-20% darker than the unvisited links.

There are 3 other states that links can have:

- hover, when a mouse is over an element
- focus, similar to hover but for keyboard users
- active, when a user is clicking on a link

Since hover and focus have similar meanings, it is useful to give them the same styles.

Though hover and focus have similar meanings, they have different interaction patterns. If you choose a subtle hover state, you should have a more easily identifiable focus state.

Hovering over a link is a directed activity, where the user knows where they are in the page and only needs to identify whether that spot is linked. Focus is an undirected activity, where the user needs to discover where their focus has moved to after shifting focus from the previous location.

Color Contrast

Color contrast refers to the **difference between two colors**. Contrast is low between navy blue and black. Contrast is high between white and black. WebAIM, a non-profit web accessibility organization, provides a [color contrast calculator](#) to help you determine the contrast in your website design. The WCAG 2.0 requires a ratio of 4.5:1 on normal text to be [AA compliant](#).

Sufficient Font Size

Make your text easy to read. By making your text large enough, you increase the usability of your site and make the content easier to understand. 14px is the smallest text should be.

Associate Labels with Inputs

Labels inform the user what an input field is for. You can connect the label to the input by using the **for** attribute in the label. This will allow the user to click the label and focus on the input field.

```
1 <label for="username">Username</label>
2 <input type="text" id="username" name="login" />
```

Labels work for radio buttons as well. Since it works using the **id** field *and not the name*, each input for the group gets its own label.

```
1 <input type="radio" id="user_group_blogger"
2 name="user_group" value="blogger" />
3 <label for="user_group_blogger">Blogger</label>
4
5 <input type="radio" id="user_group_designer"
6 name="user_group" value="designer" />
7 <label for="user_group_designer">Designer</label>
8
<input type="radio" id="user_group_developer"
name="user_group" value="developer" />
<label for="user_group_developer">Developer</label>
```

Placeholder Text in Forms

Placeholder text shows the user an example of what to type. When a user puts their cursor in the field, the placeholder text will disappear, while the label remains.

```
1 <label for="name">Name</label>
2 <input type="text" id="name" name="name" placeholder="John
Smith" />
```

Use placeholders to suggest the type of data a field requires, and not as a substitute for the field label.

Descriptive Buttons

The web is filled with buttons that have unclear meanings. Remember the last time you used ‘OK’ or ‘submit’ on your login form? Choosing better words to display on your buttons can make your website easier to use. Try the pattern *[verb] [noun]* — Create user, Delete File, Update Password, Send Message. Each describes what will happen when the user clicks the button.

JavaScript Best Practices

Many themes use JavaScript to provide interactivity, animation or other enhancements. These best practices will help ensure your code works efficiently and does not cause conflicts with your content or plugins.

Use Included Libraries

There are many useful JavaScript libraries you may want to include when building your theme. Did you know that WordPress comes bundled with dozens of popular libraries? Check out this [list of default scripts included with WordPress](#).

A common mistake made by beginning theme and plugin developers is to bundle their theme or plugin with their own version of the library. This may conflict with other plugins that enqueue the version bundled with WordPress. As a best practice, make your theme compatible with the version of your favorite library included with WordPress.

Warning:

Do not try to use your own version of a JavaScript library that is already [bundled](#) with WordPress. Doing so may break core functionality and conflict with plugins.

If you feel you MUST replace the WordPress version with one of your own... well... the answer is still: don't do it. The versions of JavaScript libraries used by WordPress may include custom tweaks that WordPress needs to operate. Any time you override these libraries, you risk breaking your WordPress instance. Moreover, plugins created by other authors should be written to be compatible with WordPress's version of these libraries as well. Adding your own version may (and often does!) conflict with plugins.

Standard JavaScript

Javascript is growing in popularity for web developers, and it's being used to accomplish for a variety of tasks. Here are some best practices to use when writing your JavaScript

- Avoid Global Variables
- Keep your JavaScript unobtrusive
- Use closures and the [module pattern](#)
- Stick to a coding style. Use The [WordPress Javascript Coding Standard](#).
- Validate and Lint Your Code – [JSLint.com](#)
- Ensure your site still works without JavaScript first – then add JavaScript to provide additional capabilities. This is a form of [Progressive Enhancement](#).
- Lazy load assets that aren't immediately required.

- Don't use jQuery *if you don't need to* — There's a great site that shows you how to do some common tasks with plain old JavaScript — [you might not need jQuery](#)

jQuery

Including jQuery in your theme

[jQuery](#) is a popular JavaScript library to make working with JavaScript easier and more reliable across browsers. If you use jQuery, be sure to [follow the handbook guidelines on including JavaScript](#). Giving your theme's enqueued .js files a dependency array of `array('jquery')` ensures that the jQuery script has been downloaded and loaded before your theme's code.

Using `$ #`

Using `$`

Because the copy of jQuery included in WordPress loads in (link needed)

`noConflict()` mode, use this wrapper code in your theme's .js files to map “\$” to “jQuery”:

```
1 ( function( $ ) {
2     // Your code goes here
3 }( jQuery );
```

This wrapper (called an Immediately Invoked Function Expression, or IIFE) lets you pass in a variable—jQuery—on the bottom line, and give it the name “\$” internally. Within this wrapper you may use `$` to select elements as normal.

Selectors

Every time you select an element with jQuery, it performs a query through your document's markup. These queries are very fast, but they do take time—you can make your site respond faster by re-using your jQuery objects instead of using a new query. So instead of repeating selectors:

```
1 // Anti-pattern
2 $('.post img').addClass('theme-image');
3 $('.post img').on('click', function() { /* ... */ });
```

it is recommended to **cache your selectors** so you can re-use the returned element without having to repeat the lookup process:

```
1 var $postImage = $('.post img');
2 // All image tags within posts can now be accessed through
3 $postImage
4 $postImage.addClass('theme-image');
$postImage.on('click', function() { /* ... */ });
```

Event Handling

When you use jQuery methods like `.bind` or `.click`, jQuery creates a new browser event object to handle processing the requested event. Each new event created takes a small amount of memory, but the amount of memory required goes up the more events you bind. If you have a page with a hundred anchor tags and wanted to fire a `logClick` event handler whenever a user clicked a link, it is very easy to write code like this:

```
1 // Anti-pattern
2 $('a').click( logClick );
```

This works, but under the hood you have asked jQuery to create a new event handler for every link on your page.

Event delegation is a way to accomplish the same effect with less overhead. Because events in jQuery “bubble”—that is, a click event will fire on a link, then on the link’s surrounding `<p>` tag, then on the `div` container, and so on up to the `document` object itself—we can put a single event handler higher up in the page structure, and still catch the click events for all of those links:

```
1 // Bind one handler at the document level, which is
2 triggered
3 // whenever there is a "click" event originating from an
// "a" tag
$(document).on('click', 'a', logClick);
```

Theme Testing

The [Theme Unit Test data](#) is a WordPress import file will fill a WordPress site with enough stub data (posts, media, users) to test a theme.

The [Theme Unit Tests](#) are manual tests to walk through to test theme functionality and how the theme responds to edge-cases of content and settings.

Validating Your Theme

Validating the markup of your theme is the process of ensuring that the web pages conform to web standards defined by various organizations. These standards ensure web pages are interpreted in the same way by different browsers, search engines, and other web clients.

Conforming to standards and regulations is one of many ways you can make your theme universally understood. Make sure your code and styles validate across the board. That means they have to meet the standards set by the [W3C Organization](#) and pass a variety of validations for [CSS](#) and [XHTML](#).

Not all validators check for the same things. Some only check CSS, others XHTML, and others for accessibility. If you are sincere in presenting standardized pages to the public, test them with several validators. The World Wide Web Consortium sets the standards and also hosts a variety of [web page validators](#).

Validation Techniques

Validating your WordPress site means more than just checking the front page for errors. A theme's template files are loaded in a modular fashion. While you may fix all the errors associated with the `index.php` and `sidebar.php` on your front page, errors may still exist within other template files such as `single.php`, `page.php`, `archives.php`, or `category.php`. Validate pages that load each of your theme template files.

The [Your WordPress](#) section in the [WordPress Forums](#) is dedicated to helping WordPress users in getting feedback about their sites. WordPress volunteers will do that for you for free. Be sure and read the [WordPress Site Reviews Guidelines](#).

Validation Checklist

To help you validate your WordPress site, here is a quick checklist:

1. Validate your HTML / CSS / Feeds (see below)
2. Validate accessibility – [WCAG Guidelines](#), Section 508 Standards and WAI standards
3. Check different browsers

4. Re-validate HTML and CSS

Validate HTML

- [The W3C's HTML Validation Service](#) (URL and upload)
- [W3C's Collection of Validators](#)
- [W3C Tidy Online](#)
- [Silk Tide Online Validator for Errors and Accessibility](#)
- [Windows GUI Interface for TIDY](#)
- [Site Report Card Validator](#)
- [Valet Webthing.com](#)
- [Watson Addy's Validator](#)
- [Alpine Internet HTML Validator](#)
- [AnyBrowser's HTML Validation](#)
- [Cynthia Says Validator](#)
- [Doctor-HTML Validator](#)
- [HTMLvalidator.com's Validator](#)
- [Software QA and Testing Resource Center](#)
- [HTML Tag Checker](#)
- [W3.org Tidy Validator](#)
- [HTMLHelp's File Upload HTML Validator](#) (upload)
- [Searchengineworld's HTML File Upload Validator](#) (upload)

Validate CSS

- [W3.org's CSS Validator](#) (URL, upload, and direct paste in)
- [Style-Sheets.com Validator](#) (browser specific)
- [W3C Style sheet validator](#)
- [WDG and HTMLhelp.com's CSS Validator and Checker](#) (URL, upload, and direct paste)

Validate Feeds

- [Feedvalidator.org](#) – for Atom and RSS feeds.
- [Experimental RSS 1.0 Validator](#)
- [Redlands RSS 1.0 Validator](#)

Validation Resources and Articles

- [Writing Code in Your Posts](#)

- [Mark Freeman's Many Links to Validation Resources](#)
- [Squarefree's Bookmarklets](#) (JavaScript for web page testing)
- [Validator.com](#)
- [Understanding How HTML Validators Work](#)
- [You Call That Web Site Testing?](#)

Related

- [Accessibility](#)

Plugin API Hooks

A theme should work well with WordPress plugins. Plugins add functionality by using actions and filters, which are collectively called hooks (see [Plugin API](#) for more information).

Most hooks are executed internally by WordPress, so your theme does not need special tags for them to work. However, a few hooks need to be included in your theme templates. These hooks are fired by special Template Tags:

[wp_head\(\)](#)

Goes at the end of the `<head>` element of a theme's `header.php` template file.

[wp_body_open\(\)](#)

Goes at the begining of the `<body>` element of a theme's `header.php` template file.

[wp_footer\(\)](#)

Goes in `footer.php`, just before the closing `</body>` tag.

[wp_meta\(\)](#)

Typically goes in the `Meta` section of a Theme's menu or sidebar.

[comment_form\(\)](#)

Goes in `comments.php` directly before the file's closing tag (`</div>`).

Take a look at a core theme's templates for examples of how these hooks are used.

Releasing Your Theme

This section covers the requirements and submission process for releasing your theme into the [WordPress Theme Directory](#). If you've followed the instructions in this handbook, your theme is almost ready for release into the directory.

The first requirement to releasing your theme is to make sure you have – at minimum – the [required theme files](#) before submitting your theme for review.

Once you've confirmed the required theme files, thorough [testing](#) of the theme – including content – must be completed. Following the [theme review guidelines](#) will help ensure acceptance of your theme.

After completing everything above, having the proper [documentation](#) is the final step before [submitting your theme for approval](#). After submitting your theme for review, the theme reviewer may request other changes to your theme.

Important Notices

As we have a large number of tickets in the review queue, reviewers are allowed to close tickets as “not-approved” for themes with more than 3 distinct required issues. If you don't wish to wait extra months for the next review, take a time to fully understand [Theme Review Guidelines](#) and strictly follow it.

Tip:

“[How to do a review \(Draft\)](#)” is the another practical guide even for the theme authors to know the detail about review process.

First Three Checks

Below three checks are very basic ones that many theme authors are missing at the first time.

1. Did you understand the rules of WordPress Theme in repository?

- Themes must be compatible with the GNU General Public License v2, or any later version.

- Plugin territory and non-design related functionality are not allowed. Adding shortcodes and custom post types are not allowed in themes.
- and [more](#).

2. Did you run Theme Sniffer plugin?

The Theme Sniffer plugin is the test tool using [WordPress-Theme](#) standard forked from WordPress Coding Standards. For more details, refer to [this document](#).

3. Did you run Theme Check plugin?

The Theme Check plugin is an easy way to test your theme and make sure it's up to spec with the latest theme review standards. For more details, refer to [this document](#).

Required Theme Files

Before you submit your theme for review, you must make sure that certain files are included. These files must follow template file standards set by the Theme Review Team. In addition to the files below, there are several other standard template files that we recommend you use, as discussed on the [Organizing Theme Files](#) page.

Required Theme Files

1. **style.css**

Your theme's main [stylesheet](#) file. This file will also include information about your theme, such as author name, version number, and plugin URL, in its header.

2. **index.php**

The main [template file](#) for your theme. This will be the template for the homepage on your site unless a static front page is specified. If you *only* include this template file, it must include all functionality of your theme. However, you can use as many relevant template files as you want in your theme.

3. **comments.php**

The comment template which is included wherever comments are allowed. This file should provide support for threaded comments and trackbacks, and should style author comments differently than user comments. See the [Comments](#) page for more information.

5. **screenshot.png**
6. In the WordPress.org theme directory, screenshot.png acts as a visual indicator of what your theme looks like. It is visible both in the web view and in the admin dashboard. Note: this screenshot can be no larger than 1200x900px.

While these files are the only files required by the theme review team for acceptance into the WordPress.org theme directory, you may use other template files. Of course, any file mentioned in the tutorial in this handbook may be used in your theme.

Testing

If you've followed this handbook, you'll already have a good grasp on the testing required before submitting your theme to the WordPress.org theme directory. If you haven't, this page will give you a quick refresher.

Testing is incredibly important before releasing a theme. You maybe have built the most beautiful WordPress theme, but if it breaks when someone tries to comment or insert an image, your theme isn't ready for real world usage.

Before testing your theme, make sure you've setup a development environment. There are a number of ways to setup your environment, many of which are documented on the [Setting up a Development Environment](#) page.

Theme Unit Test

After you've setup your development environment, you'll need test content to test your theme with. While you can create your own test content, the theme review team has created the [Theme Unit Test](#), which includes many different types of content. This will help ensure that your theme works in circumstances you may not have anticipated.

The Theme Unit Test is a WordPress export file that can be imported into any WordPress installation by using the WordPress Importer. You should import it into your local development environment.

WordPress Settings

Making tweaks and changes to your WordPress installation is another good way to ensure that you've built your theme to handle numerous scenarios. Use the following settings to test your theme.

General

Set the Site Title to something long and set the Tagline to something even longer. These settings will test how your theme handles edge cases for site titles and taglines.

Reading

Set “Blog pages show at most” to 5. This setting will ensure that index/archive pagination is triggered.

Discussion

Enable “Threaded Comments” at least 3 levels deep. This setting will facilitate testing of your theme’s comment list styling.

Enable “Break comments into pages” and set 5 comments per page to test the pagination and styling of comments.

Media

Remove the values for the large size of media to test the theme’s `$content_width` setting.

Permalinks

Change the permalink setting a few times to ensure your theme can handle various URL formats.

For more setup instructions, take a look at the [Theme Unit Test](#) page in the WordPress Codex.

WordPress Beta Tester

WordPress releases happen three times a year. It’s a good idea to test your theme against the next version of WordPress so you can anticipate issues when the next version is released. This can be done easily with the [WordPress Beta Tester](#) plugin. The plugin makes it easy to download either the latest nightly version of WordPress or the latest branch version (for minor bugfix releases). This is especially useful when anticipating a new major release or developing for an upcoming feature.

Testing and debugging tools

Theme Check

Each theme goes through an automated check before a reviewer even sees it. If there are any immediate problems with the theme, identified by the automated check, the theme will be rejected with notes on how to resolve the issues. The [Theme Check](#) plugin adds a dashboard link under Appearance so you can run the exact same checks that

WordPress.org does right from your administration panel. Doing this prior to uploading your theme lets you know what needs to be addressed prior to submission. Running the check will give you a list of any warnings your theme has generated and what items are required for the theme to be accepted in the WordPress.org theme directory, as well as any recommended items that may be missing from your theme.

Developer

The [Developer](#) plugin is really just a tool to automatically download and install some of the plugins you'll want when developing your theme. Some of the ones discussed in this handbook will already be installed and active. Others you can install as soon as you activate the plugin.

Debug Bar

[Debug Bar](#) pushes all debug messages to a separate page where they are listed in an easy-to-read layout and organized by type of message. There are also a number of [other plugins](#) that add on to Debug Bar, extending its features or adding more information.

Log Deprecated Notices

[Log Deprecated Notices](#) displays a list of the deprecated function notices in your theme and where the code can be found. This should be run, at minimum, after every major release of WordPress, so you can resolve and remove any deprecated code and functions from your theme.

Browser testing

When submitting your theme to WordPress.org, it's expected that it works well in modern browsers and at any resolution. You should test your theme against a number of popular browsers before submitting, both mobile and desktop. Many browsers have built-in features making it easy to test, for example the [Chrome Developer Tools](#), [Firefox Developer Tools](#), and the [Internet Explorer / Microsoft Edge tools](#).

Validation

Likewise, your theme should use valid HTML5 and CSS code. There are a variety of tools that will test your site for valid code, include [this HTML5 validator](#) and [this CSS validator](#).

Theme Review Guidelines

The [WordPress Theme Review Team](#) provides and maintains the theme review guidelines as part of their task as WordPress Contributors and Developers for the [WordPress Theme Directory](#). Some of these guidelines are required before themes are included in the theme directory. Other guidelines are recommendations for best practices in developing your theme and making it available to as many people as possible. This handbook uses both recommendations and requirements from the theme review guidelines to ensure the theme you develop is accessible to everyone.

The requirements for being included in the WordPress.org theme directory can be found [here](#).

Get Involved

The WordPress Theme Review Team is [open to anyone](#) and is a great way to get a better understanding of how themes are developed. To become a member of the WordPress Theme Review Team, read through the [WordPress Theme Review Team's site](#).

Writing Documentation

Documentation is important for themes as it provides a way for users to understand what a theme does and does not support. Likewise, documenting the code of your theme will make it easier for other theme developers to customize your theme, likely with a [child theme](#).

Here's a list of **requirements** and **recommendations** for your theme's documentation.

- Themes are **required** to provide end-user documentation of any design limitations or extraordinary installation/setup instructions.
- Themes are **required** to include a [readme.txt file](#), using the plugin directory's readme.txt markdown format. New themes need to follow this rule as of October 25th, 2018. Old themes have a 6 months grace time from this date.

Submitting Your Theme to WordPress.org

Before themes are added to the WordPress Theme Directory, they are closely reviewed by the [Theme Review team](#) to make sure they adhere to basic [guidelines](#). This review ensures that WordPress users across the globe can download themes that are high quality and secure.

Need Help?

If you have theme development questions, please post them in the [Developing with WordPress](#) forum. Volunteers from around the world are ready to assist you with the development of your theme.

Guidelines

Make sure you review the [Theme Review Guidelines](#) before uploading a theme. If you have questions about these guidelines, you can ask them in the [#themereview](#) channel in [Slack](#). Anyone with a WordPress.org account can access the Making WordPress Slack.

Testing With Sample Data

The [WordPress Theme Review Team](#) will be reviewing your theme using the sample data from the [Theme Unit Test](#). Before uploading your theme for review, please test it with this sample export data.

Uploading Your Theme

When you are ready to submit your theme for review, please upload your theme ZIP at [Themes > Upload](#). Future updates are to be uploaded via the same page. More information about the theme review process is available on the [Theme Review Team's website](#).

Updating Your Theme

Regardless of whether your theme is approved or waiting for review, if you are uploading a theme update, simply increase the version inside of style.css and upload the themename.zip file again, just like you do with a new theme. Increasing version will NOT change queue position of your theme (if still waiting for review).

References

This section contains lists of [Template Tags](#) and [Conditional Tags](#).

[Template Tags](#) are used in your Template Files to display information dynamically or otherwise customize your site.

[Conditional Tags](#) are a boolean data type that can be used in your Template Files to alter the display of content depending on the conditions that the current page matches.

List of Template Tags

Complete List of Template Tags

Template tags files are stored in the [wp-includes](#) directory. The files have the suffix of “-template.php” to distinguish them from other WordPress files. There are 9 template tags files:

- [wp-includes/general-template.php](#)
- [wp-includes/author-template.php](#)
- [wp-includes/bookmark-template.php](#)
- [wp-includes/category-template.php](#)
- [wp-includes/comment-template.php](#)
- [wp-includes/link-template.php](#)
- [wp-includes/post-template.php](#)
- [wp-includes/post-thumbnail-template.php](#)
- [wp-includes/nav-menu-template.php](#)

Tags

General tags

[wp-includes/general-template.php](#)

- [get_header\(\)](#)
- [get_footer\(\)](#)
- [get_sidebar\(\)](#)
- [get_template_part\(\)](#)
- [get_search_form\(\)](#)
- [wp_loginout\(\)](#)
- [wp_logout_url\(\)](#)
- [wp_login_url\(\)](#)
- [wp_login_form\(\)](#)
- [wp_lostpassword_url\(\)](#)
- [wp_register\(\)](#)
- [wp_meta\(\)](#)
- [bloginfo\(\)](#)
- [get_bloginfo\(\)](#)
- [get_current_blog_id\(\)](#)
- [wp_title\(\)](#)
- [single_post_title\(\)](#)
- [post_type_archive_title\(\)](#)
- [single_cat_title\(\)](#)
- [single_tag_title\(\)](#)
- [single_term_title\(\)](#)
- [single_month_title\(\)](#)
- [get_archives_link\(\)](#)
- [wp_get_archives\(\)](#)
- [calendar_week_mod\(\)](#)
- [get_calendar\(\)](#)
- [delete_get_calendar_cache\(\)](#)
- [allowed_tags\(\)](#)
- [wp_enqueue_script\(\)](#)

Author tags

[wp-includes/author-template.php](#)

- [the_author\(\)](#)
- [get_the_author\(\)](#)
- [the_author_link\(\)](#)
- [get_the_author_link\(\)](#)

- [the_author_meta\(\)](#)
- [the_author_posts\(\)](#)
- [the_author_posts_link\(\)](#)
- [wp_dropdown_users\(\)](#)
- [wp_list_authors\(\)](#)
- [get_author_posts_url\(\)](#)

Bookmark tags

[wp-includes/bookmark-template.php](#) and [wp-includes/bookmark.php](#)

- [wp_list_bookmarks\(\)](#)
- [get_bookmark\(\)](#)
- [get_bookmark_field\(\)](#)
- [get_bookmarks\(\)](#)

Category tags

[wp-includes/category-template.php](#)

- [category_description\(\)](#)
- [single_cat_title\(\)](#)
- [the_category\(\)](#)
- [the_category_rss\(\)](#)
- [wp_dropdown_categories\(\)](#)
- [wp_list_categories\(\)](#)
- [single_tag_title\(\)](#)
- [tag_description\(\)](#)
- [the_tags\(\)](#)
- [wp_generate_tag_cloud\(\)](#)
- [wp_tag_cloud\(\)](#)
- [term_description\(\)](#)
- [single_term_title\(\)](#)
- [get_the_term_list\(\)](#)
- [the_terms\(\)](#)
- [the_taxonomies\(\)](#)

Comment tags

[wp-includes/comment-template.php](#)

- [cancel_comment_reply_link\(\)](#)
- [comment_author\(\)](#)
- [comment_author_email\(\)](#)

- [comment_author_email_link\(\)](#)
- [comment_author_IP\(\)](#)
- [comment_author_link\(\)](#)
- [comment_author_rss\(\)](#)
- [comment_author_url\(\)](#)
- [comment_author_url_link\(\)](#)
- [comment_class\(\)](#)
- [comment_date\(\)](#)
- [comment_excerpt\(\)](#)
- [comment_form_title\(\)](#)
- [comment_form\(\)](#)
- [comment_ID\(\)](#)
- [comment_id_fields\(\)](#)
- [comment_reply_link\(\)](#)
- [comment_text\(\)](#)
- [comment_text_rss\(\)](#)
- [comment_time\(\)](#)
- [comment_type\(\)](#)
- [comments_link\(\)](#)
- [comments_number\(\)](#)
- [comments_popup_link\(\)](#)
- [get_avatar\(\)](#)
- [next_comments_link\(\)](#)
- [paginate_comments_links\(\)](#)
- [permalink_comments_rss\(\)](#)
- [previous_comments_link\(\)](#)
- [wp_list_comments\(\)](#)

Link tags

[wp-includes/link-template.php](#)

- [the_permalink\(\)](#)
- [user_trailingslashit\(\)](#)
- [permalink_anchor\(\)](#)
- [get_permalink\(\)](#)
- [get_post_permalink\(\)](#)
- [get_page_link\(\)](#)
- [get_attachment_link\(\)](#)
- [wp_shortlink_header\(\)](#)
- [wp_shortlink_wp_head\(\)](#)
- [edit_bookmark_link\(\)](#)
- [edit_comment_link\(\)](#)

- [edit_post_link\(\)](#)
- [get_edit_post_link\(\)](#)
- [get_delete_post_link\(\)](#)
- [edit_tag_link\(\)](#)
- [get_admin_url\(\)](#)
- [get_home_url\(\)](#)
- [get_site_url\(\)](#)
- [home_url\(\)](#)
- [site_url\(\)](#)
- [get_search_link\(\)](#)
- [get_search_query\(\)](#)
- [the_feed_link\(\)](#)

Post tags

[wp-includes/post-template.php](#)

- [body_class\(\)](#)
- [next_image_link\(\)](#)
- [next_post_link\(\)](#)
- [next_posts_link\(\)](#)
- [post_class\(\)](#)
- [post_password_required\(\)](#)
- [posts_nav_link\(\)](#)
- [previous_image_link\(\)](#)
- [previous_post_link\(\)](#)
- [previous_posts_link\(\)](#)
- [single_post_title\(\)](#)
- [the_category\(\)](#)
- [the_category_rss\(\)](#)
- [the_content\(\)](#)
- [the_excerpt\(\)](#)
- [the_excerpt_rss\(\)](#)
- [the_ID\(\)](#)
- [the_meta\(\)](#)
- [the_tags\(\)](#)
- [the_title\(\)](#)
- [get_the_title\(\)](#)
- [the_title_attribute\(\)](#)
- [the_title_rss\(\)](#)
- [wp_link_pages\(\)](#)
- [get_attachment_link\(\)](#)
- [wp_get_attachment_link\(\)](#)

- [the_attachment_link\(\)](#)
- [the_search_query\(\)](#)
- [is_attachment\(\)](#)
- [wp_attachment_is_image\(\)](#)
- [wp_get_attachment_image\(\)](#)
- [wp_get_attachment_image_src\(\)](#)
- [wp_get_attachment_metadata\(\)](#)
- [get_the_date\(\)](#)
- [single_month_title\(\)](#)
- [the_date\(\)](#)
- [the_date_xml\(\)](#)
- [the_modified_author\(\)](#)
- [the_modified_date\(\)](#)
- [the_modified_time\(\)](#)
- [the_time\(\)](#)
- [the_shortlink\(\)](#)
- [wp_get_shortlink\(\)](#)

Post Thumbnail tags

[wp-includes/post-thumbnail-template.php](#)

- [has_post_thumbnail\(\)](#)
- [get_post_thumbnail_id\(\)](#)
- [the_post_thumbnail\(\)](#)
- [get_the_post_thumbnail\(\)](#)

Navigation Menu tags

[wp-includes/nav-menu-template.php](#)

- [wp_nav_menu\(\)](#)
- [walk_nav_menu_tree\(\)](#)

See Also

- [Conditional Tags](#)

List of Conditional Tags

•

Conditional Tags are a boolean data type that can be used in your Template Files to alter the display of content depending on the conditions that the current page matches. They tell WordPress what code to display under specific conditions. Conditional Tags usually work with PHP [if /else](#) Conditional Statements and have close relation with WOrdPress [Template Hierarchy](#).

Warning: You can only use conditional query tags after the [WP Query](#) is set up or with [action hook](#).

Complete List of Conditional Tags

- [is_front_page\(\)](#)
- [is_home\(\)](#)
- [is_front_page\(\)](#)
- [is_home\(\)](#)
- [is_admin\(\)](#)
- [is_network_admin\(\)](#)
- [is_admin_bar_showing\(\)](#)
- [is_single\(\)](#)
- [is_sticky\(\)](#)
- [is_post_type_hierarchical\(\\$post_type \)](#)
- [is_post_type_archive\(\)](#)
- [is_comments_popup\(\)](#)
- [comments_open\(\)](#)
- [pings_open\(\)](#)
- [is_page\(\)](#)
- [is_page_template\(\)](#)
- [is_category\(\\$category \)](#)
- [is_tag\(\)](#)
- [is_tax\(\)](#)
- [has_term\(\)](#)
- [term_exists\(\\$term, \\$taxonomy, \\$parent \)](#)
- [is_taxonomy_hierarchical\(\\$taxonomy \)](#)
- [taxonomy_exists\(\\$taxonomy \)](#)
- [is_author\(\)](#)
- [is_date\(\)](#)
- [is_year\(\)](#)

- `is_month()`
- `is_day()`
- `is_time()`
- `is_new_day()`
- `is_archive()`
- `is_search()`
- `is_404()`
- `is_paged()`
- `is_attachment()`
- `wp_attachment_is_image($post_id)`
- `is_local_attachment($url)`
- `is_singular()`
- `post_type_exists($post_type)`
- `is_main_query()`
- `is_new_day()`
- `is_feed()`
- `is_trackback()`
- `is_preview()`
- `in_the_loop()`
- `is_dynamic_sidebar()`
- `is_active_sidebar()`
- `is_active_widget($widget_callback, $widget_id)`
- `is_blog_installed()`
- `is rtl()`
- `is_multisite()`
- `is_main_site()`
- `is_super_admin()`
- `is_user_logged_in()`
- `email_exists($email)`
- `username_exists($username)`
- `is_plugin_active($path)`
- `is_plugin_inactive($path)`
- `is_plugin_active_for_network($path)`
- `is_plugin_page()`
- `is_child_theme()`
- `current_theme_supports()`

- `has_post_thumbnail($post_id)`
- `wp_script_is($handle, $list)`

The Conditions For ...

All of the Conditional Tags test to see whether a certain condition is met, and then returns either TRUE or FALSE. The conditions under which various tags output TRUE is listed below. Those tags which can accept parameters are so noted.

The Main Page

- `is_home()`

The Front Page

- `is_front_page()`

The Blog Page

- `is_front_page()`
- `is_home()`

A Single Post Page

- `is_single()`

A PAGE Page

- `is_page()`
- `is_page_template()`

Has Post Thumbnail

- `has_post_thumbnail($post_id)`

A Single Page, a Single Post, an Attachment or Any

Other Custom Post Type

- `is_singular()`

A Category Page

- `is_category($category)`

A Tag Page

- `is_tag()`
- `has_tag()`

A Taxonomy Page (and related)

- `is_tax()`
- `has_term()`
- `term_exists($term, $taxonomy, $parent)`
- `is_taxonomy_hierarchical($taxonomy)`
- `taxonomy_exists($taxonomy)`

An Author Page

- `is_author()`

A Date Page

- `is_date()`
- `is_year()`
- `is_month()`
- `is_day()`
- `is_time()`
- `is_new_day()`

Any Archive Page

- `is_archive()`

A Search Result Page

- `is_search()`

A 404 Not Found Page

- `is_404()`

Is Dynamic SideBar

- `is_dynamic_sidebar()`

Is Sidebar Active

- `is_active_sidebar()`

Is Widget Active

- `is_active_widget($widget_callback, $widget_id)`

Is User Logged in

- `is_user_logged_in()`

Email Exists

- `email_exists($email)`

Username Exists

- `username_exists($username)`

A Paged Page

- `is_paged()`

Right To Left Reading

- `is rtl()`

An Attachment

- `is_attachment()`

Attachment Is Image

- `wp_attachment_is_image($post_id)`

A Local Attachment

- `is_local_attachment($url)`

Post Type Exists

- `post_type_exists($post_type)`

Is Main Query

- `is_main_query()`

A New Day

- `is_new_day()`

A Syndication

- `is_feed()`

A Trackback

- `is_trackback()`

A Preview

- `is_preview()`

Has An Excerpt

- `has_excerpt()`

Has A Nav Menu Assigned

- `has_nav_menu()`

Is Blog Installed

- `is_blog_installed()`

Part of a Network (Multisite)

- `is_multisite()`
- `is_main_site()`
- `is_super_admin()`

An Active Plugin

- `is_plugin_active($path)`
- `is_plugin_inactive($path)`
- `is_plugin_active_for_network($path)`
- `is_plugin_page()`

A Child Theme

- `is_child_theme()`

Theme supports a feature

- `current_theme_supports()`

Is Previewed in the Customizer

- `is_customize_preview()`

Credits

This list of credits is taken from revisions as pages are migrated over.

- [@ancawonka](#)
- [@Anthonynotes](#)
- [@atachibana](#)
- [@austingunter](#)
- [@BandonRandon](#)
- [@BFTrick](#)

- [@BigActual](#)
- [@bkozma](#)
- [@boborchard](#)
- [@Brainfestation](#)
- [@celloexpressions](#)
- [@charliehanger](#)
- [@claymonk](#)
- [@code_poet](#)
- [@crondeau](#)
- [@davidjlaietta](#)
- [@digisavvy](#)
- [@DrewAPicture](#)
- [@enricchi](#)
- [@esmi](#)
- [@fabianapsimoes](#)
- [@fiveo](#)
- [@grapplerulrich](#)
- [@hanni](#)
- [@hflashbrooke](#)
- [@hardeepasrani](#)
- [@iandstewart](#)
- [@jackreichert](#)
- [@jacobmc](#)
- [@JasonM4563](#)
- [@jazzs3quence](#)
- [@jboydston](#)
- [@jcastaneda](#)
- [@jcrglobalcaplaw](#)
- [@JenniferBourn](#)
- [@jerrysarcastic](#)
- [@jgclarke](#)
- [@jhoffm34](#)
- [@jim-spencer](#)
- [@jimdoran](#)
- [@juhise](#)
- [@juliekuehl](#)
- [@kadamwhite](#)
- [@kafleg](#)
- [@karmatosed](#)
- [@kenshino](#)
- [@kenyasullivan](#)
- [@kpdesign](#)

- [@lettergrade](#)
- [@linux-garage](#)
- [@lizkaraffa](#)
- [@lorax](#)
- [@mariawoothemescom](#)
- [@markel](#)
- [@martydia](#)
- [@missybunnie](#)
- [@mt_Suzette](#)
- [@nao](#)
- [@NikV](#)
- [@Otto42](#)
- [@ozzyr](#)
- [@philiparthurmoore](#)
- [@PriscillaBiju](#)
- [@rachelbaker](#)
- [@Raporteur](#)
- [@rahulsprajapati](#)
- [@rclations](#)
- [@rhauptman](#)
- [@RussE](#)
- [@ryanr14](#)
- [@samuelsidler](#)
- [@sarahovenall](#)
- [@semblance_er](#)
- [@sewmyheadon](#)
- [@SeReedMedia](#)
- [@sheebaabraham](#)
- [@siobhan](#)
- [@stubbs123](#)
- [@suastex](#)
- [@sushil-adhikari](#)
- [@Techdoode](#)
- [@thepixelista](#)
- [@topher1kenobe](#)
- [@TJNowell](#)
- [@xfrontend](#)
- [@viniciusloureco](#)
- [@vrm](#)

Feedback

Giving Feedback

We welcome suggestions to improve the article, whether it is to add new topics or to rectify mistakes, you're invited to help.

You can do so in the following ways

- Connect to [Slack](#) and join [#docs](#) and make your suggestion there, one of the Documentation Team members will help
- Join the [Documentation Team](#) as an editor and edit away!

In-page feedback forms are currently planned for the handbooks.