Desarrollo de Plugins de Wordpress

Introducción

Crear plugins en WordPress es una excelente manera de personalizar y extender las funcionalidades de tu sitio de manera modular. Los plugins permiten agregar nuevas características, modificar el comportamiento existente y, en general, adaptar WordPress a tus necesidades específicas sin alterar el núcleo del sistema.

¿Qué es un Plugin en WordPress?

En WordPress, un plugin es un conjunto de archivos que añade funcionalidades específicas a tu sitio. Puede contener código PHP, estilos CSS, scripts JavaScript y otros recursos necesarios para implementar una característica particular. Los plugins son independientes del tema, lo que significa que puedes cambiar el aspecto de tu sitio sin perder las funcionalidades proporcionadas por los plugins.

Beneficios de Crear un Plugin:

- **Personalización:** Los plugins permiten personalizar tu sitio de acuerdo con tus necesidades sin tocar el código principal de WordPress.
- **Reutilización:** Puedes reutilizar el mismo plugin en múltiples sitios, lo que ahorra tiempo y esfuerzo en la implementación de funcionalidades comunes.
- **Escalabilidad:** Los plugins son escalables y pueden crecer a medida que evolucionan tus necesidades o la complejidad de tu sitio.
- **Compatibilidad:** Al mantener la lógica personalizada en plugins, facilitas la actualización de WordPress sin perder funcionalidades.

Conceptos básicos

Cuando WordPress se actualiza a una nueva versión, anula sus archivos principales. Debido a esto, si agregas una funcionalidad personalizada a un sitio de WordPress modificando directamente el núcleo de WordPress, tus cambios se eliminarán al actualizar WordPress. Esto lleva a uno de los conceptos centrales de desarrollo de WordPress: cualquier funcionalidad que desee agregar o modificar debe realizarse mediante plugins.

Un plugin de WordPress es esencialmente una o más funciones definidas en archivos PHP, ya que PHP es el principal lenguaje de programación que impulsa WordPress. También suele tener otros 3 componentes: hooks (action hookss y filtre books), shortcodes y widgets. Estos son los elementos principales del desarrollo de complementos de WordPress.

Hooks

Los hooks son características de WordPress que le permiten manipular un proceso en un punto específico sin cambiar los archivos principales de WordPress. Por lo tanto, los hooks proporcionan una forma para que su complemento se conecte al funcionamiento del núcleo de WordPress. Puede asociar fragmentos de código o funciones con enlaces para ejecutarlos en varios momentos. Un hook se puede aplicar tanto a la acción (hook de acción) como al filtro (hook de filtro).

Antes de profundizar, cubramos rápidamente algunas diferencias entre estos dos:

Una acción es un proceso en WordPress. Un hook de acción le permite agregar un proceso. Puedes trabajar con acciones usando la función add_action() en WordPress. Ejemplos comunes de acciones son crear, leer o guardar una publicación en WordPress. Puede asociar funciones PHP o fragmentos de código con acciones. Incluso puedes crear tu propia acción y asociarle código. Una acción le permite agregar funcionalidad a un complemento.

Puede conectarse a una acción y ejecutar su funcionalidad personalizada. Por ejemplo, puedes asociar tu función, custom_function() a la acción de guardar una publicación. Debe utilizar la función add_action() como se muestra a continuación.

```
function custom_function( $post_id ) {
   //do something
}
add_action( 'save_post','custom_function' );
```

Este fragmento de código garantiza que cada vez que WordPress guarde una publicación, se ejecutará la función custom_function().

Un filtro es un hook que modifica un proceso. Los filtros ayudan a manipular los datos existentes, sin la necesidad de alterar su fuente. Puede utilizar la función apply_filters() para utilizar un hook de filtro. Se necesitan dos argumentos obligatorios: el nombre del filtro y el valor que se filtrará.

```
echo apply_filters( 'filter_name','filter_variable' );
```

Además, puede utilizar la función add_filter() para crear un filtro personalizado. Usted define el nombre del filtro que desea llamar, junto con la función que se llamará para modificar el filtro.

```
function modify_value( $filter ) {
   //change $filter and return new value
}
add_filter( 'original_filter','modify_value' );
```

Por ejemplo, vamos a crear un filtro que modifica el título de un post para convertirlo en mayúsculas. El hook de filtro toma el título de la publicación usando the_title y luego ejecuta la función de título en mayúsculas para modificar el título existente para que esté todo en mayúsculas:

```
add_filter('the_title', 'uppercase_title');
function uppercase_title ($title) {
        // do the transformation only in single post page
        if(is_single() && 'post' == get_post_type()) {
                $title = strtoupper($title);
        }

  return $title;
}
```

Bloques

Los bloques han sido una parte importante de cómo funcionan los complementos de WordPress desde que se lanzó el editor de bloques ("Gutenberg") en WordPress 5.0.

Dependiendo de la funcionalidad de su complemento, es posible que desee considerar el uso de bloques para permitir a los usuarios interactuar con tu plugin, como insertar contenido de plugin o incluso crear contenido usando tu plugin.

Sin embargo, para utilizar bloques en tu plugin, necesitará tener un conocimiento sólido de JavaScript, Node.js, React y Redux, lo que ha desanimado a algunos desarrolladores, El desarrollo de complementos de WordPress solía estar mucho más centrado en PHP. Siempre puedes comenzar sin bloques y agregarlos más tarde si encajan con tu plugin.

Shortcodes

Cuando desarrollas un complemento, este no tiene acceso directo al tema de WordPress. Para comunicarse con el tema de WordPress y mostrar información al usuario, debe utilizar un código corto. Los códigos cortos permiten a los usuarios insertar un elemento HTML dinámico en una publicación o página.ç

Aquí hay un ejemplo de un código corto que agrega el texto "Hola mundo" a tu publicación.

```
//Register shortcode
add_shortcode( 'hello_world_shortcode','hello_world_output' );

//define function to show output
function hello_world_output( $atts, $content = '', $tag ){
    $html = '';
    $html .= 'Hello World';
    return $html;
}
```

Widgets

Los widgets brindan a los desarrolladores otra forma de mostrar el contenido de su complemento al usuario final. WordPress tiene una clase WP_widget en PHP, que debes ampliar para crear un widget para tu complemento.

Ahora que cubrimos los conceptos básicos del desarrollo de complementos de WordPress, exploremos los pasos clave para crear un complemento personalizado.

Primer paso: Investigar y planificar

Hay miles de herramientas en el directorio de plagian de WordPress. Por lo tanto, lo primero que querrás hacer es investigar un poco para ver si tu idea ya existe.

Sin embargo, incluso si es así, aún podrías seguir adelante con tu plan. Es posible que desee explorar plugins similares y descubrir cómo podría mejorarlos. Alternativamente, puedes complementar lo que ya está disponible con algo como tu propio tipo de publicación personalizada y funciones adicionales.

Es posible que también desees comprobar el estado de los plugins existentes. Por ejemplo, si un plugin no se ha actualizado durante algún tiempo o no es compatible con la última versión de WordPress, podría existir la oportunidad de adoptarlo o proporcionar una mejor solución.

También puedes consultar la cantidad de instalaciones activas para ver si hay un gran mercado para el tipo de plugin que tienes en mente. También es una buena idea probar el plugin en su propio sitio para ver qué hace bien y qué se podría hacer mejor.

También querrás considerar cómo comercializarás tu plugin. Por ejemplo, algunos desarrolladores crean un sitio web dedicado a sus productos. Si planeas monetizar tu plugin, deberá pensar tanto en el precio como en las opciones de suscripción.

Finalmente, querrás leer sobre los estándares de codificación de WordPress. Esto es particularmente importante si planeas compartir tu plugin con otras personas. Estos estándares de codificación son un conjunto de pautas y mejores prácticas que los desarrolladores deben intentar seguir al crear temas y plugins para WordPress.

Antes de comenzar a trabajar con el código real y los archivos de tu plugin, querrás trabajar un poco para preparar los conceptos básicos. Primero, querrás determinar claramente los requisitos de su complemento antes de comenzar a codificar. Aquí Hay algunas preguntas comunes que debería poder responder:

- ¿Cuáles son las características del complemento?

- ¿Cómo controlarán los usuarios esas características?

- ¿Qué información debe aceptar el complemento de los usuarios?

- ¿Cómo se verá el diseño del front-end (si corresponde?)?

- ¿El complemento se integrará con otros complementos o servicios?

- ¿Qué posibles problemas de compatibilidad podría tener el complemento?

Las respuestas a estas preguntas afectarán la forma en que enfoques la codificación de tu plugin. Si deseas publicar el plugin públicamente, también deberás pensar en un nombre para tu plugin. Para evitar problemas con el nombre de tu plugin, aquí tienes algunos consejos:

Comprueba si ya existe un complemento con ese nombre, ya que esto podría causar confusión.

Ten en cuenta los problemas de marcas registradas. Por ejemplo, no puedes usar "WordPress" en el nombre oficial de tu complemento, aunque puedes usar la abreviatura "WP".

Del mismo modo, ten cuidado al utilizar marcas comerciales de otras marcas. Por ejemplo, muchos plugins que usaban "Instagram Feed" en su nombre tuvieron que cambiar sus nombres

Segundo paso: Crear la estructura de ficheros y carpetas

El directorio predeterminado de WP para almacenar el código del plugin en el back-end es /wp-content/plugins/. La forma de estructurar tu plugin dentro de este directorio dependerá de la complejidad del plugin. El nombre del directorio es el mismo que el nombre de su plugin, en minúsculas y guiones en lugar de espacios.

Recomendamos tener un único archivo PHP que contenga todo el código del plugin (/wp-content/plugins/my-plugin/my-plugin.php). Esta estructura es ideal para un plugin simple que cumple una función pequeña.

Si planea trabajar con un plugin que tiene muchos recursos, puede organizar su plugin según la función del código y los archivos PHP. Puede crear directorios como activos para archivos CSS y JavaScript, i18n para archivos de localización, plantillas y widgets.

Para plugins más complejos, puede crear una vista MVC, con directorios para modelo, vista y controlador dentro del directorio my-plugin. Esto ayuda a depurar más tarde en menos tiempo. En nuestro ejemplo simple y directo del complemento Hello World, crearemos el directorio hello-world con un único archivo PHP, hello-world.php dentro de él.

Tercer paso: Comenzar a crear el archivo del plugin

Una vez que cree su directorio de plugins y agregue archivos dentro de él, deberá agregar el encabezado del archivo. El encabezado del archivo es un bloque de comentarios PHP que contiene información sobre el plugin. Puede encontrar el contenido de un encabezado de archivo de muestra en el Codex de WordPress.

Después de agregar el encabezado del archivo, aparecerá en la lista de complementos de su administrador de WordPress.

Las siguientes líneas constituyen los encabezados del complemento y van a la parte superior del archivo del plugin.

```
<?php
/**
 * Hello World
 *
 * @package HelloWorld
 * @author Your Name
 * @copyright 2020 Your Name
 * @license GPL-2.0-or-later
 *
 * @wordpress-plugin
 * Plugin Name: Hello World
 * Plugin URI: https://mysite.com/hello-world
 * Description: Prints "Hello World" in WordPress admin.
 * Version: 0.0.1
 * Author: Your Name
 * Author URI: https://mysite.com
```

```
 * Text Domain: hello-world
 * License: GPL v2 or later
 * License URI: http://www.gnu.org/licenses/gpl-2.0.txt */
...
?>
```

Veamos el significado de estas líneas de código:

Hello World**:** Nombre del plugin o descripción corta.

@package HelloWorld: Define el paquete al que pertenece el plugin. En este caso, el paquete se llama “HelloWorld".

@author Your Name: El autor del plugin. Sustituye "Your Name" con tu nombre o el nombre del desarrollador.

@copyright 2020 Your Name: El año de copyright y el titular de los derechos de autor. Sustituye "Your Name" con el titular del copyright.

@license GPL-2.0-or-later: Especifica la licencia bajo la cual se distribuye el plugin. En este caso, la Licencia Pública General de GNU, versión 2.0 o posterior.

@wordpress-plugin: Etiqueta que indica que es un plugin de WordPress.

Plugin Name: Hello World: Nombre oficial del plugin que se mostrará en el panel de administración de WordPress.

Plugin URI: https://mysite.com/hello-world: URL del sitio web relacionado con el plugin. En este caso, el enlace al plugin.

Description: Prints "Hello World" in WordPress admin.**:** Descripción corta del plugin que se mostrará en el panel de administración de WordPress.

Version: 0.0.1: Número de versión del plugin. Este número debe incrementarse cada vez que realices una actualización.

Author: Your Name: El autor del plugin. Sustituye "Your Name" con tu nombre o el nombre del desarrollador.

Author URI: https://mysite.com: URL del sitio web del autor o del desarrollador del plugin.

Text Domain: hello-world: Dominio de texto utilizado para la internacionalización y localización del plugin. License: GPL v2 or later: Información sobre la licencia del plugin. En este caso, es la Licencia Pública General de GNU, versión 2.0 o posterior.

License URI: http://www.gnu.org/licenses/gpl-2.0.txt: URL que apunta a la página que contiene el texto completo de la licencia.

Estos comentarios son esenciales para la correcta identificación, documentación y distribución del plugin. Además, proporcionan información clave sobre el plugin que puede ser utilizada por otros desarrolladores y por WordPress.

Cuarto paso: Añadir código al plugin

Si bien has creado un plugin vacío, todavía no logra nada. Necesitamos agregarle funcionalidad ahora. El manual de plugins de WordPress debería servir como guía. Este es el paso en el que le das vida a tu idea.

En nuestro ejemplo de plugin simple, crearemos una página de administración de WordPress con el texto "Hola mundo". Cuando agrega una página de administración, también agrega un elemento de menú.

```
<?php
function print_hello_world_title()  {
  echo "<h1>Hello World</h1>";
}

function hello_world_admin_menu()  {
  add_menu_page(
    'Hello World',// page title
    'Hello World Menu Title,// menu title
    'manage_options',// capability
    'hello-world',// menu slug
    'print_hello_world_title'  // callback function
  );
}

add_action( 'admin_menu', 'hello_world_admin_menu' );

?>
```

El fragmento de código anterior agrega un elemento de menú usando el hook de acción admin_menu y ejecuta la función hello_world_admin_menu. En la función, utilizamos la función incorporada add_menu_page(), que agrega un elemento de menú y una página para el complemento en el administrador de WordPress. Agregamos el título de la página, el título del menú, la capacidad, el slug del menú y una función de devolución de llamada. Definimos el contenido de la página en la función de devolución de llamada, donde simplemente imprimimos el encabezado "Hola mundo”.

La función `add_menu_page` es una función clave en WordPress que te permite agregar elementos al menú de administración de WordPress. Este puede ser un punto de entrada para funciones personalizadas, enlaces a páginas o incluso la creación de menús completos dentro del área de administración. Aquí hay una descripción detallada de `add_menu_page`:

```
add_menu_page(
    string $page_title,
    string $menu_title,
    string $capability,
    string $menu_slug,
    callable $function = '',
    string $icon_url = '',
    int $position = null
);
```

La función add_menu_page es una función clave en WordPress que te permite agregar elementos al menú de administración de WordPress. Este puede ser un punto de entrada para funciones personalizadas, enlaces a páginas o incluso la creación de menús completos dentro del área de administración. Aquí hay una descripción detallada de add_menu_page:

Sintaxis:

add_menu_page(string $page_title, string $menu_title, string $capability, string $menu_slug, callable $function = '', string $icon_url = '', int $position = null);

Parámetros:

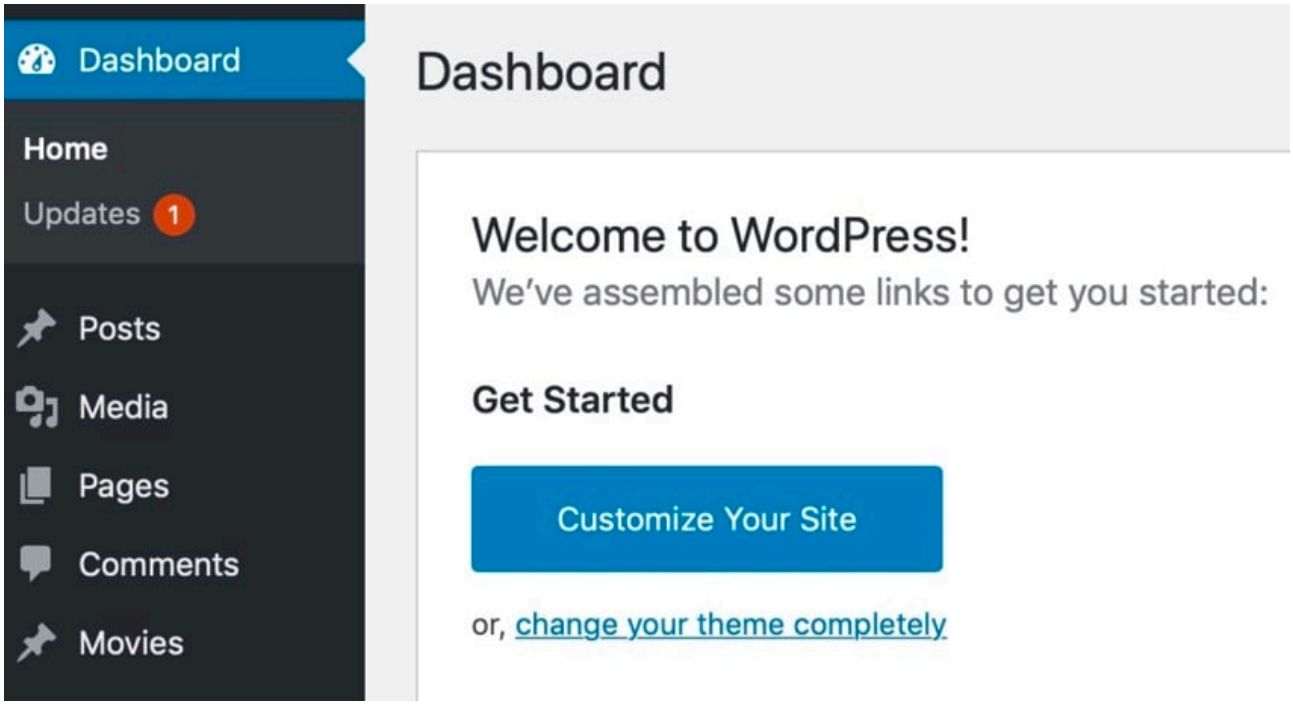
- $page_title (cadena): Título que se mostrará en la página del navegador cuando el menú esté seleccionado.
- $menu_title (cadena): Título del menú que se mostrará en la barra lateral de administración.
- $capability (cadena): Capacidad mínima requerida para acceder a esta página. Puede ser un nivel de capacidad como 'manage_options' o un nombre de capacidad personalizada.
- $menu_slug (cadena):Identificador único para esta página del menú. Debería ser único entre todas las páginas y subpáginas del menú. También se utiliza en la URL de la página.
- $function (callable): Función que se ejecutará cuando se haga clic en el elemento del menú. Puede ser una función propia o el nombre de una función existente.
- $icon_url (cadena): URL del icono que se mostrará junto al título del menú. Puede ser una URL de imagen o una clase de icono de Dashicons.
- $position (int, opcional): Posición en la que se mostrará el menú en la barra lateral. Si no se especifica, se colocará en la parte inferior del menú.

Ejemplo de Uso:

```
// Añadir un elemento de menú "Mi Plugin" al menú de administración
add_action('admin_menu', 'agregar_menu_mi_plugin');

function agregar_menu_mi_plugin() {
    add_menu_page(
        'Mi Plugin',
        'Mi Plugin',
        'manage_options',
        'mi-plugin',
        'mi_pagina_principal',
        'dashicons-admin-plugins',
```

```
        20
    );
}

// Página que se muestra al hacer clic en "Mi Plugin"
function mi_pagina_principal() {
    // Contenido de la página principal del plugin
    echo '<div class="wrap">';
    echo '<h1>Mi Plugin</h1>';
    echo '<p>Bienvenido a la página principal de Mi Plugin.</p>';
    echo '</div>';
}
```

Explicación:



- En este ejemplo, `add_menu_page` se usa dentro de la acción `admin_menu`.
- Se crea un nuevo elemento de menú llamado "Mi Plugin" con el título "Mi Plugin" en la barra lateral de administración.
- La función `mi_pagina_principal` se ejecutará cuando se haga clic en este elemento de menú.

- Se utiliza el icono `dashicons-admin-plugins` y se coloca en la posición 20 en la barra lateral.
-

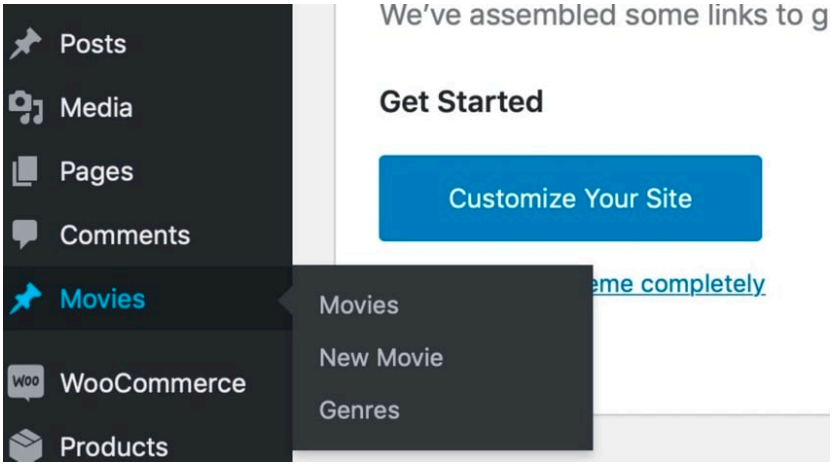
En resumen, `add_menu_page` es esencial para personalizar el área de administración de WordPress y proporciona una forma sencilla de agregar elementos al menú que enlazan con tus propias funciones y páginas.

Vamos a crear otro plugin de ejemplo. En este caso será un plugin que va a permitir crear un custom-post-type de Películas y una taxonomía llamada Géneros. Lo vamos a llamar Register Peliculas. Llamo a mi carpeta películas-register-post-types, y al archivo del plugin peliculas-register-post-types.php.

```
<?
/*
Plugin Name: Películas Register Post Types
Plugin URI: http://midominio.com
Description: Plugin que crea un post type películas
Version: 1.0
Author: Yo mismo
Autor URI: http://www.yomismo.com
License: GPLv2 or later
Text Domain: películas-register
*/
```

Ahora es el momento de escribir la primera función en nuestro plugin. Empieza por hacer tu plugin y añadir las llaves que contendrán el código. Aquí está el mío:

```
function peliculas_register_post_type(){
        $labels = array(
                "name"              =>  __("Movies", "películas-register"),
                "singular_name"     =>  __("Movie", "películas-register"),
                "add_new"           =>  __("New Movie", "películas-register"),
                "add_new_item"      =>  __("Add New Movie", "películas-register"),
                "new_item"          =>  __("New Movie", "películas-register"),
```




```
                "edit_item"         =>  __("Edit Movie", "películas-register"),
                "view_item"         =>  __("View Movie ", "películas-register"),
                "search_items"      =>  __("Search Movies", "películas-register"),
```

```
                “not_found”             =>  __(“No Movies Found”, “películas-register”),
                “not_found_in_trash”        =>   __(“No Movies Found in Trash”, “películas-register”)
        }
```

```
$args = array(
                “labels” => $labels,
                 'has_archive' => true,
                'public' => true,
                 'hierarchical' => false,
                  'supports' => array(
                        ‘title',
                         'editor',
                        'excerpt',
                        ‘custom-fields',
                        ‘thumbnail',
                         'page-attributes'
                ),
                 'rewrite'   => array( 'slug' => 'movies' ),
                 'show_in_rest' => true
                );
        register_post_type('movies', $args);
}
```

Esto incluye todas las etiquetas y argumentos para tu tipo de mensaje y (crucialmente) la función register_post_type() que es proporcionada por WordPress.

He usado películas como tipo de publicación aquí, ya que estoy creando un sitio de revisión de películas imaginarias. Tal vez quieras usar algo diferente.

Ahora, si guardas tu archivo y vuelves a tu sitio, verás que nada ha cambiado. Eso es porque no has activado tu código. El método que usamos para activar la función aquí es enganchándolo a un hook de acción proporcionado por WordPress, el hook de init. Cuando se utiliza una función proporcionada por WordPress (como register_post_type) se encuentra que hay un hook que se debe utilizar. Puedes encontrar detalles en la entrada del manual de WordPress para registrar tipos de correo personalizados.

Así que añadamos el hook. Bajo tu código, y fuera de las llaves, añades esta línea:

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

Usamos la función add_action() para enganchar nuestro código a un hook de acción, con dos parámetros: el nombre del hook de acción y el nombre de nuestra función.

Ahora intenta guardar tus archivos y vuelve a tu sitio. Verás que el tipo de mensaje personalizado se ha añadido a tu menú de administración.

Ahora vamos a añadir una función extra, para registrar una taxonomía personalizada. Debajo del código que has escrito hasta ahora, añade esto:

```
function peliculas_register_taxonomy() {
        // books
        $labels = array(
                'name' => __( 'Genres' , 'películas-registe' ),
                'singular_name' => __( 'Genre', 'películas-registe' ),
                'search_items' => __( 'Search Genres' , 'películas-registe' ),
                'all_items' => __( 'All Genres' , 'películas-registe' ),
                'edit_item' => __( 'Edit Genre' , 'películas-registe' ),
                'update_item' => __( 'Update Genres' , 'películas-registe' ),
                  'add_new_item' => __( 'Add New Genre' , 'películas-registe' ),
                 'new_item_name' => __( 'New Genre Name' , 'películas-registe' ),
                'menu_name' => __( 'Genres' , 'películas-registe' ),
 );

$args = array(
        'labels' => $labels,
         'hierarchical' => true,
        'sort' => true,
        'args' => array( 'orderby' => 'term_order' ),
        'rewrite' => array( 'slug' => 'genres' ),
        'show_admin_column' => true,
        'show_in_rest' => true
      );

register_taxonomy( 'tutsplus_genre', array( 'tutsplus_movie' ), $args);
}

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

De nuevo, puede que quieras cambiar el nombre de tu taxonomía personalizada. Aquí he hecho que la taxonomía se aplique al tipo de post que acabo de registrar (el tercer parámetro de la función register_taxonomy). Si le diste un nombre diferente a tu mensaje, asegúrate de editarlo.

Ahora guarda tu archivo y echa un vistazo a tus pantallas de administración. Cuando pase el cursor por encima de tu tipo de post en el menú de administración, verás la nueva taxonomía.

Si necesitas usar estilos o scripts personalizados en tu plugin, puedes añadirlos directamente al archivo del plugin, pero eso no es la mejor práctica. En su lugar, deberías crear hojas de estilo y scripts como archivos separados en la carpeta del plugin y ponerlos en cola, usando una función proporcionada por WordPress.

Imaginemos que quieres agregarle estilo a tu tipo de post personalizada. Podrías añadirlo al tema, pero quizá quieras añadir algún estilo específico al plugin para que el tipo de publicación destaque sobre los demás tipos de publicaciones de cualquier tema.

Para ello, creas una nueva carpeta dentro de la carpeta de plugins llamada css (o estilos, depende de ti). Dentro de esa carpeta, crea una hoja de estilo llamada style.css, o puedes darle un nombre más específico para mayor claridad. Voy a llamar a la mía movies.css.

Entonces necesitas encolar ese archivo en tu plugin para que pueda ser usado por WordPress. Añade esto a tu archivo principal del plugin, encima de las funciones que ya tienes. Me gusta añadir la cola de espera e incluye primero en mi archivo de plugin para que pueda ver qué otros archivos se están activando.

```
function movie_styles() {
wp_enqueue_style( 'movies',  plugin_dir_url( __FILE__ ) . '/css/movies.css' );
}
add_action( 'wp_enqueue_scripts', 'movie_styles' );
```

Si guardas el archivo, no verás ninguna diferencia en las pantallas de administración, pero si has añadido entradas del tipo de entrada personalizada y tu hoja de estilo incluye el estilo para ellas, ahora lo verás en el front-end de tu sitio.

Nota que el hook usado para poner en cola tanto las hojas de estilo como los scripts es el mismo: ambos usan wp_enqueue_scripts. No hay un hook separado para los estilos.

El encolado de scripts funciona de manera muy similar. Sigue estos pasos:

- Añade una carpeta de scripts o js a tu carpeta de plugins.
- Guarda tus archivos de script en esa carpeta.
- Pon en cola el script de la misma manera que la hoja de estilo anterior, reemplazando la función wp_enqueue_style() por wp_enqueue_script().

Otra opción al desarrollar el plugin es crear archivos PHP adicionales, conocidos como archivos de inclusión. Si tienes muchos de ellos, puedes crear varias carpetas para diferentes tipos de archivos de inclusión, o puedes crear una sola carpeta llamada “includes".

Por ejemplo, en nuestro plugin de tipo post personalizado, podríamos crear algún código para variar la forma de salida del contenido de la página, usando el hook de filtro the_content para modificar el código que se está ejecutando cada vez que el contenido se emite en una página de productos.

En lugar de añadir este código al archivo principal del plugin, podrías añadirlo a un archivo separado llamado movie-content.php y luego escribir el código en ese archivo para la forma en que el contenido se produce para las películas.

Para incluir este archivo en el plugin, se agrega una carpeta llamada includes al plugin y luego se agrega el archivo content-movie.php.

Para incluir ese archivo en tu plugin, añades este código al principio para el archivo principal del plugin:

include(plugin_dir_path(__FILE__) . 'includes/movie-content.php');

No necesitas enganchar esto a una acción o a un hook de filtro, solo usa la función include_once() en tu archivo de plugin. Eso entonces llamará al código del archivo de inclusión como si estuviera en tu archivo principal de plugin en ese momento.

Buenas prácticas

Puedes entrar rápidamente al desarrollo de plugins de WordPress utilizando las herramientas adecuadas. Un editor de texto con el que te sientas cómodo, un cliente FTP para transferir rápidamente archivos entre tu máquina local y el servidor, y un servidor de desarrollo para probar tu plugin en el servidor lo ayudarán a crear tu plugin a través de iteraciones rápidas.

Crear un plugin desde cero requiere mucho tiempo y esfuerzo. Si bien no existe un proceso estándar para crear un plugin, puede optar por el desarrollo de plugins a través de un modelo estándar. El uso de un texto estándar ahorra mucho tiempo al reutilizar el código.

Mientras desarrollas un plugin, utiliza las funciones integradas de WordPress siempre que sea posible para evitar retrabajos y acortar el tiempo de desarrollo web. Cumple con los estándares de codificación de WordPress mientras desarrollas tu plugin.

Utilice la estructura MVC para garantizar una estructura coherente que otros puedan agregar a su complemento cómodamente en una etapa posterior.

Actualiza tu plugin para garantizar el cumplimiento de las últimas versiones de PHP y WordPress. Esto mantiene tu sitio web a salvo de riesgos de seguridad.

Tabla	Acción	Filas	Tipo	Cotejamiento	Tamaño	Residuo a depurar
wp_commentmeta	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_520_ci	48 KB	-
wp_comments	Examinar Estructura Buscar Insertar Vaciar Eliminar	1	InnoDB	utf8mb4_unicode_520_ci	96 KB	-
wp_links	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_520_ci	32 KB	-
wp_options	Examinar Estructura Buscar Insertar Vaciar Eliminar	131	InnoDB	utf8mb4_unicode_520_ci	64 KB	-
wp_postmeta	Examinar Estructura Buscar Insertar Vaciar Eliminar	1	InnoDB	utf8mb4_unicode_520_ci	48 KB	-
wp_posts	Examinar Estructura Buscar Insertar Vaciar Eliminar	3	InnoDB	utf8mb4_unicode_520_ci	80 KB	-
wp_termmeta	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_520_ci	48 KB	-
wp_terms	Examinar Estructura Buscar Insertar Vaciar Eliminar	1	InnoDB	utf8mb4_unicode_520_ci	48 KB	-
wp_term_relationships	Examinar Estructura Buscar Insertar Vaciar Eliminar	1	InnoDB	utf8mb4_unicode_520_ci	32 KB	-
wp_term_taxonomy	Examinar Estructura Buscar Insertar Vaciar Eliminar	1	InnoDB	utf8mb4_unicode_520_ci	48 KB	-
wp_usermeta	Examinar Estructura Buscar Insertar Vaciar Eliminar	18	InnoDB	utf8mb4_unicode_520_ci	48 KB	-
wp_users	Examinar Estructura Buscar Insertar Vaciar Eliminar	1	InnoDB	utf8mb4_unicode_520_ci	64 KB	-
12 tablas	Número de filas	158	InnoDB	latin1_swedish_ci	656 KB	0 B

Siempre sanitice y escape los datos

Al escribir un plugin, es muy importante proteger la salida de tu plugin mediante el uso de las funciones adecuadas de sanitización de datos de WordPress. De lo contrario, tu plugin y todo tu sitio web corren el riesgo de ser pirateados. Tomemos como ejemplo la siguiente declaración PHP:

echo $mi_variable_personalizada;

Escrito de esta manera, permite a los intrusos colocarle código JavaScript o MySQL personalizado que luego puede provocar ataques de inyección XSS y/o MySQL. Es por eso que desea utilizar la función de seguridad incorporada esc_html y en su lugar escribir:

echo esc_html($mi_variable_personalizada)

La función esc_html, tal como sugiere su nombre, escapa de todos los códigos HTML y JavaScript en su variable, haciéndola más segura. Hay muchas funciones utilizadas de manera similar; consulte el Códice de WordPress para obtener documentación sobre todas las funciones.

Usa WPDB siempre que sea posible

Cuando comienzas a escribir un plugin que realiza operaciones CRUD personalizadas (Crear, Leer, Actualizar, Eliminar) en la base de datos, se recomienda encarecidamente utilizar la clase de abstracción de la base de datos de WordPress llamada wpdb.

Hay muchas ventajas al utilizar WPDB. Uno de los beneficios más importantes es la seguridad. En lugar de escribir sus métodos para proteger sus consultas, wpdb ya proporciona métodos de seguridad integrados para proteger sus consultas de ataques a bases de datos como $wpdb->prepare.

Además, la clase $wpdb puede ahorrarle mucho tiempo.

Usa siempre la internacionalización (i11n) y haga que su plugin sea fácilmente traducible

La internacionalización es el proceso de desarrollar su complemento de manera que pueda traducirse fácilmente a otros idiomas. Por ejemplo, un botón "Guardar" codificado como este no sería compatible con i11n.

<botón id=”ejemploBotón”>Guardar</botón>

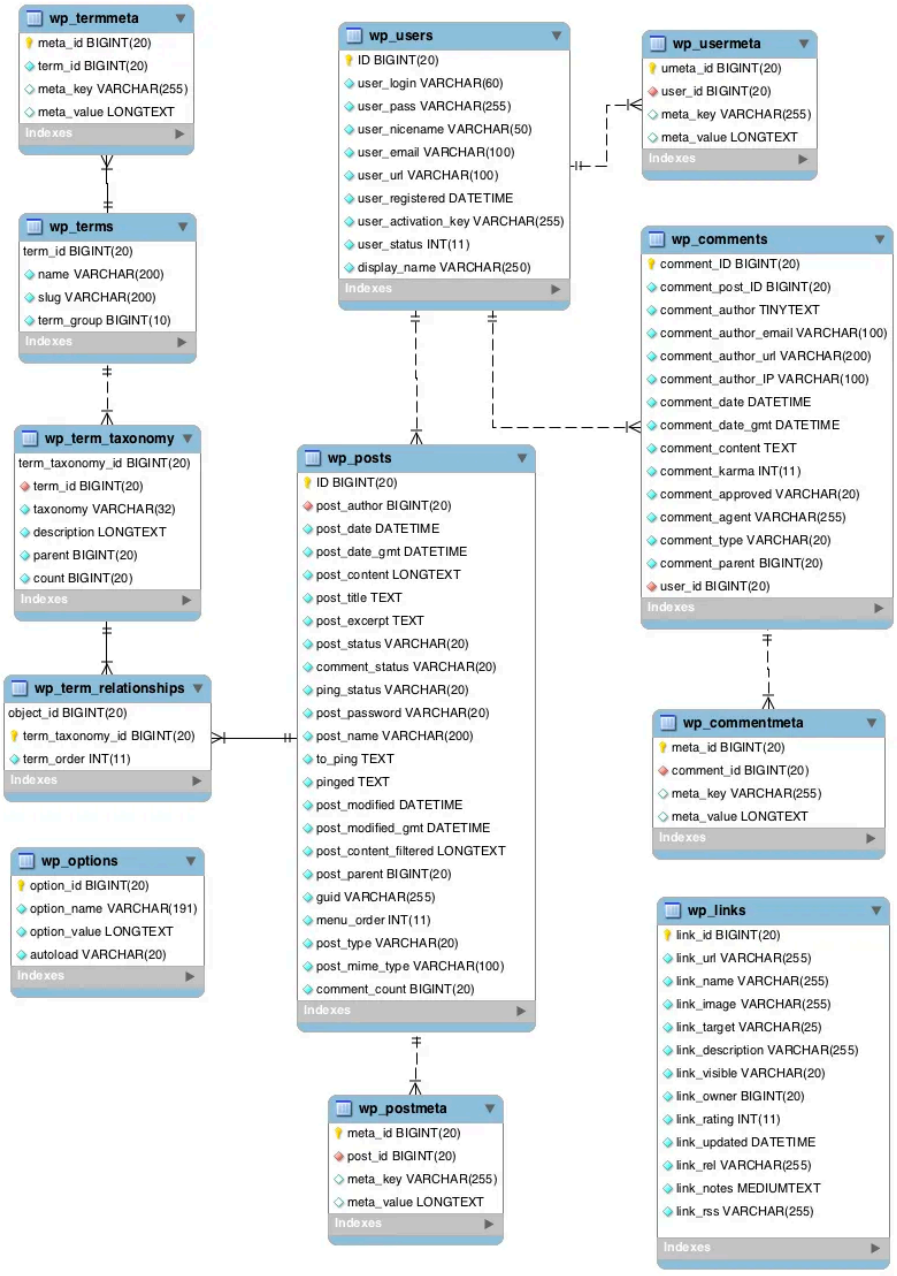
En cambio, debe codificarse así para garantizar que sea traducible:

<button id=”exampleButton”><?php _e('Guardar', 'text_domain'); ?></botón>

Para asegurarse de que está haciendo bien i11n, siga la documentación oficial de internacionalización de WordPress.

Prefijar funciones y agregar un espacio de nombres a la clase

Al escribir un complemento por primera vez, es posible que tenga la tentación de escribir una función como la siguiente:



función obtener_datos() {
 // hacer algo...
}

Si bien esto podría funcionar para usted cuando trabaje en su entorno de desarrollo local, las posibilidades de colisión de nombres con otros complementos serán muy altas ya que puede haber algunos complementos instalados en su sitio wp usando el mismo nombre de función, lo que producirá un error fatal de PHP.

Para resolver este problema, se recomienda a los desarrolladores anteponer sus funciones. Por ejemplo, en lugar de escribir get_data (), cambiará el nombre de la función de get_data() a your_prefix_get_data().

Cambie your_prefix para que coincida con el nombre del directorio de su complemento o dominio de texto. En nuestro ejemplo, usaremos hello_world _get_data().

Además, es posible que desee asegurarse de que no existan funciones que llamen al mismo nombre de función, por lo que puede hacer algo como:

```
si (! function_exists ("hola_mundo_obtener_datos")) {
  función hola_mundo_get_data() {
   // hacer algo...
  }
}
```

Este método no solo evita una colisión de nombres, sino que también hace que su función se pueda sobrescribir a través de Tema o Tema secundario.

Si no quieres empezar un plugin desde cero, te recomiendo que uses un boilerplate, una especie de plantilla preparada para que sólo tengas que añadir tu código, proporcionándote toda la estructura necesaria para tu plugin.

https://github.com/DevinVinson/WordPress-Plugin-Boilerplate/blob/master/plugin-name/admin/class-plugin-name-admin.php

Internacionalización del plugin

Para internacionalizar un plugin, los pasos son idénticos a los del caso de un theme, con la diferencia de la cabecera del plugin.

```
/*
 Plugin Name: My Rad Plugin
 Plugin URI: https://myradplugin.com
 Description: Custom Plugin That Makes My Site Rad
 Author: Rad Plugin Creator
 Version: 1.0
 Author URI: https://radplugincreator.com
 Text Domain: rad-plugin
 Domain Path: /languages/
 */
```

A partir de aquí, los pasos a dar son los mismos que se han de dar para internacionalizar un tema.

Bases de datos en Wordpress

Si nos dedicamos a cualquier cosa relacionada con **WordPress** o tenemos pensado hacerlo, no podemos dejar de lado la comprensión de su base de datos. Dependiendo del nivel en el que nos movamos, tendremos que ahondar más o menos en su comprensión pero en cualquier caso, nunca está de más conocer sus tablas y cómo está estructura.

Vamos a intentar resumir la estructura de de la base de datos de **WordPress** de manera que sea lo más comprensible posible para todos y todas.

Cuando instalamos **WordPress** se crean **12 tablas** iniciales. Nota: Ante de continuar debemos saber que no deberíamos utilizar el **prefijo por defecto** que nos ofrece WordPress que no es otro que «**wp_**«. Deberíamos cambiarlo por otro por motivos de seguridad.

Tras realizar una instalación de **WordPress** sin nada más, nos econtraremos estas tablas en nuestro **phpMyAdmin**:

Esta es la estructura de tablas que nos vamos a encontrar en una nueva instalación de **WordPress**.

A continuación vamos a resumir un poco cada una de las tablas para más adelante pasar a detallarlas un poco más cada una de ellas.

Tabla	Utilidad
wp_commentmeta	Esta tabla contiene contenido adicional sobre los comentarios.
wp_comments	Aquí están los datos de los comentarios.
wp_links	Esta tabla está obsoleta desde la versión 3.5 de WordPress. Aquí se almacenan o almacenaban los enlaces.
wp_options	Aquí se almacenan los datos de configuración de WordPress además de datos sobre configuración de plugins, temas, etc..
wp_postmeta	Información relacionada con los posts. Esta tabla está relacionada con la tabla wp_posts.
wp_posts	Se almacena todo el contenido de WordPress como entradas, páginas, ficheros, etc…
wp_termmeta	Metadados relacionados con las categorías.
wp_terms	Aquí se almacenan las categorías y etiquetas.

Tabla	Utilidad
wp_terms_relationships	En esta tabla se establece la relación entre los posts con las categorías, etiquetas, etc..
wp_term_taxonomy	Establece las características de las taxonomías como las entradas, páginas, etc…
wp_usermeta	Tabla con información adicional sobre los usuarios de la tabla wp_users.
wp_users	En esta tabla es donde se almacenan todos los usuarios con su información básica como contraseña, email, etc..

Relaciones entre las tablas de la base de datos
Para comprender un poco mejor la estructura de la base de datos de **WordPress** debemos entender cómo están relacionas entre ellas.
A continuación podemos ver un esquema de las relaciones entre las tablas obtenido del codex de WordPress.org.

En la imagen de la siguiente página podemos ver gráficamente las relaciones entre las tablas de la base de datos de **WordPress** en sus últimas versiones.

Detallaremos un poco más la relación de cada una de las tablas más adelante.

Tabla wp_commentmeta
Como ya hemos comentado anteriormente, en la tabla **wp_commentmeta** se almacenan los datos adicionales de o sobre los comentarios que se almacenan en una instalación de **WordPress**.

meta_id	Id principal de la tabla
comment_id	Id del comentario (tabla wp_comments)
meta_key	Clave del valor
meta_value	Meta o valor del campo

Destacaremos que el cambio **comment_id** hace referencia al id de la tabla **wp_comments** por lo tanto queda claro que ambas están relacinonada a través de este campo.

Para obtener datos de la tabla **wp_commentmeta** podemos utilizar la función get_comment_meta() que nos devolverá el campo meta de un comentario.

```
metacomentario = get_comment_meta( $comment->comment_ID, 'mimeta', true );
```

Tabla wp_comments
Esta es la tabla donde se almacenan o guardan los **comentarios** que se han realizado en nuestro **WordPress**. Tanto los comentarios que se han aprobado como los que no.

comment_ID	Id del comentario
comment_post_Id	Id del post (tabla wp_posts)
comment_author	Autor del comentario
comment_author_email	Email del autor del comentario
comment_author_url	Url del autor del comentario
comment_author_IP	Ip del autor del comentario
comment_date	Fecha y hora del comentario
comment_date_gmt	Fecha y hora del comentario (gmt)
comment_content	Contenido del comentario
comment_karma	Meta o valor del campo
comment_approved	Aprobación del comentario (0, 1 o spam)
comment_agent	Browser, sistema operativo, etc..
comment_type	Tipo (pingback | trackback)
comment_parent	Comentario padre del actual
user_id	id del usuario si está registrado

Para obtener datos de la tabla **wp_comments** podemos utilizar la función get_comments() que nos devolverá el campo meta de un comentario.

```
<?php
  // Mostrar por pantalla los autores de los comentarios del post con id 15

  $commentarios = get_comments('post_id=15');
  foreach($comments as $comment) :
```

```
        echo($comment->comment_author);
  endforeach;
?>
```

Tabla wp_options
Como ya hemos comentado al principio de esta guía en esta tabla se guardarán y gestionaran las **opciones de** nuestra instalación de **WordPress**.

Esta tabla es algo especial puesto que, en principio, no tiene ninguna relación con ninguna otra tabla de la base de datos de **WordPress**.

option_id	Id de la opción
option_nam	Nombre de la opción
option_value	Valor de la opción
autoload	Si la opción se carga automaticament

Para obtener datos de la tabla wp_options:
// Obtener el email del administrador

```
$admin_email = get_option( 'admin_email' );
```

Tabla wp_postmeta
Cada meta data de un post se almacena en la tabla **wp_postmeta**.
Algunos plugins también pueden utilizar esta tabla para guardar su propia información.

meta_id	Id de la información (meta)
post_id	Id del post asociado
meta_key	Clave del meta
meta_value	Valor del meta (información)

Para obtener datos de esta tabla:

// Obtener la información adicional o meta del post actual a través de su ID

```
$meta = get_post_meta( get_the_ID() );
```

Tabla wp_posts
Aunque todas las tablas de la base de datos son muy importantes, la tabla **wp_posts** es por lo menos una de las más imporntes sino la más importante.

Debemos tener en cuenta que aquí, en la tabla **wp_posts** se almacena la información de los posts de **WordPress** lo que incluye entradas, páginas, etc…

ID	Id de la información (meta)
post_author	Autor del post
post_date	Fecha del post
post_date_gmt	Fecha del post (gmt)
post_content	Contenido del post
post_title	Título del post
post_excerpt	Extracto del post
post_status	Estado del post (publicado, borrador, etc..)
comment_status	Estado comentarios (abiertos?)
ping_status	Estado de los pings (abiertos?)
post_password	Contraseña del post
post_name	Nombre del post
to_ping	Url a la que enviar pingback
pinged	Url a la que se ha enviado pingback
post_modified	Fecha y hora de modificarción del post
post_modified_gmt	Fecha y hora de modificarción del post (gmt)
post_content_filtered	Usado por algunos plugins de cache para guardar uan versión en caché
guid	Url permanente al post
menu_order	Número de orden en el menu
post_type	Tipo de post (post, page…)
post_mime_type	El Mime type de los ficheros subidos
comment_count	Número de comentarios del post

Para obtener datos de esta tabla:

// Obtener los últimos 15 posts

```
$args = array(
  'numberposts' => 10
);

$latest_posts = get_posts( $args );
```

Tabla wp_termmeta
En este tabla se almacenan los metadatos relacionados con las categorías.

meta_id	Id de la información (meta)
term_id	Id del término
meta_key	Clave del meta
meta_value	Valor del meta (información)

Para obtener datos de esta tabla:

```
// Obtener el valor del término 'color'

$color = get_term_meta( $term_id, 'color', true );
```

Tabla wp_terms
En la tabla wp_terms se guardan tanto las categorías como las etiquetas para los posts y páginas.

term_id	Id de término
name	Nombre del término
slug	Slug del término
term_group	Agrupación de terminos (alias)

Para obtener datos de esta tabla:

```
// Obtener todas la información de los términos de la categoría con ID 1

$term = get_term( 1 , 'category' );
echo $term->name;
```

Tabla wp_term_relationships
Los posts están relacionados con categorías y etiquetas por la tabla **wp_terms** y esta asociación es mantenida en la tabla **wp_term_relationships**.

object_id	Id de término
term_taxonomy_id	Nombre del término
term_order	Slug del término

Para obtener datos de esta tabla:
// Devuelve una lista de términos de taxonomía de productos que se aplican a $post

```
$product_terms = wp_get_object_terms( $post->ID,  'product' );

if ( ! empty( $product_terms ) ) {
    if ( ! is_wp_error( $product_terms ) ) {
        echo '<ul>';
            foreach( $product_terms as $term ) {
                    echo '<li><a href="' . esc_url( get_term_link( $term->slug, 'product' ) ) . '">' .
esc_html( $term->name ) . '</a></li>';
            }
        echo '</ul>';
    }
}
```

Tabla wp_term_taxonomy
Esta tabla almacena la información que describe la taxonomía como categorías, tags, cpt... para las entradas de la tabla wp_terms.

term_taxonomy_id	Id del témino de taxonomía
term_id	Id del término
taxonomy	Nombre de la taxonomía
description	Descripción
parent	Padre
count	Número

Tabla wp_usermeta
En la tabla wp_usermeta se almacenan datos adicionales de los usuarios

umeta_id	Id de la tabla
user_id	Id del usuario (tabla wp_users)
meta_key	Clave de la información o meta
meta_value	Valor de la información o meta

Obtener datos de la tabla wp_usermeta
// Obtener y mostrar toda la información meta de un usuario a través de su id

```
<?php
```

```
  $metauser = get_user_meta( 1 );
  print_r( $metauser );
?>
```

Tabla wp_users
Como su nombre indica, aquí se almacenan los **datos de los usuarios**: *nombre, email, etc*…
Esta tabla está directamente relacionada con la tabla **wp_usermeta**.

Id	Id del usuario
user_login	Login o username para acceder
user_pass	Contraseña del usuario (encriptada)
user_nicename	Nombre del usuario que se muestra
user_email	Email del usuario
user_url	Campo url del usuario
user_registered	Fecha y hora en que se registró el usuario
user_activation_key	Se usa para reestablecer las contraseñas
user_status	Se usaba antes
display_name	nombre del usuario elegido para mostrar (user_login, user_nicename…

Obtener datos de la tabla wp_users

```
// Obtener una lista de todos los usuarios con el nombre 'oscar'

$usuarios_oscar = get_users( array( 'search' => 'oscar' ) );
```

La clase $wpbd

WordPress define una clase llamada $wpdb, que contiene un conjunto de funciones utilizadas para interactuar con una base de datos. Su objetivo principal es proporcionar

una interfaz con la base de datos de WordPress, pero puede utilizarse para comunicarse con cualquier otra base de datos apropiada.

WordPress esta creado mediante PHP el cual permite correr las funciones mysql_query() y mysql_fetch_array(), pero no es recomendable usarlas directamente dentro de WordPress por las siguientes razones:

- La clase wpdb proporciona mejoras de seguridad para proteger tus consultas contra inyección o ataques de tipo SQL.

- En una posible migración o cambio de motor de base de datos por ejemplo de MySQL a PostgreSQL podrían no funcionar y romper todo.

WordPress proporciona una variable de objeto global $wpdb que es una instancia de la clase wpdb definida en /wp-includes/wp-db.php. Por defecto, $wpdb es la instancia para hablar con la base de datos de WordPress.

Para acceder con $wpdb su código PHP de WordPress, debe declarar $wpdb como una variable global utilizando la palabra global, o utilice el superglobal $GLOBALS de la siguiente forma:

```
// Declarando $wpdb como global global
$wpdb;
$results = $wpdb->get_results( 'SELECT * FROM wp_options WHERE option_id = 1', OBJECT );

// Utilizando el superglobal
$GLOBALS $results = $GLOBALS['wpdb']->get_results( 'SELECT * FROM wp_options WHERE option_id = 1', OBJECT );
```

Es importante no olvidar agregar la linea global $wpdb antes de comenzar a realizar nuestras funciones.

El objeto $wpdb no está limitado a las tablas predeterminadas creadas por WordPress; se puede utilizar para leer datos de cualquier tabla de la base de datos (como tablas personalizadas de

complementos agregados). Por ejemplo, para seleccionar información de una tabla personalizada llamada wp_form, se puede hacer lo siguiente:

```
$myrows = $wpdb->get_results( "SELECT id, name FROM wp_form" );
```

Con la clase wpdb podemos interactuar con la base de datos creando diferentes tipos de consultas pero hoy solo voy a explicar las 4 mas básicas las cuales corresponden al CRUD (CREAR, LEER, ACTUALIZAR Y BORRAR).

Pero antes de esto necesitamos tener creada una tabla dentro de nuestra base de datos, esto es muy utilizado en ocasiones cuando instalamos un plugin el cual necesita crear tablas de forma independiente y de esta forma se realiza. Todas estas pruebas las colocaremos en el archivo functions.php

```
/****************** Crear tabla con la clase wpdb *****************/
```

```
function crear_base() {

  global $wpdb;

    // Con esto creamos el nombre de la tabla y nos aseguramos que se cree con el mismo prefijo que ya tienen las otras tablas creadas (wp_form).
    $table_name = $wpdb->prefix . 'form';

    // Declaramos la tabla que se creará de la forma común.
    $sql = "CREATE TABLE $table_name (
      `id` int(11) NOT NULL AUTO_INCREMENT,
      `nombre` varchar(255) NOT NULL,
      `email` varchar(255) NOT NULL,
      UNIQUE KEY id (id)
    );";

    // upgrade contiene la función dbDelta la cuál revisará si existe la tabla.
    require_once( ABSPATH . 'wp-admin/includes/upgrade.php' );

    // Creamos la tabla
    dbDelta($sql);

}

// Ejecutamos nuestra funcion en WordPress
add_action('wp', 'crear_base');
```

Con esto hemos agregado una nueva tabla a toda nuestra base de datos con el nombre wp_form y ahora si procederemos a realizar operaciones en nuestra tabla.

Insertar

Como observamos anteriormente se creo una nueva tabla con tres campos (id, nombre, email) el **id** no lo tocaremos pues solo sirve para autoincrementarse cada vez que se agreguen mas datos, entonces vamos a insertar un **nombre** y un **email** para cada fila creada.

La manera de insertar una nueva fila se realiza de la siguiente forma:

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

table: Es de tipo (string) y corresponde al nombre de la tabla donde se insertara la fila o el campo.

data: Es la información a agregar y debe ir por medio de un (array).

format: Este es opcional de tipo (array|string) es una matriz de formatos que se asignarán a cada uno de los valores en los datos.

En nuestro caso quedaría de la siguiente forma:

```
/****************** Insertar *****************/
```

```
function insertar_wpdb(){

  global $wpdb;

  $wpdb->insert( 'wp_form',

    array(
      'nombre' => 'Andres Dev',
      'email' => 'hola@andres-dev.com'
    )
  );
}

// Ejecutamos nuestro funcion en WordPress
add_action('wp', ‘insertar_wpdb');
```

Notase que estoy utilizando el action hook wp el cual se ejecuta luego de haber actualizado el WordPress entonces para que puedan ejecutar cada una de las funciones es necesario solo actualizar WP.

Actualizar
Ahora actualizaremos la fila creada por medio de la siguiente estructura:

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

En nuestro caso quedaría de la siguiente forma:

```
/****************** Actualizar *****************/

function actualizar_wpdb(){
  global $wpdb;

  $wpdb->update( 'wp_form',
    // Datos que se remplazarán
    array(
      'nombre' => 'Bogotá',
      'email' => 'www.andres-dev.com'
    ),
    // Cuando el ID del campo es igual al número 1
    array( 'ID' => 1 )
  );
}

// Ejecutamos nuestro funcion en WordPress
add_action('wp', ‘actualizar_wpdb');
```

Leer
Ahora vamos a leer la información que está en la tabla wp_form y que sea mostrado en nuestro WP:

```
/****************** Leer *****************/
```

```
function leer_wpdb(){

  global $wpdb;

  $registros = $wpdb->get_results( "SELECT nombre, email FROM wp_form" );

   echo "Registro #1. Nombre: " . $registros[0]->nombre . ", Email: " . $registros[0]->email . "<br/>";

  }

// Ejecutamos nuestro funcion en WordPress
add_action('wp', 'leer_wpdb');
```

Lo que realizamos fue mostrar la consulta desde el header de nuestro WP.

Borrar

Y por último vamos a borrar nuestro registro por medio de la siguiente estructura que nos brinda la clase wpdb la cual es muy similar a la de insertar.

```
<?php $wpdb->delete( $table, $where, $where_format = null ); ?>
```

En nuestro caso quedaría de la siguiente forma:

```
/****************** Borrar *****************/

function borrar_wpdb(){

  global $wpdb;

  $wpdb->delete('wp_form', array('ID' => 1 ));

}

// Ejecutamos nuestro funcion en WordPress
add_action('wp', 'borrar_wpdb');
```

Funciones mas utilizadas

Las cuatro funciones vistas anteriormente hacen parte de las básicas utilizadas siempre que deseamos interactuar con una base de datos de cualquier tipo, por esta razón les dejaré a continuación un listado de otras que también puede utilizar con la clase wpdb.

- get_var(): Devuelve una variable (un único resultado, un valor,..)
- get_row(): Devuelve una fila de una tabla de la base de datos
- get_col(): Devuelve una columna de una tabla de la base de datos
- get_results(): Devuelve una lista de resultados (las mas común)
- insert(): Para realizar interacciones $wpdb->insert($table, $data, $format)

- replace(): Para actualizar tablas y reemplazos
- update(): Para actualizar una fila
- delete(): Para borrar una fila
- query(): Para cualquier consulta
- prepare(): Se usa para proteger de ataques de inyección de sql

Lo visto anteriormente es la forma correcta de trabajar con la base de datos de WordPress ya sea si estamos creando una plantilla, un plugin o cualquier otra cosa que se nos ocurra, no olvides comentar si te ha gusta este artículo y compartirlo con toda la comunidad

Por otro lado WordPress provee otras clases para trabajar con las tablas de Usuarios , Comentarios , WP_Query para los post de forma mas cómoda.

- WP_Query – clase para trabajar con post y paginas
- WP_Comment_Query – Clase para trabajar con comentarios
- WP_User_Query – Clase para trabajar con Usuarios

Por ejemplo si queremos obtener los usuarios de tipo suscriptor se podria hacer con una query (estariamos operando entre varias tablas) pero de esta manera podemos pasar argumentos y facilitar un poco el trabajo.

En este ejemplo vemos como devolver una lista de usuarios con rol suscriptor
$user_query = new WP_User_Query(array('role' => 'Subscriber'));

La clase wpdb nos ayuda para trabajar con nuestras propias consultas pero no quiere decir que sea el mejor metodo por ejemplo para crear una pagina, para ello nos podemos apoyar en funciones que ya trae WordPress para trabajar de forma mas sencilla que el hacer una insercion de toda la vida.

Ejemplo de crear un articulo con wp_insert_post()

```
$simple_area = wp_insert_post(
                        array(
                                'post_title'     => __( 'Simple Area', 'simple-wp' ),
                                'post_content'   => 'Esto es un ejemplo',
                                'post_status'    => 'publish',
                                'post_author'    => 1,
                                'post_type'      => 'page',
                                'comment_status' => 'closed'
                        )
                );
```

WP_Query

Si utilizas WordPress a menudo, sabrás que es difícil filtrar entradas por varios parámetros simultáneamente; tales como etiquetas, categorías o custom post types. Al menos es complicado con las herramientas que nos facilita WordPress por defecto.

Pero la evolución de WordPress es clara y sus desarrolladores han incluido herramientas para hacer este tipo de filtrados mucho más sencillos para nosotros; gracias a wp_query() podemos filtrar los contenidos de WordPress con cualquier criterio que se nos ocurra.

¿Qué es WP_Query?

WP_Query es una clase, una de las más importantes del núcleo de WordPress. Se encarga de determinar la consulta necesaria a la base de datos de acuerdo a la información que se está solicitando y, además, guarda este tipo de consultas frecuentes para optimizar la carga de la página.

Por otro lado, se encarga de permitirnos a los desarrolladores realizar consultas a la base de datos de una forma segura y sencilla, sin alterar el código de WordPress ni hacer “virguerías“.

Seguridad
Esta función nos permite, como dije antes, solicitar contenidos a la base de datos siguiendo una serie de criterios bastante complejos, pero lo hace protegiendo nuestro sistema de ataques SQL así como otro tipo de inseguridades que, de programarlo nosotros desde cero, habría más riesgo de dejar al descubierto.

Sencillez
Esta clase genera un objeto y nos evita tener que conocer la base de datos a fondo. Todo “habla por sí mismo”, no hace falta conocer el código interno de WordPress, ni las relaciones de la base de datos, ni nada por el estilo.

Olvídate también de hacer las consultas a la base de datos a mano; limítate a crear los argumentos (como veremos más adelante) que le pasarás a WP_Query y llama a la clase.

¿Cómo funciona WP_Query?

El loop de WordPress por defecto
Para entender el funcionamiento de WP_Query, vamos a ver cómo es un loop estándar:

```
<?php
  if(have_posts()) :
    while(have_posts()) :
      the_post();
?>
      <h1><?php the_title() ?></h1>
      <div class='post-content'><?php the_content() ?></div>
<?php
    endwhile;
  else :
```

```
?>
    Vaya, no hay entradas.
<?php
  endif;
?>
```

El mismo loop con WP_Query

```
<?php
  $args = array('cat' => 4);
  $category_posts = new WP_Query($args);

  if($category_posts->have_posts()) :
    while($category_posts->have_posts()) :
      $category_posts->the_post();
?>
      <h1><?php the_title() ?></h1>
      <div class='post-content'><?php the_content() ?></div>
<?php
    endwhile;
  else:
?>
    Vaya, no hay entradas.
<?php
  endif;
?>
```

Sí, es diferente, ¿verdad? Vamos a ver estas diferencias:

1. **Construir la consulta**
   En una página de categoría por defecto (por ejemplo category.php), WordPress sabe que ha de mostrar las entradas de esa categoría. Al hacerlo con WP_Query, debemos indicar la categoría nosotros mismos. Tranquilo, veremos esto mejor más adelante.

2. **Inicializar WP_Query y consultar**
   Cuando inicializamos WP_Query con el array de argumentos, esto colocará tanto las entradas correspondientes a dicha consulta como otras cosas.

3. **Creando el loop**
   Aquí puedes utilizar todas las funciones que usas normalmente, sólo tienes que tratarlo como un objeto en vez de llamarlas directamente; fíjate en los ejemplos siguientes:
   - $category_posts->have_posts() en vez de have_posts().
   - $category_posts->the_post() en vez de the_post().

4. **Trabaja como siempre**
   Una vez que has entendido el punto anterior (incluir $category_posts-> delante de las funciones de WordPress) puedes trabajar con total normalidad; como lo has hecho siempre.

No obstante, la variable global $post está también disponible. Esto significa que puedes si utilizas un loop personalizado como el del ejemplo anterior peligra que tengas problemas. Asegúrate de guardar el valor de esa variable global antes de utilizar un loop personalizado con WP_Query para después restaurarlo a su estado original:

```
<?php
  $temp_post = $post; // Guardamos el objeto de $post temporalmente
  $my_query = new WP_Query();
  while($my_query->have_posts()) {
    // Aquí va el loop
  }
  $post = $temp_post; // Restauramos el valor original del $post original.
?>
```

Valores por defecto
Como todo, WP_Query tiene unos valores por defecto que deberás tener en cuenta si lo utilizas:

- posts_per_page: valor por defecto que define el número de entradas a mostrar.
- post_type: siempre será “post” por defecto, hay que agregar o modificar si queremos utilizarlo con custom post types.
- post_status: mostrará los que tengan una visibilidad “pública” por defecto.

Puedes leer más sobre los parámetros en el Codex.

Consejos finales
WP_Query te permite hacer muchas cosas, modificar y utilizar WordPress como un framework de desarrollo dejando su facilidad de uso intacta, pero ¡cuidado! Puede traer muchos problemas si se utiliza mal.

Si hacemos que WP_Query realice consultas muy pesadas en un servidor compartido (de esos baratitos) es posible que nos comamos la memoria RAM del servidor y nos llamen la atención (o, peor aún, nos echen la web abajo para prevenir la caída total del servidor).

Mantén siempre presente que más consultas a la base de datos (o ficheros o consultas en general) conllevan un mayor consumo de recursos dentro del servidor.

Como consejo final, mantén presente siempre el tipo de consulta por defecto que realiza WordPress en una página, ¿para qué vamos a utilizar WP_Query para traer las últimas entradas del blog a la portada si es algo que ya está ahí por defecto?

WP_Comment_Query

WordPress nos da una serie de clases para trabajar con los elementos mas importantes, en este caso existe una clase solo para trabajar con comentarios y nos va a permitir realizar consultas de manera facil y comoda.
Y la clase en cuestion es WP_Comment_Query

```
<?php
```

```
$args = array(
  // args here
);
// Opcion 1
$comments_query = new WP_Comment_Query;
$comments = $comments_query->query( $args );

// Opcion 2 y la mas comoda
$comment_query = new WP_Comment_Query( $args );
?>
```

Aqui teneis dos formas de llamar a WP_Comment_Query, donde la opcion 2, se le pasa en su constructor un array con los argumentos o digamos lo que queremos buscar.

En los argumentos le pasaremos las opciones que queremos, por ejemplo los últimos comentarios, los comentarios spam ,… lo que mas nos interese.

Bucle

Una vez que tenemos los resultados de la búsqueda en nuestra variable $comments, lo primero que hacemos es preguntar si existen comentarios, siempre debemos hacer esta comprobación, no solo aquí sino en todos lados, asi comprobamos si viene a vacio y evitamos errores.
Si existen comentarios, entonces con el foreach podremos recorrerlo e ir mostrando los datos.

```
<?php
$args = array(
  // argumentos aqui
);
// hacemos la peticion a la base de datos
$comments = new WP_Comment_Query( $args );
// Comienza el bucle
if ( $comments ) {
	foreach ( $comments as $comment ) {
		echo '<p>' . $comment->comment_content . '</p>';
	}
} else {
	echo 'No comments found.';
}
?>
```

Como veis es sencillo y es el ejemplo que viene en la pagina oficial

Si queréis saber que información viene en los comentarios siempre podemos hacer lo siguiente y con var_dump pintaremos la informacion. Con la etiqueta pre lo pintaremos mejor y sera mas legible.

```
// Comienza el bucle
if ( $comments ) {
	foreach ( $comments as $comment ) {
```

```
                echo '<pre>';
            var_dump($comment);
            echo '</pre>';
        }
} else {
        echo 'No comments found.';
}
// Comienza el bucle
if ( $comments ) {
        foreach ( $comments as $comment ) {
                echo '<pre>';
            var_dump($comment);
            echo '</pre>';
        }
} else {
        echo 'No comments found.';
}
```

Argumentos en $args.

```
<?php
                $args = array(
                        'author_email' => '',
                        'author__in' => '',
                        'author__not_in' => '',
                        'include_unapproved' => '',
                        'fields' => '',
                        'ID' => '',
                        'comment__in' => '',
                        'comment__not_in' => '',
                        'karma' => '',
                        'number' => '',
                        'offset' => '',
                        'orderby' => '',
                        'order' => 'DESC',
                        'parent' => '',
                        'post_author__in' => '',
                        'post_author__not_in' => '',
                        'post_id' => 0,
                        'post__in' => '',
                        'post__not_in' => '',
                        'post_author' => '',
                        'post_name' => '',
                        'post_parent' => '',
                        'post_status' => '',
                        'post_type' => '',
                        'status' => 'all',
                        'type' => '',
                        'type__in' => '',
                        'type__not_in' => '',
                        'user_id' => '',
                        'search' => '',
```

```
                'count' => false,
                'meta_key' => '',
                'meta_value' => '',
                'meta_query' => '',
                'date_query' => null, // See WP_Date_Query
            );
?>
```

Como veis hay una gran cantidad de campos que podemos usar para filtar nuestras busquedas. Solo hay que poner los que necesitemos, no es necesario rellenarlos todos
Por ejemplo si quereis buscar los comentarios por el email del autor que lo ha creado podeis hacerlo asi:

```
$comment_query = new WP_Comment_Query( array( 'author_email' => 'xxxx') );
```

Wp_User_Query

WP_User_Query es una clase en WordPress que facilita la consulta y recuperación de usuarios del sitio. Permite a los desarrolladores realizar consultas avanzadas para obtener listas de usuarios basadas en diversos criterios, como roles, capacidades, metadatos de usuario y más.

Creación de una instancia de WP_User_Query:
Puedes crear una instancia de WP_User_Query pasando un conjunto de parámetros como argumentos. Estos parámetros especifican los criterios de búsqueda para la consulta.

```
$args = array(
    'role'         => 'editor',
    'number'       => 5,
    'orderby'      => 'user_registered',
    'order'        => 'DESC',
);

$user_query = new WP_User_Query($args);
```

Ejecución de la consulta:
Una vez que hayas configurado los parámetros de tu consulta, puedes ejecutarla para obtener los resultados.

```
$users = $user_query->get_results();

foreach ($users as $user) {
    // Acceder a la información del usuario
    $user_id = $user->ID;
    $user_login = $user->user_login;
    // ... más información del usuario
}
```

Parámetros comunes:
Algunos parámetros comunes que puedes utilizar en la configuración de WP_User_Query incluyen:

'role': Filtra por el rol del usuario.

'number': Número máximo de usuarios a recuperar.

'orderby' y 'order': Ordena los resultados por un campo específico en orden ascendente o descendente.

Consulta basada en metadatos de usuario:
También puedes realizar consultas basadas en metadatos de usuario utilizando el

parámetro 'meta_query'. Esto es útil cuando necesitas filtrar usuarios según datos personalizados almacenados en metadatos.

```
$args = array(
    'meta_query' => array(
        array(
            'key'     => 'custom_field',
            'value'   => 'desired_value',
            'compare' => '=',
        ),
    ),
);

$user_query = new WP_User_Query($args);
```

Menús de administración

Menús y submenús en el panel admin de WordPress, En muchos de nuestros plugins se hace necesario un menú de enlaces a las diferentes secciones que componen dicho plugin. Hoy os explicamos paso a paso cómo modificar menús y submenús del panel de administración de WordPress.

Como siempre, lo más aconsejable es mirar la documentación de WordPress. En el Codex encontraréis mucha más información acerca de este tema.

Los menús de WordPress son algo complejos de administrar y establecer tan funcionales como nos gustaría. Esto es algo que debemos aprender ya, antes de entrar en materia.
En primer lugar, las funciones básicas de gestión de menús:

- **add_menu_page**: Añade una página de menú superior (a la que se le pueden colgar «submenús»).
- **add_submenu_page**: Añade un submenú a un menú superior.
- **remove_menu_page**: Elimina una página de menú superior.

- **remove_submenu_page**: Elimina un submenú.

Por supuesto, estamos hablando de menús independientes. En el caso de que lo que queramos sea añadir menús a otros menús ya existentes, tenemos funciones específicas para ello: add_dashboard_page() (Añade a «Escritorio»), add_options_page() (Añade a «Ajustes»), etc. que podemos consultar en el Codex.

Normalmente, las páginas que vamos a usar en nuestro plugin serán llamadas a través de wp-admin/admin.php?page=<pagina>, ya que de esta forma se incluyen en el sistema WordPress. De otro modo, la página se ejecutaría sin usar la funcionalidad de WordPress.

Este es uno de los problemas que nos tocará solventar más adelante, a la hora de ver qué página depende de quién. Debemos tener en cuenta que los archivos .php referenciados deben existir para que los menús enlacen correctamente.

La especificación de add_menu_page es la siguiente:

```
<?php add_menu_page( $page_title, $menu_title,
$capability, $menu_slug, $function, $icon_url, $position ); ?>
```

- **page_title**: Título de la página en el navegador.
- **menu_title**: Texto que aparecerá en el menú.
- **capability**: permiso mínimo que debe tener un usuario para acceder a esta sección (¿Dudas sobre permisos? Consulta nuestra entrada sobre roles y permisos de hace unos días).
- **menu_slug**: cadena a la cual se dirige el enlace del menú, en caso de no existir valor para el siguiente parámetro (function).
- **function** (opcional): Función que se realizará al pulsar en el enlace. También puede ser una ruta a un archivo. De ser nulo, enlazará a la cadena escrita en «menu_slug«.
- **icon_url** (opcional): url del pequeño icono (20×20) de la sección.
- **position** (opcional): número que determina la posición en el menú. Si se establece una posición ya en uso, se sobreescribirá.

Para add_submenu_page tenemos la siguiente especificación:

```
<?php add_submenu_page( $parent_slug, $page_title,
$menu_title, $capability, $menu_slug, $function ); ?>
```

- **parent_slug**: aquí debemos escribir el menu_slug del menú superior del que cuelga este submenú.
- **page_title**: Título de la página en el navegador.
- **menu_title**: Texto que aparecerá en el menú.

- **capability**: permiso mínimo que debe tener un usuario para acceder a esta sección.
- **menu_slug**: cadena a la cual se dirige el enlace del menú, en caso de no existir valor para el siguiente parámetro (function).
    - **function**: Función que se realizará al pulsar en el enlace. También puede ser una ruta a un archivo. De ser nulo, enlazará a la cadena escrita en «menu_slug«.

Pongamos un ejemplo simple: vamos a crear un menú «Colores» y tres submenús «Rojo«, «Azul» y «Amarillo«.


```
<?php function my_custom_menu() {
        add_menu_page ( 'Colores', 'Colores', 'read', 'colores.php', '', '', 101);
        add_submenu_page( 'colores.php', 'Rojo', 'Rojo', 'read', 'rojo.php', '');
        add_submenu_page( 'colores.php', 'Azul', 'Azul', 'upload_files', 'azul.php', '');
        add_submenu_page( 'colores.php', 'Amarillo', 'Amarillo', 'read', 'amarillo.php', '');
}
add_action( 'admin_menu', 'my_custom_menu'); ?>
```

Es importante englobar estas funciones dentro de una función, que luego pasaremos con add_action en el momento en que WordPress ejecuta el hook ‘admin_menu‘.
Así, veríamos algo como esto en nuestro menú:
Como vemos, se nos ha repetido «Colores«. Es algo que el menú de WordPress hace por defecto si encuentra submenús.

Nuestras dos soluciones son:

1. Agregar un submenú justo después de agregar el menú, cuyo menu_slug sea el mismo que el parent_slug, y controlemos de esta forma el texto que aparece. Esto no elimina el enlace.

2. Eliminar el enlace con la función remove_submenu_page.

Así que ya que estamos, os presento las funciones de eliminar menus y submenús:

```
<?php remove_menu_page( $menu_slug ); ?>
```

Esta función elimina un menú superior. En nuestro caso, remove_menu_page(‘colores.php’); eliminaría todo el menú «Colores«.

Si queremos eliminar un submenú:

```
<?php remove_submenu_page( $menu_slug, $submenu_slug ); ?>
```

Para seguir con el ejemplo, remove_submenu_page(‘colores.php’, ‘rojo.php’); eliminaría sólo el submenú «Rojo«.

Por lo tanto, si queremos eliminar el submenú «Colores» que nos ha salido duplicado, bastaría con añadir a nuestra función:

```
remove_submenu_page( 'colores.php', 'colores.php' );
```

Ahora sí, nuestro menú queda como esperábamos:

El código final sería:

```
<?php
/*
Plugin Name: Colores
Plugin URI: https://www.codigonexo.com/
Description: Menu de prueba.
Version: 1.0
Author: Juan Viñas
Author URI: https://www.codigonexo.com/
*/
function my_custom_menu() {
	add_menu_page ( 'Colores', 'Colores', 'read', 'colores.php', '', '', 101);
	add_submenu_page( 'colores.php', 'Rojo', 'Rojo', 'read', 'rojo.php', '');
	add_submenu_page( 'colores.php', 'Azul', 'Azul', 'upload_files', 'azul.php', '');
	add_submenu_page( 'colores.php', 'Amarillo', 'Amarillo', 'read', 'amarillo.php', '');
	remove_submenu_page( 'colores.php','colores.php' );
}

add_action( 'admin_menu', 'my_custom_menu');
?>
```

Deberías tener en cuenta que, de esta forma, al pulsar en uno de los enlaces (si el archivo existe), nuestro menú lo redigirá a través de /wp-admin/admin.php?page=<menu_slug>, sobre todo a la hora de trabajar y navegar con rutas. Estás en la carpeta wp-admin, no en la carpeta de tu plugin.

Otros menús en la administración

Pues bien, como ya hemos visto esto, me gustaría explicar cómo crear un sub-menú de un menú existente. Pero no de un menú como el que creamos en el tutorial anterior puesto que eso ya lo vimos. Lo único que tienes que hacer es utilizar la función add_submenu_page() e indicar el menú padre del que descenderá tu sub-menú.

Ahora lo que vamos a hacer es crear un sub-menú bajo los menús que ya existen en WordPress por defecto ya que cada uno tiene su función específica. No te preocupes porque es muy sencillo.

Para dejar claro de a qué me estoy refiriendo cuando digo crear un sub-menú de un menú existente, me refiero a crear sub-menú de los menús que vienen por defecto en la instalación de WordPress. Aprovecho la lista que pongo a continuación para indicar qué función corresponde a cada uno de los menús:

- Escritorio –> add_dashboard_page()
- Entradas –> add_posts_page()
- Medios –> add_media_page()
- Páginas –> add_pages_page()
- Comentarios –> add_comments_page()
- Apariencia –> add_theme_page()
- Plugins –> add_plugins_page()
- Usuarios –> add_users_page()
- Herramientas –> add_management_page()
- Ajustes –> add_options_page()

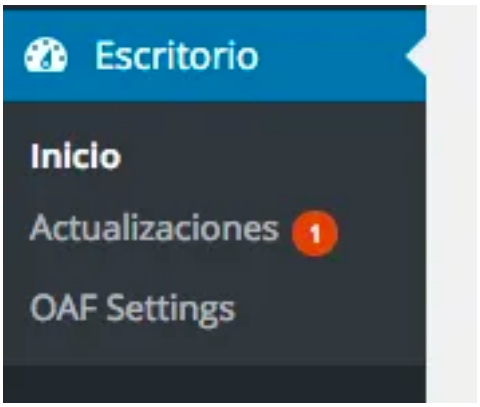
A continuación crearé el directorio y el fichero para este plugin. En este caso le voy a llamar «OAF Create Sub-menu«. Y tanto el directorio como el fichero php tendrán el nombre de: «oaf-create-submenu«.

El código fuente inicial para el fichero «oaf-create-submenu.php» es el siguiente:

```
<?php
/*

Plugin Name: OAF Create Sub-menu
```

```
Plugin URI: https://oscarabadfolgueira.com/programacion-de-plugins-wordpress-crear-un-sub-menu-en-un-menu-existente
Description: This plugin creates an admin sub-menu in existing menu
Version: 1.0
Author: Oscar Abad Folgueira
Author URI: https://oscarabadfolgueira.com
License: GPLv2
License URL: http://www.gnu.org/licenses/gpl-2.0.html

*/

add_action( 'admin_menu', 'oaf_create_submenu');




function oaf_create_submenu() {

	add_dashboard_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

}

function oaf_create_submenu_function () {

}

?>
```

En primer lugar observamos que hemos creado el add_action en el hook «admin_menu»



com en el ejemplo del anterior tutorial. Hasta aquí bien porque también le estamos diciendo que ejecute la función «oaf_create_submenu«.

Ahora bien, la función que utilizamos en «add_dashboard_page()» que nos permitirá añadir elementos de sub-menú en el menú «Escritorio» del panel de administración de WordPress.

Ya veis que en este caso, a diferencia de la función que ya habíamos utilizado de «add_submenu_page()«, no debemos indicarle el menú «padre» puesto que ya está en la función en sí misma.

Y ¿Qué es lo que hace esto?

Pues no hace otra cosa que crear un plugin. Que al activarlo nos crea un sub-menú que desciende directamente del menú «**Escritorio**» como se puede ser a continuación:
Interesante, ¿verdad?

Supongo que ya estaréis pensando la cantidad de cosas que se pueden hacer con esto. Y eso que no es nada del otro mundo porque todavía no hemos creado las páginas de opciones.

Para utilizar el resto de funciones que he indicado anteriormente, os pongo el código fuente y algunas capturas con el plugin completo:

```
<?php
/*

Plugin Name: OAF Create Sub-menu
Plugin URI: yo.com
Description: This plugin creates an admin sub-menu in existing menu
Version: 1.0
Author: Yo
Author URI: yo.com
License: GPLv2
License URL: http://www.gnu.org/licenses/gpl-2.0.html

*/

add_action( 'admin_menu', 'oaf_create_submenu');

function oaf_create_submenu() {

	add_dashboard_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

	add_posts_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

	add_media_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

	add_pages_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

	add_comments_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );
```

```
	add_theme_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

	add_users_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

	add_management_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

	add_options_page ( 'OAF Settings', 'OAF Settings', 'manage_options', 'oaf_create_submenu_plugin', 'oaf_create_submenu_function' );

}

function oaf_create_submenu_function () {

}
?>
```

Widgets

Un Widget es una pequeña porción de código que nos permitirá extender alguna funcionalidad de nuestra web. Como mínimo se podrán aplicar en el header y el sidebar, pero también podemos implementarlos en otros lugares si el tema que usemos nos lo permite, como en el aside.

También hay que tener en cuenta que muchos Widgets son los mismos elementos usados como bloques en Gutenberg. Esto quiere decir que también podremos aplicarlos dentro de nuestras páginas y posts.

Muchas veces nos gustaría mostrar cierta información en nuestra barra lateral (o Sidebar) de WordPress, y recurrimos a añadir un Widget de texto plano con HTML. Esto está bien, es simple y cómodo, pero sólo para texto estático. En el momento en que queremos algo más dinámico, no podemos (pues no podemos insertar código PHP). Es por ello que lo ideal es crear un Widget para WordPress con el contenido que queramos.
Vamos a empezar con un ejemplo simple, y lo iremos complicando poco a poco.

Creando el Widget
Un Widget se compone de dos partes:
- Una función que instancia al Widget.
- Una clase para nuestro Widget, que extiende al objeto WP_Widget.

Lo ideal es crear un plugin para nuestro Widget, pero debo decir que sí, que también podéis añadir la función que instancia el Widget en el archivo functions.php. Nosotros crearemos un nuevo plugin, cuyo código es simplemente este:

```
<?php
/*
Plugin Name: Mi primer Widget
Plugin URI: https://www.codigonexo.com/
```

```
Description: Crea un Widget para añadir a cualquier Sidebar.
Version: 1.0
Author: Codigonexo
Author URI: https://www.codigonexo.com/
*/

/*
 Función que instancia el Widget
 */
function mpw_create_widget(){
    include_once(plugin_dir_path( __FILE__ ).'/includes/widget.php');
    register_widget('mpw_widget');
}

add_action('widgets_init','mpw_create_widget');

?>
```

Vale, ¿Esto qué hace?
Tras declarar la información del plugin, tenemos la función mpw_create_widget (mpw viene de «Mi Primer Widget», el nombre de la función podéis cambiarlo como queráis, claro). Esta función hace dos cosas:

1. Incluir el archivo widget.php, que crearemos en la carpeta wp-content/<nombre_de_tu_plugin>/includes/widget.php. Esta es mi forma de tener los archivos organizados, quizá vosotros prefiráis no usar la carpeta includes, o usar /includes/widget/widget.php. En nuestro caso, lo haremos así.

2. Registrar el widget en el sistema con la función «register_widget». Como parámetro, recibe el nombre que le daremos a la clase del Widget que vamos a crear, en este caso, mpw_widget.

La clase de nuestro Widget
Si no hemos creado el archivo widget.php, es el momento de hacerlo. Nuestra clase debe extender a la clase WP_Widget, y debe tener cuatro funciones básicas:

```
<?php
class mpw_widget extends WP_Widget {

    function mpw_widget(){
        // Constructor del Widget.
    }

    function widget($args,$instance){
        // Contenido del Widget que se mostrará en la Sidebar
    }

    function update($new_instance, $old_instance){
        // Función de guardado de opciones
    }
```

```
    function form($instance){
        // Formulario de opciones del Widget, que aparece cuando añadimos el Widget a una Sidebar
    }
}
?>
```

Esta es una plantilla válida para cualquier Widget que os planteéis crear. Ahora, detengámonos a explicarlas una a una.

La primera función debe llamarse igual que la clase. Es la función constructora, la que se instancia una vez que el Widget es llamado. En esta función, lo básico que debemos hacer es llamar al constructor de WP_Widget con las opciones del Widget:

```
    function mpw_widget(){
        $widget_ops = array('classname' => 'mpw_widget', 'description' => "Descripción de Mi primer Widget" );
        $this->WP_Widget('mpw_widget', "Mi primer Widget", $widget_ops);
    }
```

Esto hará que nuestro Widget sea visible en la zona de Widgets de WordPress.

La siguiente función es «widget». Esta función es la que genera el contenido que se muestra en la zona del Widget, lo que verán tus usuarios en el Front End. Luego lo complicaremos, pero de momento, veamos un ejemplo simple:

```
function widget($args,$instance){
    echo $before_widget;
    ?>
    <aside id='mpw_widget' class='widget mpw_widget'>
```

MI PRIMER WIDGET

¡Este es mi primer Widget!

```
        <h3 class='widget-title'>Mi Primer Widget</h3>
        <p>¡Este es mi primer Widget!</p>
    </aside>
    <?php
    echo $after_widget;
}
```

Activando el Widget
Llegados a este punto, supongo que habéis activado el plugin desde el gestor de plugins de WordPress. Hecho esto, vamos a Apariencia > Widgets, y encontraremos nuestro Widget en la lista de Widgets disponibles. Tan solo tenemos que arrastrarlo a la Sidebar que mejor nos convenga.

Si no ves tu Widget en la lista, algo has pasado por alto. Revisa las instrucciones hasta aquí.

Finalmente, vamos al Front End y vemos el resultado.
Complicando el Widget con configuración interna
Si nuestro Widget está hecho sólo para mostrar información concreta, y no requiere de configuración interna, habríamos acabado aquí, no serían necesarias más funciones. En el caso de que requiera configuración, entonces debemos crear las funciones update y form.

```
function update($new_instance, $old_instance){
    $instance = $old_instance;
    $instance["mpw_texto"] = strip_tags($new_instance["mpw_texto"]);
    // Repetimos esto para tantos campos como tengamos en el formulario.
    return $instance;
}
```

La función update se encarga de guardar en la base de datos la configuración establecida para el Widget. Suele seguir una estructura similar a la que vemos siempre, cambiando los parámetros de los campos.

La función form es la que muestra el formulario de configuración del Widget en el Back End de WordPress. Por ejemplo, vamos a mostrar un texto:

```
function form($instance){
```


```
        ?>
         <p>
              <label for="<?php echo $this->get_field_id('mpw_texto'); ?>">Texto del Widget</
label>
                   <input class="widefat" id="<?php echo $this->get_field_id('mpw_texto'); ?>"
name="<?php echo $this->get_field_name('mpw_texto'); ?>" type="text" value="<?php
echo esc_attr($instance["mpw_texto"]); ?>" />
         </p>
         <?php
    }
```

Las funciones get_field_id y get_field_name las usamos para que el guardado del Widget sea correcto y coherente en cuanto a parámetros.

Ahora debemos adaptar nuestra función widget para que acepte y muestre este texto:

```
    function widget($args,$instance){
        echo $before_widget;
        ?>
        <aside id='mpw_widget' class='widget myp_widget'>
            <h3 class='widget-title'>Mi Primer Widget</h3>
            <p><?=$instance["mpw_texto"]?></p>
        </aside>
        <?php
        echo $after_widget;
```

```
}
```

Y podemos comprobar que, efectivamente, aparece el texto que hemos colocado:

Veamos otro widget distinto que nos aporte una nueva funcionalidad, como puede ser un contador de visitas. Para esto tendremos que extender la clase WP_Widget como hacemos con la estructura siguiente:

```
<?php
class widget_webheroe extends WP_Widget {
	//Función constructora de la clase WP_Widget donde declaramos el identificador, el nombre y la descripción
	function __construct() {
		parent::__construct(

		);
	}
	public function widget( $args, $instance ) {
	  //Aquí se aplica el código interesante de tu Widget
	}
	public function form( $instance ) {
		//Aquí se coloca el código del formulario visible en el back end
	}
	public function update( $new_instance, $old_instance ) {
		//Aquí se registran las variables que se facilitan en el back end
	}
}
```

Creamos el método Constructor

Este método es el que nos permitirá añadir el identificador del Widget, el nombre que aparecerá en el listado de Widgets y la descripción del mismo.

Debemos estar atentos de usar parent::__construct, ya que otras declaraciones están obsoletas. Como parent::WP_Widget(), $this->WP_Widget() o WP_Widget::WP_Widget()

```
<?php
//Función constructora de la clase WP_Widget donde declaramos el identificador, el nombre y la descripción
	function __construct() {
		parent::__construct(
			//Identificador
			'widget_webheroe',
			//Nombre
			'Contador Webheroe',
			//Descripción
			array( 'description' => 'Contador de Webheroe' )
		);
	}
```

Creamos la función Widget

Aquí es donde tenemos el alma del Widget. Nuestro mecanismo estará incluido en este método. Los argumentos $args[] sirven para incluir algún código antes o después del widget. Además, también contamos con el filtro widget_title que nos permitirá manipularlo sin necesidad de entrar en la página de Widgets o en el código del mismo.

Como podéis ver he intentado comentar todo el código de una forma clara para que se entienda fácilmente. He incluido un condicional para decidir si incrementar el valor del contador o no dependiendo de si es una página o un post.

Puede parecer que sería más lógico usar !is_admin en su lugar, pero debido al funcionamiento de esta clase seguiría incrementándose en varias unidades cada vez que carguemos el panel de control de WordPress. También he usado un bucle while para asegurar el incremento de valor cuando sea necesario.

```
<?php
public function widget( $args, $instance ) {

		$title = apply_filters( 'widget_title', $instance['title'] );

		echo $args['before_widget'];

		//Si el título existe
		if (!empty($title)){
			echo $args['before_title'] . $title . $args['after_title'];
		}

		/*El código de tu Widget empieza aquí*/

		$option = get_option($this->option_name);//Obtenemos el array con las opciones de este Widget

		$counter = $option[$this->number]['counter'];//Obtenemos el valor que nos interesa

		//Verificamos si es una página o un post
		if(is_page() || is_single()){
			$sum_counter = intval($counter) + 1;//Sumamos uno al valor

			$option[$this->number]['counter'] = $sum_counter;//Sobreescribimos el valor de la variable

			//Creamos un bucle que se repita hasta que verifique que el valor se ha actualizado
			$updated = false;
			while($updated === false){
				$updated = update_option($this->option_name, $option);
			}

		}else{
			//Si no es una página ni un post, el valor no se incrementará
			$sum_counter = $counter;
		}
```

```
		//Mostramos el Widget en el front end
		?>

	<p style="color:#fff">Ha recibido <span style="font-size:50px;font-
weight:800;color:#ada"><?php echo $sum_counter; ?></span> visitas</p>
		<?php

		/*Aquí termina el código de tu Widget*/

		echo $args['after_widget'];

	}
```

Registrar nuestro Widget

Ahora tendremos que hacer uso del hook register_widget. Como su nombre indica, es el gancho que nos permitirá registrar la clase anterior y hacer que funcione y aparezca en el listado de Widgets.

Debemos tener cuidado de no usar este hook hasta haber terminado con la creación de nuestra clase, ya que podría dar un error fatal:

```
<?php
add_action( 'widgets_init', function(){
	//Registro de la clase que hace funcionar el Widget
	register_widget( 'widget_webheroe' );
});
```

Ahora ya podremos visitar nuestra sección de Widgets en el panel de control Apariencia/ Widgets. Y lo localizaremos dentro del grupo «Widgets».

Creamos el método Form

Aquí debemos presentar el código que será mostrado en el Back End. Es decir, en la página de edición de Widgets

```
<?php
public function form( $instance ) {
		//Comprobar si hemos colocado un título al Widget. Si no, se colocará el
título por defecto
		if(isset($instance[ 'title' ])){
			$title = $instance[ 'title' ];
		}else{
			$title = 'Widget Contador de WebHeroe';
		}



		//Aquí se crea el formulario que aparecerá para poder cambiar el título
		?>
		<p>
			<label for="<?php echo $this->get_field_id( '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
        }
```

Rematamos con el método Update
En este método nos limitaremos a crear el título y colocar el contador a 0.

```
<?php
public function update( $new_instance, $old_instance ) {
                $instance = array();
                $instance['title'] = ( ! empty( $new_instance['title'] ) ) ?
strip_tags( $new_instance['title'] ) : '';
                $instance['counter'] = '0';
                return $instance;
        }
```

Shortcodes

Un shortcode es un código abreviado que se puede insertar directamente en el editor de texto de una entrada o página.

Este shortcode muestra un contenido específico, que puede ser texto, imágenes, código HTML, JavaScript, funciones PHP o cualquier tipo de código web que queramos. Los shortcodes en WordPress se representan así:

[nombre_del_shortcode]

nombre_del_shortcode es el nombre específico del shortcode. No puede haber 2 shortcodes con el mismo nombre.

Además, estos shortcodes pueden aceptar parámetros. Los parámetros son valores que se utilizan dentro del contenido para personalizarlo. Estos parámetros se escriben así:

[products limit="4" columns="4" on_sale="true"]

El shortcode se llama products y lo personalizas con 3 parámetros: el número de productos a mostrar, el número de columnas y si son productos rebajados.

Crear un Shortcode en WordPress
WordPress dispone de una función para añadir shortcodes llamada add_shortcode. El modo de crear un shortcode es con una función que incluya el código a mostrar y después añadirla como shortcode:

```
function nombre_de_la_funcion() {
        return '<p>Contenido a mostrar</p>';
}

function register_shortcodes(){
```

```
	add_shortcode('nombre_del_shortcode', 'nombre_de_la_funcion');
}

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

El shortcode a insertar en la web sería: [nombre_del_shortcode]

Este es el funcionamiento básico para crear el shortcode. Como ves en el ejemplo he utilizado return para mostrar el contenido.

Si vas a mostrar contenido HTML, JavaScript, etc., el modo más cómodo de hacerlo es recoger todo el código en una variable que es la que devolverá el mismo. Para hacerlo puedes usar las funciones ob_start y ob_get_contents. El modo de utilizarlas sería así:

```
function nombre_de_la_funcion() {
ob_start(); ?>
// Contenido html/javascript
<?php
$output = ob_get_contents();
	ob_end_clean();
	return $output;
}
```

Si en lugar de estas funciones y return utilizas echo para mostrar el contenido, el contenido puede no mostrarse correctamente, sin respetar los contenedores div.

Veamos a ver un ejemplo práctico que te servirá para después añadir los parámetros al shortcode.

Voy a mostrar las últimas entradas del blog y quiero que sean 3 columnas (usaré Bootstrap para representarlo).

```
function cvw_home_posts() {
	$args = array(
		'post_type' => 'post',
		'posts_per_page' => 3
	);
	$the_query = new WP_Query( $args );

	ob_start();
	if ( $the_query->have_posts() ) :
	?>
	<div class="content-home-blog row">
		<?php while ( $the_query->have_posts() ) : $the_query->the_post(); ?>
			<div class="col-md-4 col-sm-4 col-xs-12">
				<a href="<?php the_permalink() ?>" rel="bookmark" title="">
					<?php the_post_thumbnail('large'); ?>
				</a>
				<h5><?php the_title(); ?></h5>
				<div class="entry-content">
					<?php the_excerpt(); ?>
```

```
<a class="link-post" href="<?php the_permalink() ?
```

¿Por qué lo usamos Lorem Ipsum?

LEER

¿Qué es Lorem Ipsum?

Lorem Ipsum es simplemente el texto de relleno de las imprentas y archivos de texto. Lorem Ipsum ha sido el texto de relleno estándar de las industrias desde el año 1500, cuando un impresor (N. del T. persona que se dedica a la imprenta) desconocido usó una galería de textos y los mezcló de tal [...]

LEER

En un lugar de la Mancha

En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lantejas los viernes, algún palomino [...]

LEER

```
>">LEER</a>
                                    </div>
                                </div>
                        <?php
                        endwhile; ?>
        </div>
<?php
            wp_reset_postdata();
    endif;

    $result = ob_get_contents();
    ob_end_clean();
    return $result;
}
```

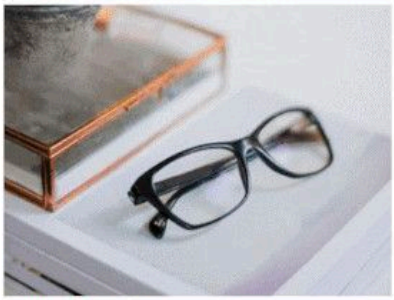
En la función he creado una query para obtener todas las entradas. Para ello le he pasado como argumentos que el tipo de post sea post y que el número de resultados sea 3. Después he creado el bucle y añadido el HTML necesario para maquetarlo. He puesto la imagen destacada, el título y el extracto.

Con este shortcode como base ahora se pueden plantear parámetros que lo personalicen más. Por ejemplo, puedes hacer que muestre entradas de una categoría concreta para hacer secciones en la página de inicio.

Para añadir parámetros lo primero que tienes que hacer en la función es añadirle una variable que los recoja:

function cvw_home_posts($atts) {

Después dentro de la función pasar los valores por defecto en caso de que no se haya introducido algún parámetro. En mi caso voy a poner un parámetro cat y su valor por defecto será todas las categorías:



Título de la entrada

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

LEER

¿Qué es Lorem Ipsum?

Lorem Ipsum es simplemente el texto de relleno de las imprentas y archivos de texto. Lorem Ipsum ha sido el texto de relleno estándar de las industrias desde el año 1500, cuando un impresor (N. del T. persona que se dedica a la imprenta) desconocido usó una galería de textos y los mezcló de tal [...]

LEER

En un lugar de la Mancha

En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lantejas los viernes, algún palomino [...]

LEER

```
$atts = shortcode_atts( array(
        'cat' => '',
    ), $atts );
```

Y por último le paso a los argumentos de la query para la categoría a mostrar:

```
$args = array(
          'post_type' => 'post',
          'posts_per_page' => 3,
          'category_name' => $atts['cat']
    );
```

A la hora de montar el shortcode añado el parámetro cat y pongo el nombre de la categoría:

[cvw_home_posts cat=»nuevos»]

Y ahora me muestra las entradas sólo de esa categoría:

Conclusiones

Los shortcodes en WordPress te permiten crear contenido personalizado de manera fácil y rápida para enriquecer las Entradas y Páginas de tu sitio web. Los shortcodes son

pequeñas cadenas de texto que se convierten en *funciones* o *bloques de contenido*, y son especialmente útiles para la creación de botones, formularios de contacto, galerías de imágenes e incluso para mostrar información en widgets.

Estos códigos cortos también pueden ser reutilizados en varias partes de un sitio web, lo que significa que una vez que se ha creado un shortcode, puede ser mostrado en diferentes páginas de la web. Además, los shortcodes facilitan la actualización del contenido, ya que los cambios se aplican en todos los lugares donde se ha utilizado el shortcode. El uso de shortcodes es una forma eficiente y fácil de crear contenido personalizado en tu web con WordPress.

Permisos y roles

Podemos definir los roles de WordPress como las capacidades o permisos que tiene cada perfil de usuario en esta aplicación.

En otras palabras, los roles de WordPress te permiten definir y controlar a qué funcionalidades tienen acceso y a cuáles no los usuarios que creas en tu aplicación.

En una misma instalación de WordPress pueden coexistir perfiles de usuario con diferentes roles. Es decir, puedes crear perfiles de usuario que pueden escribir, pero no publicar entradas en tu blog; y otros tienen capacidad para administrar etiquetas o categorías, instalar o desinstalar plugins, etc.

Por eso es tan importante que existan diferentes roles de WordPress. ¿Te imaginas que cada persona que tenga acceso al panel de administración de tu web pudiese hacer lo que quiere con ella? Sería un auténtico desastre.

Roles de WordPress: Qué son y cómo gestionarlos

Loa roles y permisos de WordPress es un concepto que muchas veces pasamos por alto, pero es de mucha importancia para la gestión de un blog o de una página web, especialmente cuando son varios los usuarios que trabajan en ella.

Si tienes una página web y eres tú el único que se encarga del diseño, de las publicaciones o de su seguridad, quizá sea algo que nunca vayas a necesitar. Al fin y al cabo, con el usuario *admin* que WordPress crea por defecto tienes todos los privilegios y permisos para realizar cualquier tipo de cambio en tu página.

Pero, ¿qué pasa si tu web es gestionada por dos o más personas? ¿Y si quieres concederle acceso a WordPress a tu agencia de *copywriting*, pero no quieres que tengan acceso a todas las opciones y funcionalidades de tu web? Precisamente para esto se crearon los roles de WordPress.

¿Qué son los roles de WordPress?

Podemos definir los roles de WordPress como las capacidades o permisos que tiene cada perfil de usuario en esta aplicación. En otras palabras, los roles de WordPress te permiten definir y controlar a qué funcionalidades tienen acceso y a cuáles no los usuarios que creas en tu aplicación.

En una misma instalación de WordPress pueden coexistir perfiles de usuario con diferentes roles. Es decir, puedes crear perfiles de usuario que pueden escribir, pero no publicar entradas en tu blog; y otros tienen capacidad para administrar etiquetas o categorías, instalar o desinstalar plugins, etc.

Si has instalado WordPress en tu plan de hosting, habrás visto que, por defecto, se crea un usuario admin. Este rol de WordPress es el que tiene todos los privilegios y permisos de la web, es decir, la capacidad de hacer los cambios que quiera. Desde instalar un plugin hasta borrar la aplicación por completo.

Por eso es tan importante que existan diferentes roles de WordPress. ¿Te imaginas que cada persona que tenga acceso al panel de administración de tu web pudiese hacer lo que quiere con ella? Sería un auténtico desastre.

Tipos de roles o perfiles de usuario en WordPress
Cuando quieres crear un nuevo usuario en tu web verás que, además del todopoderoso admin, tu aplicación muestra por defecto los siguientes roles de WordPress:

- **Administrador**. Este rol de usuario no tiene ninguna restricción o limitación. Es el único rol de WordPress que tiene todas las capacidades; crear o eliminar perfiles de usuario, actualizar el core o plugins y plantillas de tu aplicación, editar temas, modificar o personalizar el escritorio de WordPress, administrar las publicaciones y la categorización del contenido, subir archivos, crear nuevas páginas o eliminar las que ya están publicadas… Todos los cambios que te puedas imaginar pueden ser ejecutados con el rol de administrador.

- **Editor**. El rol de editor tiene menos capacidades que el de administrador. No tiene control sobre el core, plugins o plantillas que se utilizan en tu web, pero sí puede gestionar comentarios, categorías, etiquetas o controlar la publicación de contenido. Puede crear, modificar o borrar entradas y páginas que hayan sido creadas por él mismo o por cualquier otro usuario.

- **Autor**. Un usuario con el rol de autor puede subir archivos y crear, editar, publicar o eliminar sus propias entradas. Aunque sea un rol con muchas menos capacidades que los dos anteriores, es recomendable que solo concedas este permiso a una persona de confianza. El hecho de poder publicar su propio contenido, hace que su actividad en WordPress esté completamente expuesta a tus visitas.

- **Colaborador**. Tiene muchas más limitaciones que el rol de autor, pero es mucho más seguro si trabajas con personas ajenas tu empresa u organización. Tan solo pueden subir su propio contenido, pero la publicación del mismo tan solo puede realizarla el rol de administrador o editor.

- **Suscriptor**. Es un rol que no tiene mucha utilidad en WordPress. Tiene únicamente la capacidad de leer entradas del blog o páginas ya publicadas (es decir, lo mismo que cualquier otro usuario que no esté registrado). La única ventaja de este rol de usuario es que puede responder o dejar comentarios en cualquier post de tu blog sin necesidad de crear un nuevo registro.

Cómo gestionar los roles de WordPress desde el backend

Cada vez que creas un nuevo usuario en WordPress, puedes asignarle el rol que consideres en ese momento. Si llegado el momento necesitas modificarlo y concederle más o menos capacidades, desde la sección «Usuarios» del menú de WordPress puedes hacerlo sin problema. A continuación te explico cómo crear uno y cómo modificarlo.

Crear un nuevo rol de usuario en WordPress

En el menú de WordPress localiza el apartado «Usuarios» y pulsa sobre «Añadir nuevo». Se abrirá una nueva pantalla en la que puedes introducir todos los datos relativos a este perfil de usuario: nombre de usuario, correo, nombre y apellidos, contraseña y el rol de WordPress que quieras asignarle.

Modificar el rol de un usuario en WordPress

Si lo que necesitas es modificar el rol de un usuario que ya tienes creado en WordPress, puedes hacerlo sin problema a través del administrador de WordPress.

En el apartado «Usuarios» localiza el que deseas modificar y pulsa en «Editar».Además de modificar el rol de WordPress, también puedes realizar cambios en la biografía, la imagen de perfil, los datos de acceso, el correo electrónico, etc.

La clase WP_Roles

Capacidades / Rol de usuario	Administrador	Editor	Autor	Colaborador	Suscriptor
Administrar plugins	✓				
Administrar temas o plantillas	✓				
Gestionar perfiles de usuario	✓				
Actualizar WordPress, temas o plugins	✓				
Editar y personalizar el escritorio	✓				
Moderar comentarios	✓	✓			
Gestionar categorías	✓	✓			
Gestionar etiquetas	✓	✓			
Editar y eliminar cualquier página o entrada	✓	✓			
Subir archivos a la biblioteca de medios	✓	✓	✓		
Publicar sus propias entradas	✓	✓	✓		
Borrar sus propias entradas	✓	✓	✓		
Crear o editar sus propias entradas	✓	✓	✓	✓	
Leer entradas y páginas ya publicadas	✓	✓	✓	✓	✓

WordPress implementa roles y capacidades con la API de Roles de Usuario, la mayoría de las cuales se basan en la clase central WP_Roles. Puedes encontrar su fuente en el archivo wp-includes/class-wp-roles.php.

Si miras en la base de datos, encontrarás que los roles están dentro de un arreglo con sus nombres de roles definidos. La clave rolename almacena el nombre del rol de usuario como un valor de la clave del name y todas las capacidades en una matriz separada como un valor de la clave de la capabilty.

```
array (
    'rolename' => array (
        'name' => 'rolename',
        'capabilities' => array()
```

```
    )
)
```

La WP_Roles class define muchos métodos. Puedes llamarlos en cualquier parte del código para interactuar con la API de Roles de Usuario.
Nota: WordPress incluye otra clase principal llamada WP_Role (note el singular «Role»). Se usa para extender la API de Roles de Usuario.
Cuando descompongas el valor de la clave de wp_user_roles, se verá algo como esto:

```
array (
  'administrator' =>
  array (
    'name' => 'Administrator',
    'capabilities' =>
    array (
      'switch_themes' => true,
      'edit_themes' => true,
      'activate_plugins' => true,
      // [...rest of the lines cut off for brevity...]
    ),
  ),
  'editor' =>
  array (
    'name' => 'Editor',
    'capabilities' =>
    array (
      'moderate_comments' => true,
      'manage_categories' => true,
      'manage_links' => true,
      // [...rest of the lines cut off for brevity...]
    ),
  ),
  'author' =>
  array (
    'name' => 'Author',
    'capabilities' =>
    array (
      'upload_files' => true,
      'edit_posts' => true,
      'edit_published_posts' => true,
      // [...rest of the lines cut off for brevity...]
    ),
  ),
  'contributor' =>
  array (
    'name' => 'Contributor',
    'capabilities' =>
    array (
      'edit_posts' => true,
      'read' => true,
      // [...rest of the lines cut off for brevity...]
    ),
```

```
  ),
  'subscriber' =>
  array (
    'name' => 'Subscriber',
    'capabilities' =>
    array (
      'read' => true,
      'level_0' => true,
    ),
  ),
)
```

Es un conjunto multidimensional en el que a cada función se le asigna un nombre y se le otorga un conjunto de capacidades. De manera similar, WordPress almacena las capacidades basadas en el usuario en la tabla wp_usermeta con el nombre de la meta-llave wp_capabilities.

Nota: El prefijo wp_ puede ser diferente en su configuración. Depende del valor de la variable global $table_prefix en el archivo wp-config.php de tu sitio.

Capacidad especial: Carga no filtrada
La carga no filtrada es una capacidad especial que no está asignada a ningún rol de usuario por defecto, incluyendo Administrador o Superadministrador. Permite a un usuario subir archivos con cualquier extensión (por ejemplo, SVG o PSD), no sólo los que están en la lista blanca de WordPress.

Nota: Puedes obtener una lista de los tipos de mímica y las extensiones de archivo soportadas por WordPress usando la función wp_get_mime_types().

Para habilitar esta capacidad, necesitas agregar el siguiente fragmento de código a tu archivo wp-config.php. Define la constante antes de la línea que te pide que dejes de editar.

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

Después de definir esta constante, puedes darle a cualquier usuario en un sitio único de WordPress la capacidad de subir sin filtrar. Sin embargo, en una instalación multisitio sólo un Super Admin puede tener esta capacidad.

Por ejemplo, si quieres asignar la capacidad de carga_no filtrada a un Editor, puedes añadir el siguiente código en cualquier parte de tu código de WordPress (lo ideal es que lo ejecutes sólo en la activación del tema/plugin):

```
<?php

  $role = get_role( 'editor' );
  $role->add_cap( 'unfiltered_upload' );

?>
```

Más adelante en este artículo hablaremos de cómo agregar o personalizar las capacidades de todos los roles de usuario o de usuarios específicos.

Capacidades primitivas vs. Metacapacidades
Hay principalmente dos tipos de capacidades en WordPress:

- Capacidades primitivas: Estas capacidades se conceden a funciones particulares. Los usuarios con estos roles heredan las capacidades primitivas automáticamente.

- Meta Capacidades: Estas capacidades no se conceden a ningún papel por defecto. WordPress comprueba un determinado objeto en su código y base de datos, como un post, una página, un usuario o cualquier taxonomía, y si la lógica se comprueba, «mapea» una meta capacidad a una o más capacidades primitivas.

Por ejemplo, WordPress otorga a los autores la capacidad de edit_posts para sus propios posts para que puedan editarlos. Sin embargo, esta capacidad no les permite editar los mensajes de otros usuarios. Aquí es donde las meta-capacidades ayudan.

WordPress utiliza la función map_meta_cap() para devolver un conjunto de capacidades primitivas ligadas a un objeto específico. Luego las compara con el objeto del usuario para comprobar si el usuario puede editar el mensaje.

Algunos otros ejemplos de meta capacidades son read_post, delete_post, remove_user, y read_post. Los veremos con más detalle en la sección de capacidades personalizadas más abajo.

¿Cómo personalizar los roles de usuario de WordPress existentes?
Puedes añadir capacidades a las funciones de usuario existentes para aumentar su nivel de acceso. Por ejemplo, puedes dar a los editores el poder de administrar los plugins. O quizás quieras que los Colaboradores moderen los comentarios de sus propios posts. Aprendamos a hacer eso.

¿Cómo añadir capacidades a un rol de usuario?

Puedes añadir una capacidad a un rol de usuario o a cualquier usuario específico usando la función add_cap() WordPress. Usaré un plugin personalizado llamado Personalizar Rol de Usuario para mostrar cómo usar esta función para darle al rol de Editor el poder de administrar plugins.

```
<?php

/*
Plugin Name:  Customize User Role
Version:  1.0
Description:  Demonstrating how to customize WordPress User Roles.
Author:  Yp
Author URI:  https://www.yo-com/
License: GPLv2 or later
License URI: https://www.gnu.org/licenses/gpl-2.0.html
Text Domain:  customize-user-role
*/
```

WordPress recomienda ejecutar esta función en la activación de plugins o temas, ya que los ajustes que añade se almacenan en la base de datos en la tabla wp_options bajo el

campo wp_user_roles. Es ineficiente ejecutar esta función cada vez que se carga cualquier página, ya que las tablas de la base de datos seguirán siendo sobrescritas en cada carga de página.

Como estoy usando un plugin, usaré la función register_activation_hook() para engancharme a la acción que se ejecuta al activar un plugin. Hay muchas maneras de hacer esto, pero estoy usando una implementación robusta basada en clases para asegurarme de que no haya conflictos.

```
// this code runs only during plugin activation and never again
function sal_customize_user_role() {
    require_once plugin_dir_path( __FILE__ ).'includes/class-sal-customize-user-role.php';
    Sal_Customize_User_Role::activate();
}

register_activation_hook( __FILE__, 'sal_customize_user_role' );
```

El código anterior se ejecuta sólo una vez durante la activación del plugin. La función enganchada sal_customize_user_role hace referencia a una clase personalizada llamada Sal_Customize_User_Role.

He definido esta clase en un archivo separado llamado class-sal-customize-user-role.php y la he colocado dentro de la carpeta raíz de mi plugin dentro de una subcarpeta llamada includes, pero puedes nombrarla como quieras.

```
<?php

class Sal_Customize_User_Role {
    public static function activate() {
        // get the Editor role's object from WP_Role class
        $editor = get_role( 'editor' );

        // a list of plugin-related capabilities to add to the Editor role
        $caps = array(
                'install_plugins',
                'activate_plugins',
                'edit_plugins',
                'delete_plugins'
        );

        // add all the capabilities by looping through them
        foreach ( $caps as $cap ) {
            $editor->add_cap( $cap );
        }
    }
}
```

Aquí hay una explicación detallada del código anterior:

- Empieza definiendo la clase y su función a la que has hecho referencia en el archivo principal del plugin.

- La función get_role('editor') recupera el objeto de rol Editor de la clase core WP_Role y lo asigna a la variable $editor.

- La gestión de los plugins requiere cuatro capacidades: install_plugins, activate_plugins, edit_plugins y delete_plugins. Pero la función add_cap() acepta sólo un parámetro. Por lo tanto, necesitamos incluir todas las capacidades dentro de un array. Definir la matriz $caps para contener todas estas capacidades. Si sólo se añade una capacidad, entonces no hay necesidad de definir un array.

- La función add_cap($cap) añade todas las capacidades definidas en la matriz $caps haciendo un bucle a través de todas ellas usando la función foreach () PHP.

Guarda todos tus archivos de plugin y luego activa el plugin desde tu panel de administración. Ahora entremos en el dashboard del Editor para ver los cambios.

Después de agregar las capacidades relacionadas con los plugins a su rol de usuario, los editores pueden ver el menú de plugins en su menú de administración.

Puedes comprobar las capacidades asignadas a cada rol de usuario viendo el valor de la clave wp_user_roles almacenado en la tabla wp_options de la base de datos de tu sitio WordPress.

Aquí están las capacidades que encontré asignadas al rol de Editor:

```
'editor' =>
  array (
    'name' => 'Editor',
    'capabilities' =>
    array (
      'moderate_comments' => true,
      'manage_categories' => true,
      // [...lines cut off for brevity...]
      'install_plugins' => true,
      'activate_plugins' => true,
      'edit_plugins' => true,
    ),
  ),
```

Fíjense en las últimas tres líneas que dan a los editores la capacidad de manejar los plugins.

Si deseas eliminar estas capacidades, puede engancharse a la función register_deactivation_hook() y utilizar la función remove_cap() para eliminar las capacidades de desactivación de los plugins, tal y como añadimos estas capacidades en la activación de los plugins.

Ahora que has aprendido a añadir capacidades a un rol de usuario, es hora de aprender a quitar capacidades de un rol de usuario.

Nota: También puedes engancharte a la acción after_switch_theme para disparar este código durante la activación del tema (y/o tema infantil). Aquí, debes incluir el código en el archivo functions.php de tu tema o tema secundario (recomendado).

¿Cómo eliminar las capacidades de un rol de usuario?

A veces, puede que quieras eliminar una capacidad de un rol de usuario. Puede ejecutar la función remove_cap() para eliminar una capacidad de un rol o de un usuario específico. Por ejemplo, es una excelente idea eliminar la capacidad delete_published_posts del rol de usuario Author.

¡Acabemos con esto!

Voy a crear un nuevo plugin personalizado llamado Personalizar el rol de autor para empezar. Al igual que antes, ejecutaré este código sólo una vez enganchándolo a la función register_activation_hook().

```
<?php

/*
Plugin Name:  Customize Author Role
Version:  1.0
Description:  Demonstrating how to customize WordPress Author Role.
Author:  Yo
Author URI:  https://www.yo.com/
License: GPLv2 or later
License URI: https://www.gnu.org/licenses/gpl-2.0.html
Text Domain:  customize-author-role
*/

// this code runs only during plugin activation and never again
function sal_customize_author_role() {
        require_once  plugin_dir_path(  __FILE__  ).'includes/class-sal-customize-author-role.php';
   Sal_Customize_Author_Role::activate();
}

register_activation_hook( __FILE__, 'sal_customize_author_role' );
```

A continuación, definiré la clase Sal_Customize_Author_Role dentro del archivo class-sal-customize-author-role.php. He referenciado ambos recursos en el archivo principal del plugin de arriba.

```
<?php
class Sal_Customize_Author_Role {
   public static function activate() {
      // get the Editor role's object from WP_Role class
      $author = get_role( 'author' );
```

```
        // remove the capability to delete published posts from an Author role
        $author->remove_cap( 'delete_published_posts' );
    }
}
```

La función remove_cap('delete_published_posts') eliminará la capacidad de eliminar los posts publicados del rol de Autor.

Es hora de guardar todos los archivos del plugin y luego activar el plugin. Ahora, entra en el panel de control del autor y mira los cambios.

La opción Basura ya no está disponible para los artículos publicados por los autores. Sin embargo, aún pueden borrar sus publicaciones no publicadas que tengan el estado de Borrador o Pendiente.

Si quieres deshabilitar incluso esta capacidad, entonces también necesitas eliminar la capacidad de delete_posts del rol de Autor.

Adición o eliminación de capacidades para usuarios específicos

Si quieres añadir capacidades a un usuario específico, en lugar de un rol de usuario completo, entonces puedes usar la función de clase WP_User::add_cap() para añadir la capacidad.

```
// get the user object by their ID
$user = new WP_User( $user_id );

// add the capability to the specific user
$user->add_cap( $cap );
```

Puedes usar la función get_user_by() para recuperar el ID de cualquier usuario usando su correo electrónico, nombre de usuario de inicio de sesión o slug.

Del mismo modo, se pueden eliminar las capacidades de un usuario específico utilizando la función de clase WP_User::remove_cap().

```
// get the user object by their ID
$user = new WP_User( $user_id );

// add the capability to the specific user
$user->add_cap( $cap );
```

Como antes, ejecuta estas funciones sólo en la activación de plugins o temas para mantener tu código optimizado.

Nota: Tanto add_cap() como remove_cap() son métodos objeto de la clase WP_Role. No puedes llamarlos directamente en tu código. Necesitas acceder a ellos usando la función get_role() o la variable global $wp_roles.

Duplicar un rol de usuario

Puede crear un nuevo rol de usuario clonando todas las capacidades de un rol de usuario existente. Así es como puedes hacerlo:

```
add_role( 'clone', 'Clone', get_role( 'administrator' )->capabilities );
```

En el ejemplo anterior, estoy creando un nuevo rol llamado Clon con las mismas capacidades de un Administrador. Ejecutando este código en el tema o la activación del plugin se asegurará de que el rol clonado se añada sólo una vez.

¿Cómo crear roles de usuario personalizados en WordPress?

Las capacidades de edición de los roles de usuario predeterminados es una forma rápida de personalizarlos. Pero si buscas editar muchas capacidades de un rol, entonces es una buena idea crear un nuevo rol de usuario personalizado en conjunto. De esta manera puedes establecer las capacidades exactas que quieres para cada rol en tu sitio.

Para crear un rol de usuario personalizado, necesitas usar la función add_role(). Acepta tres parámetros.

```
add_role(  $role, $display_name, $capabilities );
```

Los dos primeros parámetros deben ser cadenas (y necesarios) para que la función funcione. Definen el nombre de la nueva función personalizada y el nombre para mostrar, respectivamente. El último parámetro es opcional y debería ser un array. Se puede utilizar para asignar todas las capacidades de la nueva función.

Vamos a crear un rol de usuario personalizado llamado Administrador de la Comunidad que puede moderar los comentarios y editar los mensajes en todo el sitio. Así es como puedes hacerlo:

```
<?php

/*
Plugin Name:  Add Community Manager Role
Version:  1.0
Description:  Add a Custom User Role called 'Community Manager'
Author:  Yo
Author URI:  https://www.yo.com/
License: GPLv2 or later
License URI: https://www.gnu.org/licenses/gpl-2.0.html
Text Domain:  add-community-manager-role
*/

// this code will run only once on plugin activation and never again
function add_community_manager_role() {
   add_role(
       'community_manager',
       __('Community Manager', 'add-community-manager-role'),
       array(
```

```
            'read' => true,
            'moderate_comments' => true,
            'edit_posts' => true,
            'edit_other_posts' => true,
            'edit_published_posts' => true
        )
    );
}

register_activation_hook( __FILE__, 'add_community_manager_role' );
```

Como antes, la función add_role() se ejecuta una sola vez al activar el plugin y nunca más. Guarda el archivo y activa el plugin en tu panel de administración. Ahora deberías poder asignar el rol de Administrador de la Comunidad tanto a los usuarios nuevos como a los existentes.

También puedes verificar las capacidades asignadas a este nuevo rol verificando el valor del campo wp_user_roles bajo la tabla wp_options dentro de tu base de datos. Esto es lo que encontré en la base de datos de mi sitio:

```
array (
  'administrator' =>
    // [...]
  'editor' =>
    // [...]
  'author' =>
    // [...]
  'contributor' =>
    // [...]
  'subscriber' =>
    // [...]
  'community_manager' =>
  array (
    'name' => 'Community Manager',
    'capabilities' =>
    array (
      'read' => true,
      'moderate_comments' => true,
      'edit_posts' => true,
      'edit_other_posts' => true,
      'edit_published_posts' => true,
    ),
  ),
)
```

Al final aparece el nuevo papel que acabamos de añadir con todas sus capacidades. Puedes editar este rol más adelante añadiendo o quitando capacidades.
Probando un nuevo papel de usuario

Antes de asignar el nuevo rol de usuario a cualquier usuario real, es esencial probar si funciona como se pretende. Aquí tienes una lista de control que puedes seguir para probarlo:

1. Crear una cuenta de usuario de prueba y asignarle el nuevo rol de usuario.

2. Entra con el usuario de prueba y asegúrate de que todas sus capacidades funcionan como se pretende. Por ejemplo, si le ha concedido la capacidad de editar los mensajes publicados, entonces vaya a cualquier mensaje y compruebe si puede editarlo. Cuantas más capacidades le hayas asignado al rol, más tiempo pasarás probándolas todas.

3. A continuación, intenta visitar cualquier enlace de administración de nivel superior directamente en tu navegador. Probé esto visitando la pantalla de configuración de WordPress directamente, y como era de esperar, WordPress no me dejó entrar.
   El mensaje de «acceso denegado» mostrado por WordPress

4. Borra el usuario de la prueba después de que hayas terminado de probarla.

¡Eso es más o menos así! Ahora puedes asignar el nuevo rol a los usuarios de tu sitio.

¿Cómo eliminar los roles de usuario de WordPress?

Puedes eliminar cualquier rol de usuario de WordPress usando la función remove_role( ). Acepta un solo argumento, que es el nombre del rol. Por ejemplo, puedes eliminar el rol de colaborador ejecutando el siguiente código en cualquier lugar de tu sitio:

```
remove_role( 'contributor' );
```

A diferencia de la función add_role() que seguirá actualizando la base de datos si no se ejecuta en un plugin o en la activación de un tema, la función remove_role() se ejecuta sólo si el rol existe. Dado que cualquier rol pasado como argumento es eliminado la primera vez que se ejecuta, no hay que preocuparse de dónde se ejecuta esta función.

Sin embargo, para evitar futuros conflictos, elimina el código después de que el papel sea eliminado de la base de datos.

Creación de capacidades personalizadas en WordPress

La edición de los roles de usuario existentes y la creación de nuevos roles personalizados utilizando las capacidades incorporadas de WordPress es suficiente para la mayoría de los casos de uso, pero es posible que desees definir nuevas capacidades para las características introducidas por su código personalizado (utilizando un plugin o un tema).
Luego puedes utilizar esas capacidades personalizadas para definir nuevos papeles o añadirlos a los ya existentes.

Por ejemplo, WooCommerce añade capacidades y funciones adicionales junto con sus amplias características de comercio electrónico. Algunas de las capacidades que añade son:

- Permitir la administración de la configuración de WooCommerce
- Crear y editar productos
- Ver los informes de WooCommerce

Usando estas capacidades, añade dos nuevos roles de usuario: Cliente y Gerente de la tienda.

El rol de Cliente es casi similar al rol de Suscriptor, excepto que los usuarios con el rol de Cliente pueden editar la información de su cuenta y ver los pedidos actuales/anteriores. El rol de Administrador de tienda incluye todas las capacidades de un Editor, además de que también se les conceden todas las capacidades de WooCommerce.

Otros plugins que introducen capacidades y/o roles personalizados incluyen The Events Calendar, Visual Portfolio, WPML y WP ERP.

Si te profundizas en la documentación de todos estos plugins, se observará que vinculan casi todas sus capacidades personalizadas a los tipos de postales personalizados definidos por ellos. En el caso de WooCommerce, son los tipos de postales

personalizadas de Productos y Pedidos, mientras que en otros son Eventos, Portafolios, Traducciones y Clientes respectivamente.

Aprendamos a crear capacidades personalizadas atadas a un tipo de poste personalizado. Primero, configura un plugin y registra el tipo de correo personalizado que quieras. En mi ejemplo, estoy registrando un nuevo tipo de mensaje personalizado llamado «Historias«.

```
<?php

/*
Plugin Name:    Custom Post Type and Capabilities
Version:        1.0
Description:    Register a custom post type and define custom capabilities tied into it.
Author:         Yo
Author URI:     https://www.yo.com/
License:        GPLv2 or later
License URI:    https://www.gnu.org/licenses/gpl-2.0.html
Text Domain:    custom-post-type-capabilities
*/

// register a custom post type, in this case it's called "story" //
function cpt_story_init() {
    $labels = array(
        'name'                  => _x( 'Stories', 'custom-post-type-capabilities' ),
        'singular_name'         => _x( 'Story', 'custom-post-type-capabilities' ),
            'menu_name'                     => _x( 'Stories', 'Admin Menu text', 'custom-post-type-
capabilities' ),
```

```
        'name_admin_bar'        => _x( 'Story', 'Add New on Toolbar', 'custom-post-type-capabilities' ),
        'add_new'               => __( 'Add New', 'custom-post-type-capabilities' ),
        'add_new_item'          => __( 'Add New Story', 'custom-post-type-capabilities' ),
        'new_item'              => __( 'New Story', 'custom-post-type-capabilities' ),
        'edit_item'             => __( 'Edit Story', 'custom-post-type-capabilities' ),
        'view_item'             => __( 'View Story', 'custom-post-type-capabilities' ),
        'all_items'             => __( 'All Stories', 'custom-post-type-capabilities' ),
        'search_items'          => __( 'Search Stories', 'custom-post-type-capabilities' ),
        'parent_item_colon'     => __( 'Parent Stories:', 'custom-post-type-capabilities' ),
        'not_found'             => __( 'No stories found.', 'custom-post-type-capabilities' ),
        'not_found_in_trash'    => __( 'No stories found in Trash.', 'custom-post-type-capabilities' ),
        'featured_image'        => _x( 'Story Cover Image', 'custom-post-type-capabilities' ),
        'set_featured_image'    => _x( 'Set cover image', 'custom-post-type-capabilities' ),
        'remove_featured_image' => _x( 'Remove cover image', 'custom-post-type-capabilities' ),
        'use_featured_image'    => _x( 'Use as cover image', 'custom-post-type-capabilities' ),
        'archives'              => _x( 'Story archives', 'custom-post-type-capabilities' ),
        'insert_into_item'      => _x( 'Insert into story', 'custom-post-type-capabilities' ),
        'uploaded_to_this_item' => _x( 'Uploaded to this story', 'custom-post-type-capabilities' ),
        'filter_items_list'     => _x( 'Filter stories list', 'custom-post-type-capabilities' ),
        'items_list_navigation' => _x( 'Stories list navigation', 'custom-post-type-capabilities' ),
        'items_list'            => _x( 'Stories list', 'custom-post-type-capabilities' ),
    );

    $args = array(
        'labels'             => $labels,
        'public'             => true,
        'menu_icon'          => 'dashicons-book',
        'publicly_queryable' => true,
        'show_ui'            => true,
        'show_in_menu'       => true,
        'query_var'          => true,
        'rewrite'            => array( 'slug' => 'story' ),
        'capability_type'    => array ( 'story', 'stories' ),
        'map_meta_cap'       => true,
        'has_archive'        => true,
        'hierarchical'       => false,
        'menu_position'      => 6,
        'supports'           => array( 'title', 'editor', 'author', 'thumbnail', 'excerpt', 'comments' ),
        'show_in_rest'       => true,
    );

    register_post_type( 'story', $args );
}

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

Aquí hay un desglose del guión anterior:

- Utiliza la función register_post_type() para registrar su tipo de puesto personalizado. Puedes engancharte a la acción init para ejecutar esta función.

- La función register_post_type() acepta dos argumentos. El primero es el nombre del tipo de puesto personalizado y el segundo es un array que contiene todos los argumentos para registrar el tipo de puesto.

- La variable $args contiene todos los argumentos que pasarás a la función register_post_type(). Uno de sus argumentos (‘labels’) es en sí mismo un array definido separadamente como la variable $label.

- Fíjate en el argumento ‘capability_type’ => ‘post’. Es el tipo de capacidad predeterminado utilizado por WordPress para construir las capacidades de lectura, edición y borrado para el tipo de post personalizado.

- Para crear sus capacidades personalizadas, debes reemplazar el valor del argumento capability_type con el nombre preferido de sus capacidades personalizadas. Acepta una cadena o un array como argumento. El array es útil si el plural de su capacidad personalizada no sigue la sintaxis estándar del sufijo (por ejemplo, libro/libros vs. cuento/cuentos).

- También puedes usar el argumento de capabilities para nombrar las nuevas capacidades de forma diferente a lo que hace WordPress automáticamente.

- Debes asignar tus capacidades personalizadas a las capacidades primitivas de WordPress. Establezca el argumento map_meta_cap en true para que WordPress sepa que necesita mapear las capacidades personalizadas como se sugiere.

A continuación, debes añadir las capacidades personalizadas a los roles que quieres dar acceso al tipo de publicación personalizada de Stories. Para este ejemplo, estoy otorgando la capacidad a los roles de Administrador y Editor.

```
// add the custom capabilities to the desired user roles
$roles = array( 'editor','administrator' );

foreach( $roles as $the_role ) {

   $role = get_role($the_role);

        $role->add_cap( 'read' );
        $role->add_cap( 'read_story');
        $role->add_cap( 'read_private_stories' );
        $role->add_cap( 'edit_story' );
        $role->add_cap( 'edit_stories' );
        $role->add_cap( 'edit_others_stories' );
        $role->add_cap( 'edit_published_stories' );
        $role->add_cap( 'publish_stories' );
        $role->add_cap( 'delete_others_stories' );
        $role->add_cap( 'delete_private_stories' );
        $role->add_cap( 'delete_published_stories' );
}
```

Guarda el archivo y luego activa el plugin. Ahora deberías ver el enlace y el panel de Historias en tu panel de Administrador o Editor.

Si compruebas las capacidades disponibles en tu sitio, también verás todas las capacidades relacionadas con las historias que hemos añadido. Aquí, estoy usando el plugin View Admin As para comprobar las capacidades.