

WordPress

Introducción al Desarrollo de Plugins

Andrés Villarreal

Tabla de contenido

- Acerca de este libro 1.1
- Parte 1: Introducción 1.2
  - A quién está dirigido este libro 1.2.1
  - Presentación y objetivos de este libro 1.2.2
  - Qué es un plugin y para qué sirve 1.2.3
  - Cómo crear un plugin básico 1.2.4
- Parte 2: Plugin API 1.3
  - Programación orientada a eventos 1.3.1
  - Acciones 1.3.2
  - Filtros 1.3.3
  - Uso de clases y objetos en eventos 1.3.4
  - Remover eventos del registro 1.3.5
- Parte 3: Personalización de contenidos 1.4
  - Internacionalización y localización 1.4.1
  - Post Types predefinidos 1.4.2
  - Custom Fields y Meta Boxes 1.4.3
  - Custom Post Types 1.4.4
  - Taxonomías 1.4.5
- Parte 4: Personalización de opciones 1.5
  - Menús de administración 1.5.1
  - Options API 1.5.2
  - Settings API 1.5.3
- Parte 5: Prácticas recomendadas 1.6
  - Estilos y scripts en menús de administración 1.6.1
  - Pluggable functions 1.6.2
  - By-Passing 1.6.3
  - Must-Use Plugins 1.6.4
  - Recomendaciones finales 1.6.5

WordPress: Introducción al Desarrollo de Plugins

Por Andrés Villarreal

Este libro presenta una serie de herramientas básicas para la construcción de plugins propios para WordPress, combinadas con sugerencias acerca de optimización y buenas prácticas de desarrollo. Consiste en capítulos cortos que intentan explicar de la manera más directa posible un conjunto de conceptos importantes a la hora de desarrollar plugins.

Los contenidos presentados están destinados a un perfil bastante específico: desarrolladores web con experiencia en WordPress, con conocimientos de instalación y configuración de la plataforma (incluyendo plugins y themes), nociones fuertes de HTML y CSS, y nociones al menos básicas de PHP y programación en general. Como punto de partida se asume que el lector ha desarrollado previamente sitios web con WordPress, ya sea para clientes o proyectos personales.

Si bien el libro apunta a desarrolladores con poca experiencia en las partes más técnicas de WordPress, está estructurado y pensado para que programadores más experimentados también puedan utilizar sus contenidos como punto de referencia para sus propios desarrollos.

Propósito

Existen muchos libros acerca de desarrollo de plugins para WordPress, varios de ellos excelentes. Sin embargo, aquellos que considero buenos fueron escritos en inglés. Muy pocos se tradujeron al español, e inicialmente no estuvieron pensados para el público de habla hispana. Salvo por unas pocas fuentes confiables, se cuenta con muy poco material de consulta en español dedicado a WordPress, y solo una parte ínfima consiste en material técnico. Con eso en mente, el presente libro es al menos un intento inicial de resolver ese problema.

Estado actual: Beta

El libro está en etapa de revisión. Los ejemplos de código son 100% funcionales, aunque pueden ser modificados en el futuro para una mejor comprensión. Tengo planeado agregar imágenes ilustrativas para algunos conceptos, y realizar algunas modificaciones de estilo para facilitar la lectura.

Licencias

Los contenidos textuales de este libro están publicados bajo la licencia BY-NC-ND 4.0 de Creative Commons, la cual indica que no pueden ser reproducidos sin autorización expresa del autor, ni utilizados con fines comerciales. Tampoco se permite la redistribución de trabajos derivados del original.

Los ejemplos de código PHP, JavaScript, HTML y CSS presentados se publican bajo la licencia GPL2, por lo cual pueden ser reutilizados, redistribuidos y modificados libremente, cualquiera sea su fin.

Parte 1: Introducción

Contenidos:

- A quién está dirigido este libro
- Presentación y objetivos de este libro
- Qué es un plugin y para qué sirve
- Cómo crear un plugin básico

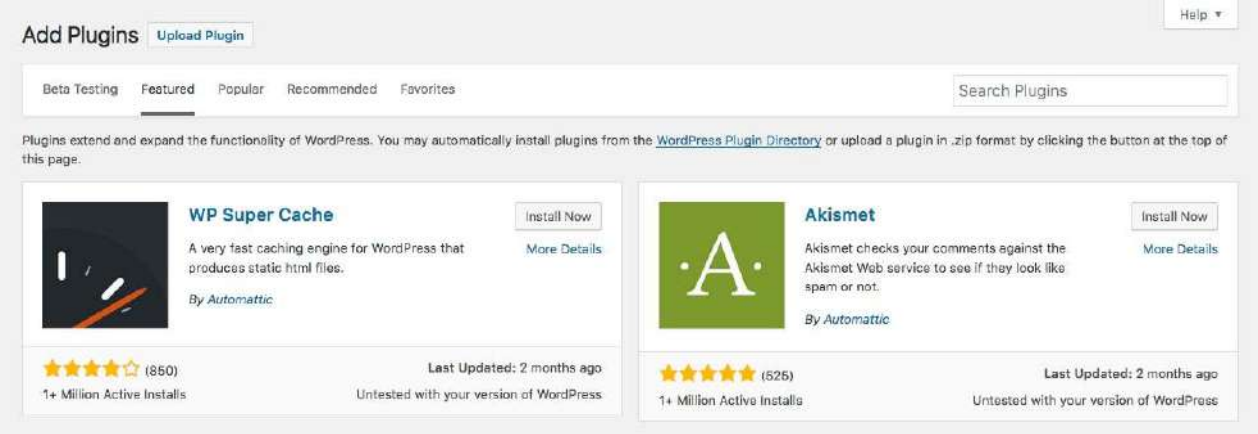
A quién está dirigido este libro

Los contenidos de este libro están destinados a un perfil bastante específico: desarrolladores web con experiencia en WordPress, con conocimientos de instalación y configuración de la plataforma (incluyendo themes y plugins), nociones fuertes de HTML y CSS, y nociones al menos básicas de PHP y programación en general.

Se presupone que el lector ya trabajó con WordPress para la creación de sitios web destinados a clientes o para fines personales, y que es capaz de investigar por su cuenta acerca de los conceptos estudiados en el libro, los cuales de ninguna manera se presentan como una manera definitiva de desarrollar plugins, sino como introductorios a un gran número de diferentes posibilidades.

Por otra parte, si bien no es el foco principal, este libro también está pensado y estructurado de forma tal que programadores más experimentados puedan utilizar los contenidos presentados como punto de referencia para sus propios desarrollos.

Presentación y objetivos de este libro



Al trabajar con WordPress es posible buscar un plugin tanto a través de nuestro propio sitio como desde el directorio oficial, entre varias otras opciones. Dependiendo de cómo lo hayamos encontrado, un plugin puede instalarse desde la sección de administración de WordPress, o descomprimiendo un archivo zip (o similar) dentro del directorio `wp-content/plugins`. Sin embargo, con solo instalar plugins no necesariamente podemos saber cómo estos funcionan internamente, cómo se construyen, cómo llegan a hacer lo que hacen. Por esa razón, en este libro vamos a sumergirnos en lo que es el funcionamiento y la arquitectura de un plugin desde un punto de vista técnico, trabajando directamente sobre código y estudiando las herramientas que ofrece WordPress para construir nuestras propias extensiones.

A grandes rasgos, la definición más sencilla que se puede dar de un plugin es que sirve para agregar funcionalidad adicional a una instalación de WordPress, algo que no está disponible a partir del momento en el que WordPress se instala. Es común encontrarnos con un requisito que no se puede cumplir con los plugins disponibles en algún directorio, y vernos obligados a hacerlo nosotros mismos, o pedirle a un programador que lo haga. Debido a esto, si estamos interesados en profesionalizar nuestro trabajo con WordPress, necesitamos tener ciertas nociones básicas acerca de cómo se constituye un plugin.

Como diferentes plugins suelen responder a diferentes necesidades, no se usan exactamente las mismas herramientas para construir cada uno, sino que muchas de ellas van a variar de un plugin a otro. Sin embargo, podría decirse que hay un núcleo de herramientas que van a ser usadas en una cantidad enorme de plugins, y en este libro vamos a detenernos principalmente en ellas.

Las más importantes de estas herramientas van a ser las diferentes APIs que WordPress nos ofrece para extender su funcionalidad básica. La API predominante para desarrollo de plugins es la **Plugin API**, y también vamos a revisar otras muy importantes como la

Options API y la **Settings API**.

Como vamos a estar trabajando continuamente con código, algo que va a ser necesario para poder seguir eficientemente los contenidos de este libro es contar con un conocimiento al menos básico de PHP. Un excelente punto de partida para quienes necesiten aprender acerca del lenguaje o reforzar sus conocimientos es el curso interactivo de PHP de Codecademy. También es conveniente tener cerca la documentación oficial de PHP.

Durante el transcurso de varios capítulos, además, vamos a revisar ciertas prácticas de optimización de código, y a estudiar diferentes métodos para que nuestros plugins sean extensibles, es decir dejarlos listos para que otros desarrolladores puedan seguir construyendo sus propias extensiones a partir de nuestro código.

Qué es un plugin y para qué sirve

En un sentido conceptual, podríamos decir que todo sitio o aplicación web se divide en tres partes: **contenido**, **presentación** y **funcionalidad**. **El contenido es aquella información variable que nuestro sitio o aplicación le muestra al usuario final**, y que alguien con los permisos adecuados puede agregar, modificar o eliminar. En la mayoría de las aplicaciones modernas es provisto de manera externa por un usuario, sin necesidad de modificar de manera manual los archivos que constituyen a la aplicación, y normalmente queda guardado en algún tipo de base de datos. **La presentación es la forma en la que esa información se le muestra al usuario**, y tiene que ver con la implementación de algún tipo de diseño gráfico sobre una interfaz; es básicamente cómo se ve nuestro proyecto. **La funcionalidad tiene que ver con todos los procesos internos que manejan el contenido** (por ejemplo la carga, edición y eliminación) y lo dejan preparado para ser presentado al usuario.

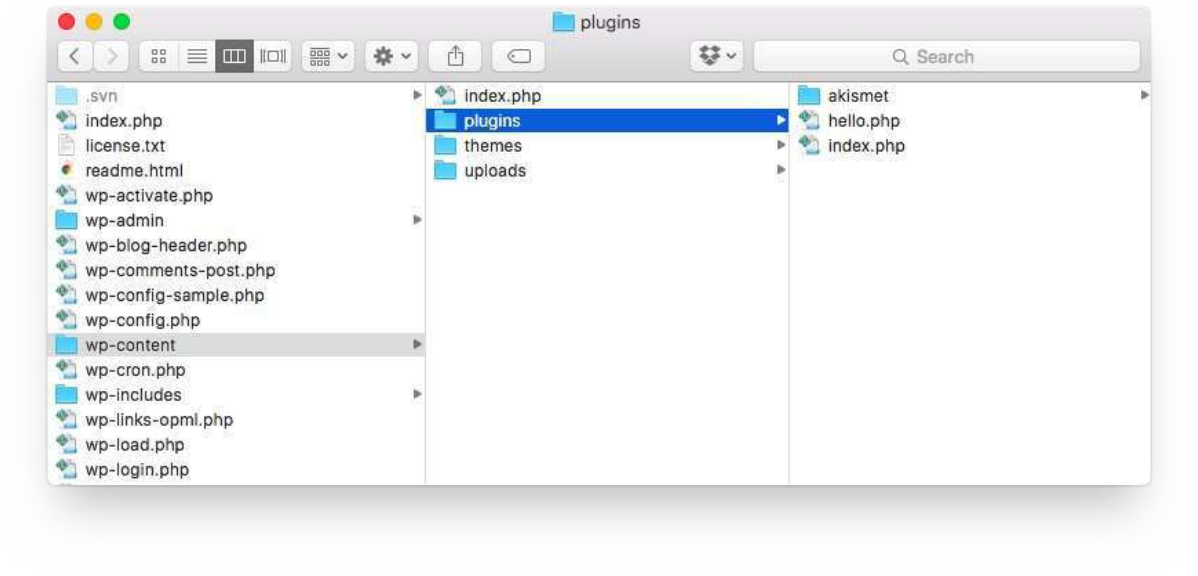
WordPress, como todo CMS (por *Content Management System*) o sistema de gestión de contenidos, se encarga por su cuenta de gran parte de las cuestiones técnicas relacionadas con la manipulación de información. Sin embargo, muchas veces vamos a encontrarnos con que necesitamos poder manejar algún tipo de información que no está disponible desde la instalación, o que se ejecuten ciertos procesos internos invisibles al usuario, o que necesitamos mostrar algo de una manera que no estaba prevista por las opciones de presentación que tenemos a nuestro alcance. Ese es el momento en el cual entran en acción los plugins y themes.

Lo que se dice más típicamente en el ámbito de los desarrolladores que trabajan con WordPress es que, mientras los themes están pensados para manejar cuestiones de presentación, los plugins apuntan exclusivamente a agregar nueva funcionalidad. Sin embargo, en este libro vamos a ser un poco más específicos acerca de estas definiciones, y a decir que eso no siempre es tan así. Eso pasa porque **la funcionalidad y la presentación no siempre son dos conceptos inseparables**, sino que a veces están muy entrelazados. Y si bien siempre es una excelente práctica intentar separarlos cuanto sea posible, a veces se presentan algunas situaciones problemáticas en las que es complicado distinguirlos. Por eso nos vamos a encontrar muy típicamente con themes que manejan algunas cuestiones de funcionalidad, o con plugins que ofrecen algún tipo de presentación. Incluso hay cosas que bien podrían ser incluidas tanto en plugins como en themes, y en ese punto nos vemos obligados a decidir dónde es mejor ubicarlas.

Teniendo en cuenta todo esto, podemos decir que **los themes, más que tener que ver específicamente con la presentación, en realidad tienen el foco puesto en ella, sin dejar completamente de lado la funcionalidad**. Son aquello que, a grandes rasgos, va a definir de qué manera se va a ver nuestro sitio, pero pueden ofrecer ciertas características de comportamiento. **Lo mismo se aplica a los plugins, pero a la inversa: tienen el foco puesto en agregar nueva funcionalidad a nuestro sitio, pero pueden ofrecer nuevas formas de visualización**.

A partir de tener presentes cuáles son estas diferencias y similitudes conceptuales básicas entre plugins y themes, podemos empezar a construir nuestros propios plugins.

Cómo crear un plugin básico

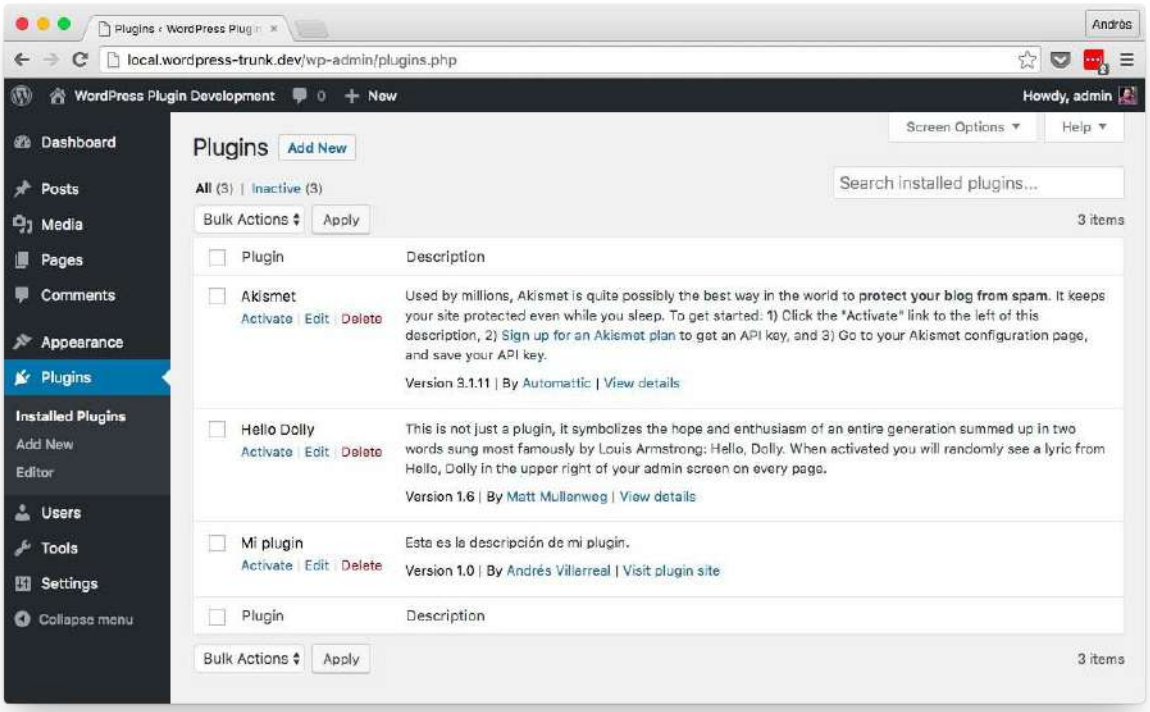


En un sentido reduccionista, los plugins son simples archivos PHP que se encuentran en la carpeta `wp-content/plugins` de nuestra instalación de WordPress. El paquete de instalación de WordPress incluye dos plugins: **Akismet** y **Hello Dolly**. Hello Dolly es un plugin que consiste en un simple archivo llamado `hello.php`, y lo podemos encontrar suelto en la carpeta de plugins. Akismet es más complejo: consiste en varios archivos, y por una cuestión de orden lo vamos a encontrar en su propia carpeta. Sin embargo, dentro de esa carpeta vamos a encontrar un archivo principal llamado `akismet.php`. La similitud entre estos dos archivos, `hello.php` y `akismet.php`, es que ambos cuentan con una sección de código comentado al inicio de cada uno, conteniendo una serie de definiciones con los datos de cada plugin. Esa porción de código, que funciona a manera de encabezado del plugin, es lo que permite que WordPress reconozca el archivo como el principal de un plugin, y que a partir de ahí lo pueda mostrar en la lista de la sección *Plugins* de nuestra instalación.

Esta forma de organización nos indica que un plugin puede estar suelto dentro de `wp-content/plugins` o dentro una carpeta propia del plugin. Lo importante es que el archivo cuente con ese encabezado en el que se declaren ciertos datos del plugin. Solamente es obligatorio que nuestro plugin tenga un nombre, pero también podemos agregar descripción, link, autor, versión, etc.

```
<?php
/*
Plugin Name: Mi plugin
Plugin URI:  http://example.org/mi-plugin
Description: Esta es la descripción de mi plugin.
Version:     1.0
Author:      Andrés Villarreal
Author URI:  http://andrezrv.com
License:     GPL2
License URI: https://www.gnu.org/licenses/gpl-2.0.html
Domain Path: /languages
Text Domain: mi-plugin
*/
```

Una vez que tenemos nuestro archivo principal con su encabezado, podemos ver que WordPress lo reconoce como tal en nuestra lista de plugins instalados. Sin embargo, todavía no está activo. Para activarlo tenemos que hacer click en el link correspondiente. Eso hace que todo el código PHP que escribamos en nuestro archivo empiece a ejecutarse, como pasa con cualquier archivo PHP que podamos ejecutar en un proyecto propio.



Como todavía no tenemos nada de código PHP propiamente dicho, más allá del encabezado, no tenemos nada ejecutándose. Podemos solucionar eso rápidamente escribiendo algo de código PHP después del encabezado, como por ejemplo un `die( 'Hello world!' );`. Esto va a hacer que, una vez que actualicemos nuestro browser en cualquier página de nuestra instalación de WordPress, se imprima *"Hello world!"* y la

ejecución del script termine, tal como dicta el comportamiento de la función `die()`. Claramente, esto no es para nada útil, pero sirve para darnos una idea de la libertad de manejo de código que tenemos a partir de ese archivo.

```
<?php
/*
Plugin Name: Mi plugin
Plugin URI:  http://example.org/mi-plugin
Description: Esta es la descripción de mi plugin.
Version:     1.0
Author:      Andrés Villarreal
Author URI:  http://andrezrv.com
License:     GPL2
License URI: https://www.gnu.org/licenses/gpl-2.0.html
Domain Path: /languages
Text Domain: mi-plugin
*/

die( 'Hello world!' );
```

Para ver más posibilidades acerca de lo que nos permite WordPress, una buena idea a la hora de empezar es estudiar *Hello Dolly*. Dentro de `hello.php` podemos ver que se declara una función, `hello_dolly()`, que toma una línea de la canción "Hello Dolly" al azar, y la imprime al ejecutarse la acción `admin_notices`, por medio del uso de la función `add_action()`. De hecho, si activamos *Hello Dolly* desde nuestra lista de plugins, podemos ver que la línea que se muestra cambia cada vez que actualizamos cualquier página en la sección de administración, reflejando lo que se define en el código del plugin.

Extracto de `hello.php`:

```
<?php
// This just echoes the chosen line, we'll position it later
function hello_dolly() {
    $chosen = hello_dolly_get_lyric();
    echo "<p id='dolly'>$chosen</p>";
}

// Now we set that function up to execute when the admin_notices action is called
add_action( 'admin_notices', 'hello_dolly' );
```

No importa ahora mismo cómo funciona esta interacción entre funciones y acciones, porque la vamos a ver con más detalle en los próximos capítulos. Lo que es importante saber en este punto es que, una vez que creamos nuestro archivo principal de plugin y lo activamos desde el panel de administración, dentro de ese archivo podemos hacer cualquier cosa que PHP nos permita hacer.

Parte 2: Plugin API

Contenidos:

- Programación orientada a eventos
- Acciones
- Filtros
- Uso de clases y objetos en eventos
- Remover eventos del registro

Programación orientada a eventos

La herramienta principal que nos ofrece WordPress para construir nuestras propias extensiones es un conjunto de funciones al que comúnmente se llama *Plugin API*, un sistema basado en el **paradigma de **Programación Orientada a Eventos**, o **Programación Basada en Eventos** (en inglés, *Event-oriented programming* o *Event-driven programming*). Es combinable con otros paradigmas populares, como el estructurado, el orientado a objetos y el funcional, y tiene unos conceptos fundamentales muy sencillos.

Este paradigma es extremadamente útil cuando necesitamos que un proceso se ejecute en algún punto determinado, pero tenemos un acceso limitado al código que se encuentra en ese punto; o cuando queremos modificar un proceso sin cambiar el código original. Es particularmente útil a la hora de extender WordPress, porque no podemos modificar libremente su código sin perder lo que hayamos hecho en futuras actualizaciones.

El paradigma de eventos dicta que, en ciertos lugares puntuales de nuestro programa, van a ocurrir determinados eventos o "sucesos" importantes, los cuales el desarrollador original debe identificar de cierta manera particular. Estos eventos, por sí mismos, no hacen nada; su función original es meramente descriptiva. Solamente van a empezar a tener algún efecto sobre el programa cuando les asignemos procesos, es decir cuando indiquemos que al ocurrir un determinado evento tiene que ejecutarse un proceso determinado.

Normalmente, vamos a tener en alguna parte de nuestro código la ejecución de un evento con un nombre dado. Supongamos que tenemos un evento llamado `mesa_servida`, el cual podría verse de esta forma:

```
<?php
evento( 'mesa_servida' );
```

Por otra parte, vamos a necesitar que, al ocurrir ese evento, también se ejecute un proceso. Vamos a suponer que, al ocurrir el evento `mesa_servida`, queremos que se procese la función `sentarse_a_comer()`. Para eso, necesitamos que la función haya sido declarada antes de que ocurra el evento.

```
<?php
function sentarse_a_comer() {
    echo 'a comer!';
}

evento( 'mesa_servida' );
```

Sin embargo, ese código por sí mismo todavía no cumple con nuestro propósito. Para que la función se procese en el momento en el que se ejecuta el evento, necesitamos asignar la función al evento.

```
<?php
function sentarse_a_comer() {
    echo 'a comer!';
}

asignar_proceso( 'mesa_servida', 'sentarse_a_comer' );

evento( 'mesa_servida' ); // Se imprime "a comer!"
```

De esta manera, al usar `asignar_proceso()` con el nombre del evento como primer parámetro y el nombre de la función como segundo parámetro, indicamos que, al ocurrir el evento `mesa_servida`, va a procesarse el código declarado dentro de la función `sentarse_a_comer`. La función asignada a un evento es lo que dentro de este paradigma se suele llamar *hook*.

El importante notar que, al menos en PHP, no es necesario que la función exista antes de asignarla a un evento, pero sí tiene que haber sido declarada antes de que el evento ocurra. La misma asignación también tiene que hacerse antes de que ocurra el evento. De esta manera, el siguiente código es equivalente al anterior:

```
<?php
asignar_proceso( 'mesa_servida', 'sentarse_a_comer' );

function sentarse_a_comer() {
    echo 'a comer!';
}

evento( 'mesa_servida' );
```

Conociendo los conceptos fundamentales de la programación orientada a eventos, podemos ver de qué manera se puede aplicar en WordPress para construir nuestros propios plugins usando los dos tipos de eventos que nos ofrece: acciones y filtros.

Acciones

Uno de los dos tipos de eventos ofrecidos por WordPress se llama ***Action***, o acción. El propósito de las acciones es permitir la ejecución de procesos propios durante la carga de una página. Algunos ejemplos de estos procesos pueden consistir en modificar información de la base de datos, enviar un mail, registrar nuevos tipos de contenido, o imprimir cierta información en el HTML procesado.

La interacción básica entre eventos y procesos es muy similar al ejemplo de la sección anterior. Las acciones se ejecutan por medio de la función `do_action()`, mientras que los hooks se registran usando `add_action()`.

```
<?php
add_action( 'mesa_servida', 'sentarse_a_comer' );

function sentarse_a_comer() {
    echo 'a comer!';
}

do_action( 'mesa_servida' );
```

Sin embargo, este uso básico a veces puede resultar un poco limitado. Por ejemplo, es posible que queramos añadir un segundo hook a `mesa_servida`, y necesitemos especificar cuál de ellos va a ejecutarse primero. Supongamos que introducimos la función `comer()`, y queremos que se ejecute inmediatamente después de `sentarse_a_comer()`.

```
<?php
add_action( 'mesa_servida', 'comer' );

function comer() {
    echo 'comiendo ...';
}

add_action( 'mesa_servida', 'sentarse_a_comer' );

function sentarse_a_comer() {
    echo 'a comer!';
}

do_action( 'mesa_servida' );
```

Con este código, la función `comer()` siempre se va a ejecutar antes de `sentarse_a_comer()`. Si no tenemos la posibilidad de cambiar la ubicación de la asignación de `add_action( 'mesa_servida', 'comer' )`, necesitamos encontrar una manera de hacer que `comer()` se ejecute después de `sentarse_a_comer()`. Para este tipo de necesidades, la función `add_action()` admite un tercer parámetro llamado `$priority`, en el cual se especifica un valor numérico. Los hooks con valores menores van a ser ejecutados con anterioridad a los hooks con valores más altos, y el valor por defecto es 10. Teniendo esto en cuenta, podemos modificar nuestro código de esta manera.

```
<?php
add_action( 'mesa_servida', 'comer', 20 );

function comer() {
    echo 'comiendo ...';
}

add_action( 'mesa_servida', 'sentarse_a_comer', 10 );

function sentarse_a_comer() {
    echo 'a comer!';
}

do_action( 'mesa_servida' );
// Se imprime "a comer!"
// Se imprime "comiendo ..."
```

Puede pasar, también, que dentro de las funciones que usamos como hooks necesitemos algún tipo de información contextual con respecto al momento en el que se ejecuta la acción. Supongamos que dentro de `sentarse_a_comer()` necesitamos evaluar cuántos comensales vamos a tener.

```
<?php
add_action( 'mesa_servida', 'sentarse_a_comer', 10 );

function sentarse_a_comer( $comensales ) {
    if ( $comensales < 5 ) {
        echo 'a comer!';
    } else {
        echo 'acá hay demasiada gente, Roberto!'
    }
}

$comensales = 10;

do_action( 'mesa_servida' );
// Se imprime "a comer!"
```

De alguna manera necesitamos hacer que ese número de comensales declarado en el mismo contexto de ejecución de `do_action( 'mesa_servida' )` llegue a la función `sentarse_a_comer()`. Para esto podemos adjuntar a `do_action()` todos los parámetros que queramos, y en `add_action()` especificar, en un cuarto parámetro llamado `$accepted_args`, el número de parámetros que va a recibir nuestro hook.

```
<?php
add_action( 'mesa_servida', 'sentarse_a_comer', 10, 1 );

function sentarse_a_comer( $comensales ) {
    if ( $comensales < 5 ) {
        echo 'a comer!';
    } else {
        echo 'acá hay demasiada gente, Roberto!';
    }
}

$comensales = 10;

do_action( 'mesa_servida' , $comensales );
// Se imprime "acá hay demasiada gente, Roberto!"
```

El valor por defecto de `$accepted_args` es 1, por lo cual, si vamos a tener un solo parámetro, podemos incluso no especificar este valor.

```
<?php
add_action( 'mesa_servida', 'sentarse_a_comer', 10 );

function sentarse_a_comer( $comensales ) {
    if ( $comensales < 5 ) {
        echo 'a comer!';
    } else {
        echo 'acá hay demasiada gente, Roberto!';
    }
}

$comensales = 10;

do_action( 'mesa_servida' , $comensales );
// Se imprime "acá hay demasiada gente, Roberto!"
```

Sin embargo, podemos pasarle a `do_action()` tantos parámetros como necesitemos en nuestra función, pero siempre especificando la cantidad en `add_action()`, en caso de que sea más de uno.

```php
<?php
add_action( 'mesa_servida', 'sentarse_a_comer', 10, 2 );

function sentarse_a_comer( $comensales, $comida ) {
    if ( $comensales < 5 ) {
        echo 'A comer!';
    } else {
        echo 'Acá hay demasiada gente, Roberto! Tenemos solamente ' . $comida;
    }
}

$comensales = 10;
$comida = 'Fideos con pesto';

do_action( 'mesa_servida', $comensales, $comida );
// Se imprime "acá hay demasiada gente, Roberto! Tenemos solamente fideos con pesto"
```

Sabiendo cómo opera este tipo de evento, también podemos agregar hooks a las acciones nativas de WordPress. Para ejemplificar esto vamos a crear un plugin muy básico, y que tiene un solo propósito: mandarle un mail al administrador del sitio cada vez que se publique un nuevo post. Para esto, definimos la función que envía el mail, y se la asignamos como hook a la acción `publish_post`.

```php
<?php
/*
Plugin Name: Aviso de Actualización
*/

add_action( 'publish_posts' ,'avisar_amigos' );

function avisar_amigos() {
    $amigos = 'juan@example.org,pedro@example.org';

    mail( $amigos, 'Actualización de blog', 'Acabo de actualizar mi blog: http://blog.
example.com' );
}
```

Una vez que activemos este nuevo plugin y publiquemos un nuevo post, vamos a poder ver que llega un nuevo mail a las casillas de correo especificadas en la función.

Nótese que no estamos llamando directamente a `do_action( 'publish_posts' );`. Esto es porque ese llamado se va a estar haciendo en algún lugar del código base de WordPress, que llamamos Core. WordPress cuenta con una lista bastante larga de acciones propias a las que podemos asignar nuestros propios hooks, la cual nos puede servir a manera de guía.

Si queremos, también podemos crear y usar acciones propias. Por ejemplo, si yo quisiera que, una vez que se envió el mail al administrador, también se envíe un mail a una casilla de correo específica, puedo ejecutar una acción una vez que se envía el primer mail, y asignarle como hook una segunda función.

```
<?php
/*
Plugin Name: Aviso de Actualización
*/

add_action( 'publish_posts' ,'avisar_amigos' );

function avisar_amigos() {
    $amigos = 'juan@example.org,pedro@example.org';

    if ( mail( $amigos, 'Actualización de blog', 'Acabo de actualizar mi blog: http://
blog.example.com' ) ) {
        do_action( 'email_enviado' );
    }
}

add_action( 'email_enviado' , 'avisarme' );

function avisarme() {
    mail( 'yo@example.org', 'Se publicó post y se envió mail a amigos' );
}
```

De esta manera, si el envío del primer mail fue exitoso, se va a ejecutar la acción `email_enviado`. Como esta acción tiene asignada la función `avisarme()` como hook, va e enviarse un segundo mail a la dirección de correo especificada en dicha función.

Filtros

El otro tipo de eventos que nos ofrece WordPress son los ***Filters***, o filtros. Este tipo de eventos ya no pone tanto el foco en la ejecución de procesos, como pasa con las acciones, sino en la manipulación de datos internos de la aplicación. Por ejemplo, un filtro puede ser utilizado para cambiar el valor de una variable, o modificar el valor de retorno de una función. Usos típicos de los filtros pueden ser: activar o desactivar ciertas características de la aplicación, modificar alguna parte del HTML que se va a imprimir, cambiar valores de configuración, o alterar consultas a la base de datos.

Una característica importante de los filtros es que siempre tienen un valor de retorno, a diferencia de las acciones. Debido a esto, los filtros siempre están asignados a una variable, a una evaluación o a un valor de retorno de una función o método.

Teniendo en cuenta esta diferencia, su uso es muy similar al de las acciones. En algún punto de nuestro código vamos a tener un llamado a la función `apply_filters()`, que es equivalente a `do_action()`. Los parámetros que va a recibir esta función son el nombre del evento y el valor que va a tener por defecto.

```
<?php
$comensales = apply_filters( 'cantidad_de_comensales', 10 );
```

Ahora bien, antes de que ese código se ejecute, necesitamos definir cuáles van a ser los filtros que se asignen al evento. Esto podemos hacerlo a través de la función `add_filter()`.

```
<?php
add_filter( 'cantidad_de_comensales', 'nueva_cantidad_de_comensales' );

function nueva_cantidad_de_comensales( $comensales ) {
    if ( viene_ramon() ) {
        $comensales++;
    }

    return $comensales;
}

$comensales = apply_filters( 'cantidad_de_comensales', 10 );
```

De esta manera, al momento de definirse la variable `$comensales`, el valor por defecto (10) se le va a pasar a la función `nueva_cantidad_de_comensales()`, y va a sumar 1 en caso de que Ramón venga (dando por resultado 11). Noten que antes de finalizar la función que

opera como filtro siempre necesitamos devolver un valor; de lo contrario el valor que estemos filtrando va a terminar siendo `null`.

Los filtros, al igual que las acciones, pueden recibir una prioridad de ejecución (`$priority`) y un número de argumentos (`$accepted_args`) como tercer y cuarto parámetro, respectivamente.

```
<?php
add_filter( 'cantidad_de_comensales', 'descartar_vegetarianos', 20, 2 );

function descartar_vegetarianos( $comensales, $comida ) {
    if ( 'asado' == $comida && viene_jose() ) {
        $comensales--;
    }

    return $comensales;
}

add_filter( 'cantidad_de_comensales', 'nueva_cantidad_de_comensales', 10 );

function nueva_cantidad_de_comensales( $comensales ) {
    if ( viene_ramon() ) {
        $comensales++;
    }

    return $comensales;
}

$comida = 'asado';
$comensales = apply_filters( 'cantidad_de_comensales', 10, $comida );
```

De esta forma, el primer filtro a ejecutarse va a ser `nueva_cantidad_de_comensales()`, por más que se haya declarado en segundo lugar. No necesitamos especificar la cantidad de argumentos para ese filtro, porque el valor por defecto es 1, y solamente necesitamos la primera variable, `$comensales`. En segundo lugar se va a ejecutar la función `descartar_vegetarianos()`, que va a restar un comensal en caso de que la comida sea asado y venga José. Al asignar este filtro sí especificamos que vamos a recibir dos parámetros, ya que necesitamos las variables `$comensales` y `$comida`, que se están pasando al filtro por medio de `apply_filters()`.

Al igual que pasa con las acciones, también tenemos muchos filtros que vienen con WordPress por defecto. Uno de ellos es `the_content`, que se aplica al contenido de cada post o página antes de ser impreso. Supongamos que queremos crear un nuevo plugin que modifique levemente este contenido. Para eso necesitamos crearlo, con su correspondiente encabezado, activarlo desde la sección de administración, y tener dentro de nuestro código algo como esto:

```php
<?php
/*
Plugin Name: Modify Content
*/

add_filter( 'the_content', 'modify_post_content' );

function modify_post_content( $content ) {
    if ( is_single() ) {
        $content = '<p>Esto es un post.</p>' . $content;
    }

    return $content;
}
```

Una vez que guardemos este código en nuestro plugin y refresquemos cualquier página correspondiente a un post, vamos a ver el texto que agregamos en nuestra función inmediatamente arriba del resto del contenido.

De la misma forma que cuando usamos acciones predefinidas por WordPress, no estamos haciendo en ningún momento el llamado a `apply_filters()` con `the_content` como primer parámetro, ya que WordPress mismo se encarga de hacer ese llamado en algún punto de su ejecución. Y también al igual que con las acciones, podemos definir nuestros propios eventos en los que se van a aplicar filtros.

```php
<?php
/*
Plugin Name: Modify Content
*/

add_filter( 'additional_post_content', 'show_thumbnail_message' );

function show_thumbnail_message( $additional_post_content ) {
    if ( ! has_post_thumbnail() ) {
        $additional_post_content .= '<p>Este post no tiene una imagen destacada.</p>';
    }

    return $additional_post_content;
}

add_filter( 'the_content', 'modify_post_content' );

function modify_post_content( $content ) {
    if ( is_single() ) {
        $content = apply_filters( 'additional_post_content', '<p>Esto es un post.</p>'
 ) . $content;
    }

    return $content;
}
```

Con este agregado, es decir definiendo un evento al modificar el contenido, y asignando un filtro a ese nuevo evento, podemos hacer que se muestre el mensaje "Este post no tiene una imagen destacada" antes del post, en el caso de que efectivamente no tenga una imagen. De lo contrario, el texto va a ser "Esto es un post". Podemos ver ese cambio reflejado al refrescar la página de cualquier post sin imagen destacada.

También para los filtros hay disponible una lista muy grande de aquellos que WordPress ofrece por defecto en su documentación oficial, con un detalle del momento en el que se aplica cada uno de estos eventos.

Ahora que ya manejamos los conceptos principales de la programación orientada a eventos, podemos continuar viendo algunas prácticas un poco más avanzadas al usar la Plugin API.

Uso de clases y objetos en eventos

Hasta ahora solamente estuvimos viendo cómo asignar funciones a eventos. Sin embargo, cuando creemos nuestros plugins, o cuando estemos mirando el código de plugins de terceros, nos vamos a encontrar con que necesitamos usar métodos de clases, en lugar de funciones, para llevar a cabo algún proceso.

La solución más simple a este problema es crear funciones que sirvan a manera de contenedores de los métodos de clase. Por ejemplo, podemos tener algo como esto.

```
<?php
class My_Class {
    function my_method() {
        //
    }
}

add_action( 'my_action', 'my_method_wrapper' );

function my_method_wrapper() {
    $obj = new My_Class;
    $obj->my_method();
}

do_action( 'my_action' );
```

De esta manera tenemos una función que instancia el objeto que queremos usar y ejecuta el método que necesitamos. El problema que tenemos acá es que es la función misma la que crea un nuevo objeto, el cual no va a ser accesible por fuera de ella, y probablemente necesitemos usar ese objeto para otras cosas por fuera de la función. Una solución a esto sería pasar el objeto como parámetro a la función desde `do_action()`.

```php
<?php
class My_Class {
    public function my_method() {}
}

add_action( 'my_action', 'my_method_wrapper' );

function my_method_wrapper( $obj ) {
    $obj->my_method();
}

$obj = new My_Class;

do_action( 'my_action', $obj );
```

Es una posibilidad, y puede servirnos para muchos casos, pero esta propuesta depende de que tengamos acceso al llamado a `do_action()` y de que tengamos control de lo que se le pasa como parámetro. Eso es algo que no va a pasar cuando estemos interactuando con eventos nativos de WordPress.

Este problema se resuelve pasando directamente el método como parámetro a `add_action()` o `add_filter()`. Para eso necesitamos que el segundo parámetro sea un array con dos elementos: el primero va a corresponder al objeto o al nombre de la clase que contiene nuestro método, y el segundo va a ser el nombre del método.

```php
<?php
class My_Class {
    public function my_method() {}
}

$obj = new My_Class;

add_action( 'my_action', array( $obj, 'my_method' ) );

do_action( 'my_action' );
```

Lo que necesitamos para que esto funcione es que el objeto haya sido instanciado antes de hacer nuestro llamado a `add_action()`.

Otra posibilidad es asignar el método al evento directamente dentro de un método de la clase misma, por ejemplo el constructor. De esta manera, cuando el objeto se instancia, directamente se asigna el método al evento. Esto es lo mismo que decir que el objeto "sabe" cuáles de sus métodos van a asignarse a un evento.

```php
<?php
class My_Class {
    public function __construct() {
        add_action( 'my_action', array( $this, 'my_method' ) );
    }

    public function my_method() {}
}

$obj = new My_Class;

do_action( 'my_action' );
```

Además, tenemos la posibilidad de asignar métodos estáticos a eventos, es decir métodos que no necesitan conocer el contexto de una instancia de la clase, tal como pasa con las funciones. Si ya contamos con el objeto instanciado, es decir si ya hicimos un llamado a `$obj = new My_Class`, podemos asignar el método estático usando el objeto, tal como lo veníamos haciendo.

```php
<?php
class My_Class {
    public static function my_method() {}
}

$obj = new My_Class;

add_action( 'my_action', array( $obj, 'my_method' ) );

do_action( 'my_action' );
```

Sin embargo, esto puede no ocurrir; puede darse el caso en el que necesite usar ese método estático de mi clase antes de que la clase misma sea instanciada. En ese caso, en lugar de pasar el objeto como primer elemento del hook, voy a pasar directamente el nombre de la clase.

```php
<?php
class My_Class {
    public static function my_method() {}
}

add_action( 'my_action', array( 'My_Class', 'my_method' ) );

do_action( 'my_action' );
```

Algo muy importante a tener en cuenta a la hora de asignar métodos de clase a eventos es que esos métodos siempre tienen que ser de acceso público; WordPress va a mostrar un error si un método usado como hook es privado o está protegido. Por ejemplo, esto no va a funcionar:

```
<?php
class My_Class {
    private function my_method() {}

    protected function my_other_method() {}
}

$obj = new My_Class;

add_action( 'my_action', array( $obj, 'my_method' ) );
add_action( 'my_action', array( $obj, 'my_other_method' ) );

do_action( 'my_action' );
```

Una recomendación en materia de extensibilidad que suele hacerse entre los desarrolladores que trabajan con WordPress es que los métodos de clase que vayamos a crear no sean nunca privados, o que lo sean solo en casos extremadamente necesarios. En lugar de hacerlos privados, la recomendación es dejarlos como protegidos, para que de esta manera otros desarrolladores puedan extender nuestras clases desde sus propios plugins si lo necesitan.

Por otra parte, nótese que, si bien estuvimos usando acciones para los ejemplos de este capítulo, estas mismas formas de manipular métodos también son por completo aplicables a filtros.

Todos estas son formas totalmente válidas de trabajar con métodos en la implementación del paradigma de programación orientada a eventos que hace WordPress, pero ninguna es una solución definitiva. La solución óptima depende mucho de las características del plugin con el que estemos trabajando, por lo cual está en nosotros probar y evaluar cuál nos conviene más para resolver un problema determinado.

Remover eventos del registro

Algo que podemos llegar a necesitar mientras desarrollamos nuestras propias extensiones es que ciertas acciones o filtros se dejen de ejecutar. Por ejemplo, podemos querer desactivar alguna funcionalidad nativa de WordPress, o de un plugin de terceros, o que alguna de nuestras acciones o filtros solamente se ejecuten en determinados contextos. Para lograr eso, WordPress nos ofrece cuatro funciones:

- `remove_action()`
- `remove_filter()`
- `remove_all_actions()`
- `remove_all_filters()`

Cuando asignamos un filtro o una acción a un evento, WordPress lo guarda en una lista, más específicamente en la variable global `$wp_filters`, donde también queda el detalle de sus prioridades y la cantidad de argumentos que acepta. Lo que nos permiten estas funciones es remover de esa lista los procesos que necesitemos.

Por ejemplo, supongamos que tenemos dos eventos, una acción y un filtro, y a cada uno de ellos queremos asignarle una función.

```
<?php
add_action( 'my_action', 'my_action_callback', 10, 1 );

function my_action_callback() {
    echo 'Hello world!';
}

add_filter( 'my_filter', 'my_filter_callback', 10, 1 );

function my_filter_callback( $value ) {
    return 'some other value';
}

do_action( 'my_action' );

$my_value = apply_filters( 'my_filter', 'some value' );
```

Sin embargo, en algún punto en el futuro vamos a necesitar que esas funciones que asignamos dejen de ejecutarse, pero por las características de nuestra extensión no podemos remover ni las asignaciones ni las declaraciones de funciones. No podemos simplemente remover código. En ese punto es donde necesitamos usar estas nuevas funciones de las que venimos hablando.

```php
<?php
add_action( 'my_action', 'my_action_callback', 20, 1 );

function my_action_callback() {
    echo 'Hello world!';
}

add_filter( 'my_filter', 'my_filter_callback', 20, 1 );

function my_filter_callback( $value ) {
    return 'some other value';
}

remove_action( 'my_action', 'my_action_callback', 20 );
remove_filter( 'my_filter', 'my_filter_callback', 20 );

do_action( 'my_action' );

$my_value = apply_filters( 'my_filter', 'some value' );
```

De esta manera logramos que las funciones que asignamos previamente con `add_action()` y `add_filter()` dejen de ejecutarse.

Para usar correctamente `remove_action()` y `remove_filter()` necesitamos tener en cuenta dos cosas: en primer lugar, tienen que ser llamadas después de que los hooks hayan sido asignados, pero antes de que se ejecuten las acciones y filtros. En segundo lugar, ambas funciones reciben un tercer parámetro, que es equivalente a la prioridad con la que se asignaron previamente los hooks que ahora estamos removiendo del registro. Este parámetro no es obligatorio, y su valor por defecto es 10. Si no lo especificamos, tenemos que asegurarnos de que la prioridad del hook también sea 10, o que tampoco esté especificada.

Ahora bien, ¿qué pasa cuando los hooks que queremos remover son métodos de clases? Si tenemos acceso al objeto instanciado de la clase, podemos hacerlo de esta manera, que es la más intuitiva:

```php
<?php
class My_Class {
    public function my_method() {}
}

$obj = new My_Class;

add_action( 'my_action', array( $obj, 'my_method' ) );

remove_action( 'my_action', array( $obj, 'my_method' ) );

do_action( 'my_action' );
```

En cambio, si en algún momento se deja de tener acceso al objeto (por ejemplo, si el nombre de la variable que lo contiene pasa a ser usado para otra cosa) y el método es estático, puede usarse el nombre de la clase para remover el callback.

```php
<?php
class My_Class {
    public static function my_method() {}
}

add_action( 'my_action', array( $obj, 'my_method' ) );

remove_action( 'my_action', array( 'My_Class', 'my_method' ) );

do_action( 'my_action' );
```

Por último, tenemos las funciones `remove_all_actions()` y `remove_all_filters()`. Estas nos permiten remover todos los callbacks que hayan sido asignados a una acción o filtro determinados, sin necesidad de especificar más que el nombre del evento. Por ejemplo, podemos suponer que tenemos dos hooks asignados a una acción y otros dos hooks asignados a un filtro.

```php
<?php
add_action( 'my_action', 'my_action_callback', 10 );
add_action( 'my_action', 'my_other_callback', 20 );

function my_action_callback() {
    echo 'Hello world!';
}

function my_other_action_callback() {
    echo 'Hello again!';
}

add_filter( 'my_filter', 'my_filter_callback', 10, 1 );
add_filter( 'my_filter', 'my_other_filter_callback', 20, 1 );

function my_filter_callback( $value ) {
    return 'some other value';
}

function my_other_filter_callback( $value ) {
    return 'some other different value';
}

do_action( 'my_action' );

$my_value = apply_filters( 'my_filter', 'some value' );
```

Podemos remover muy fácilmente todos los hooks para cada evento haciendo algo así:

```php
<?php
add_action( 'my_action', 'my_action_callback', 10 );
add_action( 'my_action', 'my_other_callback', 20 );

function my_action_callback() {
    echo 'Hello world!';
}

function my_other_action_callback() {
    echo 'Hello again!';
}

add_filter( 'my_filter', 'my_filter_callback', 10, 1 );
add_filter( 'my_filter', 'my_other_filter_callback', 20, 1 );

function my_filter_callback( $value ) {
    return 'some other value';
}

function my_other_filter_callback( $value ) {
    return 'some other different value';
}

remove_all_actions( 'my_action' );
remove_all_filters( 'my_filter' );

do_action( 'my_action' );

$my_value = apply_filters( 'my_filter', 'some value' );
```

Tenemos que tener en cuenta que ambas son funciones para usar con mucho cuidado, ya que no es muy común querer remover todos los hooks para un evento. Sin embargo, es bueno saber que contamos con ellas, y suelen ser muy útiles mientras estamos desarrollando nuestras extensiones, particularmente con fines de testing.

Parte 3: Personalización de contenidos

Contenidos:

- Internacionalización y localización
- Post Types predefinidos
- Custom Fields y Meta Boxes
- Custom Post Types
- Taxonomías

Internacionalización y Localización

Una posibilidad que WordPress ofrece al hacer la instalación es elegir el lenguaje en el que se va a mostrar nuestro sitio. Gracias a esta opción, si se selecciona un lenguaje que no sea inglés, WordPress va a traducir automáticamente todos los textos que estén preparados para ser mostrados en el idioma elegido.

Una recomendación típica de extensibilidad es que todos los textos escritos en el código de plugins y themes sean traducibles, es decir que deben ser internacionalizados. De esta manera, alguien que quiera traducir un plugin internacionalizado a su propio idioma va a poder hacerlo simplemente siguiendo unos pocos pasos.

Para que un plugin pueda ser traducido hacen falta dos cosas: en primer lugar, contar con una palabra clave para que los textos que introducidos puedan ser reconocidos como propios del plugin; en segundo lugar que los textos sean invocados con las funciones de localización que provee WordPress. Adicionalmente se puede ofrecer una plantilla para traducciones. Esto no es obligatorio, pero resulta muy útil para los traductores.

El requisito de la palabra clave es el más sencillo de cumplir: basta con agregarla como dato en el encabezado del plugin, bajo el nombre "Text Domain". Por convención, el nombre del dominio suele ser igual al del plugin, pero en minúsculas y con los espacios separados por guiones. Este valor también suele ser igual al nombre de la carpeta del plugin.

```
<?php
/*
 * Plugin Name: My Plugin
 * Text Domain: my-plugin
 */
```

Opcionalmente, se puede especificar, en el valor "Domain Path", la ruta relativa a la carpeta donde se encuentren los archivos de traducción que podamos ofrecer por defecto.

```
<?php
/*
 * Plugin Name: My Plugin
 * Text Domain: my-plugin
 * Domain Path: languages/
 */
```

Una vez que la tenemos, debemos usar la función `load_textdomain()` para que se carguen automáticamente todas las posibles traducciones existentes de nuestro plugin.

```php
<?php
/*
 * Plugin Name: My Plugin
 * Text Domain: my-plugin
 * Domain Path: languages/
 */

load_plugin_textdomain( 'my-plugin' );
```

Las funciones de localización son varias, aunque las más comunes son `__()` y `_e()`. `__()` devuelve la traducción del texto que se le pasa como primer parámetro, mientras que `_e()` la imprime. Cada vez que se las llame se debe especificar, como segundo parámetro, el dominio al cual pertenece el texto a traducir. Teniendo eso en cuenta, puede retomarse alguno de los ejemplos de código anteriores e internacionalizarlo.

```php
<?php
/*
 * Plugin Name: My Plugin
 * Text Domain: my-plugin
 * Domain Path: languages/
 */

load_plugin_textdomain( 'my-plugin' );

add_action( 'my_action', 'my_action_callback', 20, 1 );

function my_action_callback() {
    _e( 'Hello world!', 'my-plugin' );
}

add_filter( 'my_filter', 'my_filter_callback', 20, 1 );

function my_filter_callback( $value ) {
    return __( 'some other value', 'my-plugin' );
}

$my_value = apply_filters( 'my_filter', 'some value' );

do_action( 'my_action' );
```

Un problema con el que es común encontrarse es que parte del texto a traducir dependa de valores variables. Esto es justamente un problema porque `gettext`, la librería que usa WordPress para gestionar sus traducciones, no puede leer información dinámica dentro de las cadenas de texto a traducir. Por lo tanto, algo así no puede ser traducido:

```
<?php
_e( 'Hello World! The current year is ' . date( 'Y' ), 'my-plugin' );
```

Para resolver este problema, podemos combinar nuestros textos a traducir con las funciones nativas de PHP `printf()` y `sprintf()`. El ejemplo anterior se puede replantear de manera correcta de esta forma:

```
<?php
printf( __( 'Hello World! The current year is %s', 'my-plugin' ), date( 'Y' ) );
```

Por último, se puede ofrecer una plantilla con textos localizables. La plantilla se trata de nada más que un archivo de texto con el nombre del dominio y con extensión POT (*Portable Object Template*), y un formato como el siguiente:

```
# my-plugin.pot
msgid "Hello world!"
msgstr ""

msgid "some other value"
msgstr ""
```

La propiedad `msgid` corresponde al texto original a traducir, y `msgstr` es la traducción propiamente dicha. En nuestro template este segundo valor va a estar en blanco, porque todavía no es una traducción, sino una herramienta para desarrollar nuevas traducciones.

Muchas veces podemos contar con un número bastante grande de textos a traducir, por lo cual no es muy viable incluir todos manualmente en el archivo POT. Para esto tenemos programas que pueden generar el archivo POT de manera automática, como por ejemplo Poedit.

Estos son simplemente los conceptos básicos de internacionalización y localización, y es todo lo que hace falta saber al momento de crear plugins que puedan ser traducidos. No vamos a tocar el tema de las traducciones propiamente dichas, ya que eso escapa un poco al desarrollo de plugins, y además hay muchas maneras de llevarlas a cabo.

Post Types predefinidos

Al construir una aplicación o un sitio web con cierto nivel de complejidad, es muy común que en algún momento se necesite algún tipo de contenido diferente de los que ya tenemos, alguna nueva entidad para la que sea posible crear, editar y modificar registros.

Si nuestra aplicación está basada en PHP y MySQL, normalmente se tiende a diseñar e implementar nuevas tablas en la base de datos para que contengan la información de las nuevas entidades. A continuación se escribe el código que gestione la carga, edición y eliminación de contenidos, el acceso de los usuarios de la aplicación a esos contenidos, y las relaciones e interacciones con otras entidades.

Lo que suele pasar con este enfoque es que, a medida que la aplicación se vuelve más grande y se necesitan más tipos de contenido, también se vuelve más complejo el mantenimiento de nuestras entidades, ya que sus datos y funcionalidad pueden variar mucho de una a otra, por tener diferentes características.

Sin embargo, si se trabaja con un framework, a menudo este suele proveer una serie de herramientas para crear nuevos tipos de contenido de una manera previamente definida y estandarizada, de forma que se fuerce una cierta coherencia interna para lo que vayamos a crear.

En el caso de WordPress, esta serie de herramientas esta cubierta por la API de ***Post Types***, la cual permite interactuar con tipos de contenidos nuevos o ya existentes siguiendo un conjunto de reglas y entendiendo ciertos conceptos fundamentales.

Por defecto, con su instalación básica, WordPress presenta cinco tipos de contenido: posts, páginas, revisiones, archivos y menús de navegación. Todos tienen diferencias entre ellos, las cuales se pueden notar al cargar o editar información, y todos están construidos usando la API de Post Types. Esta API permite manejar diferentes tipos de datos para diferentes tipos de contenidos de una forma estandarizada.

Custom Fields y Meta Boxes

La manera más rápida de agregar valores extra a los contenidos de un sitio es por medio del uso de campos personalizados, o *Custom Fields*. Estos campos personalizados se hacen visibles en la pantalla de creación o edición de un contenido cuando se chequea el campo correspondiente en *Opciones de Pantalla*. Esto va a hacer que WordPress muestre un menú en el que se puede agregar un nuevo campo con un nombre a elección y su correspondiente valor.

Puede obtenerse e imprimirse en pantalla ese valor desde un plugin de esta manera:

```
<?php
add_filter( 'the_title', 'apply_title_color' );

function apply_title_color( $title ) {
    $color = get_post_meta( get_the_ID(), 'title_color', true );

    if ( $color ) {
        $title = '<span style="color: ' . $color . '">' . $title . '</span>';
    }

    return $title;
}
```

O mostrar todos los valores de campos personalizados previamente cargados de esta forma:

```
<?php
add_filter( 'the_content', 'show_post_meta' );
function show_post_meta( $content ) {
    ob_start();

    the_meta();
    $content .= ob_get_contents();

    ob_end_clean();

    return $content;
}
```

La ventaja de esta forma de gestionar campos personalizados es que es rápida. Hace falta muy poco código para mostrar nuevos datos. Es muy útil para salir de un apuro o con fines de testing, pero para desarrollo de plugins dista bastante de ser óptima. En primer lugar, porque los campos tienen que ser creados manualmente, y se requiere que el usuario

conozca el nombre de estos campos para poder ingresar información. Esto no es una muy buena práctica, porque parte de la lógica de la aplicación se manejaría como contenido, y no como código.

Por otra parte, este método también es limitante en cuanto a cómo se carga la información: no se puede modificar el formato de los campos personalizados. Su formato por defecto puede ser muy útil para ingresar texto, pero si se necesita manejar otro tipo de información, como un array, o elegir entre una cantidad limitada de opciones, los campos personalizados no alcanzan.

La solución es crear *meta boxes*. Las meta boxes son, básicamente, campos personalizados con esteroides. A la hora de guardar y obtener información funcionan casi de la misma manera que los campos personalizados, pero brindan control completo sobre el HTML que generan, por lo cual se puede crear cualquier tipo de campo para que los usuarios ingresen datos, en lugar de limitarnos a un input de texto. Además, manejan el nombre del campo internamente, por lo cual no es necesario que el usuario lo conozca a la hora de ingresar datos. También ofrecen control absoluto acerca de cómo se guarda esa información, por lo cual, si se necesita hacer algún tipo de validación de datos o correr algún tipo de proceso sobre la información ingresada antes de guardarla, puede hacerse sin problema.

Agregar meta boxes es muy sencillo, y para hacerlo es necesario asignar una función al evento `add_meta_boxes`, el cual corresponde al momento en el que WordPress registra este tipo de información.

```
<?php
add_action( 'add_meta_boxes', 'register_meta_box' );

function register_meta_box() {
    add_meta_box(
        $id,       // Atributo "id" impreso en el contenedor HTML. String.
        $title,    // Título de meta box. String.
        $callback, // Función que imprime nuestro HTML. String o función anónima.
        $screen,   // Opcional. Vista o vistas donde va a aparecer la meta box. Nombre
s de post types u objeto WP_Screen.
    );
}
```

Teniendo en cuenta estos parámetros, la función se vería similar a esta:

```php
<?php
add_action( 'add_meta_boxes', 'register_meta_box' );

function register_meta_box() {
    add_meta_box(
        'my-meta-box'
        __( 'My meta box', 'my-plugin' ),
        'my_metabox_callback',
        'post', // También son válidos 'page', array( 'post', 'page' ) y current_scree
n().
    );
}

function my_metabox_callback() {
    echo 'Hello world!'
}
```

Aunque con este código todavía no es posible cargar nueva información, ya puede verse una nueva meta box en las pantallas de edición y creación de posts. La función `add_meta_box()` también acepta otros tres parámetros opcionales más: `$context`, para definir en qué sección de la pantalla va a aparecer la meta box; `$priority`, para indicar qué prioridad tiene dentro del contexto en el que se la registra; y `$callback_args`, para enviarle información adicional a nuestro callback dentro de un array.

El siguiente paso es imprimir, a través del callback, algún tipo de información que un usuario pueda cargar. Para eso simplemente puede crearse un campo de formulario dentro de la función e imprimirlo.

Con este código puede verse un campo de tipo select dentro de la meta box:

```php
<?php
add_action( 'add_meta_boxes', 'register_meta_box' );

function register_meta_box() {
    add_meta_box(
        'my-meta-box'
        __( 'My meta box', 'my-plugin' ),
        'my_metabox_callback',
        'post', // También son válidos 'page', array( 'post', 'page' ) y current_scree
n().
    );
}

function my_metabox_callback() { ?>
    <select name="_my_selectable_option">
        <option value="value-1"><?php _e( 'Value 1', 'my-plugin' ); ?></option>
        <option value="value-2"><?php _e( 'Value 2', 'my-plugin' ); ?></option>
    </select>
<?php
}
```

Con lo que logrado hasta ahora, un usuario puede seleccionar información, pero aún no puede guardarla. Para eso vamos a necesitar una nueva acción en el evento `save_post`, donde se chequee y se guarde la información que estamos enviando.

```php
<?php
add_action( 'add_meta_boxes', 'register_meta_box' );
function register_meta_box() {
    add_meta_box(
        'my-meta-box'
        __( 'My meta box', 'my-plugin' ),
        'my_metabox_callback',
        'post', // También son válidos 'page', array( 'post', 'page' ) y current_scree
n().
    );
}

function my_metabox_callback() { ?>
    <select name="_my_selectable_option">
        <option value="value-1"><?php _e( 'Value 1', 'my-plugin' ); ?></option>
        <option value="value-2"><?php _e( 'Value 2', 'my-plugin' ); ?></option>
    </select>
<?php
}

add_action( 'save_post', 'save_my_meta_box_data' );
function save_my_meta_box_data( $post_id ) {
    $data = get_post_meta( $post_id, '_my_selectable_option', true );

    if ( empty( $data ) ) {
        add_post_meta( $post_id, '_my_selectable_option', $_POST['_my_selectable_optio
n'], true );
    } elseif ( $data != $_POST['_my_selectable_option'] ) {
        update_post_meta( $post_id, '_my_selectable_option', $_POST['_my_selectable_op
tion'] );
    }
}
```

Este código sirve para guardar la información ingresada en la meta box al guardar un post. Lo que hace es chequear si ya existen datos para el post actual con el nombre del campo creado. Si no existen, los ingresa en la base de datos; si existen, y además son distintos de los que estamos enviando al guardar, los actualiza. Sin embargo, no se está haciendo ningún chequeo de seguridad, por lo cual un usuario malicioso podría ingresar cualquier tipo de información en ese campo, provocando problemas bastante graves. Es por eso que necesitamos ser más específicos en nuestros chequeos.

```php
<?php
/**
 * Necesitamos chequear:
 *
 * - Que la información que hace falta actualizar se esté enviando.
 * - Que la actualización se esté haciendo en el momento y lugar correctos.
 * - Que el usuario tenga permisos para modificar la información.
 * - Que la información a actualizar sea válida.
```

```
 */

add_action( 'add_meta_boxes', 'register_meta_box' );
function register_meta_box() {
    add_meta_box(
        'my-meta-box'
        __( 'My meta box', 'my-plugin' ),
        'my_metabox_callback',
        'post', // También son válidos 'page', array( 'post', 'page' ) y current_scree
n().
    );
}

function my_metabox_callback() {
    wp_nonce_field( '_my_selectable_option', '_my_selectable_option_nonce' );
?>
    <select name="_my_selectable_option">
        <option value="value-1"><?php _e( 'Value 1', 'my-plugin' ); ?></option>
        <option value="value-2"><?php _e( 'Value 2', 'my-plugin' ); ?></option>
    </select>
<?php
}

add_action( 'save_post', 'save_my_meta_box_data', 10, 2 );
function save_my_meta_box_data( $post_id, $post ) {
    // Si no se reciben datos, salir de la función.
    if ( ! isset( $_POST['_my_selectable_option'] ) ) {
        return;
    }

    // Si no se aprueba el chequeo de seguridad, salir de la función.
    if ( ! isset( $_POST['_my_selectable_option_nonce'] ) || ! wp_verify_nonce( $_POST[
'_my_selectable_option_nonce'], '_my_selectable_option' ) ) {
        return;
    }

    $post_type = get_post_type_object( $post->post_type );

    // Si el usuario actual no tiene permisos para modificar el post, salir de la func
ión.
    if ( ! current_user_can( $post_type->cap->edit_post, $post_id ) ) {
        return;
    }

    // Convertimos los datos ingresados a un formato válido para nuestro campo.
    $valid_data = esc_html( $_POST['_my_selectable_option'] );

    $data = get_post_meta( $post_id, '_my_selectable_option', true );

    if ( empty( $data ) ) {
        add_post_meta( $post_id, '_my_selectable_option', $valid_data, true );
    } elseif ( $data != $_POST['_my_selectable_option'] ) {
        update_post_meta( $post_id, '_my_selectable_option', $valid_data );
```

```
    }
}
```

En el primer chequeo, en caso de que no se reciba la información esperada, simplemente se sale de la función sin guardar. El segundo chequeo es algo más complejo, pero sirve para introducir un concepto muy importante en cuestión de seguridad: los *nonces*.

Un *nonce* es, básicamente, un tipo de dato generado por WordPress que consiste en un valor numérico variable, y que permite chequear que un proceso dado se esté ejecutando en el momento y desde el lugar adecuados. Está pensado para cuando se necesita ejecutar operaciones delicadas, que requieren un cierto nivel de seguridad, como el almacenamiento de información en la base de datos.

Para poder chequear que un nonce sea válido antes de guardar datos, primero se lo debe crear. Es por eso que dentro del callback que genera el HTML de nuestra meta box va a usarse la función `wp_nonce_field()` con dos parámetros: el primero es una palabra clave que permite identificar el contexto en el que se genera el nonce, y el segundo es el nombre del dato que va a contener el valor numérico del nonce y va a llegar por medio de la variable global `$_POST` cuando se guarde la entrada.

Al chequear un nonce debe confirmarse que el dato que lo contiene está creado, lo cual hacemos con `isset()`. También se necesita chequear con `wp_verify_nonce()` que el dato numérico corresponda al contexto definido. Si alguna de estas dos cosas no pasa, se sale de la función sin guardar información. Si se pasan ambos chequeos, se continúa.

Por último, debe confirmarse que los datos que se van a ingresar tengan el formato correcto. Por ejemplo, si lo que se desea guardar en la base de datos es un array, pero recibimos un string, se debe lo debe convertir en un array; si se espera texto plano pero se recibe HTML, se debe convertir a texto plano. Este proceso se llama *sanitización*, y permite asegurarse de que un usuario malicioso no va a poder ingresar algún tipo de información que dañe la base de datos. Además, garantiza que los datos introducidos en el sistema siempre van a ser los esperables.

En el ejemplo se usa la función `esc_html()` para asegurarse de que cualquier texto que pueda ser ingresado como HTML se convierta en texto plano. Hay una cantidad bastante grande de funciones de sanitización provistas por WordPress, las cuales se pueden chequear en la documentación oficial. Algunas de esas funciones son más recomendables para mostrar datos (lo cual se llama *escape*), y otras son preferibles a la hora de guardarlos (lo cual se llama *validación*).

Custom Post Types

Una necesidad típica al construir sitios con WordPress es la de gestionar tipos de contenido diferentes de los que se incluyen por defecto. Normalmente esto se logra con la instalación de un plugin que agregue un nuevo post type. También es común encontrarse con themes que lo hagan, pero no es una práctica recomendable, salvo en casos muy específicos, ya que al cambiar el theme se perdería el nuevo post type.

Para mostrar cómo funciona este tipo de solución, vamos a crear un plugin que agregue a WordPress características de portfolio. Este plugin va permitir que un usuario cargue sus proyectos de trabajo, con una imagen asociada a cada uno de ellos, y también va a ofrecer opciones para cargar el nombre del cliente del trabajo, su fecha de finalización, y un link externo al proyecto terminado.

A partir de este momento vamos a poner en práctica una recomendación típica al trabajar con WordPress y otros CMSs: prefijar los nombres de funciones y clases con una palabra clave similar al nombre del componente que estamos creando. Nuestro plugin se va a llamar simplemente **Portfolio**, por lo cual el prefijo va a ser `portfolio_`. Hacemos esto para evitar colisiones con funciones de otros plugins, o incluso nativas de WordPress, que puedan llegar a tener el mismo nombre que las nuestras. Vale aclarar que esta es solo una de las maneras de resolver conflictos de nombres, y que a partir de PHP 5.3 podemos usar *namespaces*, pero no siempre podemos estar seguros de que nuestro plugin va a ser instalado en un servidor con PHP 5.3 o superior, por lo cual tenemos que evaluar qué es lo más conveniente. Otra opción sería decidir que nuestro plugin va a soportar una versión mínima de PHP, que puede ser 5.3 o superior. Por el momento vamos a usar prefijos, ya que es la solución más comúnmente usada.

```php
<?php
add_action( 'init', 'portfolio_create_project_post_type' );

function portfolio_create_project_post_type() {
    register_post_type( 'project', array(
            /**
             * @arg $labels
             *
             * Define etiquetas de texto utilizadas por el post type.
             */
            'labels' => array(
                'name' => __( 'Projects', 'portfolio' ),          // Nombre (en plural
) para este tipo de contenido.
                'singular_name' => __( 'Product' , 'portfolio' ), // Nombre (en singul
ar) para este tipo de contenido.
            ),
            /**
             * @arg $supports
             *
             * Define características soportadas por el post type.
             */
            'supports' => array(
                'title',     // Permite ingresar título
                'editor',    // Permite usar el editor de texto
                'author',    // Permite definir y modificar el autor
                'excerpt',   // Permite usar el campo de extracto de la entrada
                'thumbnail', // Permite ingresar imagen destacada
                'comments'   // Permite ingresar comentarios en la entrada
            ),
            'public' => true, // Hace accesibles las entradas desde el front-end.
            'has_archive' => true, // Hace accesible el sumario de entradas de este ti
po.
        )
    );
}
```

Para crear un nuevo post type, vamos a usar la función `register_post_type()` al momento de ejecutarse el evento `init`. La función va a recibir dos parámetros: el primero es el nombre interno del post type, y el segundo es un array conteniendo su descripción. Dentro de este array podemos usar una serie de valores bastante completa, a partir de los cuales nuestro post type va a contar o no con ciertas características. Todos estos valores se describen con detalle en la documentación oficial de la función, y en este momento solamente vamos a usar aquellos que nos van a permitir que nuestros proyectos se vean de la manera más similar posible a un post. Vamos a definir el nombre del post type en plural y en singular, vamos a indicar cuáles de los campos predefinidos por WordPress necesitamos, y a indicar que las entradas que generemos puedan verse desde la parte pública de nuestro sitio, y también a través de una página de archivo.

De esta manera, cuando guardemos nuestro código, vamos a ver una nueva sección en el menú de administración: ***Projects***. Podemos cargar un nuevo proyecto, con su correspondiente título, contenido, extracto, autor e imagen destacada, y una vez que lo guardemos WordPress nos va a dar un link para visualizarlo en el front-end. Si todo anduvo bien, deberíamos ver algo muy parecido a un post, quizás con algunas diferencias menores, dependiendo del theme que estemos usando.

Ahora bien, todavía no podemos ingresar el resto de los datos que dijimos que íbamos a necesitar: cliente, fecha de finalización y link externo. Para esto necesitamos agregar a nuestro plugin lo que aprendimos acerca de meta boxes. Vamos a implementar el código necesario para que puedan verse los campos que queremos agregar, y la funcionalidad necesaria para guardar esta información adicional.

```
<?php
add_action( 'add_meta_boxes', 'portfolio_register_meta_box' );

function portfolio_register_meta_box() {
    add_meta_box(
        'portfolio-meta-box'
        __( 'Project Data', 'my-plugin' ),
        'portfolio_meta_box_callback',
        'project', // Indicamos que la meta box se muestre solo para projects.
    );
}

function portfolio_meta_box_callback( WP_Post $post ) {
    wp_nonce_field( '_project_data', '_project_data_nonce' );

    $client   = get_post_meta( $post->ID, '_project_client', true );
    $end_date = get_post_meta( $post->ID, '_project_end_date', true );
    $url      = get_post_meta( $post->ID, '_project_url', true );
?>
    <div class="portfolio-project-data-field">
        <label for="_project_client"><?php _( 'Client', 'portfolio' ); ?></label>
        <input type="text" name="_project_client"><?php echo $client; ?></input>
    </div>

    <div class="portfolio-project-data-field">
        <label for="_project_end_date"><?php _( 'End Date', 'portfolio' ); ?></label>
        <input type="text" name="_project_end_date"><?php echo $end_date; ?></input>
    </div>

    <div class="portfolio-project-data-field">
        <label for="_project_url"><?php _( 'Link', 'portfolio' ); ?></label>
        <input type="text" name="_project_url"><?php echo $url; ?></input>
    </div>
<?php
}
```

```
add_action( 'save_post', 'portfolio_save_project_data', 10, 2 );

function portfolio_save_project_data( $post_id, $post ) {
    // Si no se reciben los datos esperados, salir de la función.
    if ( ! isset( $_POST['_project_client'] ) || ! isset( $_POST['_project_end_date']
) || ! isset( $_POST['_project_url'] ) ) {
        return;
    }

    // Si no se aprueba el chequeo de seguridad, salir de la función.
    if ( ! isset( $_POST['_project_data_nonce'] ) || ! wp_verify_nonce( $_POST['_proje
ct_data_nonce'], '_project_data' ) ) {
        return;
    }

    $post_type = get_post_type_object( $post->post_type );

    // Si el usuario actual no tiene permisos para modificar el post, salir de la func
ión.
    if ( ! current_user_can( $post_type->cap->edit_post, $post_id ) ) {
        return;
    }

    // Convertimos los datos ingresados a formatos válidos para nuestros campos.
    $client   = sanitize_text_field( $_POST['_project_client'] );
    $end_date = sanitize_text_field( $_POST['_project_end_date'] );
    $client   = esc_url_raw( $_POST['_project_url'] );

    // Guardamos el nombre del cliente.
    if ( empty( $client ) ) {
        add_post_meta( $post_id, '_project_client', $client, true );
    } elseif ( $client != $_POST['_project_client'] ) {
        update_post_meta( $post_id, '_project_client', $client );
    }

    // Guardamos la fecha de finalización.
    if ( empty( $end_date ) ) {
        add_post_meta( $post_id, '_project_end_date', $end_date, true );
    } elseif ( $end_date != $_POST['_project_end_date'] ) {
        update_post_meta( $post_id, '_project_end_date', $end_date );
    }

    // Guardamos el link externo.
    if ( empty( $url ) ) {
        add_post_meta( $post_id, '_project_url', $url, true );
    } elseif ( $url != $_POST['_project_url'] ) {
        update_post_meta( $post_id, '_project_url', $url );
    }
}
```

Hasta el momento, ya sabemos cómo crear un nuevo post type, cómo adjuntarle datos adicionales y cómo guardar esa información. Sin embargo, todavía no podemos ver en el front-end los datos que cargamos en nuestra meta box. Una solución posible a eso es filtrar los datos del evento `the_content`, y mostrar nuestra información al final.

```
<?php
add_filter( 'the_content', 'portfolio_custom_content' );

function portfolio_custom_content( $content ) {
    $post_id = get_the_ID();

    if ( $client = esc_html( get_post_meta( $post_id, '_project_client', true ) ) ) {
        $content .= '<strong>' . __( 'Client:', 'portfolio' ) . '</strong> ' . $client
 . '<br />';
    }

    if ( $end_date = esc_html( get_post_meta( $post_id, '_project_end_date', true ) )
) {
        $content .= '<strong>' . __( 'End Date:', 'portfolio' ) . '</strong> ' . $end_
date . '<br />';
    }

    if ( $url = esc_url( get_post_meta( $post_id, '_project_url', true ) ) ) {
        $content .= '<strong>' . __( 'Link:', 'portfolio' ) . '</strong> ' . $url . '<
br />';
    }

    return $content;
}
```

Taxonomías

Los posts cuentan con dos tipos de taxonomías para organizar contenidos: las categorías y las etiquetas (o tags). La diferencia fundamental entre estos dos tipos de taxonomías es que las categorías pueden organizarse jerárquicamente, es decir que puede haber categorías principales, y además otras secundarias que las tengan como padres, mientras que las etiquetas están siempre al mismo nivel.

El problema con el que nos encontramos al trabajar con *custom post types* es que no tenemos disponibles por defecto esas dos taxonomías para ordenar nuestros nuevos contenidos. Para hacerlo necesitamos crear taxonomías propias. Y con ese fin es que WordPress ofrece la función `register_taxonomy()`.

Usarla es muy sencillo: como primer parámetro necesitamos el nombre que va a tener nuestra nueva taxonomía (podemos usar *"Category"*, *"Tag"*, o cualquier otro que se nos ocurra); en el segundo parámetro debemos especificar el nombre del post type al que la vamos a aplicar (o un array de post types, si vamos a aplicarla a más de uno); y nuestro tercer parámetro (opcional) es una lista de argumentos con las características de la taxonomía. Entre ellos, si va a ser jerárquica o no, y una lista de textos que WordPress va a mostrar en diferentes partes del sitio web cuando se use la taxonomía. El detalle de todo lo que se puede usar dentro de estos parámetros se puede ver en la documentación oficial.

Vamos a agregar, entonces, dos taxonomías: una que se comporte de la misma manera que las categorías de posts, y otra que se comporte como las etiquetas.

```php
<?php
add_action( 'init', 'portfolio_register_category_taxonomy' );

function portfolio_register_category_taxonomy() {
    register_taxonomy( 'project_category', 'project' , array(
            'labels' => array(
                'name'          => __( 'Categories', 'portfolio' ),
                'singular_name' => __( 'Category', 'portfolio' ),
            ),
            'hierarchical' => true,
        )
    );
}

add_action( 'init', 'portfolio_register_tag_taxonomy' );

function portfolio_register_tag_taxonomy() {
    register_taxonomy( 'project_tag', 'project' , array(
            'labels' => array(
                'name'          => __( 'Tags', 'portfolio' ),
                'singular_name' => __( 'Tag', 'portfolio' ),
            ),
            'hierarchical' => false,
        )
    );
}
```

Al guardar el código y recargar la sección de administración, vamos a ver que bajo la sección *Projects* aparecen dos nuevos links: *Categories* y *Tags*. Accediendo a cada uno de ellos vamos a poder crear nuestras propias categorías y etiquetas, y asignárselas a nuestros proyectos. Estos nuevos datos que ingresamos como categorías y etiquetas son manejados por WordPress internamente como términos, o *terms*, y podemos ver la lista completa de funciones relacionadas con términos en la documentación de la función `get_term()`.

Con esto ya podemos asignar categorías y etiquetas a nuestros proyectos, pero todavía no podemos verlas en el front-end del sitio. Para resolver esto, una posible solución es filtrar la información del evento `the_content`.

```php
<?php
add_filter( 'the_content', 'portfolio_project_categories', 20 );

function portfolio_project_categories( $content ) {
    if ( taxonomy_exists( 'project_category' ) ) {
        $taxonomy = get_taxonomy( 'project_category' );
        $terms    = get_the_terms( get_the_ID(), 'project_category' );

        if ( is_array( $terms ) ) {
            $category_links = array();

             foreach ( $terms as $term ) {
                $category_links[] = '<a href="' . esc_url( get_term_link( $term ) ) .
'">' . $term->name . '</a>';
             }

            $categories = join( ', ', $category_links );

            $content .= '<div class="portfolio-project-taxonomy"><strong>' . $taxonomy[
'labels']['name'] . '</strong> ' . $categories . '</div>';
        }
    }

    return $content;
}

add_filter( 'the_content', 'portfolio_project_categories', 30 );

function portfolio_project_tags( $content ) {
    if ( taxonomy_exists( 'project_tag' ) ) {
        $taxonomy = get_taxonomy( 'project_tag' );
        $terms    = get_the_terms( get_the_ID(), 'project_tag' );

        if ( is_array( $terms ) ) {
            $tag_links = array();

             foreach ( $terms as $term ) {
                $tag_links[] = '<a href="' . esc_url( get_term_link( $term ) ) . '">'
 . $term->name . '</a>';
             }

            $tags = join( ', ', $tag_links );

            $content .= '<div class="portfolio-project-taxonomy"><strong>' . $taxonomy[
'labels']['name'] . '</strong> ' . $tags . '</div>';
        }
    }

    return $content;
}
```

Lo que hacemos en ambas funciones es chequear si la taxonomía ya existe. En caso afirmativo, obtenemos sus datos y la lista de términos que hayan sido creados bajo dicha taxonomía. A continuación construimos el código HTML que se va a agregar al contenido del proyecto, incluyendo el nombre de la taxonomía (*Category* y *Tag*) y una lista con los nombres de todos los términos creados para la misma, cada uno enlazando a su respectivas página de archivo. Una vez armada la pieza de código HTML, la agregamos al final del contenido original del post.

Parte 4: Personalización de opciones

Contenidos:

- Menús de administración
- Options API
- Settings API

Menús de administración

Es bastante corriente que nuestros plugins necesiten algún tipo de configuración global, más allá de la información de podemos manipular por medio de post types y meta boxes. Para estos casos, generalmente necesitamos crear alguna pantalla de administración propia de nuestro plugin, donde el usuario pueda configurar las opciones que definan su funcionamiento.

Con este fin es que WordPress nos ofrece tres tipos de herramientas: **menús de administración**, ***Options API*** y ***Settings API***. Cada una consiste en un conjunto de funciones que nos permiten definir cómo se van a manejar las configuraciones internas de nuestro plugin. En esta clase vamos a ver la primera, los menús de administración.

Los menús de administración son esas secciones que vemos en la barra lateral de navegación en nuestro panel de administración. Cada uno de ellos tiene un link principal, y además pueden tener un segundo nivel de links, que vemos cuando estamos navegando un menú o cuando pasamos el mouse por encima del link principal. Estos links secundarios son lo que llamamos sub-menús.

Por último tenemos las páginas de opciones, que son aquellos links que vemos ubicados debajo del menú *Settings* (o *Configuración*).

Para agregar un menú usamos la función `add_menu_page()`, y la asignamos como acción al evento `admin_menu`. Con este ejemplo de código podemos crear un menú y asociarle una interfaz muy básica.

```php
<?php
add_action( 'admin_menu', 'portfolio_menu_page' );

function portfolio_add_menu_page() {
    add_menu_page(
        __( 'Portfolio Settings', 'portfolio' ), // Texto del título que aparece en la
 página de opciones.
        __( 'Portfolio', 'portfolio' ),          // Texto que aparece en el link princ
ipal del menú.
        'manage_options',                        // Permiso que debe tener el usuario
para ver el menú.
        'portfolio-settings',                    // Slug, string que permite identific
ar internamente el menú.
        'portfolio_menu_page',                   // Nombre de la función que imprime e
l HTML de la página de opciones.
        'dashicons-art',                         // Icono del menú. Podemos usar Dashi
cons o una URL.
        10,                                      // Posición en la que aparece el menú
 en la barra de navegación.
    );
}

function portfolio_menu_page(){
    _e( 'Set Portfolio options here.', 'portfolio' );
}
```

Cuando guardemos nuestro código y refresquemos el browser, vamos a ver que contamos con un nuevo menú de administración. Haciendo click en el link del menú, vamos a ver el texto que imprimimos dentro de nuestra función callback.

También podemos, en lugar de crear un nuevo menú para nuestra página de opciones, ubicar un link debajo de Settings. Esto es lo que hacen muchos desarrolladores de plugins para que la barra de navegación no se vea tan larga.

El procedimiento es muy similar a agregar un menú, pero usando la función `add_options_page()`. Simplemente reemplazamos el código anterior por este:

```
<?php
add_action( 'admin_menu', 'portfolio_menu_page', 999 );

function portfolio_add_menu_page() {
    add_options_page(
        __( 'Portfolio Settings', 'portfolio' ), // Texto del título que aparece en la
 página de opciones.
        __( 'Portfolio', 'portfolio' ),          // Texto que aparece en el link princ
ipal del menú.
        'manage_options',                        // Permiso que debe tener el usuario
para ver el menú.
        'portfolio-settings',                    // Slug, string que permite identific
ar internamente el menú.
        'portfolio_menu_page',                   // Nombre de la función que imprime e
l HTML de la página de opciones.
    );
}

function portfolio_menu_page(){
    _e( 'Set Portfolio options here.', 'portfolio' );
}
```

Cuando refresquemos el browser después de guardar, la ubicación de nuestro link va a haber cambiado, pero vamos a seguir viendo la misma página. Lo único que no podemos desde definir desde `add_options_page()` es un ícono, ya que los sub-menús y links páginas de opciones no tienen ninguno, ni la posición en la que va a aparecer el link. Para la posición podemos jugar un poco con las prioridades de `add_action()` hasta que el link se vea donde pretendemos. En el ejemplo hay una prioridad alta (999) para que aparezca lo más cerca posible del final de la lista.

Una tercera alternativa, más ordenada, es ubicar el link a nuestra página de opciones dentro del menú que se creó automáticamente para nuestro post type. Como ya vimos al agregar un menú principal, todo menú cuenta con un slug o palabra clave para que pueda ser reconocido internamente, y para agregar un sub-menú necesitamos conocer el slug del menú principal.

En el caso de los post types, el nombre del slug es un poco más rebuscado que el de los que venimos definiendo, pero es fácilmente reconocible: `edit.php?post_type={nombre_del_post_type}`. En nuestro caso, como nuestro post type es `project`, el slug del menú va a ser `edit.php?post_type=project`. Teniendo esto en cuenta, la forma de registrar un sub-menú es prácticamente igual al ejemplo anterior, pero usando la función `add_submenu_page()`.

```
<?php
add_action( 'admin_menu', 'portfolio_menu_page', 999 );

function portfolio_add_menu_page() {
    add_submenu_page(
        'edit.php?post_type=project',            // Slug del menú principal.
        __( 'Portfolio Settings', 'portfolio' ), // Texto del título que aparece en la
 página de opciones.
        __( 'Portfolio Settings', 'portfolio' ), // Texto que aparece en el link princ
ipal del menú.
        'manage_options',                        // Permiso que debe tener el usuario
para ver el menú.
        'portfolio-settings',                    // Slug, string que permite identific
ar internamente el menú.
        'portfolio_menu_page',                   // Nombre de la función que imprime e
l HTML de la página de opciones.
    );
}

function portfolio_menu_page(){
    _e( 'Set Portfolio options here.', 'portfolio' );
}
```

Al refrescar el browser van a ver que el link cambió nuevamente de ubicación, y ahora está en el último lugar del menú de *Projects*.

Algo importante a tener en cuenta es que, si estamos mirando la página de administración que acabamos de crear, al cambiar el link de lugar y refrescar la página, WordPress puede mostrar un error. Esto no es porque hayamos hecho algo mal, sino porque al cambiar la ubicación del link también puede cambiar la URL que se le asigna, pero con volver a la página principal del panel de administración vamos a poder ver el link en su nueva ubicación, y acceder a él desde su nueva URL sin problemas.

Options API

Ya vimos cómo crear una página de administración para nuestro plugin, pero todavía no podemos guardar opciones a través de esa página. Para poder gestionar nuestras opciones, WordPress nos ofrece la Options API, un conjunto de funciones muy sencillas para manipular configuraciones internamente en nuestros plugins.

Para poder ingresar información desde nuestra página de opciones, primero tenemos que ofrecer algún campo con el que el usuario quiera interactuar. Por el momento vamos a agregar una única opción: la de mostrar proyectos solo a usuarios registrados. Para eso, vamos a retomar la función que usamos para mostrar la página de administración:

```
<?php
function portfolio_menu_page() {
    $settings = get_option( 'portfolio_settings' );
    $checked  = isset( $settings['show_logged_only'] ) ? $settings['show_logged_only']
 : false;

    wp_nonce_field( '_portfolio_settings', '_portfolio_settings_nonce' );

    ?>
        <form method="post">
            <input id="portfolio-settings-show-logged-only" name="portfolio_settings[s
how_logged_only]" type="checkbox" value="true" <?php checked( $checked ); ?>/>
            <label for="portfolio-settings-show-logged-only"><?php _e( 'Only show proj
ects to logged-in users', 'portfolio' ); ?></label>

            <input type="submit" class="button button-save" value="<?php _e( 'Save opt
ions', 'portfolio' ); ?>" />
        </form>
    <?php
}
```

Al refrescar la página vamos a ver aparecer la opción. Noten como, en las dos primeras líneas de esta función, chequeamos si nuestro plugin guardó datos previamente, y a partir de eso decidimos si el checkbox tiene que mostrarse tildado o no. También creamos el nonce que nos va a permitir validar los datos cuando se envíen.

Ahora tenemos que ver cómo se guardan esos datos. El proceso es muy parecido a lo que hacíamos con las meta boxes.

```
<?php
add_action( 'admin_init', 'portfolio_save_settings' );
```

```
function portfolio_save_settings() {
    // Si no se aprueba el chequeo de seguridad, salir de la función.
    if ( ! isset( $_POST['_portfolio_settings_nonce'] ) || ! wp_verify_nonce( $_POST['
_portfolio_settings_nonce'], '_portfolio_settings' ) ) {
        return;
    }

    // Si el usuario actual no tiene permisos para actualizar la configuración, salir
de la función.
    if ( ! current_user_can( 'manage_options' ) ) {
        return;
    }

    // Chequeamos si el dato se envió para decidir si mostramos los proyectos a todos
usuarios o no.
    if ( isset( $_POST['portfolio_settings']['show_logged_only'] ) ) {
        // Si el valor se envío, chequeamos si tiene un valor permitido. Si lo tiene,
mostramos proyectos solo a usuarios registrados.
        $accepted_values = array( true, 'true' );
        $show_logged_only = in_array( $_POST['portfolio_settings']['show_logged_only']
, $accepted_values );
    } else {
        // Si el valor no se envió, mostramos proyectos a todos los usuarios.
        $show_logged_only = false;
    }

    // Creamos un flag para evaluar si ya tenemos la configuración guardada en al base
 de datos.
    $settings_empty = false;

    // Obtenemos la configuración del plugin, si existe.
    $settings = get_option( 'portfolio_settings' );

    // Si no hay una configuración previa guardada en la base de datos, lo indicamos,
y creamos un array vacío para contenerla.
    if ( empty( $settings ) ) {
        $settings_empty = true;
        $settings = array();
    }

    // Si el valor de la configuración que estamos modificando no existe o es diferent
e al que evaluamos, lo cambiamos por el nuevo valor.
    if ( ! isset( $settings['show_logged_only'] ) || $settings['show_logged_only'] !=
$show_logged_only ) {
        $settings['show_logged_only'] = $show_logged_only;
    }

    if ( $settings_empty ) {
        // Si la configuración está vacía, la creamos.
        add_option( 'portfolio_settings', $settings );
    } else {
        // Si la configuración no está vacía, la actualizamos.
        update_option( 'portfolio_settings', $settings );
```

```
    }
}
```

Con estos chequeos y el uso de `add_option()` y `update_option()` podemos crear y modificar la configuración de nuestro plugin. Es importante tener en cuenta que cada opción manejada por estas funciones genera una nueva entrada en la base de datos, y si para cada configuración que necesitemos en nuestro plugin creamos una entrada nueva, estamos complejizando la base de datos, lo cual, a la larga, puede llevar a problemas de rendimiento para sitios web con bases de datos grandes. Por eso lo que hacemos acá, y lo que cada vez se recomienda más, es usar una sola opción para guardar nuestra configuración, e incluir todos los datos que necesitamos dentro de un array.

Ahora bien, necesitamos ver cómo reflejar en el front-end estos cambios que introdujimos. Para eso vamos a volver a asignar una función a `the_content`, como en los ejemplos anteriores, pero en vez agregarle cosas al contenido, lo vamos a modificar por completo en caso de que el usuario no esté logueado, y que solo queramos mostrar proyectos a usuarios registrados. Para eso vamos a usar una prioridad con un número alto, y así asegurarnos de que la función se ejecute lo más tarde posible.

```
<?php
add_filter( 'the_content', 'portfolio_project_show_logged_in_only', 999 );

function portfolio_project_show_logged_in_only( $content ) {
    $settings = get_option( 'portfolio_settings' );

    // Si se elige mostrar solo a usuarios registrados y el usuario actual no está log
ueado, reemplazamos el contenido por un mensaje.
    if ( ! empty( $settings['show_logged_in_only'] ) && ! is_user_logged_in() ) {
        $content = __( 'You don\'t have permissions to view this content.', 'portfolio'
 );
    }

    return $content;
}
```

De todas maneras, si bien la Options API es la herramienta que WordPress usa internamente y recomienda para manejar configuraciones propias, a la hora de manejar el guardado de datos y generación de campos en menús de administración, no es la mejor solución que tenemos a nuestro alcance, ya que tiene una gran cantidad de limitaciones. Es importante entender cómo funciona y cuáles son sus funciones principales, porque en desarrollo de plugins es casi inevitable usarla, pero para lo que acabamos de hacer, la mejor solución que nos ofrece WordPress es la ***Settings API***, que es el tema del próximo capítulo.

Settings API

Ya vimos cómo se maneja la *Options API* para guardar la configuración de nuestro plugin, y recomendamos usar una única opción para guardar todos nuestros datos, construyendo un array. También vimos que para guardar nuestra configuración tenemos que hacer una serie de chequeos de seguridad que pueden llegar a ser tediosos, o pueden volver nuestras funciones muy largas y complicadas de mantener.

Todos estos problemas se pueden resolver usando la *Settings API*. Lo que nos permite esta API es sacarnos de encima gran parte de los procesos de seguridad y validación que necesitamos para guardar nuestros datos, y en consecuencia vuelve nuestro código más estable, más sencillo de mantener, y más extensible.

Una diferencia que vamos a tener con respecto a lo visto anteriormente es que no vamos a imprimir de manera directa el HTML que genera nuestros campos. En cambio, nuestros campos tienen que ser registrados en la *Settings API*, y asignados a una sección.

```
<?php
add_action( 'admin_init', 'portfolio_settings_api_init' );

function portfolio_settings_api_init() {
     // Registramos una sección para nuestro campo.
     add_settings_section(
        'portfolio-general-settings-section',          // Texto del tag `id` de la sec
ción.
        __( 'General Settings', 'portfolio' ),         // Título de la sección.
        'portfolio_general_settings_section_callback', // Nombre de la función que imp
rime el HTML de la sección.
        'portfolio-settings'                           // Slug del menú donde debe apa
recer la sección.
    );

     // Registramos un campo asociado a la sección.
     add_settings_field(
        'portfolio-settings-show-logged-only-setting',             // Texto del tag `
id` del campo.
        __( 'Only show projects to logged-in users', 'portfolio' ), // Título del camp
o.
        'portfolio_settings_show_logged_only_callback',            // Nombre de la fu
nción que imprime el HTML del campo.
        'portfolio-settings',                                      // Slug del menú d
onde debe aparecer el campo.
        'portfolio-general-settings-section'                       // ID de la secció
n a la que pertenece el campo.
    );
```

```
     // Registramos nuestro campo como setting.
    // El primer parámetro es el nombre de la opción que estamos usando para guardar l
a configuración.
    // El segundo parámetro es el identificador de nuestra setting dentro del array de
 opciones.
    // El tercer parámetro es el nombre de la función que va a sanitizar los datos de
la opción.
    // Esta función se ocupa de validar nuestros datos por medio de nonces.
     register_setting( 'portfolio_settings', 'show_logged_only', 'portfolio_sanitize_s
how_logged_only' );
}

// Callback para la sección.
function portfolio_general_settings_section_callback() {
    _e( 'Configure the general plugin settings here' );
}

// Callback para el campo. Solamente imprimimos el input y su label, no el tag form ni
 el botón de confirmación.
function portfolio_settings_show_logged_only_callback() {
    $settings  = get_option( 'portfolio_settings' );
    $checked = isset( $settings['show_logged_only'] ) ? $settings['show_logged_only']
: false;

    ?>
        <input id="portfolio-settings-show-logged-only" name="portfolio_settings[show_
logged_only]" type="checkbox" value="true" <?php checked( $checked ); ?>/>
        <label for="portfolio-settings-show-logged-only"><?php _e( 'Only show projects
 to logged-in users', 'portfolio' ); ?></label>
    <?php
}

// Función de sanitización para nuestro campo.
function portfolio_sanitize_show_logged_only( $data ) {
    // Chequeamos si el valor ingresado está permitido. Si lo está, mostramos proyecto
s solo a usuarios registrados.
    $accepted_values = array( true, 'true' );
    $show_logged_only = in_array( $data, $accepted_values );

    return $show_logged_only;
}
```

Con este código ya podemos guardar nuestros datos automáticamente, sin necesidad de interactuar de manera directa con la *Options API* ni de chequear nonces. Sin embargo, todavía no podemos usarlo, porque lo que hicimos aún no se ve reflejado en la página de administración. Para eso necesitamos modificar nuestra función `portfolio_menu_page()` de la siguiente manera:

```
<?php
function portfolio_menu_page() {
    ?>
        <form action="options.php" method="POST">
            <?php settings_fields( 'portfolio_settings' ); // Imprime el HTML necesari
o para las validaciones. ?>
            <?php do_settings_sections( 'portfolio-general-settings-section' ); // Imp
rime nuestra sección y los campos asociados. ?>
            <?php submit_button(); // Imprime el botón de confirmación. ?>
        </form>
    <?php
}
```

Una vez que actualizamos a este código, pasan varias cosas: nuestra antigua función de guardado, `portfolio_save_settings()` ya no se usa más, ya que todas las validaciones son hechas por la Settings API, y podemos eliminarla tranquilos. También contamos con más funciones que antes, pero más chicas y más fáciles de mantener. Además, la sanitización de datos está desacoplada del proceso de guardado. Como toda la registración de secciones y de campos es manejada internamente por WordPress, con mirar un poco la documentación oficial de la API y sus otras funciones podemos ver que es muy fácil mostrar o no mostrar nuevas opciones condicionalmente, según el contexto en el que estemos, qué usuario está viendo la página, en qué menú estamos, etc. Es una mejora sustancial sobre el método anterior, con el que las cosas no se podían hacer de manera tan automática.

Parte 5: Prácticas recomendadas

Contenidos:

- Estilos y scripts en menús de administración
- Pluggable functions
- By-Passing
- Must-Use Plugins
- Recomendaciones finales

Carga de estilos y scripts en menú de administración

Un escenario muy común en el desarrollo de plugins es que, una vez que creamos nuestras propias pantallas de opciones, queramos modificar sus estilos por medio de CSS, o su comportamiento por medio de JavaScript. Para eso vamos a necesitar incluir los archivos en los que tengamos nuestros estilos y nuestros scripts.

Por ejemplo, podemos tener un archivo CSS que aplique un borde rojo a todos los elementos input que sean del tipo "text":

```
input[type="text"] {
    border: 1px solid red;
}
```

Y también un archivo JavaScript que ocasione que, al hacer click en un botón de envío de formulario, pida una confirmación.

```
jQuery( 'form' ).submit( function() {
    var confirmed = confirm( 'Please confirm form submission' );
    return confirmed;
} );
```

Si ya hicieron los cursos de desarrollo de themes para WordPress, habrán visto que estos archivos pueden cargarse desde el front-end con las funciones `wp_register_style()`, `wp_register_script()`, `wp_enqueue_style()` y `wp_enqueue_script()`, usándolas dentro de una función asignada al evento `wp_enqueue_scripts`. Un caso de uso de estas funciones sería este:

```
<?php
add_action( 'wp_enqueue_scripts', 'portfolio_enqueue_style' );

function portfolio_enqueue_style() {
    wp_register_style( 'portfolio-admin-styles', plugins_url( 'css/style.css', false )
;
    wp_enqueue_style( 'portfolio-admin-styles' );
}

add_action( 'wp_enqueue_scripts', 'portfolio_enqueue_script' );

function portfolio_enqueue_script() {
    wp_register_script( 'portfolio-admin-scripts', plugins_url( 'css/scripts.js', false
 );
    wp_enqueue_script( 'portfolio-admin-scripts' );
}
```

Este código va a hacer que, en el HTML generado por WordPress, se impriman los tags `<link>` y `<script>` que se ocupan de cargar nuestros archivos CSS y JavaScript. Sin embargo, esta acción `wp_enqueue_scripts` solamente tiene efecto cuando estamos en el front-end de nuestro sitio, y no en la parte administrativa. Sin embargo, existe otro evento similar que se ejecuta solamente en la sección de administración, llamado `admin_enqueue_scripts`. Para aplicarlo, podemos reformular nuestro ejemplo de la siguiente manera:

```
<?php
add_action( 'admin_enqueue_scripts', 'portfolio_enqueue_style' );

function portfolio_enqueue_style() {
    wp_register_style( 'portfolio-admin-styles', plugins_url( 'css/style.css', false )
;
    wp_enqueue_style( 'portfolio-admin-styles' );
}

add_action( 'admin_enqueue_scripts', 'portfolio_enqueue_script' );

function portfolio_enqueue_script() {
    wp_register_script( 'portfolio-admin-scripts', plugins_url( 'css/scripts.js', false
 );
    wp_enqueue_script( 'portfolio-admin-scripts' );
}
```

Una vez que pasamos a usar este evento para cargar nuestros archivos, vamos a ver que los tags se imprimen correctamente en el HTML, y que los estilos y el comportamiento se asignan a a los elementos a los que apuntamos desde nuestros archivos CSS y JavaScript.

Un potencial problema de esto que acabamos de hacer es que esos archivos que cargamos van a aparecer en todas las páginas que visitemos dentro de la sección de administración, incluso cuando no tengan ningún efecto sobre lo que se ve en pantalla. Si los archivos son muy largos, van a tardar más tiempo que archivos más cortos en ser descargados. Y si no necesitamos que siempre se carguen, el hecho de que aparezcan siempre puede causar problemas sobre la performance del sitio donde se instaló nuestro plugin. Por eso, una recomendación muy útil para cuando estamos trabajando con hojas de estilos o scripts muy largos, y que por ende van a llevar más tiempo para ser descargados que archivos más cortos, es solamente cargarlos en las páginas en las que sea necesario usarlos. Para poder hacer esto, WordPress suele ofrecer herramientas que nos permiten conocer el contexto de la página que estamos viendo. Una de ellas es la función `current_screen()`.

Usando `current_screen()` podemos obtener bastante información acerca de la página que estamos viendo. Si solamente queremos cargar estos archivos en la página de administración que creamos en las clases anteriores, lo que tenemos que hacer es usar `current_screen()` para asegurarnos de que la página sea la indicada antes de cargar los archivos, modificando nuestro código de esta forma:

```
<?php
add_action( 'admin_enqueue_scripts', 'portfolio_enqueue_style' );

function portfolio_enqueue_style() {
    if ( 'portfolio-settings' != current_screen()->id ) {
        return;
    }

    wp_register_style( 'portfolio-admin-styles', plugins_url( 'css/style.css', false )
;
    wp_enqueue_style( 'portfolio-admin-styles' );
}

add_action( 'admin_enqueue_scripts', 'portfolio_enqueue_script' );

function portfolio_enqueue_script() {
    if ( 'portfolio-settings' != current_screen()->id ) {
        return;
    }

    wp_register_script( 'portfolio-admin-scripts', plugins_url( 'css/scripts.js', false
 );
    wp_enqueue_script( 'portfolio-admin-scripts' );
}
```

Pluggable Functions

Una característica que nos ofrece WordPress es la de reservar ciertos nombres de funciones para que podamos utilizarlos para reemplazar funcionalidad nativa por la nuestra. A estas funciones, en la terminología propia de WordPress, nos referimos como ***Pluggable Functions***.

Estas funciones tienen la característica de ser definidas recién después de que todos los plugins se hayan cargado, y chequean si la función ya se declaró antes, usando `function_exists()`. Lo que hacen es algo como esto:

```
<?php
if ( ! function_exists( 'wp_mail' ) ) :
function wp_mail( $args... ) {
    // Code here…
}
endif;
```

Esto permite que podamos definir nuestras propias funciones con ese nombre antes de que WordPress lo haga.

```
<?php
// En nuestro plugin…

function wp_mail( $args... ) {
    // Code here…
}

// En WordPress…

if ( ! function_exists( 'wp_mail' ) ) :
function wp_mail( $args... ) {
    // Code here…
}
endif;
```

Si hacemos esto, cada vez que se llame a la función `wp_mail()` se va a ejecutar nuestro código para esa función en lugar del código de WordPress.

Esta posibilidad de redefinir funciones nativas es extremadamente útil para plugins que necesitan llevar a cabo sus propios procesos de envío de mails, validación de usuarios, generación de contraseñas, etc.

En nuestros plugins también podemos crear nuestras propias *Pluggable Functions*, para permitir a otros desarrolladores que redefinan funcionalidad de nuestros plugins cuando lo necesiten.

```
<?php
// En un plugin de terceros...
function my_function() {
    _e( 'Hello John!', 'third-party-plugin' );
}

// En nuestro plugin…
if ( ! function_exists( 'my_function' ) ) :
function my_function() {
    _e( 'Hello world!', 'my-plugin' );
}
endif;
```

Este mismo principio de extensibilidad es aplicable a clases, por medio de `class_exists()`:

```
<?php
// En un plugin de terceros…
class My_Class {
    public function __construct() {}
}

// En nuestro plugin…
if ( ! class_exists( 'My_Class' ) ) :
class My_Class {
    public function __construct() {}
}
endif;
```

Hay que tener en cuenta que, si alguien quiere redefinir nuestras funciones, tiene que encargarse de declarar las suyas antes de que se declaren las nuestras. Como no podemos estar seguros del orden en el que WordPress va a cargar los archivos de plugins que vaya encontrando, lo que se puede hacer es definir nuestras *Pluggable Functions* dentro de un evento, para que otro desarrollador pueda usar ese mismo evento e insertar sus propias funciones especificando una prioridad más alta.

```php
<?php
#my-plugin.php

add_action( 'plugins_loaded', 'load_pluggable_functions', 10 );

function load_pluggable_functions() {
    require( dir( __FILE__ ) . '/my-pluggable-functions.php' );
}

#my-pluggable-functions.php

if ( ! function_exists( 'my_function' ) ) :
function my_function() {
    _e( 'Hello world!', 'my-plugin' );
}
endif;

#third-party-plugin.php

add_action( 'plugins_loaded', 'load_third_party_functions', 1 );

function load_third_party_functions() {
    require( dir( __FILE__ ) . '/third-party-functions.php' );
}

#third-party-functions.php

function my_function() {
    _e( 'Hello world!', 'my-plugin' );
}
```

De esta forma, otro desarrollador puede usar el mismo evento que usamos nosotros para cargar sus funciones, pero al cargarse su archivo de funciones con una prioridad más alta que el nuestro, las suyas se van a declarar primero, y las nuestras van a ser ignoradas, al no pasar el chequeo de `if( ! function_exists() )`.

Si bien esta es una práctica muy útil a la hora de hacer nuestros plugins extensibles, tampoco tenemos que abusar de ella. Va a haber muchas ocasiones en las que no queremos que nuestras funciones puedan ser redeclaradas en plugins de terceros, y solamente deberíamos permitirlo con aquellas que sepamos que alguien puede llegar a necesitar personalizar.

Funciones extensibles (By-Pass)

El *by-passing* es una práctica que permite crear una vía alternativa para la ejecución del proceso interno de una función. A diferencia de las funciones extensibles, en este caso no se reemplaza una función por otra, sino que se ingresa a la función original, se ejecuta un proceso diferente, y se detiene la ejecución del resto del código. En WordPress podemos lograrlo utilizando filtros.

```
<?php
// Función original:

function my_original_function() {
  if ( apply_filters( 'my_original_function_hook', false ) ) {
    return; // Se detiene la ejecución de la función en caso de que el filtro resuelva
 "true".
  }

  echo 'Hola mundo!';
}

// Función de by-pass:

function my_by_pass_function() {
  echo 'Chau mundo!';

  return true; // Devolvemos "true". Esto es importante para que la ejecución de la fu
nción original se detenga.
}

// Seteamos el filtro de by pass:
add_filter( 'my_original_function_hook', 'my_by_pass_function' );

// Llamamos a la función original:
my_original_function(); // Se imprime 'Chau mundo!'
```

Must-Use Plugins

Existe un segundo tipo de plugins, que no es tan popular como el que estuvimos viendo hasta ahora. Son plugins que no necesitan activación, y que van a estar siempre activos, siempre y cuando se encuentren en una ubicación específica de la instalación de WordPress.

Estamos hablando de los ***Must-Use Plugins***, también conocidos como `mu-plugins`. Al igual que los plugins comunes y corrientes, se trata de archivos PHP que van a estar incluidos en la carpeta `wp-content/mu-plugins`. Esta carpeta no viene pre-creada en las instalaciones de WordPress, sino que tenemos que crearla a mano. Una vez que la tenemos, podemos empezar a incluir archivos PHP dentro de ella.

Los `mu-plugins` tienen algunas diferencias con los plugins comunes, además de su ubicación. La primera diferencia notoria es que no necesitan un encabezado correctamente formateado para ser reconocidos como plugins. WordPress reconoce el encabezado cuando lo encuentra, pero no es un requisito obligatorio. Simplemente con tener un archivo con extensión PHP dentro del directorio `wp-content/mu-plugins`, WordPress lo reconoce como plugin. Podemos ver la lista de `mu-plugins` que tenemos instalados reflejada en la misma página de plugins de la sección de administración.

Otra diferencia importante es que los `mu-plugins` no pueden estar en subcarpetas, como sí pueden los plugins comunes. Podemos usar archivos que estén en esas subcarpetas, pero necesitamos cargarlos con un `include` o `require` desde el archivo principal.

Debido a estas diferencias, no todos los plugins normales pueden funcionar como `mu-plugins`, es decir que no siempre se va a dar el caso de que un plugin funcione correctamente si se lo mueve a la carpeta `mu-plugins` y se incluye su archivo principal desde un `mu-plugin`. Esto sucede porque los `mu-plugins` están pensados para ser utilizados como archivos que introduzcan muy poca funcionalidad, y que generalmente modifiquen o impongan algún tipo de comportamiento dentro del sitio que no pueda ser modificado o desactivado por nadie que no tenga acceso a los archivos. Es una práctica muy usada por desarrolladores que trabajan para usuarios finales, ya que les sirve para introducir procesos personalizados específicos de cada proyecto.

Recomendaciones finales

Dominando los contenidos que estuvimos viendo durante todo el libro, el lector debería estar para empezar a programar plugins por su cuenta. De todas formas, para que este trabajo sea más cómodo y más eficiente, puede seguirse una serie de recomendaciones que suele hacer una gran parte de los desarrolladores que trabajan con WordPress.

En primer lugar está el uso del Codex, la documentación oficial de WordPress donde pueden encontrar información bastante detallada acerca de funciones, clases, filtros, acciones, APIs, y prácticamente cualquier herramienta de código que ofrezca WordPress, incluyendo ejemplos de uso.

Además del Codex, WordPress también brinda el Plugin Developer Handbook, una especie de manual de referencia básico acerca de las prácticas más recomendadas a la hora de desarrollar plugins.

También vamos a contar con Coding Guidelines para PHP, HTML, CSS y JavaScript. Se trata de una serie de documentos en los que se recomienda seguir ciertos estándares de formato a la hora de escribir código, para que lo que hacemos le resulte más legible y familiar a la mayor cantidad posible de desarrolladores.

Por último, una gran recomendación para aquellos que trabajan con WordPress es colaborar con otros desarrolladores de plugins o de themes. Dado que el código de los plugins que se suben al directorio oficial de WordPress es público, cualquiera puede revisarlo, sugerir modificaciones y mejorar el producto. También pueden involucrarse en el desarrollo del Core de WordPress, resolviendo bugs, encargándose de nueva funcionalidad, ayudando a hacer traducciones, actualizando documentación, etc. Incluso pueden ayudar con consultas de soporte en los foros oficiales o de sitios relacionados. Las formas de colaborar son muchísimas, las pueden ver listadas en WordPress.org, y son algo a tener muy en cuenta, porque al colaborar no solo se aprende muchísimo acerca de WordPress, sino que además se pueden llegar a crear relaciones de trabajo muy interesantes con otras personas.