Creando una capa de abstracción con PHP y mysqli

PHP

mysqli, es el conector para bases de datos MySQL recomendado por PHP, para interactuar desde tu aplicación con una base de datos MySQL. Pero crear una capa de abstracción genérica, reutilizable y orientada a objetos, suele ser un dolor de cabeza. En este artículo, veremos como lograr crear una capa de abstracción a bases de datos, simple y muy fácil de usar.

Escrito por: **Eugenia Bahit** (Arquitecta GLAMP & Agile Coach)

Eugenia es **Arquitecta de Software**, **docente** instructora de tecnologías **GLAMP** (GNU/Linux, Apache, MySQL, Python y PHP) y **Agile coach** (UTN) especializada en Scrum y eXtreme Programming. Miembro de la **Free Software Foundation** e integrante del equipo de **Debian Hackers**.

Webs:
Cursos de programación a Distancia: www.cursosdeprogramacionadistancia.com
Agile Coaching: www.eugeniabahit.com

Redes sociales:
Twitter / Identi.ca: **@eugeniabahit**

Si alguna vez intentaste crear una capa de abstracción a bases de datos, utilizando el conector mysqli en estilo orientado a objetos, sabrás que hacerlo es un gran dolor de cabeza.

Pero **¿qué es exactamente mysqli y en qué se diferencia de mysql?** Básicamente, como bien se define en el manual oficial de PHP, mysqli es **"una extensión mejorada del conector mysql"**. Entre las principales diferencias, se encuentran además, sus dos grandes ventajas:

- Permite trabajar en estilo orientado a objetos (también continúa proveyendo soporte para estilo procedimental);

- Nos facilita una forma segura de filtrado de datos en sentencias SQL, para prevenir inyecciones SQL;

Sin dudas, mysqli es una gran ventaja frente al antiguo conector. Tiene una gran cantidad de clases, métodos, constantes y propiedades muy bien documentados[1]. Sin embargo, entender la documentación puede ser una tediosa tarea, en la cual, hallar un principio y un fin, se podrá convertir en la peor pesadilla de tu vida.

Así que entonces ¡Manos a la obra! Y a crear una capa de abstracción con mysqli orientado a objetos.

Recetario para crear una capa de abstracción a bases de datos con mysqli

Lo primero que debemos tener en cuenta, es que nuestra capa de abstracción deberá proveer de métodos públicos, que puedan ser llamados de forma estática, para crear un objeto conector, no sea necesario.

Para poder lograr una capa de abstracción genérica, **la clave** es utilizar **`ReflectionClass`**[2] para crear una instancia de **`mysqli_stmt`** y mediante `ReflectionClass->getMethod`, invocar al método **`bind_param`**. De lo contrario, preparar una consulta SQL y enlazarle los valores a ser filtrados, será imposible.

> *Ten en cuenta que para seguir los ejemplos de este artículo, es necesario contar con la **versión 5.3.6** o superior, de **PHP**.*

Propiedades

Nuestra capa de abstracción, tendrá una única propiedad pública, destinada a almacenar los datos obtenidos tras una consulta de selección. El resto de las propiedades, serán de ámbito protegido, accesibles solo desde nuestra clase y clases que hereden de ésta.

```
class DBConnector {

    protected static $conn;        # Objeto conector mysqli
    protected static $stmt;        # preparación de la consulta SQL
    protected static $reflection;  # Objeto Reflexivo de mysqli_stmt
    protected static $sql;         # Sentencia SQL a ser preparada
    protected static $data;        # Array conteniendo los tipo de datos
                                     más los datos a ser enlazados (será
                                     recibido como parámetro)
```

1 http://php.net/manual/es/book.mysqli.php
2 http://php.net/manual/es/class.reflectionclass.php

```
    public static $results;        # Colección de datos retornados por una
                                     consulta de selección

}
```

La consulta SQL, deberá ser seteada en los modelos (clases) donde se requiera, incluyendo marcadores de parámetros (embebidos con el signo ?), en la posición donde un dato deba ser enlazado. Un ejemplo de ella, podría ser el siguiente:

```
$sql = "INSERT INTO productos (categoria, nombre, precio)
        VALUES (?, ?, ?)";
```

Mientras que el array `$data`, deberá contener, como primer elemento, una string con el tipo de datos y los elementos siguientes, serán los datos a ser enlazados (todos de tipo *string*):

```
$data = array("isbd",
              "{$categoria}", "{$nombre}", "{$descripcion}", "{$precio}");
```

El primer elemento, siempre representará el tipo de datos correspondiente al marcador de parámetro que se desea enlazar. Siendo los tipos de datos posibles: ***s*** (string), ***i*** (entero), ***d*** (doble) y ***b*** (blob).

Métodos

Conectar a la base de datos:

```
    protected static function conectar() {
        self::$conn = new mysqli(DB_HOST, DB_USER, DB_PASS, DB_NAME);
    }
```

Requerirá 4 constantes predefinidas (se recomienda definir en un archivo `settings`): `DB_HOST`, `DB_USER`, `DB_PASS` y `DB_NAME`.

Preparar una sentencia SQL (con marcadores de parámetros):

```
    protected static function preparar() {
        self::$stmt = self::$conn->prepare(self::$sql);
        self::$reflection = new ReflectionClass('mysqli_stmt');
    }
```

La clase `ReflectionClass` de PHP, cumple un papel fundamental: solo a través de ella podemos crear un objeto `mysqli_stmt` reflexivo, siendo ésta, la única alternativa que tenemos para preparar sentencias SQL con marcadores de parámetros, de forma dinámica.

La propiedad estática `$sql` (`self::$sql`) será creada por el único método público que tendrá la clase.

Enlazar los datos con la sentencia SQL preparada:

```
protected static function set_params() {
    $method = self::$reflection->getMethod('bind_param');
    $method->invokeArgs(self::$stmt, self::$data);
}
```

La propiedad estática `$data` que se pasa como segundo parámetro a invokeArgs, también será seteada por el único método público.

En este método (`set_params`), la variable temporal `$method`, llama reflexivamente (a través del método `getMethod` de `reflectionClass`), al método `bind_param` de la clase `mysqli`. En la siguiente instrucción, a través del método `invokeArgs` de `ReflectionClass`, le pasa por referencia a `bind param`, los datos a ser enlazados con la sentencia preparada (almacenados en el array `$data`). Podría decirse que `invokeArgs`, se comporta de forma similar a `call_user_func_array()`.

Enlazar resultados de una consulta de selección:

```
protected static function get_data($fields) {
    $method = self::$reflection->getMethod('bind_result');
    $method->invokeArgs(self::$stmt, $fields);
    while(self::$stmt->fetch()) {
        self::$results[] = unserialize(serialize($fields));
    }
}
```

Este método es uno de los más “complejos y rebuscados”, que incluso cuenta con algunas “pseudo-magias” un tanto “raras” como el uso de las funciones `serialize` y `unserialize` en la la misma instrucción. Pero analicémoslo detenidamente.

El parámetro `$fields` será recibido a través del único método público que crearemos en nuestra capa de abstracción (este método, a la vez, recibirá este dato, también como parámetro, pero opcional).

Este parámetro, será un array asociativo, donde las claves, serán asociadas al nombre de los campos, y el valor de esas claves, al dato contenido en el campo.

Si tuviese la siguiente consulta SQL:

```
SELECT nombre, descripcion, precio FROM producto WHERE categoria = ?
```

Mi array asociativo, podría paracerse al siguiente:

```
$fields = array("Producto" => "",
                "Descripción" => "",
                "Precio" => "");
```

`mysqli->bind_result()` enlazará el campo producto.nombre a la clave Producto,

producto.descripcion a la clave Desripción y producto.precio a la clave Precio.

Las instrucciones:

```
$method = self::$reflection->getMethod('bind_result');
$method->invokeArgs(self::$stmt, $fields);
```

se comportan de forma similar, a sus homónimas en el método set_params. Llama reflexivamente al método `bind_result` de la clase `mysqli` y le pasa por referencia, el array `$fields`.

En el bucle `while`, estamos asociando iterativamente los datos obtenidos a nuestra propiedad pública `$results`. Pero ¿cómo lo hacemos? ¿para qué serializar y deserializar los datos?:

```
while(self::$stmt->fetch()) {
    self::$results[] = unserialize(serialize($fields));
}
```

En cada iteración, `stmt->fetch` nos está retornando nuestro array `$fields`, asociado al registro de la tabla, sobre el cuál se está iterando. Es decir, que en cada iteración, `stmt->fetch` nos retornará algo como esto:

```
// contenido del array $fields
array("Producto" => "HD Magazine",
      "Descripción" => "Magazine digital de edición mensual sobre Software
                        Libre, Hacking y Programación",
      "Precio" => "0.00");
```

Pero nuestro array `$fields`, ha sido pasado por referencia. Ergo, en cada iteración, su contenido será modificado.

Si a mi propiedad estática `$results`, le agrego como elemento, un array que está siendo modificado por referencia en el momento que lo estoy asignando a mi propiedad estática, mi propiedad estática, será también, modificada en cada iteración.

Para prevenir eso, serializo mi array `$fields` y lo almaceno en `$results` serializado. Pero como luego necesitará recuperarlo, ahorro un paso y lo deserializo en la misma iteración. Al serializarlo, estoy "mágicamente" emulando una "inmutabilidad" de los datos asociados.

Cerrar conexiones abiertas:

```
protected static function finalizar() {
    self::$stmt->close();
    self::$conn->close();
}
```

Un método público que ejecute todas las acciones:

```
public static function ejecutar($sql, $data, $fields=False) {
    self::$sql = $sql;  # setear la propiedad $sql
    self::$data = $data;  # setear la propiedad $data
    self::conectar();  # conectar a la base de datos
    self::preparar();  # preparar la consulta SQL
    self::set_params();  # enlazar los datos
    self::$stmt->execute();  # ejecutar la consulta
    if($fields) {
        self::get_data($fields);
    } else {
        if(strpos(self::$sql, strtoupper('INSERT')) === 0) {
            return self::$stmt->insert_id;
        }
    }
    self::finalizar();  # cerrar conexiones abiertas
}
```

La estructura de control de flujo condicional, que utiliza el método `ejecutar()`, es la encargada de discernir si se trata de una consulta de "lectura" a la base de datos para así, llamar al método `get_data`, o si se trata de una consulta de "escritura" (INSERT, UPDATE o DELETE). En ese caso, verifica si es una escritura de tipo "INSERT" para retornar la id del nuevo registro creado.

Código completo de la capa de abstracción

```
class DBConnector {

    protected static $conn;
    protected static $stmt;
    protected static $reflection;
    protected static $sql;
    protected static $data;
    public static $results;

    protected static function conectar() {
        self::$conn = new mysqli(DB_HOST, DB_USER, DB_PASS, DB_NAME);
    }

    protected static function preparar() {
        self::$stmt = self::$conn->prepare(self::$sql);
        self::$reflection = new ReflectionClass('mysqli_stmt');
    }

    protected static function set_params() {
        $method = self::$reflection->getMethod('bind_param');
        $method->invokeArgs(self::$stmt, self::$data);
    }

    protected static function get_data($fields) {
        $method = self::$reflection->getMethod('bind_result');
        $method->invokeArgs(self::$stmt, $fields);
```

```
            while(self::$stmt->fetch()) {
                self::$results[] = unserialize(serialize($fields));
            }
        }

        protected static function finalizar() {
            self::$stmt->close();
            self::$conn->close();
        }

        public static function ejecutar($sql, $data, $fields=False) {
            self::$sql = $sql;
            self::$data = $data;
            self::conectar();
            self::preparar();
            self::set_params();
            self::$stmt->execute();
            if($fields) {
                self::get_data($fields);
            } else {
                if(strpos(self::$sql, strtoupper('INSERT')) === 0) {
                    return self::$stmt->insert_id;
                }
            }
            self::finalizar();
        }

    }
```

¿Cómo utilizar la capa de abstracción creada?

En todos los casos, siempre será necesario invocar estáticamente al método ejecutar, pasándole al menos dos parámetros: la sentencia SQL a preparar y un array con los datos a enlazar a la sentencia SQL preparada:

```
$sql = "INSERT INTO productos
        (categoria, nombre, descripcion, precio)
        VALUES (?, ?, ?, ?)";

$data = array("isbd",
              "{$categoria}", "{$nombre}", "{$descripcion}", "{$precio}");

$insert_id = DBConnector::ejecutar($sql, $data);
```

Cuando se tratare de una consulta de selección, se deberá adicionar un tercer parámetro, el cuál será un array asociativo, cuyas claves, serán asociadas a los campos de la tabla:

```
$sql = "SELECT nombre, descripcion, precio
        FROM   productos
        WHERE  categoria = ?";

$data = array("i", "{$categoria}");
$fields = array("Producto" => "", "Descripción" => "", "Precio" => "");
DBConnector::ejecutar($sql, $data, $fields);
```