Trabajo de investigación de Clases Útiles

Libreria ORM - Idiorm & Paris

Alumnos

- Cardozo Ramiro FAI-1729
- Landaeta Lucia FAI-1918

Introducción

Idiorm y Paris son librerías distintas que comparten un objetivo final que es el de facilitar la implementación del ORM, pero cada una utilizando su propio enfoque.

¿Por qué la elegimos?

Idiorm

¿Qué es?

Mapeador objeto-relacional y generador de consultas fluido.

Objetivo:

Minimizar el impacto de implementar un ORM en aplicaciones de pequeña y mediana escala, haciendo énfasis en la simplicidad y rapidez.

Enfoque:

Utiliza una clase ORM construida sobre la clase PDO.

Funciona como una API de consultas SELECT y como una clase simple de Modelo CRUD (Create,Read,Update,Delete).

Instalación

Packagist: Con el identificador *j4mie/idiorm* .

Directo: descargar e incorporar el archivo idiorm.php al proyecto.

Implementación

Solo necesita realizar un include del archivo idiorm.php y configurar los datos de conexión a la BD usando el método estático **configure() .**

Ejemplo con MySQL

Dos posibles implementaciones

```
ORM::configure('mysql:host=localhost;dbname=my_database');
ORM::configure('username', 'database_user');
ORM::configure('password', 'top_secret');
```

```
ORM::configure(array(
    'connection_string' => 'mysql:host=localhost;dbname=my_database',
    'username' => 'database_user',
    'password' => 'top_secret'
));
```

Configuración

Configuraciones específicas

```
ORM::configure('driver_options', array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8'));
```

Columna de identificación

ORM asume que todas las tablas tienen como clave primaria una columna llamada id.

Hay dos formas de anular esa asunción, para todas las tablas por igual o para cada tabla por separado.

id_column

Especifica los nombres de las claves primarias de todas las tablas

Clave primaria única

```
ORM::configure('id_column', 'clave_primaria');
```

Clave primaria compuesta

```
ORM::configure('id_column', array('cl_1', 'cl_2'));
```

id_column_overrides

Específica las claves primarias para cada tabla por separado

```
ORM::configure('id_column_overrides', array(
    'usuario' => 'usuario_id',
    'rol' => 'rol_id', ));
```

Conjunto resultado

Esta configuración permite recibir una colección de resultados como un conjunto, que son más flexibles a la hora de operar.

```
ORM::configure('return_result_sets', true);
```

Registro de consultas

Esta configuración permite activar el registro de consultas de idiorm

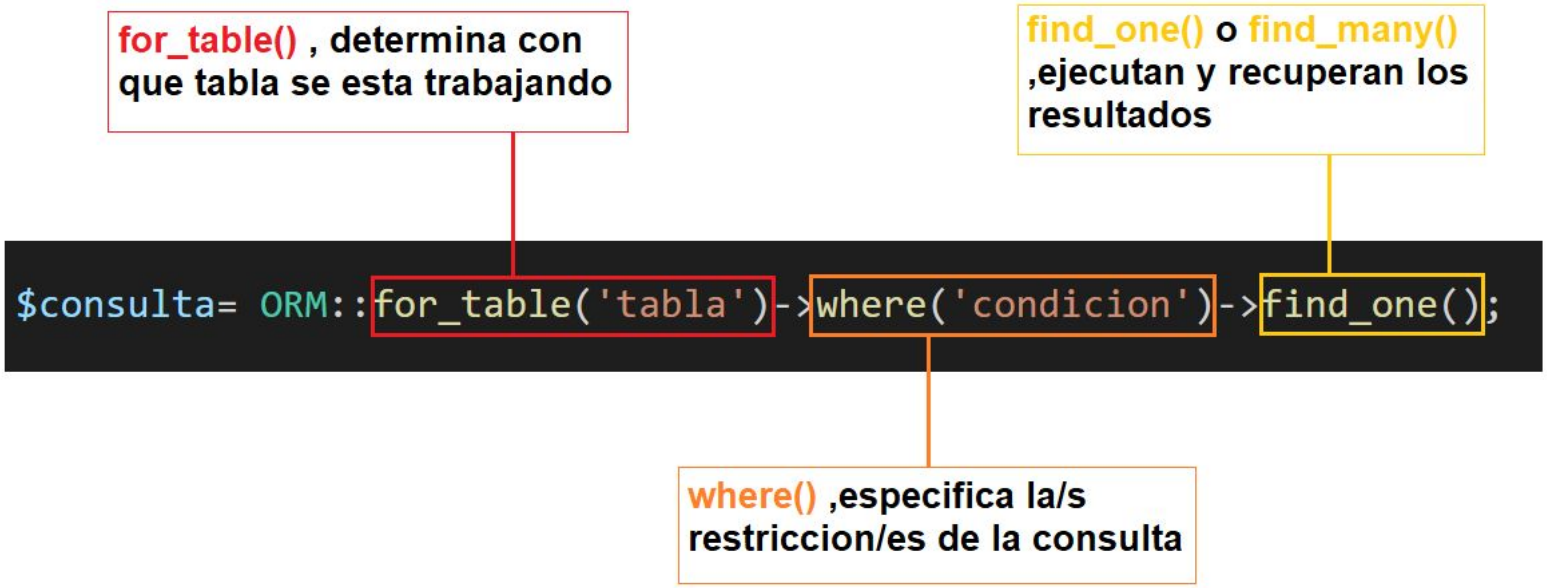
```
ORM::configure('logging', true); //por defecto viene false
```

Para acceder al registro se debe llamar a *ORM::get_last_query()* que retorna la consulta más reciente, o *ORM::get_query_log()* que retorna un arreglo con todas las consultas ejecutadas.

Consultas

Se pueden realizar consultas sin escribir SQL.

Estructura general



Recuperar registros (SELECT)

Resultados únicos

find_one() retornarán una sola instancia de la clase ORM la cual representa la fila de la BD que se solicitó, o false si no encontró coincidencia.

Ejemplo:

```
$persona = ORM::for_table('persona')->where('nombre', 'Juan Perez')->find_one();
```

Equivale a: SELECT * FROM persona WHERE nombre = 'Juan Perez'

Recuperar registro por su ID

```
$persona = ORM::for_table('persona')->find_one(5);
```

En el caso de clave compuesta

```
$persona = ORM::for_table('usuario_rol')->find_one(array(
    'idUsuario' => 34,
    'idRol' => 10 ));
```

Resultados múltiples

find_many() retornarán un arreglo de instancias de la clase ORM, donde cada elemento del arreglo corresponde a un registro. Si no se encontraron coincidencias, retornará un arreglo vacío.

Ejemplo:

```
$personas = ORM::for_table('persona')->find_many();
```

Equivale a: SELECT * FROM persona

Restricciones

where() se llama para especificar las restricciones de la consulta.

Ejemplo: Recuperar a todas las personas llamadas Juan

```
$personas = ORM::for_table('persona')->where('nombre', 'Juan')->find_many();
```

-Ordenamiento Ascendente, order_by_asc()
-Ordenamiento Descendente, order_by_desc()

Ejemplo: Recuperar a las personas registradas y ordenarlas de mayores a menores de edad y sus nombres por orden alfabético inverso.

```
$personas=ORM::for_table('persona')->order_by_asc('edad')->order_by_desc('nombre')
-> find_many();
```

Agrupar registros (GROUP BY)

Método group_by()

Ejemplo: Recuperar a las mujeres registradas y agruparlas por nombre

```
$personas=ORM::for_table('persona')->where('genero','femenino')
->group_by('nombre')->find_many();
```

Contar registros (COUNT)

Método count()

Ejemplo: obtener la cantidad total de personas en la BD

```
$cantidadPersonas = ORM::for_table('persona')->count();
```

Otros casos de consultas SELECT

Mostrar solo algunos atributos

```
$personas = ORM::for_table('persona')->select('nombre')
->select('edad')->find_many();
```

ó

```
$personas = ORM::for_table('persona')->select_many('nombre', 'edad')->find_many();
```

Equivale: SELECT nombre, edad FROM persona

Colocar un alias a la columna a mostrar

```
$personas = ORM::for_table('persona')->select('nombre', 'Nombre')->find_many();
```

Equivale: SELECT nombre AS Nombre FROM persona

Devolver valores diferentes a ...

```
$distintos = ORM::for_table('persona')->distinct()->select('nombre')->find_many();
```

Equivale: SELECT DISTINCT nombre FROM persona

Unión (INNER JOIN)

Idiorm tiene distintos tipos de JOIN (join, inner_join,left_outer_join,right_outer_join, full_outer_join).
Pero todos trabajan con los mismos 2 argumentos obligatorios.

*Es recomendado especificar las condiciones con una matriz que contiene tres componentes: la primera columna, el operador y la segunda columna

Funciones agregadas (MIN, MAX, AVG, SUM)

Todas funcionan de la misma manera.
Se proporciona el nombre de la columna para realizar la función agregada y ésta *devolverá un número entero*.

Ejemplo: Recuperar la altura mínima del registro de personas.

```
$min = ORM::for_table('persona')->min('altura');
```

Obtener datos de los objetos

Una vez recuperados los registros (ahora objetos), se pueden acceder a las atributos de estos de dos formas; usando el método get() o accediendo directamente del objeto:

```php
$persona = ORM::for_table('persona')->find_one(5);
$nombre = $persona->get('nombre');
//ó
$nombre = $persona->nombre;
```

Encapsular datos de los objetos

Se pueden encapsular todas las propiedades del objeto en un arreglo asociativo, con el método as_array().

*Opcionalmente, a éste método se lo puede llamar con los nombres de las columnas como argumentos para que el arreglo solo retorne esas claves y sus respectivos datos:

```php
$persona = ORM::for_table('persona')->find_one(5);
$juan = $persona->as_array();
//array('id' => 5, 'nombre' => 'Juan', 'edad' => 50)

$juan = $persona->as_array('nombre', 'edad');
//array('nombre' => 'Juan', 'edad' => 50)
```

Modelo

Actualizar Registros

Se modifica uno o más atributos del objeto y se llama al método save() para confirmar los cambios en la BD.

Para modificar los valores se puede cambiar el valor del atributo directamente o utilizar el método set().

```php
$persona = ORM::for_table('persona')->find_one(5);
// Las siguientes dos formas son equivalentes
$persona->set('nombre', 'Juan Perez');
$persona->edad = 20;
// Esto es equivalente a las dos asignaciones anteriores.
$persona->set(array(
    'nombre' => 'Juan Perez',
    'edad'   => 20
));
// Sincronizar el objeto con la base de datos
$persona->save();
```

Crear Registros

Se debe crear una instancia de objeto "vacía" con el método *create()*.
Luego, establecer valores a los atributos del objeto y guardarlo.

```
$persona = ORM::for_table('persona')->create();
$persona->nombre = 'Juan Perez';
$persona->edad = 40;
$person->save();
```

Eliminar Registro

Para eliminar un registro de la base de datos, se debe llamar al método *delete()*.

```
$persona = ORM::for_table('persona')->find_one(5);
$persona->delete();
```

Para eliminar más de un registro, se crea una consulta y se llama a *delete_many()*

```
$person = ORM::for_table('persona')
    ->where('codigo', 55555)
    ->delete_many();
```

*este es recomendado cuando se trabaja con conjuntos resultado.

Paris

¿Qué es?

Es una implementación de Active Record construida sobre Idiorm.

Objetivo:

Minimizar el impacto de implementar un ORM, pensado para desarrollos con BD más complejas.

Enfoque:

Implementa Asociaciones (relaciones uno a uno, uno a muchos, muchos a muchos).

Instalación

Packagist: Con el identificador *j4mie/paris* .

Directo: paris.php necesita de idiorm.php para funcionar, por lo que se deben descargar e incorporar ambos archivos al proyecto.

Implementación

Solo necesita realizar un include del archivo idiorm.php y paris.php y configurar los datos de conexión a la BD a traves de idiorm.php usando el método estático **configure() .**

Se configura de igual manera que Idiorm, exceptuando las configuraciones *id_column() y id_column_overrides()*, las cuales no son necesarias para Paris, ya que las claves se manejan de otra forma.

Modelos

Clases Modelo

Por cada entidad en la aplicación se debe crear una clase que extienda de otra llamada Modelo.

Ejemplo:

```
class Persona extends Model{

}
```

-Paris crea instancias de las clases del modelo y las rellena con propiedades que reflejan los datos de la BD.
Se puede agregar comportamiento con métodos públicos estáticos.

-Por defecto los nombres de las clases siguen la convención.
CapWords o StudlyCaps y los nombres de las tablas en la base con_guion_bajo en minúsculas.

Establecer el nombre de una tabla para una clase manualmente:

```php
class Persona extends Model{
    public static $_table = 'persona';
}
```

Paris asume que la columna de clave primaria de cada tabla se llama ‘id’.
Se puede sobreescribir este comportamiento.

```php
class Persona extends Model{
    public static $_id_column = 'id_persona';
}
```

*Paris no respetará la configuración que usamos para Idiorm respecto a las claves primarias.

Asociaciones

Paris ofrece métodos propios para describir las relaciones entre modelos de forma sencilla.
Define diferentes métodos en los modelos para hallar las tablas de las distintas relaciones y armar la instancia de su respectivo modelo.

Uno a Uno

Se implementan con el método has_one().
Ejemplo: Se tiene un modelo Usuario y cada usuario tiene un único Perfil.

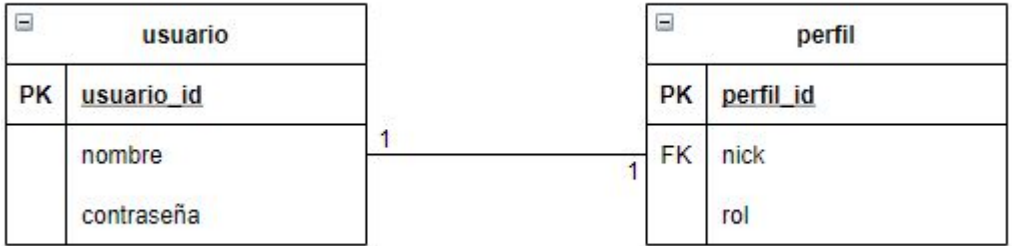
```
class Perfil extends Model {
}
class Usuario extends Model {
    public function perfil() {
        return $this->has_one('Perfil');    }     }
```

Al llamar al método perfil(), éste nos devolverá una instancia ORM con la que trabajar

```
// Seleccionamos un usuario de la BD
$usuario = Model::factory('Usuario')->find_one($usuario_id);
// Encontramos el perfil asociado a ese usuario
$perfil = $usuario->perfil()->find_one();
```

has_one() opcionalmente puede recibir hasta dos parámetros adicionales; el primero especifica el nombre de la columna en la tabla asociada elegida como clave foránea, el segundo especifica el nombre de la columna en la tabla base elegida para buscar la clave foránea:

```
return $this->has_one('Perfil', 'nick', 'nombre');
```



Uno a Muchos

Se implementan con el método has_many().

Ejemplo: tenemos un modelo Perfil y cada perfil tiene muchos Posts(o publicaciones).

La tabla perfil debe estar relacionada a la tabla post.

```
class Post extends Model {
}
class Perfil extends Model {
    public function posts() {
        return $this->has_many('Post');
    }
}
```

Al llamar al método posts(), éste nos devolverá las instancias ORM correspondientes.

```
// Seleccionamos un perfil de la BD
$perfil = Model::factory('Usuario')->find_one($perfil_id);
// Encontramos las publicaciones relacionadas a ese usuario
$posts = $perfil->posts()->find_many();
```

*Trabaja las clave foráneas igual que has_one()

Pertenece a .

Es igual a 1-1 y 1-M pero con el método belongs_to().

Se asume que la clave foránea está del **otro lado** de la relación.

```php
class Perfil extends Model {
	public function usuario() {
		return $this->belongs_to('Usuario');
	}
}
class Usuario extends Model {
	}
```

*trabaja con las claves foráneas al igual que *has_one()* y *has_many()* teniendo en cuenta que la tabla base y la tabla asociada se 'invierten'

```php
// Seleccionamos un perfil de la BD
$perfil = Model::factory('Perfil')->find_one($perfil_id);

// Encontramos el usuario asociado a ese perfil
$usuario = $perfil->usuario()->find_one();
```

Muchos a Muchos

Se implementan con el método *has_many_through()* y es necesario crear un modelo intermediario para reflejar la relación.

Ejemplo: tenemos un modelo **Usuario** y otro modelo **Rol**. Cada usuario puede tener muchos roles y, a su vez, cada rol puede tener muchos usuarios. Debemos crear un modelo **RolUsuario** (nombre como concatenación de ambos modelos ordenados alfabéticamente).

```php
class Usuario extends Model {
    public function roles() {
        return $this->has_many_through('Rol');
    }
}
class Rol extends Model {
    public function usuarios() {
        return $this->has_many_through('Usuario');
    }
}
class RolUsuario extends Model {
}
```

```php
// Selecciona un usuario de la base
$usuario = Model::factory('Usuario')->find_one($usuario_id);
// Busca los roles asociados al usuario
$roles = $usuario->roles()->find_many();
// Selecciona el primer rol
$un_rol = $roles[0];
// Encuentra todos los usuarios con ese rol asociado
$usuarios_de_un_rol = $un_rol->usuarios()->find_many();
```

Parámetros de *has_many_through()*

Este método puede recibir hasta 6 parámetros, siendo solo el primero de ellos obligatorio, y el resto opcionales.

- Primer parámetro: Nombre del modelo relacionado
- Segundo parámetro: Nombre del modelo intermediario
- Tercer parámetro: Clave de la tabla base a la tabla intermediaria
- Cuarto parámetro: Clave de la tabla relacionada a la tabla intermediaria
- Quinto parámetro: Clave foránea en la tabla base
- Sexto parámetro: Clave foránea en la tabla relacionada

Consultas

Se llama a *factory()* sobre la clase Model con el nombre la clase a trabajar.
Usa mismos métodos de Idiorm que agregan restricciones a la consulta.

```
$usuarios = Model::factory('Usuario')
    ->where('name', 'Juan')
    ->where_gte('edad', 20)
    ->find_many();
```

Existen consultas más corta y legible (disponible desde PHP +5.3)

```
$usuarios = Usuario::where('nombre', 'Juan')
    ->where_gte('edad', 20)
    ->find_many();
```

Las consultas con Paris se pueden utilizar de la misma forma que con Idiorm, aunque se diferencia en 3 puntos:

1.- No es necesario llamar al método for_table() para especificar la tabla a usar

2.- Los métodos find_one() y find_many() retornarán instancias de las clases del modelo, en vez de las de la clase ORM.

3.- Filtrado personalizado.

CRUD

Crear

```
$usuario = Model::factory('Usuario')->create();
    $usuario->nombre = 'Juan';
    $usuario->save();
```

Leer

```
$usuario = Model::factory('Usuario')->find_one($usuario_id);
echo $usuario->nombre;
```

Actualizar

```
$usuario = Model::factory('Usuario')->find_one($id);
$usuario->nombre = 'Carlos';
$usuario->save();
```

Eliminar

```
$usuario = Model::factory('Usuario')->find_one($id);
$usuario->delete();
```

Métodos personalizados

Se pueden utilizar métodos personalizados, definido en las clases del modelo.

```php
class Usuario extends Model {
    public function nombre_completo() {
        return $this->nombre . ' ' . $this->apellido;
    }
}
$usuario = Model::factory('Usuario')->find_one($id);
echo $usuario->nombre_completo();
```

Arreglos

Método *as_array()* ordena las propiedades de una instancia modelo en un arreglo asociativo

```php
$usuario = Model::factory('Usuario')->find_one(5);
$juan = $usuario->as_array();
//array('id' => 5, 'nombre' => 'Juan', 'edad' => 50)

$juan = $persona->as_array('nombre', 'edad');
//array('nombre' => 'Juan', 'edad' => 50)
```

Filtros

Método *filter()*

Ejemplo: tenemos una clase Usuario en la que queremos un método que nos devuelva la lista de administradores ya que necesitamos de la misma en repetidas ocasiones

```
class Usuario extends Model {
    public static function admins($orm) {
        return $orm->where('rol', 'admin');
    }
}
```

```
$admins = Model::factory('Usuario')->filter('admins')->find_many();
```

también

```
$admins_ordenados = Model::factory('Usuario')
                    ->filter('admins')
                    ->order_by_asc('nombre')
                    ->find_many();
```

Filtros con parámetros

Se pueden enviar parámetros hacia los filtros definidos, que se envían a través del método *filter()* luego del nombre del filtro (método) a aplicar:

```
class Usuario extends Model {
    public static function edad_admins($orm, $edad) {
        return $orm->where(
            array(
                'rol' => 'admin',
                'edad' => $edad
            )
        );
    }
}

$admins_sub20 = Model::factory('Usuario')
      ->filter('edad_admins', 20)
      ->find_many();
```

Consultas Libres (Raw Querys)

Con ambas librerías se pueden realizar consultas específicas (sin tener que utilizar los métodos provistos) a través de *raw_query()*:

```
$admins = ORM::for_table('usuario')
    ->raw_query('SELECT p.* FROM usuario p JOIN rol r ON p.rol_id = r.id WHERE r.nombre = :rol', array('rol' => 'admin'))
    ->find_many();
```

Si bien se puede referenciar a cualquier tabla en el método *for_table()*, y aún así retornará una instancia con los datos solicitados, esta instancia estará vinculada justamente a esa tabla referenciada, y si queremos utilizar, por ejemplo, el método *save()*, podría intentar guardar los cambios en la tabla equivocada. Se recomienda siempre referenciar la tabla correspondiente a la consulta.

Transacciones

Tanto en Idiorm como en Paris, se pueden trabajar transacciones, pero no proveen ningún método extra para ello. Simplemente se pueden utilizar los métodos incorporados de PDO, utilizando 'ORM' solo para manejar la conexión y configuración:

```
// Inicio de transaccion
ORM::get_db()->beginTransaction();
// Commit
ORM::get_db()->commit();
// Roll back
ORM::get_db()->rollBack();
```