TRABAJO DE INVESTIGACIÓN DE CLASES ÚTILES

Librerías ORM - “Idiorm & Paris”

Alumnos	Legajo
Cardozo Ramiro	**FAI-1729**
Landaeta Lucia	**FAI-1918**

ÍNDICE

CLASE ELEGIDA

Librerías ORM - “Idiorm & Paris”

INTRODUCCIÓN

Para comenzar cabe destacar que Idiorm y Paris son librerías distintas, pero con el mismo objetivo final; facilitar el desarrollo de ORM’s.

Idiorm se define como un ‘mapeador objeto-relacional y generador de consultas fluido’. Está pensado para minimizar el impacto de implementar un mapeo objeto-relacional en aplicaciones de pequeña y mediana escala, haciendo énfasis en la simplicidad y rapidez, de modo que no tengamos que crear muchas clases con jerarquías de herencia, y toda la complejidad que eso conlleva. En su lugar, con esta librería solo utilizaremos una clase (construida sobre PDO), ORM, que funciona como una API de consultas SELECT fluida y como una clase simple del modelo CRUD (Create, Read, Update, Delete).

En la misma línea tenemos a **Paris**, la cual es una implementación de **Active Record** construida sobre Idiorm (de modo que necesita de esta para funcionar). Básicamente tiene el mismo objetivo minimalista de Idiorm, pero está pensado para desarrollos con bases de datos más complejas y con un enfoque diferente, por lo que soporta *Asociaciones* (relaciones uno a uno, uno a muchos, muchos a muchos). Cuando nos referimos a diferentes enfoques, no hay mejor explicación que la que nos da el desarrollador al comparar sus dos librerías;

“Si prefiere pensar sobre tablas y uniones, probablemente debería usar Idiorm. Si prefiere pensar sobre objetos del modelo y relaciones, probablemente debería usar Paris.”

Además de las particularidades de cada librería, ambas comparten características como:

- Soporte a distintos tipos de base de datos (entre ellas MySQL, SQLite, PostgreSQL, Firebird).
- Utilizan encadenamiento de métodos para colecciones de modelos, para filtrar o aplicar acciones en múltiples resultados a la vez.
- Soporta múltiples conexiones a BD’s a la vez.
- Métodos compatibles con el estándar PSR-1.
- Uso de ‘prepared statements’ como protección contra ataques de SQL Injection.

IDIORM

Fuente

https://github.com/j4mie/idiorm/

Instalación

Es muy fácil de incorporar ya que solo se necesita descargar e incorporar el archivo idiorm.php al proyecto. Por otro lado se puede optar por incorporar la librería con el identificador *j4mie/idiorm* si se utiliza Packagist (repositorio principal de Composer).

Implementación

Idiorm es de fácil implementación ya que no necesita definir ninguna clase de modelo para usarlo, si no que se puede comenzar a utilizar el ORM de inmediato. Solo se necesita realizar un include del archivo idiorm.php y configurar la conexión a la base de datos a través del método estático *configure()*. En este caso al utilizar MySQL pasamos nombre de BD, username y password. Se brindan dos formas de hacerlo:

Pasar los datos con un llamado para cada uno,

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

o a través de un atajo que permite pasar varias claves/valor (configuración que desea modificar y el valor que desea establecer) a la vez

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

Configuración

En algunos casos es necesario configurar opciones específicas de controlador, la siguiente configuración permite pasar estas opciones al constructor del PDO. Por ejemplo, para forzar al controlador MySQL a utilizar UTF-8:

```
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, una para configurar todas las tablas por igual y otra para cada tabla por separado.

Configuración de *id_column*

En caso de ser necesario se puede utilizar la configuración *id_column* para especificar los nombres de las claves primarias de **todas** las tablas.
Suponiendo que su columna ID es única y se llama 'clave_primaria':

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

Si se trata de una clave primaria compuesta:

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

Nota: si la clave compuesta utiliza una columna auto_increment, ésta siempre debe ser definida primero en el array.

Configuración de *id_column_overrides*

Esta configuración se utiliza para especificar la columna ID para **cada tabla por separado**, y toma como parámetro un array asociativo que mapea cada nombre de tabla con el nombre de su columna. Por ejemplo, si cada columna ID incluye el nombre de la tabla:

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

Al igual que en id_column, si la clave es compuesta se define con un array.

Conjunto resultado

Esta configuración permite recibir una colección de resultados como un conjunto, el cual es posible utilizar para iterar uno a uno con, por ejemplo, un *foreach.* Por el momento se recomienda dejar esto definido ya que los conjuntos de resultados son más flexibles a la hora de operar, (se explicará con el método *find_result_set()* en el apartado de Resultados Múltiples).

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

Registro de consultas

Idiorm puede registrar todas las consultas que se ejecutan. Para activar esta configuración:

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

Luego, para acceder a este registro se pueden utilizar estos dos métodos estáticos; *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

Idiorm permite realizar consultas de una forma simple sin necesidad de escribir los sql de estas.
Para realizar una consulta se comienza llamando al método *for_table()* para especificar sobre que tabla se quiere trabajar.
Luego se llaman a los métodos que agregan restricciones a la consulta.

Estándar PSR-1 y camelCase

Todos los métodos definidos de la clase pueden ser llamados tanto con guiones bajos como con la notación 'camelCase', por ejemplo *find_one()* puede ser invocado como *findOne()*.

Resultados únicos

Todas las consultas que finalicen con *find_one()* retornarán una sola instancia de la clase ORM la cual representa la fila de la base de datos que solicitó, o false si no se encontró coincidencia.

Resultados múltiples

Todas las consultas que finalicen con *find_many()* retornarán un arreglo de instancias de la clase ORM, donde cada fila de la base de datos corresponde a cada elemento del arreglo. Si no se encontraron coincidencias, retornará el arreglo vacío.
Si la configuración *'return_result_sets'* se encuentra activa, como recomendamos previamente, puede finalizar la consulta con ***find_result_set()*** para obtener un conjunto de resultados, en el cual se puede ejecutar operaciones en lote, y de todas formas tratarlo como si fuese un arreglo.
Para comprender mejor la forma de operar con los resultados, sea cual sea el método que elija, a continuación dejamos los ejemplos para cada uno.

Recuperar registros (SELECT)

Para recuperar un **único registro:**
Se debe llamar al método *find_one()* que devolverá una sola instancia de la clase ORM la cual representa la fila de la base de datos que solicitó, o false si no se encontró coincidencia.
Ejemplo:

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

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

Para recuperar un solo registro por su ID se lo puede pasar directamente por *find_one()* como parámetro:

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

En el caso se utilizar una clave compuesta,

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

Para recuperar **varios registros con *find_many()*** :

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

Equivale a: SELECT * FROM persona
Para recuperar registros donde existan condiciones se llama al método *where()* para especificarlas:

Ejemplo: Recuperar todas las personas que se llamen Juan y 'setearles' el nombre en Carlos

```
$personas = ORM::for_table('persona')->where('nombre', 'Juan')->find_many();
foreach ($personas as $persona) {
    $persona->nombre = 'Carlos';
    $persona->save();
}
```

Para recuperar **varios registros con *find_result_set()*** y 'setearles' el nombre en Carlos

```
ORM::for_table('persona')->where('nombre', 'Juan')->find_result_set()
->set('nombre', 'Carlos')
->save();
```

La ventaja de este método es que se pueden utilizar métodos directamente sobre el conjunto resultado y modificar todos los registros a la vez. Aún así, se puede tratar este conjunto con foreach o count() como si de un arreglo se tratase.

Contar registros (COUNT)

Para obtener el número de filas resultado de una consulta se llama al método *count()*
Ejemplo: obtener la cantidad total de personas en la BD

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

Ordenar registros (ORDER BY)

Se cuenta con dos métodos de ordenamiento
-Ordenamiento Ascendente, *order_by_asc()*
-Ordenamiento Descendente, *order_by_desc()*
Ejemplo:

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

Agrupar registros (GROUP BY)

Se debe llamar al método *group_by()*

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

Otros casos de consultas SELECT

-Mostrar solo algunos atributos de la búsqueda

```
$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 los valores diferentes

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

Equivale: SELECT DISTINCT nombre FROM persona

Unión (INNER JOIN)

Idiorm tiene una familia de métodos para agregar diferentes tipos de JOIN a las consultas (join, inner_join, left_outer_join, right_outer_join, full_outer_join). Cada uno de estos métodos toma el mismo conjunto de argumentos.
Se utilizará el método join básico como ejemplo, pero lo mismo se aplica a los demás.
La estructura del método join cuenta con dos primeros argumentos obligatorios.
El primero es el nombre de la tabla a unir y el segundo proporciona las condiciones para la unión. Se recomienda especificar las condiciones con una matriz que contiene tres componentes: la primera columna, el operador y la segunda columna.
Ejemplo:

```
$resultados = ORM::for_table('persona')->join('persona_ profile',
array('persona.id', '=', 'persona_profile.persona_id'))->find_many();
```

Funciones agregadas (MIN, MAX, AVG, SUM)

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

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

Obtener datos de los objetos

Una vez que se obtengan los registros (ahora objetos) con las consultas, se pueden acceder a las propiedades de esos objetos (los datos en las columnas de los registros de las tablas) de dos formas; usando el método *get()* o accediendo directamente a la propiedad del objeto:

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

A su vez, se pueden encapsular todas las propiedades del objeto en un arreglo asociativo, donde cada clave corresponde a cada propiedad del objeto, con el método *as_array()*.
Opcionalmente, a éste método se lo puede llamar con los nombres de las columnas como argumentos de modo que el arreglo solo retorne con esas claves y sus respectivos datos:

```
$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

Para actualizar un registro se debe modificar uno o más atributos del objeto con el que se está trabajando y llamar al método *save()* para confirmar los cambios en la BD.
Para modificar los valores de un objeto se puede cambiar el valor del atributo directamente o utilizar el método *set()*, que además permite cambiar varios atributos a la vez a través de una matriz asociativa.
Ejemplo:

```
$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 nuevos registros

Para agregar un nuevo registro, se debe crear una instancia de objeto "vacía" con el método *create()* . Luego, establecer valores a los atributos del objeto y guardarlo.
Ejemplo:

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

Eliminar registros

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

Ejemplo:

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

Para eliminar más de un registro de la base de datos, cree una consulta y utilice el método *delete_many()* :

Ejemplo:

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

Este último método es recomendado cuando se trabaja con **conjuntos resultado**.

PARIS

Fuente

https://github.com/j4mie/paris/

Instalación

paris.php necesita de idiorm.php, ya que está construido sobre este último, y de igual manera, se deben descargar e incorporar ambos archivos al proyecto. También se puede incorporar la librería con el identificador *j4mie/paris* si se utiliza Packagist.

Implementación

Para comenzar a usar esta librería, se deben incluir idiorm.php y luego paris.php, luego configurar la conexión a la base de datos a través de Idiorm tal cual se explicó al inicio del documento, con el método *configure*.

Configuración

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

Hay algunas opciones propias de esta librería que se explicaran al momento de utilizarlas.

Modelos

Clases Modelo

Por cada entidad en la aplicación se debe crear una clase que extienda de otra clase llamada Modelo. Por ejemplo, para la clase Persona:

```
class Persona extends Model{
}
```

Paris se encarga de crear las instancias de las clases del modelo y rellenarlas con los datos de la base de datos. También se puede agregar *comportamiento* en forma de métodos públicos estáticos los cuales implementan la lógica de la aplicación.

Por defecto, Paris asume que los nombres de las clases seguirán la convención *CapWords* o *StudlyCaps* y los nombres de las tablas en la base con_guion_bajo en minúsculas, por lo que va a convertir entre ambos formatos de manera automática.
En caso de querer establecer el nombre de una tabla para una clase manualmente, se puede definir una propiedad:

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

Del mismo modo, Paris asume que la columna de clave primaria de cada tabla se llama 'id', y para sobreescribir este comportamiento podemos definir la siguiente propiedad:

```
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, diferenciándose de la mayoría de los ORM que se basa en el uso de arreglos asociativos para el mismo objetivo. Para esto debemos definir diferentes métodos en los modelos de modo que podamos hallar las tablas de dichas relaciones.
Nota: en los métodos que definimos utilizamos los nombres de las tablas, lo cual no es absolutamente necesario, pero deberían nombrarse de forma descriptiva.

Uno a Uno

Las relaciones uno a uno se implementan con el método *has_one()*.
Por ejemplo; tenemos un modelo Usuario y cada usuario tiene un único Perfil, entonces la tabla usuario debe estar relacionada a la tabla perfil. Para que esto quede especificado, en la clase Usuario debemos definir un método llamado *perfil(),* el cual llama al método de *has_one()* que recibe como parámetro el nombre de la clase del objeto relacionado.

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

De esta forma, cuando llamemos al método *perfil(),* éste nos devolverá una instancia ORM para seguir trabajando.

```
// 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();
```

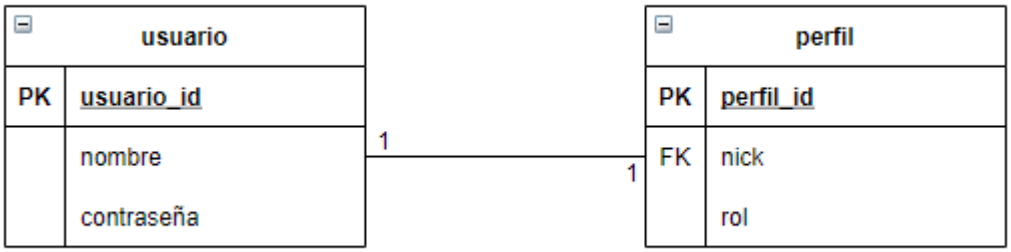
Por defecto, Paris asume que la clave foránea de (en este caso) la tabla usuario, que se encuentra en la tabla perfil, tiene el mismo nombre de la tabla base (usuario) con ‘ _id’ agregado (usuario_id). Para sobreescribir este comportamiento, debemos agregar un segundo parámetro a *has_one()* con el nombre de la tabla que vamos a utilizar. Por ejemplo, si la columna de clave foránea en perfil se llama ‘nick’:

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

A su vez, Paris asume que usuario_id es la columna de usuario en la que debe basarse para buscar la clave foránea en perfil, asumiendo que usuario_id es la clave primaria en usuario. Para sobreescribir este comportamiento, se debe agregar un tercer parámetro a *has_one()* con el nombre de la columna que vamos a utilizar. Por ejemplo, si de la tabla usuario queremos utilizar la columna ‘nombre’:

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

El siguiente diagrama es para esclarecer mejor este ejemplo y dejar claro el uso de parámetros de estos métodos ‘relacionales’.



Uno a Muchos

Las relaciones uno a muchos se implementan con el método *has_many()*. Por ejemplo; tenemos un modelo Perfil y cada perfil tiene muchos Posts(o publicaciones), entonces la tabla perfil debe estar relacionada a la tabla post.
Para que esto quede especificado, en la clase Usuario debemos definir un método llamado *posts()*, el cual llama al método *has_many()* que recibe como parámetro el nombre de la clase de los objetos relacionado (nombrado literalmente, es decir, Post).

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

De esta forma, cuando llamemos al método *posts()*, éste nos devolverá las instancias ORM correspondientes para seguir trabajando.

```
// 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();
```

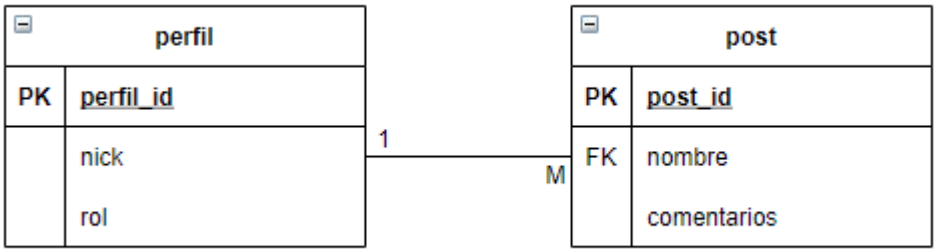
Por defecto, Paris asume que la clave foránea de (en este caso) la tabla perfil, que se encuentra en la tabla post, tiene el mismo nombre de la tabla base (perfil) con '_id' agregado (perfil_id). Para sobreescribir este comportamiento, debemos agregar un segundo parámetro a *has_many()* con el nombre de la tabla que vamos a utilizar. Por ejemplo, si la columna de clave foránea en post se llama 'nombre':

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

A su vez, Paris asume que perfil_id es la columna de perfil en la que debe basarse para buscar la clave foránea en post, asumiendo que perfil_id es la clave primaria en perfil.
Para sobreescribir este comportamiento, se debe agregar un tercer parámetro a *has_many()* con el nombre de la columna que vamos a utilizar. Por ejemplo, si de la tabla perfil queremos utilizar la columna 'nick':

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

El siguiente diagrama muestra mejor el uso de estos parámetros:



Pertenece a

Esta relación es similar a uno a uno y uno a muchos, y se usa de la misma forma pero con el método *belongs_to()*, y asumiendo que la clave foránea está del otro lado de la relación.
Tomando el ejemplo de cada usuario y su perfil;

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

De esta forma, cuando llamemos al método *usuario()*, éste nos devolverá una instancia ORM para seguir trabajando.

```
// 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();
```

Por defecto, Paris asume que la clave foránea de (en este caso) la tabla perfil, que se encuentra en la tabla usuario, tiene el mismo nombre de la tabla base (perfil) con '_id' agregado (perfil_id).
Para sobreescribir este comportamiento, debemos agregar un segundo parámetro a *belongs_to()* con el nombre de la tabla que vamos a utilizar. Por ejemplo, si la columna de clave foránea en usuario se llama 'nombre':

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

A su vez, Paris asume que perfil_id es la columna de perfil en la que debe basarse para buscar la clave foránea en usuario, asumiendo que perfil_id es la clave primaria en perfil. Para sobreescribir este comportamiento, se debe agregar un tercer parámetro *belongs_to()* con el nombre de la columna que vamos a utilizar. Por ejemplo, si de la tabla perfil queremos utilizar la columna 'nick':

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

Muchos a Muchos

Las relaciones muchos a muchos se implementan con el método *has_many_through()*. Este método, a diferencia de los anteriores, recibe hasta 6 parámetros, siendo sólo el primero de estos obligatorio, el resto sobreescribe el comportamiento por defecto.
Por ejemplo; tenemos por un lado al modelo Usuario y por otro lado un modelo Rol. Cada Usuario puede tener muchos roles y, a su vez, cada Rol puede tener muchos usuarios. Para reflejar esto en nuestro código, debemos crear un modelo intermediario, y su nombre debe ser una concatenación de los nombres de las dos clases relacionadas, en orden alfabético. En este caso, como nuestras clases son Usuario y Rol, la clase intermediaria debería llamarse RolUsuario.
Luego, volviendo a las clases relacionadas, en cada una de estas debemos crear métodos (en Rol uno que se llame usuarios, y en Usuario uno que se llame Rol) para poner invocar al método *has_many_through()* con el nombre de la clase de los objetos relacionados como parámetro. A continuación, un ejemplo sería mucho más claro:

```
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 {
}
```

De esta forma, cuando llamemos al método *roles()* o *usuarios()*, estos nos devolverán las instancias ORM correspondientes para seguir trabajando.

```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():*
Primer parámetro - **Nombre del modelo relacionado**: es obligatorio y es el nombre del modelo relacionado a la que estamos definiendo.
Segundo parámetro - **Nombre del modelo intermediario**: es opcional, por defecto Paris asume que es el nombre de los modelos relacionados concatenados y ordenados alfabéticamente.
Tercer parámetro - **Clave de la tabla base a la tabla intermediaria**: es opcional y por defecto es el nombre de la tabla base con '_id' agregado al final.
Cuarto parámetro - **Clave de la tabla relacionada a la tabla intermediaria**: es opcional y por defecto es el nombre de la tabla relacionada con '_id' agregado al final.
Quinto parámetro - **Clave foránea en la tabla base**: es opcional y por defecto lleva el nombre de la clave primaria de la tabla base.
Sexto parámetro - **Clave foránea en la tabla relacionada**: es opcional y por defecto lleva el nombre de la clave primaria de la tabla relacionada.

Consultas

Con Paris, las consultas comienzan con una invocación al método estático *factory()* sobre la clase **Model** que toma un solo argumento: el nombre del modelo de la clase sobre la que se quiere trabajar. Luego se encadenan los mismos métodos de Idiorm que agregan restricciones a la consulta.
Por ejemplo:

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

También se pueden realizar consultas de una manera 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, Paris utilizará directamente el nombre de la clase creada para esa tabla (como en el último ejemplo).
2.- Los métodos *find_one()* y *find_many()* retornarán instancias de las clases del modelo, en vez de las de la clase ORM, y funcionarán del mismo modo; *find_one()* retornará una instancia o falso si no encontró coincidencias, y *find_many()* retornarán un arreglo de instancias o un arreglo vacío si no encontró coincidencias.
3.- Filtrado personalizado (explicado en detalle más adelante).

CRUD

Al igual que con ORM, las instancias del modelo se pueden manejar exactamente igual que con Idiorm:

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

Además de estos métodos predefinidos, también se pueden utilizar los métodos personalizados que se hayan definido en las clases del modelo:

```
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

Como con Idiorm, con el método *as_array()* podemos ordenar las propiedades de una instancia del modelo en un arreglo asociativo donde cada clave corresponde a cada columna. A su vez, puede recibir los mismos argumentos opcionales para filtrar su contenido:

```
$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

Paris provee un método llamado *filter()* que puede ser encadenado a las consultas realizadas con la API de Idiorm. Este método toma como argumento el nombre de un método público estático (public static) que se encuentra en la subclase de Modelo. Cuando se invoca al método *filter()*, a la vez se invoca al método que se le pasó como parámetro, y este último recibe un objeto ORM, trabaja sobre el mismo y retornará otro objeto ORM para continuar con la ejecución de la consulta.
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');
    }
}
```

Luego, podremos utilizarlo:

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

Incluso con más consultas por delante:

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

También permite el envío de argumentos adicionales para utilizar dentro del método creado, y estos serán enviados a través del método *filter()* luego del nombre del filtro a aplicar, por ejemplo:

```
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();
```

CONCLUSIÓN

Idiorm y Paris, desde nuestra perspectiva, son muy fáciles de implementar, simplifican en gran medida la escritura de código y la lectura del mismo, y nos libra de cometer errores como lo son las malas prácticas en la escritura de sentencias SQL, añadiendo una capa de seguridad que no habíamos tenido en cuenta hasta el momento (protección contra SQLi). Sin embargo, con los ejemplos que proporcionamos junto a este documento, solo raspamos la superficie de las posibilidades que nos brindan, y creemos que pueden llegar a ser una poderosísimas herramientas para trabajar en proyectos que requieran una complejidad bastante más avanzada, basándonos en el nivel en el que venimos trabajando en esta materia en particular, que dicho sea de paso, esta documentación está adaptada para los estudiantes de Programación Web Dinámica y recomendamos que se remitan a la documentación oficial (links al final) para expandir aún más en las funcionalidades de estas herramientas.

Finalmente, vale aclarar que nuestra experiencia como programadores no va mucho más allá de lo trabajado este cuatrimestre (2do de 2021), y estamos al tanto de las advertencias que nos hemos encontrado al investigar sobre estas librerías y el patrón de arquitectura que fomenta una de ellas (Active Record). Dicho esto, extendemos estas advertencias a la hora de ponerlas en práctica instando a que se investigue un poco más, y se analice en profundidad la escala de lo que sea que se va a desarrollar, recomendando (nuevamente) el uso de estas herramientas solo en desarrollos de pequeña a mediana escala.

BIBLIOGRAFÍA

Documentación oficial Idiorm
Idiorm

Documentación oficial Paris
Paris