# Query Builder Class

https://codeigniter.es/user\_guide/database/query\_builder.html

CodeIgniter da acceso a la clase Query Builder. Esta clase permite actualizar, insertar y consultar la base de datos con un mínimo de líneas de código. Más allá de la simplicidad, un beneficio de utilizar la clase es que hace las aplicaciones independientes del SGBD, dado que las consultas son generadas por cada adaptador de SGBD. También tiene sus propias comprobaciones de seguridad, con lo cual lo hace más seguro.

De forma práctica, se crea un nuevo modelo extendiendo CI\_Model y carga la librería de la base de datos. Esto hace a la BdD accesible a través de $this->db object.

## Índice

Seleccionando Datos

Buscando Datos Específicos

Buscando Datos similares

Ordenando resultados

Limitando o Contando Resultados

Agrupando Consultas

Insertando Datos

Modificando Datos

Borrando Datos

Encandenando Metodos

Query Builder Caching

Resetting Query Builder

Class Reference

## Seleccionando Datos

Las siguientes funciones te permiten crear instrucciones SELECT de SQL.

$this->db->get()

Ejecuta la consulta de selección y devuelve el resultado. Devuelve todas las filas de una tabla.

$query = $this->db->get('mytable'); // Produces: SELECT \* FROM mytable

Tienede un segundo parámetro para establecer el límite y el offset de una clausula:

$query = $this->db->get('mytable', 10, 20);

// Executes: SELECT \* FROM mytable LIMIT 20, 10

$query = $this->db->get('mytable');

foreach (**$query->result()** as $row) {

echo $row->title;

}

$this->db->get\_compiled\_select()

Compila la consulta seleccionada como $this->db->get() pero no la ejecuta, sólo la devuelve como una cadena.

### Ejemplo:

$sql = $this->db->get\_compiled\_select('mytable');

echo $sql;

// Prints string: SELECT \* FROM mytable

El segundo parámetro te permite resetear query builder o no (por defecto se resetea):

echo $this->db->limit(10,20)->get\_compiled\_select('mytable', FALSE);

// Prints string: SELECT \* FROM mytable LIMIT 20, 10

echo $this->db->select('title, content, date')->get\_compiled\_select();

// Hay que remarcar que la segunda consulta no utiliza $this->db->from() ni le pasa el nombre de una tabla como primer parámetro. La razón es que la consulta no se ha ejecutado con $this->db->get() y por tanto no ha sido reseteada (ni con $this->db->reset\_query()).

$this->db->get\_where()

Idéntica a la función anterior, pero permite añadir un clausula “where” en el segundo parámetro en lugar de usar la función db->where():

$query = $this->db->get\_where('mytable', array('id' => $id), $limit, $offset);

$this->db->select()

Permite escribir la parte del SELECT de tu consulta:

$this->db->select('title, content, date');

$query = $this->db->get('mytable');

// Ejecuta: SELECT title, content, date FROM mytable

$this->db->select() acepta un parámetro opcional. Si FALSE, CodeIgniter no intentará proteger tu campo o los nombres de tablas. Esto es útil cuando necesitas un select compuesto donde el escape automático de los campos puede romperlos.

$this->db->select('(SELECT SUM(payments.amount) FROM payments WHERE payments.invoice\_id=4') AS amount\_paid', FALSE);

$query = $this->db->get('mytable');

$this->db->select\_max()

Escribe un SELECT MAX(campo) para tu consulta. Puedes añadir un parámetro para renombrar el resultado.

$this->db->select\_max('age');

$query = $this->db->get('members'); // Produces: SELECT MAX(age) as age FROM members

**$this->db->select\_max**('age', 'member\_age');

$query = $this->db->get('members'); // Produces: SELECT MAX(age) as member\_age FROM members

$this->db->select\_min();

Existe un equivalente para mínimo: “SELECT MIN(field)”.

**$this->db->select\_min**('age');

$query = $this->db->get('members'); // Produces: SELECT MIN(age) as age FROM members

$this->db->select\_avg()

La media se calcula de forma similar.

**$this->db->select\_avg**('age');

$query = $this->db->get('members'); // Produces: SELECT AVG(age) as age FROM members

$this->db->select\_sum()

Y también “SELECT SUM(field)”.

**$this->db->select\_sum**('age');

$query = $this->db->get('members'); // Produces: SELECT SUM(age) as age FROM members

$this->db->from()

Permite determinar la parte FROM de tu consulta:

$this->db->select('title, content, date');

**$this->db->from**('mytable');

$query = $this->db->get(); // Produces: SELECT title, content, date FROM mytable

$this->db->join()

Permite determinar la parte JOIN de tu consulta:

$this->db->select('\*');

$this->db->from('blogs');

$this->db->join('comments', 'comments.id = blogs.id');

$query = $this->db->get();

// Produce:

// SELECT \* FROM blogs JOIN comments ON comments.id = blogs.id

Se puede llamar varias veces si necesitas varios joins en una consulta.

El tipo de JOIN se puede especificar con el tercer parámetro de la función: left, right, outer, inner, left outer, y right outer.

$this->db->join('comments', 'comments.id = blogs.id', 'left');

// Produce: LEFT JOIN comments ON comments.id = blogs.id

## Buscando Datos Específicos

$this->db->where()

Esta función permite crear la clausula WHERE utilizando cuatro métodos distintos:

Método simple key/value:

**$this->db->where('name', $name);** // Produces: WHERE name = 'Joe'

El sígno igual se añade automáticamente.

Si se llama varias veces la función, se encadenan las condiciones con AND entre ellas:

$this->db->where('name', $name);

$this->db->where('title', $title);

$this->db->where('status', $status);

// WHERE name = 'Joe' AND title = 'boss' AND status = 'active'

Método personalizado key/value:

Se incluye un operador en el primer parámetro para controlar la comparación:

$this->db->where('name !=', $name);

$this->db->where('id <', $id); // Produces: WHERE name != 'Joe' AND id < 45

Método de array asociativo:

$array = array('name' => $name, 'title' => $title, 'status' => $status);

$this->db->where($array);

// Produces: WHERE name = 'Joe' AND title = 'boss' AND status = 'active'

Puedes incluir tus propios operadores utilizando este método:

$array = array('name !=' => $name, 'id <' => $id, 'date >' => $date);

$this->db->where($array);

Cadena personalizada:

Puedes escribir tus propias clausulas manualmente:

$where = "name='Joe' AND status='boss' OR status='active'";

$this->db->where($where);

$this->db->where() acepta un tercer parámetro. Si es FALSE, CodeIgniter no intentará proteger tu campo o nombres de tablas.

$this->db->where('MATCH (field) AGAINST ("value")', NULL, FALSE);

$this->db->or\_where()

Esta función es idéntica a la anterior, excepto en que varias llamadas son unidas con OR:

$this->db->where('name !=', $name);

$this->db->or\_where('id >', $id); // Produces: WHERE name != 'Joe' OR id > 50

$this->db->where\_in()

Genera una consulta WHERE field IN (‘item’, ‘item’) con AND si necesario

$names = array('Frank', 'Todd', 'James');

$this->db->where\_in('username', $names);

// Produces: WHERE username IN ('Frank', 'Todd', 'James')

$this->db->or\_where\_in()

Genera una consulta WHERE field IN (‘item’, ‘item’) con OR si necesario

$names = array('Frank', 'Todd', 'James');

$this->db->or\_where\_in('username', $names);

// Produces: OR username IN ('Frank', 'Todd', 'James')

$this->db->where\_not\_in()

Genera una consulta WHERE field NOT IN (‘item’, ‘item’) con AND si necesario

$names = array('Frank', 'Todd', 'James');

$this->db->where\_not\_in('username', $names);

// Produces: WHERE username NOT IN ('Frank', 'Todd', 'James')

$this->db->or\_where\_not\_in()

Genera una consulta WHERE field NOT IN (‘item’, ‘item’) con OR si necesario

$names = array('Frank', 'Todd', 'James');

$this->db->or\_where\_not\_in('username', $names);

// Produces: OR username NOT IN ('Frank', 'Todd', 'James')

## Buscando Datos Similares

$this->db->like()

Este método permite generar clausulas LIKE, lo cuál es útil haciendo búsquedas.

Método clave/valor:

$this->db->like('title', 'match');

// Produce: WHERE `title` LIKE '%match%' ESCAPE '!'

Si la invocas varias veces se concatenarán con AND entre ellas::

$this->db->like('title', 'match');

$this->db->like('body', 'match');

// WHERE `title` LIKE '%match%' ESCAPE '!' AND `body` LIKE '%match% ESCAPE '!'

Si quieres controlar dónde se colocará la wildcard (%), puedes usar un tercer argumento opcional. Tus opciones son ‘before’, ‘after’ y ‘both’ (el valor por defecto).

$this->db->like('title', 'match', 'before'); // Produce: WHERE `title` LIKE '%match' ESCAPE '!'

$this->db->like('title', 'match', 'after'); // Produce: WHERE `title` LIKE 'match%' ESCAPE '!'

$this->db->like('title', 'match', 'both'); // Produce: WHERE `title` LIKE '%match%' ESCAPE '!'

Método de array asociativo:

$array = array('title' => $match, 'page1' => $match, 'page2' => $match);

$this->db->like($array);

// WHERE `title` LIKE '%match%' ESCAPE '!' AND `page1` LIKE '%match%' ESCAPE '!' AND `page2` LIKE '%match%' ESCAPE '!'

$this->db->or\_like()

Este método es idéntico al anterior, excepto que varias llamadas se unen con OR:

$this->db->like('title', 'match'); $this->db->or\_like('body', $match);

// WHERE `title` LIKE '%match%' ESCAPE '!' OR `body` LIKE '%match%' ESCAPE '!'

$this->db->not\_like()

Idéntico a like(), excepto que genera una consulta NOT LIKE:

$this->db->not\_like('title', 'match'); // WHERE `title` NOT LIKE '%match% ESCAPE '!'

$this->db->or\_not\_like()

Este método es idéntico a not\_like(), excepto que varias llamadas se unen usando OR:

$this->db->like('title', 'match');

$this->db->or\_not\_like('body', 'match');

// WHERE `title` LIKE '%match% OR `body` NOT LIKE '%match%' ESCAPE '!'

$this->db->group\_by()

Te permite escribir la porción GROUP BY de tu consulta:

$this->db->group\_by("title"); // Produces: GROUP BY title

You can also pass an array of multiple values as well:

$this->db->group\_by(array("title", "date")); // Produce: GROUP BY title, date

$this->db->distinct()

Añade la palabra clave “DISTINCT” a una consulta:

$this->db->distinct();

$this->db->get('table'); // Produce: SELECT DISTINCT \* FROM table

$this->db->having()

Te permite escribir la porción HAVING de tu consulta. Hay dos sintáxis posibles, 1 argumento o 2:

$this->db->having('user\_id = 45'); // Produce: HAVING user\_id = 45

$this->db->having('user\_id', 45); // Produce: HAVING user\_id = 45

Puedes pasar también un array con multiples valores:

$this->db->having(array('title =' => 'My Title', 'id <' => $id));

// Produce: HAVING title = 'My Title', id < 45

Si estas usando una base de datos y quieres que el contenido de las consultas no se escape, puedes utilizar un tercer parámetro opcional con el valor FALSE.

$this->db->having('user\_id', 45); // Produce: HAVING `user\_id` = 45 en añgunas bases de datos como MySQL

$this->db->having('user\_id', 45, FALSE); // Produces: HAVING user\_id = 45

$this->db->or\_having()

Idéntico a having(), pero separando multiples clausulas con “OR”.

***Ordenando resultados***

$this->db->order\_by()

Permite crear una clausula ORDER BY.

El primer parámetro contiene el nombre de la columna por la que te gustaría ordenar.

El segundo parámetro nos permite determinar la dirección del resultado: ASC, DESC o RANDOM.

$this->db->order\_by('title', 'DESC');

// Produce: ORDER BY `title` DESC

También puedes pasar tu propia cadena en el primer parámetro:

$this->db->order\_by('title DESC, name ASC');

// Produce: ORDER BY `title` DESC, `name` ASC

Se pueden realizar multiples llamadas su necesitas ordenar por varios campos.

$this->db->order\_by('title', 'DESC');

$this->db->order\_by('name', 'ASC');

// Produce: ORDER BY `title` DESC, `name` ASC

Si eliges la opción RANDOM, entonces el primer parámetro será ignorado a menos que especifiques un valor de semilla numérico.

$this->db->order\_by('title', 'RANDOM');

// Produce: ORDER BY RAND()

$this->db->order\_by(42, 'RANDOM');

// Produce: ORDER BY RAND(42)

***Limitando o Contando Resultados***

$this->db->limit()

Te permite limitar el número de filas que quieres que devuelva la consulta:

$this->db->limit(10); // Produce: LIMIT 10

El segundo parámetro permite definir un offset.

$this->db->limit(10, 20); // Produce: LIMIT 20, 10 (en MySQL)

$this->db->count\_all\_results()

Te permite determinar el numero de filas en una consulta activa concreta. Las consultas aceptarán todos los restrictores Query Builder tales como where(), or\_where(), like(), or\_like(), etc. Ejemplo:

echo $this->db->count\_all\_results('my\_table'); // Produce un entero, como 25

$this->db->like('title', 'match');

$this->db->from('my\_table');

echo $this->db->count\_all\_results(); // Produce un entero, como 17

Sin embargo, esto método resetea cualquier valor que puedas haber pasado con select(). Si necesitas mantenerlos, puedes pasar FALSE como el segundo parámetro:

echo $this->db->count\_all\_results('my\_table', FALSE);

$this->db->count\_all()

Te permite determinar el numero de líneas en una tabla concreta. Envía el nombre de la tabla como primer parámetro. Ejemplo:

echo $this->db->count\_all('my\_table'); // Produce un entero

***Agrupar consultas***

Agrupar consultas te permite crear grupos de clausulas WHERE poniéndolas entre paréntesis. Esto te permitirá crear consultas con clausulas WHERE complejas. Se permiten grupos anidados.

**Ejemplo:**

$this->db->select('\*')->from('my\_table')

->group\_start()

->where('a', 'a')

->or\_group\_start()

->where('b', 'b')

->where('c', 'c')

->group\_end()

->group\_end()

->where('d', 'd')

->get();

// Genera:

// SELECT \* FROM (`my\_table`) WHERE ( `a` = 'a' OR ( `b` = 'b' AND `c` = 'c' ) ) AND `d` = 'd'

$this->db->group\_start()

Inicia un nuevo grupo al añadir un paréntesis a la clausula WHERE de la consulta.

$this->db->or\_group\_start()

Inicia un nuevo grupo añadiendo un paréntesis abierto a la clausula WHERE, precediéndola con ‘OR’.

$this->db->not\_group\_start()

Inicia un nuevo grupo añadiendo un paréntesis abierto a la clausula WHERE, precediéndola con ‘NOT’.

$this->db->or\_not\_group\_start()

Inicia un nuevo grupo añadiendo un paréntesis abierto a la clausula WHERE, precediéndola con ‘OR NOT’.

$this->db->group\_end()

Finaliza el grupo actual añadiendo un paréntesis de cierre a la clausula WHERE de la consulta.

***Insertando Datos***

$this->db->insert()

Genera an insert string based on the data you supply, and runs the query. You can either pass an array or an object to the function. Here is an example using an array:

$data = array(

'title' => 'My title',

'name' => 'My Name',

'date' => 'My date'

);

$this->db->insert('mytable', $data);

// Produce: INSERT INTO mytable (title, name, date) VALUES ('My title', 'My name', 'My date')

El primer parámetro contendrá el nombre de la una tabla, el segundo es un array asociativo de valores.

Aquí hay un ejemplo simple usando un objeto.

/\*

class Myclass {

public $title = 'My Title';

public $content = 'My Content';

public $date = 'My Date';

}

\*/

$object = new Myclass;

$this->db->insert('mytable', $object);

// Produce: INSERT INTO mytable (title, content, date) VALUES ('My Title', 'My Content', 'My Date')

El primer parámetro contendrá el nombre de la tabla, el segundo es un objeto.

$this->db->get\_compiled\_insert()

Compila la consulta de inserción como $this->db->insert() perp no ejecuta la consulta. Este método simplemente devuelve la consulta SQL como una cadena.

**Ejemplo:**

$data = array(

'title' => 'My title',

'name' => 'My Name',

'date' => 'My date'

);

$sql = $this->db->set($data)->get\_compiled\_insert('mytable');

echo $sql;

// Produce: INSERT INTO mytable (`title`, `name`, `date`) VALUES ('My title', 'My name', 'My date')

El segundo parámetro te permite indicarle si la consulta será reiniciada (reset) (aunque por defecto lo será, como $this->db->insert()):

echo $this->db->set('title', 'My Title')->get\_compiled\_insert('mytable', FALSE);

// Produce la cadena: INSERT INTO mytable (`title`) VALUES ('My Title')

echo $this->db->set('content', 'My Content')->get\_compiled\_insert();

// Produce la cadena: INSERT INTO mytable (`title`, `content`) VALUES ('My Title', 'My Content')

Hay que remarcar que la segunda consulta no utiliza $this->db->from() ni le pasa un nombre de tabla como primer parámetro. La razón es que la consulta no se ha ejecutado con $this->db->insert() que resetea los valores o directamente utilizando $this->db->reset\_query().

Nota

El método no funciona con inserciones en modo batch.

$this->db->insert\_batch()

Genera una cadena de inserción basada en los datos que le has proporcionado y ejecuta la consulta. Puedes pasarle un array o un objeto a la funcción. Un ejemplo:

$data = array(

array(

'title' => 'My title',

'name' => 'My Name',

'date' => 'My date'

),

array(

'title' => 'Another title',

'name' => 'Another Name',

'date' => 'Another date'

)

);

$this->db->insert\_batch('mytable', $data);

// Produce: INSERT INTO mytable (title, name, date) VALUES ('My title', 'My name', 'My date'), ('Another title', 'Another name', 'Another date')

El primer parámetro contendrá el nombre de la tabla, el segundo es un array asociativo de valores.

***Actualizando Datos***

$this->db->replace()

Este método ejecuta un comando REPLACE, que es básicamente el estándar SQL para DELETE + INSERT, utilizando claves PRIMARY y UNIQUE como el factor determinante. En nuestro caso, nos librará de la necesidad de emplear lógica compleja con diferentes combinaciones de llamadas select(), update(), delete() e insert().

**Ejemplo:**

$data = array(

'title' => 'My title',

'name' => 'My Name',

'date' => 'My date'

);

$this->db->replace('table', $data);

// Ejecuta: REPLACE INTO mytable (title, name, date) VALUES ('My title', 'My name', 'My date')

En el ejemplo anterior, si asumimos que el campo título es nuestra clave primaria, entonces si una fila conteniendo ‘My title’ como valor del título, esa fila será borrada con nuestra fila reemplazándola.

El uso del método set() también está permitido y todos los campos son escapados automáticamente, como con insert().

$this->db->set()

Esta función te permite dar valores para inserts o updates.

Puede usarse en lugar de pasar un array de datos directamente a las funciones insert o update:

$this->db->set('name', $name);

$this->db->insert('mytable'); // Produce: INSERT INTO mytable (`name`) VALUES ('{$name}')

Si llamas varias veces a las funciones, se montarán adecuadamente según estes haciendo un insert o un update:

$this->db->set('name', $name);

$this->db->set('title', $title);

$this->db->set('status', $status);

$this->db->insert('mytable');

set() también aceptará un tercer parámetro opcional ($escape), que evitará que los datos sean escapados si su valor es FALSE. Para ilustrar la diferencia, aquí set() se usa con y sin el parámetro de escape.

$this->db->set('field', 'field+1', FALSE);

$this->db->where('id', 2);

$this->db->update('mytable'); // da UPDATE mytable SET field = field+1 WHERE id = 2

$this->db->set('field', 'field+1');

$this->db->where('id', 2);

$this->db->update('mytable'); // da UPDATE `mytable` SET `field` = 'field+1' WHERE `id` = 2

También puedes pasarle un array asociativo a esta función:

$array = array(

'name' => $name,

'title' => $title,

'status' => $status

);

$this->db->set($array);

$this->db->insert('mytable');

Or an object:

/\*

class Myclass {

public $title = 'My Title';

public $content = 'My Content';

public $date = 'My Date';

}

\*/

$object = new Myclass;

$this->db->set($object);

$this->db->insert('mytable');

$this->db->update()

Genera una cadena de update (actualización) y ejecuta la consulta basándose en los datos que tu le proporcionan. Puedes pasar un array o un objeto a la función. Aquí hay un ejemplo utilizando un array:

$data = array(

'title' => $title,

'name' => $name,

'date' => $date

);

$this->db->where('id', $id);

$this->db->update('mytable', $data);

// Produce:

//

// UPDATE mytable

// SET title = '{$title}', name = '{$name}', date = '{$date}'

// WHERE id = $id

O puedes proporcionar un objeto:

/\*

class Myclass {

public $title = 'My Title';

public $content = 'My Content';

public $date = 'My Date';

}

\*/

$object = new Myclass;

$this->db->where('id', $id);

$this->db->update('mytable', $object);

// Produce:

//

// UPDATE `mytable`

// SET `title` = '{$title}', `name` = '{$name}', `date` = '{$date}'

// WHERE id = `$id`

Nota

Todos los valores son escapados automáticamente produciendo consultas más seguras.

Aunque se usa la función $this->db->where() también se puede proporcionar esta información directamente a la función de update como una cadena:

$this->db->update('mytable', $data, "id = 4");

O como un array:

$this->db->update('mytable', $data, array('id' => $id));

También puedes usar la función $this->db->set() descrito encima cuando se realizan updates.

$this->db->update\_batch()

Genera una cadena de update basada en los datos que proporcionas y ejecuta la consulta. Puedes pasar un array o un objeto a la función. Aquí hay un ejemplo usando un array:

$data = array(

array(

'title' => 'My title' ,

'name' => 'My Name 2' ,

'date' => 'My date 2'

),

array(

'title' => 'Another title' ,

'name' => 'Another Name 2' ,

'date' => 'Another date 2'

)

);

$this->db->update\_batch('mytable', $data, 'title');

// Produce:

// UPDATE `mytable` SET `name` = CASE

// WHEN `title` = 'My title' THEN 'My Name 2'

// WHEN `title` = 'Another title' THEN 'Another Name 2'

// ELSE `name` END,

// `date` = CASE

// WHEN `title` = 'My title' THEN 'My date 2'

// WHEN `title` = 'Another title' THEN 'Another date 2'

// ELSE `date` END

// WHERE `title` IN ('My title','Another title')

El primer parámetro contendrá el nombre de la tabla, el segundo es un array asociativo de valores, el tercero es la clave del where.

Nota

affected\_rows() no te dará resultados correctos con este método, debido a como funciona. En su lugar, update\_batch() devuelve el número de filas afectadas.

$this->db->get\_compiled\_update()

Esto funciona exactamente de la misma forma que $this->db->get\_compiled\_insert() excepto que produce una cadena UPDATE SQL en lugar de una cadena INSERT SQL.

Nota

Este método no funciona para updates batched.

***Borrando Datos***

$this->db->delete()

Genera una cadena SQL delete y ejecuta la consulta.

$this->db->delete('mytable', array('id' => $id)); // Produce: // DELETE FROM mytable // WHERE id = $id

El primer parámeter es el nombre de la tabla, el segundo es la clausula where. También puedes utilizar las funciones where() o or\_where() en lugar de pasar los datos al segundo parámetro de la función:

$this->db->where('id', $id);

$this->db->delete('mytable');

// Produce:

// DELETE FROM mytable

// WHERE id = $id

Un array de nombres de tablas puede pasarse a delete() si quieres borrar más de una tabla.

$tables = array('table1', 'table2', 'table3');

$this->db->where('id', '5');

$this->db->delete($tables);

Si quieres borrar todos los datos de una tabla, puedes usar la función truncate(), o empty\_table().

$this->db->empty\_table()

Genera una cadena SQL delete SQL y ejecuta la consulta:

$this->db->empty\_table('mytable'); // Produce: DELETE FROM mytable

$this->db->truncate()

Genera una cadena SQL truncate y la ejecuta.

$this->db->from('mytable');

$this->db->truncate();

// o

$this->db->truncate('mytable');

// Produce:

// TRUNCATE mytable

Nota

Si el comando TRUNCATE no está disponible, truncate() se ejecutará como “DELETE FROM table”.

$this->db->get\_compiled\_delete()

Esto funciona exactamente de la misma forma que $this->db->get\_compiled\_insert() excepto que produce una cadena SQL DELETE en lugar de una cadena SQL INSERT.

***Método de Encadenado***

El encadenado de métodos te permite simplificar tu sintáxis al conectar multiples funciones. Por ejemplo:

$query = $this->db->select('title')

->where('id', $id)

->limit(10, 20)

->get('mytable');

***Query Builder Caching***

Aunque no es un auténtico almacenado, Query Builder te permite guardar ciertas partes de tus consultas para reutilizarlas más adelante en la ejecución de tu script. Normalmente en QB, cuando se completa una llamada, toda la información almacenada se resetea para la próxima llamada. Con “caching” puedes evitarlo y reutilizar información fácilmente.

Las llamadas son acumulativas. Si haces dos llamadas select() almacenadas, y luego dos llamadas no almacenadas, esto dará lugar a cuatro llamadas select(). Hay tres funciones de Caching disponibles:

$this->db->start\_cache()

Se debe llamar a esta función para empezar el proceso. Todas las consultas QB del tipo correcto son almacenadas para uso posterior.

$this->db->stop\_cache()

Se puede llamar esta función para detener el almacenaje.

$this->db->flush\_cache()

Esta función vacía el almacenaje.

Un ejemplo de uso de caching

$this->db->start\_cache();

$this->db->select('field1');

$this->db->stop\_cache();

$this->db->get('tablename');

//Genera: SELECT `field1` FROM (`tablename`)

$this->db->select('field2');

$this->db->get('tablename');

//Genera: SELECT `field1`, `field2` FROM (`tablename`)

$this->db->flush\_cache();

$this->db->select('field2');

$this->db->get('tablename');

//Generates: SELECT `field2` FROM (`tablename`)

Nota

Se pueden almacenar las siguientes instrucciones: select, from, join, where, like, group\_by, having y order\_by

***Resetting Query Builder***

$this->db->reset\_query()

Resetear Query Builder te permite empezar desde cero con tu consulta sin ejecutarla primero con métodos como $this->db->get() o $this->db->insert(). Pero esto no reseteará lo que hayas almacenado con Query Builder Caching.

Esto es útil en situaciones en las que utilizamos Query Builder para generae SQL (ex. $this->db->get\_compiled\_select()) pero entonces decidimos, por ejemplo, ejecutar la consulta:

$sql = $this->db->select(array('field1','field2'))

->where('field3',5)

->get\_compiled\_select('mytable', FALSE);

// ...

// Haz algo loco con el código SQL...

// ...

$data = $this->db->get()->result\_array();

// Ejecutará y devolverá un array de resultados de la siguiente consulta:

// SELECT field1, field1 from mytable where field3 = 5;

Nota

Repetir la llamada a get\_compiled\_select() mientras estás usando la funcionalidad de Query Builder Caching y NO reseteando tus consultas dará como resultado que la caché se una dos veces. Eso puede llevar a que se seleccione el mismo campo dos veces, por ejemplo.