SQL Builder

The ClickBlocks SQL Builder provides an object-oriented way of writing SQL statements. It allows developers to use class methods and properties to specify individual parts of a SQL statement. It then assembles different parts into a valid SQL statement that can be further executed by calling method **execute()** of class **ClickBlocks\DB\DB**.

Despite the fact that the SQL Builder is a helper class that used in the base classes like **ClickBlocks\DB\DB** or **ClickBlocks\DB\AR**, you can use it in your application if you need. This approach has a number advantages:

- SQL Builder allows to construct complex SQL statements programmatically.
- SQL statements that written via SQL Builder are independent on the particular DBMS.
- SQL Builder automatically quotes table names and column names to prevent possible SQL injection attacks and escape reserved words and special characters.
- SQL Builder always uses parameter binding that also allows to prevent SQL injections.

But this also has some disadvantages:

- Separate methods of the class have complex structure of their parameters.
- SQL Builder supports only limited subset of SQL syntax according to the last SQL Standard (However it supports LIMIT construction).

There is two main ways to get an instance of SQL Builder for the particular DBMS:

- Use property **sql** of class **ClickBlocks\DB\DB**.

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// Form SQL statement.
$query = $sql->dropTable('MyTable');
// Execute SQL statement.
$db->execute($query);
```

- Use static method **getInstance()** of class **ClickBlocks\DB\SQLBuilder**. This method takes as a single parameter the name of the required DBMS.

```
// Get an instance of SQL Builder.
$sql = SQLBuilder::getInstance('mysql');
// Form SQL statement.
$query = $sql->dropTable('MyTable');
// Execute SQL statement.
$db->execute($query);
```

All methods of SQL Builder can be divided to three groups:

1. Methods allowing to build data retrieval queries.
2. Methods allowing to build data manipulation queries.
3. Methods allowing to build schema manipulation queries.

Let's examine these method groups in details.

Building data retrieval queries

The SQL Builder provides a number methods for building different parts of SELECT queries. Because all these methods return the **ClickBlocks\DB\SQLBuilder** instance, we can call them using method chaining.

- **select()** - specifies the SELECT part of the query.
- **join()** - appends a join query fragment.
- **where()** - specifies the WHERE part of the query.
- **group()** - specifies the GROUP BY part of the query.
- **having()** - specifies the HAVING part of the query.
- **order()** - specifies the ORDER BY part of the query.
- **limit()** - specifies the LIMIT part of the query.

Examples below explain usage of these methods (for MySQL database).

select()

```
select(mixed $table, mixed $columns = '*', string $distinct = null, array $options = null)
```

The **select()** method specifies the SELECT part of a query. The **$table** parameter specifies which table(s) to be selected from. This parameter can be an array of table name(s), a string or an instance of **ClickBlocks\DB\SQLExpression**. The **$columns** parameter specifies the columns to be selected, which can be an array of column names, a string or an instance of **ClickBlocks\DB\SQLExpression**. The **$distinct** and **$options** are additional select options for some DBMS.

Below are some examples:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// SELECT * FROM `MyTable`
$q = $sql->select('MyTable')->build();
// SELECT `c1`, `c2` `param` FROM `tb1` `t`, `tb2`
$q = $sql->select(['tb1' => 't', 'tb2'], ['c1', 'c2' => 'param'])->build();
// SELECT c1, COUNT(*) AS c2 FROM tb AS t
$q = $sql->select('tb AS t', 'c1, COUNT(*) AS c2')->build();
// SELECT DISTINCT c1, COUNT(*) AS c2 FROM tb AS t
$q = $sql->select(new SQLExpression('tb AS t'), new SQLExpression('c1, COUNT(*) AS c2'), 'DISTINCT')->build();
```

As you can see when **$table** or **$columns** are a string or an instance of **ClickBlocks\DB\SQLExpression** then they are taken as is, without changes.

Method **build()** that ends each chain is designed for forming SQL string and getting array of query parameter values (see **where()** method).

where()

```
where(mixed $conditions)
```

The **where()** method specifies the WHERE part of a query. The **$conditions** parameter specifies the main query conditions. It can be an array of complex structure, a string or an instance of **ClickBlocks\DB\SQLExpression**. The most typical cases of usage of the method are shown below:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// ... WHERE c1 = ? AND c2 = ?
$q = $sql->select('tb')->where(['c1' => 'a', 'c2' => 'b'])->build($data);
// ['a', 'b']
print_r($data);
// ... WHERE c1 = ? AND c2 = ? OR c3 = ? AND c4 = ?
$q = $sql->select('tb')
        ->where(['or', ['and', 'c1' => 'a', 'c2' => 'b'], ['and', 'c3' => 'c', 'c4' => 'd']])
        ->build($data);
// ['a', 'b', 'c', 'd']
print_r($data);
// ... WHERE `c1` > ? AND `c2` LIKE ? AND (`c3` = ? OR `c4` IN (?, ?, ?)) AND `c5` IS NOT NULL AND `c6` BETWEEN 10 AND 100
$q = $sql->select('tb')
        ->where([['>', 'c1', 5],
                ['LIKE', 'c2', 'a'],
                ['or', 'c3' => 111, ['IN', 'c4', [1, 2, 3]]],
                ['IS', 'c5', 'NOT NULL'],
                ['BETWEEN', 'c6', 10, 100]])
        ->build($data);
// [5, 'a', 111, 1, 2, 3, 10, 100]
print_r($data);
```

Method **build** takes as a single parameter a variable to which an array of query parameters will be assigned to.

join()

```
join(mixed $table, mixed $conditions, string $type = 'INNER')
```

The **join()** method appends a join query fragment. The **$table** parameter specifies which table(s) to be joined with. It can be an array of table name(s), a string or an instance of **ClickBlocks\DB\SQLExpression**. The **$conditions** parameter specifies the join conditions. Its syntax is the same as that in **where()**. The **$type** parameter specify the type of join part. It can be different for different DBMS.

Below are some examples:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// ... FROM `t1` INNER JOIN `t2` ON t2.ID = t1.ID AND `t2`.`c1` = ?
$q = $sql->select('t1')
         ->join('t2', ['t2.ID = t1.ID', 't2.c1' => 5])
         ->build($data);
// [5]
print_r($data);
// ... FROM `t1` LEFT JOIN `t2`, `t3` ON t2.ID = t1.ID AND t3.ID = t2.ID
$q = $sql->select('t1')
         ->join('t2', 't2.ID = t1.ID AND t3.ID = t2.ID', 'LEFT')
         ->build();
```

group()

```
group(mixed $group)
```

The **group()** method specifies the GROUP BY part of a query. The **$group** parameter specifies the group conditions. It can be an array, a string or an instance of **ClickBlocks\DB\SQLExpression**.

Below are some examples:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// ... GROUP BY `c1`, `c2`
$q = $sql->select('tb')->group(['c1', 'c2'])->build();
// ... GROUP BY CONCAT(c1, c2)
$q = $sql->select('tb')->group('CONCAT(c1 ,c2)')->build();
```

having()

```
having(mixed $conditions)
```

The **having()** method specifies the HAVING part of a query. The **$conditions** parameter specifies the having conditions. The format of this parameter is the same as the format of the argument of the method **where()**.

Below are some examples:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// ... HAVING `c1` = ? AND `c2` IN (?, ?)
$q = $sql->select('tb')
         ->having(['c1' => 5, ['IN', 'c2', ['a', 'b']]])
         ->build($data);
// [5, 'a', 'b']
print_r($data);
// ... HAVING COUNT(*) > 6 OR `c1` = ?
$q = $sql->select('tb')
         ->having(['or', 'COUNT(*) > 6', 'c1' => 3])
         ->build($data);
// [3]
print_r($data);
```

order()

```
order(mixed $order)
```

The **order()** method specifies the ORDER BY part of a query. The **$order** parameter specifies the order conditions. It can be an array, a string or an instance of **ClickBlocks\DB\SQLExpression**.

Below are some examples:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// ... ORDER BY `c1`, `c2` DESC
$q = $sql->select('tb')->order(['c1', 'c2' => 'DESC'])->build();
// ... ORDER BY CONCAT(c1, c2), `c2` ASC
$q = $sql->select('tb')->order(['CONCAT(c1, c2)', 'c2' => 'ASC'])->build();
```

limit()

```
limit(integer $limit, integer $offset = null)
```

The **limit()** methods specify the LIMIT part of a query. Note that some DBMS may not support LIMIT syntax. In this case, the SQL Builder will rewrite the whole SQL statement to simulate the function of limit.

Below are some examples:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// ... LIMIT 10
$q = $sql->select('tb')->limit(10)->build();
// ... LIMIT 5, 10
$q = $sql->select('tb')->limit(10, 5)->build();
```

Building data manipulation queries

Data manipulation queries are all queries of INSERT, UPDATE and DELETE types. The SQL Builder has the appropriate methods allowing to form these queries.

insert()

```
insert(mixed $table, mixed $columns, array $options = null)
```

The method builds INSERT SQL statement. The **$table** parameter specifies which table to be inserted into, while **$columns** specifies column values to be inserted. Value of **$columns** can be an array of of name-value pairs, a string or an instance of **ClickBlocks\DB\SQLExpression**. The **$options** parameter specifies additional options that make sense for some DBMS.

Below are some examples:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// ... INSERT INTO `tb` (`ID`, `name`) VALUES (?, ?)
$q = $sql->insert('tb', ['ID' => 5, 'name' => 'test'])->build($data);
// [5, 'test']
print_r($data);
// ... INSERT INTO `tb` (`ID`, `name`) VALUES (?, ?), (?, ?), (?, ?)
$q = $sql->insert('tb', ['ID' => [1, 2, 3], 'name' => ['a', 'b', 'c']])
         ->build($data);
// [1, 'a', 2, 'b', 3, 'c']
print_r($data);
// ... INSERT INTO `tb` (`ID`, `name`) VALUES (?, ?)
//     ON DUPLICATE KEY UPDATE `ID` = ?, `name` = ?
$q = $sql->insert('tb', ['ID' => 5, 'name' => 'test'],
```

```
                ['updateOnKeyDuplicate' => true])->build();
// [5, 'test', 5, 'test']
print_r($data);
```

update()

```
update(mixed $table, mixed $columns, array $options = null)
```

This methods is intended for building UPDATE SQL statements. The **$table** parameter specifies which table to be updated. The **$columns** parameter specifies column values to be updated. It can be an array of of name-value pairs, a string or an instance of **ClickBlocks\DB\SQLExpression**. At present, the **$options** parameter has fictitious value.

Below are some examples:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// UPDATE `tb` SET `c1` = ?, `c2` = ?
$q = $sql->update('tb', ['c1' => 5, 'c2' => 'a'])->build($data);
// [5, 'a']
print_r($data);
// UPDATE `tb` SET c1 = c1 + 1, `c2` = ?
$q = $sql->update('tb', ['c1 = c1 + 1', 'c2' => 5])->build($data);
// [5]
print_r($data);
```

delete()

```
delete(mixed $table, array $options = null)
```

This methods is intended for building DELETE SQL statements. The **$table** parameter specifies which table to be deleted. At present, the **$options** parameter has fictitious value.

Below is an example:

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// DELETE FROM `tb` ...
$q = $sql->delete('tb')->build();
```

> Note, that you can combine method **insert()**, **update()** or **delete()** with other methods of building common parts of queries.

For example,

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// UPDATE `tb` SET `c1` = ? WHERE `c2` > ? AND `c1` = ?
$q = $sql->update('tb', ['c1' => 5])
         ->where([['>', 'c2', 10], 'c1' => '100'])
         ->build($data);
// [5, 10, 100]
print_r($data);
```

Building schema manipulation queries

Besides normal data retrieval and manipulation queries, the SQL Builder also offers a set of methods for building SQL queries that can manipulate the schema of a database. In particular, it supports the following queries:

- **tableList()** - returns SQL for getting the table list of the current database.
- **tableInfo()** - returns SQL for getting the metadata of the given table.
- **columnsInfo()** - returns SQL for getting the metadata of the table columns.
- **createTable()** - returns SQL for creating a new database table.

- **renameTable()** - returns SQL for renaming a database table.
- **dropTable()** - returns SQL for removing a database table.
- **truncateTable()** - returns SQL for deleting all rows from a table.
- **addColumn()** - returns SQL for adding a new table column.
- **renameColumn()** - returns SQL for renaming a table column.
- **changeColumn()** - returns SQL for changing definition of a table column.
- **dropColumn()** - returns SQL for dropping a table column.
- **addForeignKey()** - returns SQL for adding a foreign key constraint to an existing table.
- **dropForeignKey()** - returns SQL for dropping a foreign key constraint.
- **createIndex()** - returns SQL for creating a new index.
- **dropIndex()** - returns SQL for removing an index.

You can find out more information about these methods in the class reference.

The SQL Builder has two important methods allowing to quote string values as well as table or column names for use in SQL queries: **quote()** and **wrap()**. Some examples (for MySQL):

```
// Get an instance of SQL Builder.
$sql = $db->sql;
// Create an SQL statement.
$s = 'SELECT ' . $sql->wrap('my column') . '
      FROM ' . $sql->wrap('my table', true) . '
      WHERE c1 <> ' . $sql->quote("test's") . ' AND
            c2 LIKE ' . $sql->quote('a%bc', SQLBuilder::ESCAPE_LEFT_LIKE);
// It will output something like this:
// SELECT `my column` FROM `my table` WHERE c1 <> 'test\s' AND c2 LIKE '%a\%bc'
echo $s;
```

The first parameter of method **wrap()** is the name of a column or table. And the second one is the boolean value that determines whether a table name is used. By default the second parameter is FALSE.

The first parameter of method **quote()** is any column value, and the second one is one of the `SQLBuilder::ESCAPE_*` constants that determine the format of the quoted value.

> Class **ClickBlocks\DB\DB** has aliases of these methods with the same names.