
SphinxQL Query Builder

Release 1.0.0

Jun 01, 2018

Contents

1	Introduction	1
1.1	Compatibility	1
2	CHANGELOG	3
2.1	What's New in 1.0.0	3
3	Configuration	5
3.1	Obtaining a Connection	5
3.2	Connection Parameters	5
4	SphinxQL Query Builder	7
4.1	Creating a Query Builder Instance	7
4.2	Building a Query	7
4.3	COMPILE	10
4.4	EXECUTE	10
5	Multi-Query Builder	13
6	Facets	15
7	Contribute	17
7.1	Pull Requests	17
7.2	Coding Style	17
7.3	Testing	17
7.4	Issue Tracker	17

The SphinxQL Query Builder provides a simple abstraction and access layer which allows developers to generate SphinxQL statements which can be used to query an instance of the Sphinx search engine for results.

1.1 Compatibility

SphinxQL Query Builder is tested against the following environments:

- PHP 5.6 and later
- Sphinx (Stable)
- Sphinx (Development)

Note: It is recommended that you always use the latest stable version of Sphinx with the query builder.

CHAPTER 2

CHANGELOG

2.1 What's New in 1.0.0

3.1 Obtaining a Connection

You can obtain a SphinxQL Connection with the *Foolz\SphinxQL\Drivers\Mysqli\Connection* class.

```
<?php
use Foolz\SphinxQL\Drivers\Mysqli\Connection;

$conn = new Connection();
$conn->setparams(array('host' => '127.0.0.1', 'port' => 9306));
```

Warning: The existing PDO driver written is considered experimental as the behaviour changes between certain PHP releases.

3.2 Connection Parameters

The connection parameters provide information about the instance you wish to establish a connection with. The parameters required is set with the *setParams(\$array)* or *setParam(\$key, \$value)* methods.

host

Type string

Default 127.0.0.1

port

Type int

Default 9306

socket

Type string

Default null

options

Type array

Default null

SphinxQL Query Builder

4.1 Creating a Query Builder Instance

You can create an instance by using the following code and passing a configured *Connection* class.

```
<?php

use Foolz\SphinxQL\Drivers\Mysqli\Connection;
use Foolz\SphinxQL\SphinxQL;

$conn = new Connection();
$queryBuilder = SphinxQL::create($conn);
```

4.2 Building a Query

The *Foolz\SphinxQL\SphinxQL* class supports building the following queries: *SELECT*, *INSERT*, *UPDATE*, and *DELETE*. Which sort of query being generated depends on the methods called.

For *SELECT* queries, you would start by invoking the *select()* method:

```
$queryBuilder
->select('id', 'name')
->from('index');
```

For *INSERT*, *REPLACE*, *UPDATE* and *DELETE* queries, you can pass the index as a parameter into the following methods:

```
$queryBuilder
->insert('index');

$queryBuilder
->replace('index');
```

(continues on next page)

(continued from previous page)

```
$queryBuilder
->update('index');

$queryBuilder
->delete('index');
```

Note: You can convert the query builder into its compiled SphinxQL dialect string representation by calling `$queryBuilder->compile()->getCompiled()`.

4.2.1 Security: Bypass Query Escaping

```
SphinxQL::expr($string)
```

4.2.2 Security: Query Escaping

```
$queryBuilder
->escape($value);
```

```
$queryBuilder
->quoteIdentifier($value);
```

```
$queryBuilder
->quote($value);
```

```
$queryBuilder
->escapeMatch($value);
```

```
$queryBuilder
->halfEscapeMatch($value);
```

4.2.3 WHERE Clause

The *SELECT*, *UPDATE* and *DELETE* statements supports the *WHERE* clause with the following API methods:

```
// WHERE `column` = 'value'
$queryBuilder
->where($column, $value);

// WHERE `column` = 'value'
$queryBuilder
->where($column, '=', $value);

// WHERE `column` >= 'value'
$queryBuilder
->where($column, '>=', $value)
```

(continues on next page)

(continued from previous page)

```
// WHERE ` $column ` IN ( '$value1', '$value2', '$value3' )
$queryBuilder
  ->where($column, 'IN', array($value1, $value2, $value3));

// WHERE ` $column ` NOT IN ( '$value1', '$value2', '$value3' )
$queryBuilder
  ->where($column, 'NOT IN', array($value1, $value2, $value3));

// WHERE ` $column ` BETWEEN '$value1' AND '$value2'
$queryBuilder
  ->where($column, 'BETWEEN', array($value1, $value2));
```

Warning: Currently, the SphinxQL dialect does not support the *OR* operator and grouping with parenthesis.

4.2.4 MATCH Clause

MATCH extends the *WHERE* clause and allows for full-text search capabilities.

```
$queryBuilder
  ->match($column, $value, $halfEscape = false);
```

By default, all inputs are automatically escaped by the query builder. The usage of *SphinxQL::expr(\$value)* can be used to bypass the default query escaping and quoting functions in place during query compilation. The *\$column* argument accepts a string or an array. The *\$halfEscape* argument, if set to *true*, will not escape and allow the usage of the following special characters: -, |, and “.

4.2.5 SET Clause

```
$queryBuilder
  ->set($associativeArray);
```

```
$queryBuilder
  ->value($column1, $value1)
  ->value($column2, $value2);
```

```
$queryBuilder
  ->columns($column1, $column2, $column3)
  ->values($value1_1, $value2_1, $value3_1)
  ->values($value1_2, $value2_2, $value3_2);
```

4.2.6 GROUP BY Clause

The *GROUP BY* supports grouping by multiple columns or computed expressions.

```
// GROUP BY $column
$queryBuilder
  ->groupBy($column);
```

4.2.7 WITHIN GROUP ORDER BY

The *WITHIN GROUP ORDER BY* clause allows you to control how the best row within a group will be selected.

```
// WITHIN GROUP ORDER BY $column [$direction]
$queryBuilder
    ->withinGroupOrderBy($column, $direction = null);
```

4.2.8 ORDER BY Clause

Unlike in regular SQL, only column names (not expressions) are allowed.

```
// ORDER BY $column [$direction]
$queryBuilder
    ->orderBy($column, $direction = null);
```

4.2.9 OFFSET and LIMIT Clause

```
// LIMIT $offset, $limit
$queryBuilder
    ->limit($offset, $limit);
```

```
// LIMIT $limit
$queryBuilder
    ->limit($limit);
```

4.2.10 OPTION Clause

The *OPTION* clause allows you to control a number of per-query options.

```
// OPTION $name = $value
$queryBuilder
    ->option($name, $value);
```

4.3 COMPILE

You can have the query builder compile the generated query for debugging with the following method:

```
$queryBuilder
    ->compile();
```

This can be used for debugging purposes and obtaining the resulting query generated.

4.4 EXECUTE

In order to run the query, you must invoke the *execute()* method so that the query builder can compile the query for execution and then return the results of the query.

```
$queryBuilder  
->execute();
```


CHAPTER 5

Multi-Query Builder

```
$queryBuilder  
->enqueue(SphinxQL $next = null);
```

```
$queryBuilder  
->executeBatch();
```


CHAPTER 6

Facets

7.1 Pull Requests

1. Fork [SphinxQL Query Builder](#)
2. Create a new branch for each feature or improvement
3. Submit a pull request with your branch against the master branch

It is very important that you create a new branch for each feature, improvement, or fix so that may review the changes and merge the pull requests in a timely manner.

7.2 Coding Style

All pull requests must adhere to the [PSR-2](#) standard.

7.3 Testing

All pull requests must be accompanied with passing tests and code coverage. The SphinxQL Query Builder uses [PHPUnit](#) for testing.

7.4 Issue Tracker

You can find our issue tracker at our [SphinxQL Query Builder](#) repository.