Database access

Class **ClickBlocks\DB\DB** is responsible for access to data stored in different DBMS. As a result, the underlying DBMS can be changed to a different one without requiring change of the code which uses **ClickBlocks\DB\DB** to access the data. Using this class you will be able to:

- connect to any relational database.
- execute any SQL statement.
- set explicit data type for query parameters.
- iterate query results.
- cache query results.
- use transactions.
- get additional information about database connection and a database.
- log SQL queries.

Establishing Database Connection

To connect to a database you need to create an instance of **ClickBlocks\DB\DB** and invoke method **connect()**. A data source name (DSN) is needed to specify the information required to connect to the database. A username and password may also be needed to establish the connection.

```
$db = new DB();
$db->connect($dsn, $username, $password);
```

The first argument is a DSN. The format of DSN depends on the PDO database driver in use. For more information see PDO documentation.

You can also set additional properties of the database connection by passing the array of options as the last argument of **connect()**. For example, you can establish persistent connection as follows:

```
$db = new DB();
$db->connect($dsn, $username, $password, [\PDO::ATTR_PERSISTENT => true]);
```

There is the possibility to specify connection parameters at once during creation an instance of the class:

```
$db = new DB($dsn, $username, $password, $options);
// Now, we can use "connect" without parameters.
$db->connect();
```

Once the object was created you can specify (it makes sense only for MySQL DBMS) the charset of a database connection.

```
$db = new DB();
// Set connection charset.
$db->charset = 'utf8';
// Create connection.
$db->connect($dsn, $username, $password);
```

This is equivalent of executing, after establishing connection, the following SQL statement:

```
SET NAMES 'utf8'
```

After the connection was established you can access to PDO object via access to property **pdo**:

```
// Create new connection.
$db->connect();
// Get an instance of PDO class.
$pdo = $db->pdo;
// Perform required PDO commands.
$pdo->exec('...');
...
```

If you try to get PDO object without established database connection it will be automatically established (via invoking of method **connect()**).

To check if the connection is established you can do as follows:

```
$db = new DB($dsn, $username, $password);
// Check connection.
if ($db->isConnected())
{
  // the connection is not set.
  ...
}
// Connect to a database.
$db->connect();
// Check connection again.
if ($db->isConnected())
{
  // now, the connection is set.
  ...
}
```

To close connection use the relevant method:

```
// Establish connection.
$db->connect();
// Perform some actions.
...
// Close connection.
$db->disconnect();
```

Executing SQL statements

ClickBlocks\DB\DB has method **execute()** allowing to execute any SQL statement. In the simplest case all you need is to pass to this method an SQL string to execute:

```
// Create the class object.
$db = new DB($dsn, $username, $password);
// Execute a query. Connection will be automatically established.
$db->execute('some sql query');
```

To protect your SQL statements from SQL-injection attacks you can use the parameterized queries as follows:

```
// Create the class object.
$db = new DB($dsn, $username, $password);
// Execute a parameterized query with named parameters.
$sql = 'UPDATE Users SET name = :name WHERE userID = :userID';
$db->execute($sql, [':userID' => 10, ':name' => 'John']);
// Or the same query with question mark parameters.
$sql = 'UPDATE Users SET name = ? WHERE userID = ?';
$db->execute($sql, [10, 'John']);
```

The third parameter of **execute()** method is responsible for the type of returning data. Each type is determined by the relevant class constant. There are only 6 constants:

- **DB::EXEC** - the method returns the number of rows affected by the query.
- **DB::CELL** - the method returns a scalar value corresponding the value of the required column of the first row from the retrieved rowset.
- **DB::COLUMN** - the method returns a numeric array of all values of the required column from the retrieved rowset.
- **DB::ROW** - the method returns an array of column values of the first row from the retrieved rowset.
- **DB::ROWS** - the method returns an array of all rows from the retrieved rowset.

By default the third parameter is **DB::EXEC**.

The fourth parameter of the method **execute()** is responsible for format of returning data. The value of this parameter should be one of the PDO constants like PDO::FETCH_*constant name*. For complete information see the relevant documentation.

The last two parameters of the method match the last two parameters of the method PDO::fetchAll and have a different meaning depending on the

value of the fourth parameter.

Besides of **execute()** method **ClickBlocks\DB\DB** has auxiliary methods which are wrappers over the calling of method **execute()** with the specified parameters. Below is the list of these methods:

- **cell()** - executes the SQL query and returns the value of the given column in the first row of data.

```
$sql = 'SELECT firstName, lastName, email FROM Users WHERE age > ?';
// Get email of the first found user.
$email = $db->cell($sql, [18], 2);
```

- **column()** - executes the SQL query and returns the given column of the retrieved rowset.

```
$sql = 'SELECT firstName, lastName, email FROM Users WHERE age > ?';
// Get emails of all users.
$emails = $db->column($sql, [18], 2);
```

- **row()** - executes the SQL query and returns the first row of the retrieved rowset.

```
$sql = 'SELECT firstName, lastName, email FROM Users WHERE age > ?';
// Get the first found row.
$user = $db->row($sql, [18]);
```

- **rows()** - execute the SQL query and returns all row from the retrieved rowset.

```
$sql = 'SELECT firstName, lastName, email FROM Users WHERE age > ?';
// Get all rows.
$user = $db->rows($sql, [18]);
```

- **pairs()** - this method is similar to the method **rows()** but returns a two-column result into an array where the first column is a key and the second column is a value.
- **groups()** - this method is similar to the method **rows()** but returns all rows which grouped by values of the first column.
- **couples()** - executes the SQL query and returns all rows into an array where the first column is a key and the other columns are the values.

The example below demonstrates the difference between methods **pairs()**. **groups()** and **couples()**.

Let's say we have the table "Fruit" in the database:

ID	name	color
1	apple	red
2	apple	orange
3	apple	green
4	pear	green
5	pear	yellow
6	peach	yellow
7	peach	green
8	melon	yellow
9	plum	violet

Now, we will try to get all data from this table using the methods above.

```
$sql = 'SELECT name, color FROM Fruit';
// Get pairs.
$pairs = $db->pairs($sql);
```

```
print_r($pairs);
// Get groups.
$groups = $db->groups($sql);
print_r($groups);
// Get couples.
$couples = $db->couples($sql);
print_r($couples);
```

The above script will output something like this:

```
// Pairs
Array
(
    [apple] => green
    [pear] => yellow
    [peach] => green
    [melon] => yellow
    [plum] => violet
)
// Groups
Array
(
    [apple] => Array
        (
            [0] => Array
                (
                    [color] => red
                )
            [1] => Array
                (
                    [color] => orange
                )
            [2] => Array
                (
                    [color] => green
                )
        )
    [pear] => Array
        (
            [0] => Array
                (
                    [color] => green
                )
            [1] => Array
                (
                    [color] => yellow
                )
        )
    [peach] => Array
        (
            [0] => Array
                (
                    [color] => yellow
                )
            [1] => Array
                (
                    [color] => green
                )
        )
    [melon] => Array
        (
            [0] => Array
                (
                    [color] => yellow
                )
        )
```

```
    [plum] => Array
        (
            [0] => Array
                (
                    [color] => violet
                )
        )
)
// Couples
Array
(
    [apple] => Array
        (
            [color] => green
        )
    [pear] => Array
        (
            [color] => yellow
        )
    [peach] => Array
        (
            [color] => green
        )
    [melon] => Array
        (
            [color] => yellow
        )
    [plum] => Array
        (
            [color] => violet
        )
)
```

ClickBlocks\DB\DB has three methods for executing queries of the most common types like INSERT, UPDATE and DELETE. This methods simplifies inserting, updating and deleting rows in a table.

- **insert()** - allows to add one or several rows to a database table. Method returns the ID of the last inserted row or a sequence value. Below is a few examples of usage of this method:

```
// Insert one row.
$db->insert('Fruit', ['name' => 'watermelon', 'color' => 'green']);
// Insert three rows.
$db->insert('Fruit', ['name'  => ['cherry', 'blueberry', 'watermelon'], 'color' => ['blue', 'black', 'green']]);
```

- **update()** - updates existing rows in the database. Method returns the number of rows affected by this SQL statement. Below is a few examples of usage of this method:

```
// Update one row.
$db->update('Fruit', ['color' => 'purple'], ['ID' => 2]);
// This is equivalent of the following query (for MySQL):
// UPDATE Fruit SET color = 'purple' WHERE ID = 2
...
// Update several rows according to some criteria.
$db->update('Fruit', ['color' => 'purple'],
                  ['color' => 'red', ['or', ['=', 'name', 'apple'], ['=', 'name', 'melon']]]);
// This is equivalent of the following query (for MySQL):
// UPDATE Fruit SET color = 'purple' WHERE color = 'red' AND (name = 'apple' OR name = 'melon')
```

- **delete()** - deletes existing rows in the database. Method returns the number of rows affected by this SQL statement.

```
// Delete one particular row.
$db->delete('Fruit', ['ID' => 2]);
// This is equivalent of the following query (for MySQL):
// DELETE FROM Fruit WHERE ID = 2
...
```

```
// Delete several rows.
$db->delete('Fruit', ['or', 'color' => 'red', ['and', 'name' => 'apple', 'color' => 'green']]);
// This is equivalent of the following query (for MySQL):
// DELETE FROM Fruit WHERE color = 'red' OR name = 'apple' AND color = 'green'
```

For more information about parameters of methods **update()** and **delete()** read documentation for **SQLBuilder**.

Setting of explicit data type for query parameters

When you execute some parameterized SQL statement via method **execute()** (as well as via other auxiliary methods) each value of the query parameters is bound to the statement through method **bindValue()** of **PDO()** class. This method takes as an argument the PDO data type of parameter value. By default data type is determined by converting PHP data type to PDO data type. However you can explicitly set PDO data type for each parameter of a query. At that, PDO data type can be set by using PDO::PARAM_* (see more information here). Example below displays how you can do this:

```
// The SQL statement to execute.
$sql = 'SELECT * FROM Fruit WHERE ID > :id AND color = :color';
// Execute the given query.
$fruit = $db->rows($sql, [':id'    => [3 => \PDO::PARAM_INT],
                          ':color' => ['red' => \PDO::PARAM_STR]]);
// Or for query:
$sql = 'SELECT * FROM Fruit WHERE ID > ? AND color = ?';
// We have the following command.
$fruit = $db->rows($sql, [[3     => \PDO::PARAM_INT],
                          ['red' => \PDO::PARAM_STR]]);
```

It is also possible to specify explicit data types in methods: **insert()**, **update()** or **delete()**.

```
// Insert a row with three columns, two of which
// have values with explicit specified data type.
$db->insert('MyTable', ['c1' => ['a' => \PDO::PARAM_STR],
                        'c2' => [1 => \PDO::PARAM_INT],
                        'c3' => 'auto type']);
// Update a column of a row.
// This column has the given value with explicit specified data type.
$db->update('MyTable', ['col' => ['test' => \PDO::PARAM_STR]],
                       ['col' => [true => \PDO::PARAM_BOOL]]);
// Delete a row with the given column value which has explicit data type.
$db->delete('MyTable', ['col' => [123 => \PDO::PARAM_INT]]);
```

Data reader

ClickBlocks\DB\DB has method **query()** which is similar to method **rows()**, but returns an instance of class **ClickBlocks\DB\Reader** instead of an array. This class allows to iterate all row in the retrieved set of rows. Since **ClickBlocks\DB\Reader** implements interface **Iterator** you can directly use instances of this class in **foreach**:

```
// Get an instance of data reader.
$reader = $db->query('SELECT * FROM Fruit');
// Use cycle "foreach".
foreach ($reader as $index => $row)
{
  ...
}
// Use cycle "for".
for ($reader->rewind(); $reader->valid(); $reader->next())
{
  $index = $reader->key();
  $row = $reader->current();
  ...
}
// Use cycle "while".
$reader->rewind();
while ($reader->valid())
```

```
{
  $index = $reader->key();
  $row = $reader->current();
  ...
  $reader->next();
}
```

Caching query results

Class **ClickBlocks\DB\DB** allows to cache results of query execution. Only SQL statement of type "SELECT" can be cached. You can set caching for individual groups of queries and for all queries at once.

To set caching for all SELECT queries, you need to perform one of the following actions:

- Specify in the configuration file in the section **db** the following parameters:

```
[db]
cacheExpire = 3600   ; Cache expiration time in seconds.
cacheGroup  = "--db" ; Cache group.
```

- Specify the cache expiration time and cache group in the relevant properties of the class:

```
$db->cacheExpire = 3600;  // Cache expiration time in seconds.
$db->cacheGroup = '--db'; // Cache group.
```

> The cache expiration time equal to 0 or FALSE means no caching. But if the cache expiration time will be less than 0 then the maximum cache lifetime (vault cache expiration time) will be used for query cache.

To set caching for individual queries, you need to set the regular expression pattern (PCRE compatible) that should match all required queries. You can do this by using method **addPattern()**. Some examples:

```
$db->addPattern('/^SELECT(.+)FROM MyTable(.*)/i', 3600);
// The following queries will be cached:
$rows = $db->rows('SELECT * FROM MyTable');
$rows = $db->rows('SELECT column FROM MyTable WHERE column > 5');
$rows = $db->rows('SELECT * FROM MyTable LIMIT 5, 10');
...
```

The second parameter of the method is the cache expiration time. The queries will not be cached if this parameter is 0 or FALSE. You can use this feature when the global query caching is enabled (value of property **cacheExpire** is not 0 or FALSE).

```
// Turn on caching for all queries.
$db->cacheExpire = 900;
// Turn off caching for queries retrieving data from table Fruit.
$db->addPattern('/^SELECT(.+)FROM Fruit(.*)/i', false);
// This query is cached.
$row = $db->row('SELECT * FROM MyTable LIMIT 1');
// This query is not cached.
$row = $db->row('SELECT * FROM Fruit LIMIT 1');
```

To remove pattern you can use method **dropPattern()**:

```
// Add pattern.
$db->addPattern('/^SELECT(.+)FROM Fruit(.*)/i', false);
... // some SQL executions here.
// Remove pattern.
$db->dropPattern('/^SELECT(.+)FROM Fruit(.*)/i');
```

By default **ClickBlocks\DB\DB** use the default cache object of the framework, which is returned by method **getCache()** of the class **CB**. But you can set object of cache type you need as follows:

```
// Specify your own cache object.
$db->setCache(new Cache\APC());
```

```
// Get cache object.
$cache = $db->getCache();
```

Using transactions

Some DBMS allow you to perform transactions. You can easily do this using the relevant methods:

```
// Start a transaction.
$db->beginTransaction();
try
{
  // Some SQL executions,
  ...
  // Check whether we are in the transaction.
  if ($db->inTransaction())
  {
    // We are in the transaction.
    ...
  }
  ...
  // End the transaction.
  $db->commit();
}
catch (\Exception $e)
{
  // Cancel the transaction.
  $db->rollBack();
}
```

Getting additional information

ClickBlocks\DB\DB contains a lot of methods allowing to get different information about connection, database or DBMS server. You can find out more about them by taking a look at the class reference. But two of them is mostimportant:

- **getAffectedRows** - this method returns number of rows that were affected by the last SQL statement.
- **getLastInsertID** - returns the ID of the last inserted row or sequence value.

Logging SQL queries

There is a possibility of query logging. To turn on the query logging need to specify boolean parameter **logging** in the section **db** in the configuration file. It is also necessary to set path to the log file in parameter **log**.

```
[db]
logging = 1                ; Logging is enabled.
log = '/path/to/log/file'  ; Log file.
```

The log file has simple CSV format and can be placed anywhere where you want. If you want to turn off the logging somewhere in your code you can use property **logging**:

```
// Turn off the logging.
$db->logging = false;
...
// Now, no query logging here.
...
// Turn on the logging.
$db->logging = true;
```