mef\Db Documentation

Release 1.0.0

Matthew Leverton

August 05, 2015

Contents

mef\Db is a database driver with a simple and consistent object oriented interface that currently works with both PDO and mysqli extensions. It supports prepared (named and indexed) statements and nested transactions.

Results can be iterated over using an iterator or returned as an array. Each row can be returned as an associative array, an indexed array, an object, or anything mapped by a callback function.

CHAPTER 1

User guide

1.1 Overview

1.1.1 Requirements

- PHP 5.6.0
- mysqli or PDO extension

1.1.2 Example

```
$driver = new mef\Db\Driver\PdoDriver(new PDO('sqlite::memory:'));
$driver->execute('CREATE TABLE test (
    "id" INTEGER PRIMARY KEY,
    "key" TEXT, "value" TEXT
)');

$driver->prepare(
    'INSERT INTO test VALUES (:id, :key)',
    [':id' => 1, ':key' => 'apple']
)->execute();

foreach ($driver->query('SELECT * FROM test') as $row)
{
    echo $row['id'], ' => ', $row['key'], PHP_EOL;
}
```

More complete examples are available in the `examples` directory.

1.1.3 Installation

The recommended way to install mef\Db is with Composer.

```
php composer.phar require mefworks/db:~1.0
```

Alternatively, you can download the source from BitBucket. It is compatible with PSR-4 class loaders.

```
git clone git@bitbucket.org:leverton/mefworks-db.git
```

1.1.4 License

Licensed using the MIT license.

> Copyright (C) 2006-2014 Matthew Leverton
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.2 Quickstart

This page provides a brief introduction to mef\Db. It only covers a few basic use cases. For a more comprehensive look into the various components, read the individual sections. Refer to the source code for documentation on the API itself.

1.2.1 Connect to database

To connect to a database, you must first instantiate a driver. mef\Db comes with two drivers: PDO and mysqli. Because the driver just acts as a decorator, you must first create the underlying connection.

PDO

```
$pdo = new PDO('sqlite::memory:');
$driver = new mef\Db\Driver\PdoDriver($pdo);
echo $driver->query('SELECT "Hello, World!"')->fetchValue(), PHP_EOL;
```

MySqli

```
$mysqli = new mysqli($host, $user, $password);
$driver = new mef\Db\Driver\MySqliDriver($mysqli);
echo $driver->query('SELECT "Hello, World!"')->fetchValue(), PHP_EOL;
```

The above examples will both output the same thing:

```
Hello, World!
```

1.2.2 Run queries

There are two ways to execute a query:

1. `query(string $sql) : mef\RecordSet\RecordSetInterface` - Run a select query from a raw SQL string and return a record set.
2. `execute(string $sql) : integer` - Run a data-modifying query from a raw SQL string and return the number of affected results.

If a query fails, some exception descending from `mef\Db\Exception` will be thrown.

Note: The record set that is returned by `query()` conforms to the `mef\Db\RecordSet\RecordSetInterface` interface and contains many methods that return data in various forms. This guide will only cover a few of them.

---

query()

When selecting data from the database, you must use the `query()` method. It will return a record set object. To access the data row by row, just iterate over it:

```
foreach ($driver->query('SELECT * FROM city') as $city)
{
    echo $city['name'], ' has a population of ', $city['population'], PHP_EOL;
}
```

The `foreach` value (`$city`) is a single row represented by an associative array with the name of the column being the key. The `foreach` key (not used in this example) would be the row number, beginning with `0`.

Warning: If multiple columns have the same name, then only one of them will be accessible. The exact behavior is driver-dependent, so you should avoid this situation by using SQL aliases for the columns you want to access. You may inadvertently encounter this if you are joining multiple tables with a `SELECT *` query.

To collect all of the results into an array, use the `RecordSetInterface::fetchAll()` method.

```
$results = $driver->query('SELECT * FROM city')->fetchAll();

if (count($results) > 0)
{
    echo $results[0]['name'], ' has a population of ', $results[0]['population'], PHP_EOL;
}
```

Note: Do not use `fetchAll()` if you only need to iterate over the results one time. This will unnecessarily buffer the entire results into a PHP array.

---

execute()

When modifying data (e.g., `UPDATE`, `INSERT`, and `DELETE`), you must use the `execute()` method. It will execute the query and return the number of affected rows.

```
$affectedRows = $driver->execute('DELETE city WHERE population=666');

echo 'Number of devlish cities encountered: ', $affectedRows, PHP_EOL;
```

1.2.3 Prepare statements

Any query that uses data from untrusted sources (e.g., data entered on a web form) **must** use prepared statements to avoid SQL injection attacks.

Prepared statements use placeholders for places where user data will later be filled in. There are two ways to do this:

1. Indexed parameters. Use a `?` symbol to denote a placeholder. Reference them by their 0-based index.
2. Named parameters. Use any alphanumeric name preceeded by a colon (e.g., `:name`) to denote a placeholder.

To use prepared statements:

1. Call `prepare($sql)` to create a statement.
2. Set or bind all of the parameters for the statement.
3. Call `query()` or `execute()` on the statement.

Note: `prepare()` returns a new object that conforms to the `mef\Db\Statement\StatementInterface` interface. You must use this object (not the driver) when setting the parameters and finally running the query.

Indexed Parameters

```
$st = $driver->prepare('SELECT * FROM city WHERE population > ?');
$st->setParameter(0, 1000000);

echo 'Here are some cities with one million people: ', PHP_EOL;
foreach ($st->query() as $city)
{
    echo $city, PHP_EOL;
}
```

Named Parameters

```
$st = $driver->prepare('SELECT * FROM city WHERE population > :population');
$st->setParameter(':population', 1000000);

echo 'Here are some cities with one million people: ', PHP_EOL;
foreach ($st->query() as $city)
{
    echo $city, PHP_EOL;
}
```

Note: It is not valid to mix indexed and named parameters within the same query.

1.2.4 Use transactions

Transactions are supported by objects implementing `mef\Db\Transaction\TransactionDriver`. The most feature complete driver is the `mef\Db\TransactionDriver\NestedTransactionDriver`; in order to use it, the underlying database engine must support save points and transactions.

Both the PDO and mysqli driver require that you inject the transaction driver into it.

```
$transactionDriver = new mef\Db\TransactionDriver\NestedTransactionDriver($driver);
$driver->setTransactionDriver($transactionDriver);
```

There are three methods on the driver object that power transactions:

- `startTransaction()` - Starts a transaction
- `commit()` - Commits the current transaction
- `rollBack()` - Rolls back the current transaction

With the `NestedTransactionDriver`, inner (nested) transactions are fully supported.

```
$driver->startTransaction();
$driver->execute('DELETE FROM t1');
$driver->startTransaction();
$driver->execute('INSERT INTO t1 VALUES (1)'); // will not be committed
$driver->rollBack();
$driver->commit();
```

1.3 Drivers

A driver is the main point of interaction with the database. All drivers implement the `mef\Db\Driver\DriverInterface` interface with the following methods:

- **Queries**
  - `query(string $sql) : mef\Db\RecordSet\RecordSetInterface`
  - `execute(string $sql) : integer`
  - `prepare(string $sql, array $parameters = []) : mef\Db\Statement\StatementInterface`
- **Transactions**
  - `startTransaction()`
  - `commit()`
  - `rollBack()`
- **Miscellaneous**
  - `quoteValue(string $value) : string`

The following drivers are available:

- `mef\Db\Driver\DataProviderDriver` - for unit testing
- `mef\Db\Driver\MySqliDriver` - wraps a mysqli connection
- `mef\Db\Driver\PdoDriver` - wraps a PDO connection

Each of them extend from `mef\Db\Driver\AbstractDriver`.

In addition, the `mef\Db\Driver\AbstractDecoratorDriver` can be used to extend existing drivers with additional methods.

```
// PDO driver
$pdo = new PDO('mysql:host=localhost;dbname=mydb', $user, $password);
$driver = new mef\Db\Driver\PdoDriver($pdo);

// mysqli driver
$mysqli = new mysqli('localhost', $user, $password, 'mydb');
$driver = new mef\Db\Driver\MysqliDriver($mysqli);
```

Note: The drivers take no responsibility for configuration of connections. Things like the server time zone or the connection's character set must be set via the underlying PDO / mysqli object. It's assumed that once the driver is instantiated, the underlying object will no longer be used – but because the drivers do not alter configuration, it is generally safe to use them outside the context of the driver.

1.3.1 Queries

If your query does not contain any untrusted data, then you can safely use either `query()` or `execute()`. The former is used when you need to return data (e.g., SELECT). The latter is used for data modifying queries (e.g., INSERT, UPDATE, DELETE).

Note: Any errors during queries will throw exceptions.

If your query contains untrusted data from the user (e.g., from a web form) **always** use prepared statements via the `prepare()` method. While you can use `quoteValue()` to obtain a safe string for a single value, it is not the recommended way to build queries. It is much more error prone than prepared statements.

Warning: Always use prepared statements with placeholders for user data. **Never** inject user supplied values directly into queries like this:

```
$driver->query("SELECT * FROM t1 WHERE v='" . $_POST['v'] . "'");
```

The above code will subject you to SQL injection attacks.

query()

Use `query()` to obtain data. It returns an object that implements `mef\Db\RecordSet\RecordSetInterface`. This object may be directly iterated over; each record will be represented by an associative array.

```
foreach ($driver->query('SELECT * FROM data') as $row)
{
        echo $row['column1'], PHP_EOL;
}
```

There are many other ways to retrieve data from the recordset. Refer to the RecordSets section for more information.

execute()

Use `execute()` to modify data. It returns the number of affected rows as an integer. If the driver cannot report this information, it will return `0`. It is not intended to be used to indicate errors.

```
$affectedRows = $driver->execute('DELETE FROM data WHERE v=1');
echo 'Number of rows deleted: ', $affectedRows, PHP_EOL;
```

prepare()

Use `prepare()` to safely set up a query with untrusted data. The SQL string contains placeholders that are later filled in. These placeholders can take two forms:

- *indexed* - each placeholder is represented by `?`

- *named* - each placeholder is represented by a name preceded by a colon, e.g. `:name`

A single SQL statement must be consistent: it can either contain indexed parameters or named parameters, but not both. Indexed parameters are referenced by a zero-based index from left to right; named parameters are referenced by their name, including the leading colon.

For more details, see Prepared Statements.

1.3.2 Transactions

Transactions are used so that you can commit "all or nothing." Generally they adhere to the following pattern:

```
$driver->startTransaction();

try
{
    $driver->execute($sql1);
    $driver->execute($sql2);
    $driver->commit();
}
catch (Exception $e)
{
    $driver->rollBack();
    throw $e;
}
```

The API here is stateful: when you commit or roll back, you are doing so to the currently open transaction. (It is an error to try to commit or roll back when there is no open transaction.)

Note: Before you can use transactions with the PDO and MySqli drivers, you must first set up a transaction driver.

For more details, see Transactions.

1.3.3 Miscellaneous

`quoteValue()` can be used to build a safe SQL string.

```
$sql = "SELECT * FROM t1 WHERE x='" . $driver->quoteValue($unsafeData) . "'";
```

It is not recommended to use this unless you really need to get a raw SQL string. Using prepared statements is a much better solution when you are only interested in executing some SQL with user data.

Warning: The results of this method are not guaranteed to be safe for all connections due to different character sets. It is important that you properly set your database's character set before calling this method, and that you don't use the SQL string in the future on a different character set. Refer to the PHP documentation for details on how to do this.

1.4 Prepared Statements

The database driver supports prepared statements via the `prepare` method. This will return a `mef\Db\Statement\StatementInterface` object. A query can bind values in one of two ways (but not both at the same time):

1. 0-based indexed via `?` placeholders.

2. named via `:name` placholders

```
$st = $driver->prepare('SELECT * FROM foo WHERE id=?');
$st = $driver->prepare('SELECT * FROM foo WHERE id=:id');
```

1.4.1 Setting up parameters

By value

Multiple parameters can be passed *by value* as the second argument to `prepare()`:

```
// indexed
$st = $driver->prepare('SELECT * FROM foo WHERE id=?', [42]);

// named
$st = $driver->prepare('SELECT * FROM foo WHERE id=:id', [':id' => 42]);
```

Or they can be passed later via `setParameter`.

```
// indexed
$st->setParameter(0, 42, Statement::INTEGER);

// named
$st->setParameter(':id', 42, Statement::INTEGER);
```

The third parameter is optional. If present, it must be one of the following constants (`AUTOMATIC` is the default):

- `Statement::AUTOMATIC`
- `Statement::NULL`
- `Statement::BOOLEAN`
- `Statement::INTEGER`
- `Statement::STRING`
- `Statement::BLOB`

By reference

Parameters can also be bound by reference.

```
// indexed
$st->bindParameter(0, $val, Statement::INTEGER);

// named
$st->bindParameter(':id', $val, Statement::INTEGER);
```

This is useful when you need to execute the same statement many times with different data.

```
$st = $driver->prepare('UPDATE t1 SET value=? WHERE key=?');
$st->bindParameter(0, $value);
$st->bindParameter(1, $key);

foreach (getArray() as $key => $value)
{
    $st->execute();
}
```

Note: Binding array keys may not work as expected.

```
$st->bindParameter(0, $a['value']);
$st->bindParameter(1, $a['key']);

$a = ['key' => 1, 'value' => 'foo'];
$st->execute(); // will not work
```

This won't work as `$a` has been replaced by a completely new array that is not being referenced by the `bindParameter()` call. But if individual elements of `$a` are updated, then it would work:

```
$a['key'] = 1;
$a['value'] = 'foo';

$st->execute(); // will work
```

Updating parameters in bulk

Parameters can also be sent in bulk via `setParameters` (by value) or `bindParameters` (by reference). The third parameter is an array of types; any omitted will be `Statement::AUTOMATIC`.

```
// by value
$st->setParameters([$a, $b]);
$st->setParameters([$a, $anInteger], [1 => Statement::INTEGER]);

$st->setParameters([':a' => $a, ':b' => $b]);
$st->setParameters([':a' => $a, ':b' => $anInteger], [':b' => Statement::INTEGER]);

// by reference
$st->bindParameters([&$a, &$b]);
$st->bindParameters([&$a, &$anInteger], [1 => Statement::INTEGER]);

$st->bindParameters([':a' => &$a, ':b' => &$b]);
$st->bindParameters([':a' => &$a, ':b' => &$anInteger], [':b' => Statement::INTEGER]);
```

Note: When using `bindParameters()`, you must add a reference `&` to each value, otherwise it will not work.

1.4.2 Running a prepared statement

After setting or binding all parameters, call `execute()` or `query()` as appropriate. `execute()` will return the number of affected rows, while `query()` will return a recordset.

The same prepared statement can be ran multiple times. Only the parameters that change need to be reset.

1.5 RecordSets

Calling `query()` on a driver or prepared statement will return a recordset. The data can be fetched row-by-row via an iterator or all at once as an array. Each row of the recordset can only be traversed one time, and must be done in order.

1.5.1 Naming conventions

There are many different methods associated with the recordset, depending on how you want the data returned (associative array, indexed array, etc) and if you want an array or an iterator.

The methods are named after the following conventions:

- `fetch` returns data. When combined with `All`, `Column`, or `Keyed`, it will return the remainder of the resultset as an array. Otherwise, it is only returning something from the next row.
- `get*Iterator` returns an iterator that returns one row at a time. This is the best way to get data if you simply need to process records.
- The "default" row type is an associative array.
- The methods with an explicit `Array` return an *indexed* array.
- The "keyed" methods return some sort of associative array where the key comes from data in the column.

All of the methods are listed below, followed by an overview of the main differences between them. Then the keyed methods and callbacks are covered in more detail.

Fetching one record

Type	Row Type	Method
return by value	one value from single row	`fetchValue($i = 0) : string`
return by value	associative array	`fetchRow() : array`
return by value	indexed array	`fetchRowAsArray() : array`
modify by reference	associative array	`fetchRowInto(&$array) : boolean`
modify by reference	indexed array	`fetchRowIntoArray(&$array) : boolean`
modify by reference	object	`fetchRowIntoObject(&object) : boolean`

Fetching multiple records

Key	Row Type	Method
indexed	associative array	`fetchAll() : array`
indexed	indexed array	`fetchAllAsArray() : array`
indexed	callback	`fetchAllWithCallback($cb) : array`
indexed	scalar	`fetchColumn($i = 0) : array`
associative	associative array	`fetchKeyed($key = '') : array`
associative	indexed array	`fetchKeyedAsArray($i = 0) : array`
associative	callback	`fetchKeyedWithCallback($cb, $key = '') : array`

Iterators

Key	Row Type	Method
indexed	associative array	`getIterator() : iterator`
indexed	indexed array	`getArrayIterator() : iterator`
indexed	callback	`getCallbackIterator($cb) : iterator`
indexed	scalar	`getColumnIterator($i = 0) : iterator`
associative	associative array	`getKeyedIterator($key = '') : iterator`
associative	indexed array	`getKeyedArrayIterator($i = 0) : iterator`
associative	callback	`getKeyedCallbackIterator($cb, $key = '') : iterator`

1.5.2 Fetching data

The most direct way to fetch data is via the default iterator:

```
foreach ($driver->query($sql) as $row)
{
    echo $row['columnName'], PHP_EOL;
}
```

This is equivalent to:

```
foreach ($driver->query($sql)->getIterator() as $row) ...
```

Single value

If you only need a single value from a row, you can use `fetchValue()`.

```
$value = $driver->query('SELECT COUNT(*) FROM t1')->fetchValue();
```

It defaults to using the first column, but you can specifiy a different column by passing the index (*not* the name of column). But whenever possible, it's better to just rewrite the SQL to only return the column you are interested in.

Single row

If you only need a single row, then you can use `fetchRow()` or `fetchRowAsArray()`.

```
$row = $driver->query($sql)->fetchRow();
echo $row['columnName'], PHP_EOL;

$row = $driver->query($sql)->fetchRowAsArray();
echo $row[0], PHP_EOL;
```

Note: Of course, these fetch value and row methods can be used multiple times on the same recordset. But usually it's much more succinct to use the related iterator or fetchAll/Column methods.

Fetching by reference

It's also possible to fetch *into* an array or object. These methods return `true` (more records) or `false` (no more records).

```
$rs = $driver->query($sql);

while ($rs->fetchInto($row))
{
    echo $row['columnName'], PHP_EOL;
}
```

This is the only way to fetch into an object (other than by custom callbacks):

```
$rs = $driver->query($sql);

$object = new stdClass;

while ($rs->fetchIntoObject($object))
{
```

```
    echo $object->columnName, PHP_EOL;
}
```

In both cases, the supplied array or object will have its elements or properties overridden. Anything that does not exist in the row will not be modified.

The entire recordset

The fetchAll/Column/Keyed methods return the entire recordset (or more accurately the remainder of the recordset) as an array. This is useful when you need the data as an array, so that you can perform arbitrary lookups or iterate through it multiple times.

```
$results = $driver->query($sql)->fetchAll();
```

Note: It's possible to mix things together:

```
$q = $driver->query($sql);

$firstRow = $q->fetchRow();
$remainingRows = $q->fetchAll();
```

The data will not be duplicated, as you can only iterate over each row one time.

Iterators

The iterators allow you to iterate over the recordset with the format that best suits your needs. When iterating directly over the recordset, you are using the `getIterator()` method implicitly.

To use indexed arrays:

```
foreach ($driver->query($sql)->getArrayIterator() as $row)
{
    echo $row[0], PHP_EOL;
}
```

To iterate over the nth column:

```
foreach ($driver->query($sql)->getColumnIterator() as $value)
{
    echo $value, PHP_EOL;
}
```

1.5.3 Keyed arrays

All of the `keyed` methods allow you to specify a column to use as the key, as opposed to a zero-based array. Imagine a dataset that looks like:

n	en	sp
1	one	uno
2	two	dos

With `SELECT *` and `fetchAll`, you would get an array just like that:

```
$driver->query('SELECT * FROM t1')->fetchAll()

[
  0 => ['n' => '1', 'en' => 'one', 'sp' => 'uno'],
  1 => ['n' => '2', 'en' => 'two', 'sp' => 'dos']
]
```

But say you want the *n* to be the key of the array. In that case, you need to use one of the keyed methods.

```
$driver->query('SELECT * FROM t1')->fetchKeyed()

[
  1 => ['en' => 'one', 'sp' => 'uno'],
  2 => ['en' => 'two', 'sp' => 'dos']
]
```

The keyed value is removed from the row. If there is only one value left, then a simple key => value array is returned.

```
$driver->query('SELECT n,en FROM t1')->fetchKeyed()

[
  1 => 'one',
  2 => 'two'
]
```

If there are no more columns left, then the `true` is used for the value.

```
$driver->query('SELECT en FROM t1')->fetchKeyed()

[
  'one' => true,
  'two' => true
]
```

The keyed iterator works the same way:

```
foreach ($driver->query('SELECT n,en')->getKeyedIterator() as $n => $en)
{
    echo $n, ': ', $en, PHP_EOL;
}
```

1.5.4 Callbacks

For more flexibility in how the records are returned, use a callback. Note that there is no way to directly return an object. This introduces complexities (constructor parameters, injecting dependencies, etc) that are better handled by a callback.

Callbacks always receive the entire row as an associative array. They are free to return anything.

The following illustrates using a callback to handle custom object creation:

```
$cb = function (array $row) {
    $myObject = new MyObject;
    $myObject->setFirstName($row['first_name']);
    $myObject->setLastName($row['last_name']);
    return $myObject;
};

foreach ($driver->query($sql)->getCallbackIterator($cb) as $myObject)
```

```
{
    echo $myObject->getFirstName(), PHP_EOL;
}
```

Or to retrieve them all in an array:

```
$myObjects = $driver->query($sql)->fetchAllWithCallback($cb);
```

Note: There is no single row callback because you could just as easily do this:

```
$myObject = $cb($driver->query($sql)->fetchRow());
```

Keyed callbacks

Callbacks can also be used with keyed queries. As with regular queries, those callbacks retreive the entire row as an associative array. However, keyed callbacks return an array with the key and value.

1.6 Transactions

Database drivers include the following methods to support transactions:

- `startTransaction()`
- `commit()`
- `rollBack()`

It is permissible to start multiple transactions. The `commit()` and `rollBack()` pertain to the most recently started transaction. The `mef\Db\AbstractDriver` (which all included drivers extend) outsources the implementation details to a `TransactionDriverInterface` object. If a particular implementation cannot support transactions, then some exception will be thrown. Note that these database drivers (extending `AbstractDriver`) do not have a transaction driver associated with it by default.

NestedTransactionDriver

The recommended driver to use is `NestedTransactionDriver`. It supports true nested transactions; e.g., you can roll back an inner transaction but still commit the outer one. This requires underlying support for transactions and save points.

```
$driver = new mef\Db\Driver\PdoDriver(new PDO('sqlite::memory:'));
$driver->setTransactionDriver(new mef\Db\TransactionDriver\NestedTransactionDriver($driver));

$driver->startTransaction();

// these changes are saved

$driver->startTransaction();
// these changes are lost
$driver->rollBack();

// these changes are saved

$driver->commit();
```

EmulatedNestedTransactionDriver

If the database supports transactions, but does not support save points, then the `EmulatedNestedTransactionDriver` is the best choice. Nested transactions are supported as long as every transaction is committed.

However, if any transaction is rolled back, then the entire transaction is rolled back. This happens when the outermost (first) transaction is closed. If it is attempted to be committed (despite an inner transaction failing), then an exception is thrown and it is rolled back.

PDOEmulatedNestedTransactionDriver

The `PdoEmulatedNestedTransactionDriver` extends `EmulatedNestedTransactionDriver` to use PDO's transaction methods (instead of assuming the SQL syntax). It can only be used with `PdoDriver`.