mef\Db Documentation

Release 1.0.0

Matthew Leverton

November 13, 2014

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

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 Overview

There are four basic types of objects:

- `mef\Db\Driver\DriverInterface` - The core driver that the majority of application code uses. It has methods such as `prepare`, `query`, and `execute`.
- `mef\Db\RecordSet\RecordSetInterface` - The results of a query. It can be iterated over as an array or an iterator.
- `mef\Db\Statement\StatementInterface` - A statement that has been prepared, but not yet queried. Its `query` method returns a `RecordSetInterface`.

- `mef\Db\TransactionDriver\TransactionDriverInterface` - A driver to handle transactions. The `NestedTransactionDriver` is the preferred driver. It works with any database that supports save points.

1.1.4 License

Licensed using the MIT license.

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

Drivers must 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`.

1.3.1 Queries

If your query contains untrusted data from the user (e.g., from an 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.

```
// NEVER do this
$driver->query("SELECT * FROM t1 WHERE v='" . $_POST['v'] . "'");
```

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).

Any errors during queries will throw exceptions.

query()

Use `query()` to obtain data. It returns an object that implements `mef\Db\RecordSet\RecordSetInterface`. This object may be directly iterated over.

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

There are many other ways to retrieve its data. 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.

Here is an example of the same query built with indexed and named parameters:

```
$st1 = $driver->prepare('DELETE FROM t1 WHERE v=?');

$st2 = $driver->prepare('DELETE FROM t1 WHERE v=:v');
```

You can also specify parameters as a second argument to `prepare()`:

```
$st1 = $driver->prepare('DELETE FROM t1 WHERE v=?', [42]);

$st2 = $driver->prepare('DELETE FROM t1 WHERE v=?', [':v' => 42]);
```

The return value of `prepare()` is an object implementing `mef\Db\Statement\StatementInterface`. There are three things that can be done with a prepared statement:

1. Bind or set values for the parameters,
2. Call `query()` to return a record set, or

3. Call `execute()` to return the number of affected rows.

```
// execute a prepared statement
$st = $driver->prepare('DELETE FROM t1 WHERE v=?');
$st->setParameter(0, 42);
$affectedRows = $st->execute();

// query a prepared statement
$st = $driver->query('SELECT * FROM t1 WHERE v=?');
$st->setParameter(0, 42);
foreach ($st->query() as $t1)
{
        echo $t1['v'], PHP_EOL;
}
```

For more details, see *Prepared Statements*.

1.3.2 Transactions

1.3.3 Miscellaneous

1.4 Prepared Statements

The database driver supports prepared statements via the `prepare` method. This will return a `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');
```

Parameters can be passed as the second argument:

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

Or they can be passed later via `setParameter` (by value) or `bindParameter` (by reference):

```
// by value
$st->setParameter(0, 42, Statement::INTEGER);
$st->setParameter(':id', 42, Statement::INTEGER);

// by reference
$st->bindParameter(0, $val, Statement::INTEGER);
$st->bindParameter(':id', $val, Statement::INTEGER);

$val = 42; // future calls to query/execute will use this value
```

The third parameter must be one of the following constants (`AUTOMATIC` is the default, if omitted):

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

- `Statement::STRING`
- `Statement::BLOB`

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

```
// by value
$st->setParameters([42], [Statement::INTEGER]);
$st->setParameters([':id' => 42], [':id' => Statement::INTEGER]);

// by reference
$st->bindParameters([&$val], [Statement::INTEGER]);
$st->bindParameters([':id' => &$val], [':id' => Statement::INTEGER]);
```

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 `RecordSetInterface`.

1.5 RecordSets

There are two different ways to iterate over a recordset: an iterator or an array. In addition, there are several ways to represent each row type. Each of these permutations is given its own method.

The methods are named after the following conventions:

- `fetch` returns data as an array. e.g., `fetchAll()` returns the complete recordset as an array of associative arrays.
- `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. e.g., `fetchRow` will return an associative array (keys are the names of the columns), and `getIterator()` returns an iterator that iterates over associative arrays.
- The `Array` row type is always an _indexed_ row. e.g., `fetchRowAsArray` will return an indexed array (integer keys).
- The "keyed" methods return some sort of associative array where the key comes from data in the column. That keyed value is removed from the row. If ther are no more columns left, then the `true` is used for the value. If there is exactly one column left, then it is used as the value. If there are two or more columns left, then the value becomes an associative array. e.g., "SELECT n" would result in `[1 => true, 2 => true]`. "SELECT n,en" would be `[1 => 'one', 2 => 'two']`. "SELECT n,en,sp" would be `[1 => ['en' => 'one', 'sp' => 'uno'], 2 => ['en' => 'two', 'sp' => 'dos']]`

The following table lists the names of the methods:

Traverse Type	Key	Row Type	Method
array	indexed	associative	`fetchAll()`
array	indexed	index	`fetchAllAsArray()`
array	indexed	callback	`fetchAllWithCallback($cb)`
array	indexed	scalar	`fetchColumn($i = 0)`
array	associative	associative	`fetchKeyed($key = '')`
array	associative	index	`fetchKeyedAsArray($key = '')`
array	associative	callback	`fetchKeyedWithCallback($cb, $key = '')`
iterator	indexed	associative	`getIterator()`
iterator	indexed	index	`getArrayIterator()`
iterator	indexed	callback	`getCallbackIterator($cb)`
iterator	indexed	scalar	`getColumnIterator($i = 0)`
iterator	associative	associative	`getKeyedIterator($key = '')`
iterator	associative	index	`getKeyedArrayIterator($key = '')`
iterator	associative	callback	`getKeyedCallbackIterator($cb, $key = '')`

If using an iterator, the following methods can be used to access the next row:

Type	Row Type	Method
Return by value	One value from single row	`fetchValue($i = 0)`
Return by value	Associative array	`fetchRow()`
Return by value	Indexed array	`fetchRowAsArray()`
Modify by reference	Associative array	`fetchRowInto(&$array)`
Modify by reference	Indexed array	`fetchRowIntoArray(&$array)`
Modify by reference	Object	`fetchRowIntoObject(&object)`

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. (Fetching into an object works by assuming that the object has public properties with the same name as the columns.)

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

```
$cb = function (array $row) {
    return MyObject::fromArray($row);
};

foreach ($st->getCallbackIterator($cb) as $myObject)
{
    echo $myObject->someProperty, PHP_EOL;
}
```

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`.