Phormium Documentation

Release 0.8.0

Ivan Habunek

December 25, 2016

Contents

1 Contents **3**
- 1.1 Installation . . . 3
- 1.2 Setting up . . . 4
- 1.3 Configuration . . . 5
- 1.4 Usage . . . 7
- 1.5 Transactions . . . 18
- 1.6 Events . . . 19
- 1.7 Relations . . . 22

2 License **27**

Phormium is a minimalist ORM for PHP.

It's tested on informix, mysql, postgresql and sqlite.

Could work with other relational databases which have a PDO driver, or may require some changes.

Caution: This is a work in progress. Test before using! Report any bugs on Github.

CHAPTER 1

Contents

1.1 Installation

1.1.1 Prerequisites

Phormium requires PHP 5.6 or greater with the PDO extension loaded, as well as any PDO drivers for databases to wich you wish to connect.

1.1.2 Via Composer

The most flexible installation method is using Composer.

Create a *composer.json* file in the root of your project:

```
{
    "require": {
        "phormium/phormium": "0.*"
    }
}
```

Install composer:

```
curl -s http://getcomposer.org/installer | php
```

Run Composer to install Phormium:

```
php composer.phar install
```

To upgrade Phormium to the latest version, run:

```
php composer.phar update
```

Once installed, include *vendor/autoload.php* in your script to autoload Phormium.

```
require 'vendor/autoload.php';
```

1.1.3 From GitHub

The alternative is to checkout the code directly from GitHub:

```
git clone https://github.com/ihabunek/phormium.git
```

In your code, include and register the Phormium autoloader:

```
require 'phormium/Phormium/Autoloader.php';
\Phormium\Autoloader::register();
```

Once you have installed Phormium, the next step is to set it up.

1.2 Setting up

Unlike some ORMs, Phormium does not automatically generate the database model or the PHP classes onto which the model is mapped. This has to be done manually.

1.2.1 Configure database connections

Create a JSON configuration file which contains database definitions you wish to use. Each database must have a DSN string, and optional username and password if required.

```
{
    "databases": {
        "testdb": {
            "dsn": "mysql:host=localhost;dbname=testdb",
            "username": "myuser",
            "password": "mypass"
        }
    }
}
```

For details on database specific DSNs consult the PHP documentation.

A more detailed config file reference can be found in the configuration chapter.

1.2.2 Create a database model

You need a database table which will be mapped. For example, the following SQL will create a MySQL table called *person*:

```
CREATE TABLE person (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(100),
    birthday DATE,
    salary DECIMAL
);
```

The table does not have to have a primary key, but if it doesn't Phormium will not perform update or delete queries.

1.2.3 Create a Model class

To map the *person* table onto a PHP class, a corresponding Model class is defined. Although this class can be called anything, it's sensible to name it the same as the table being mapped.

```php
class Person extends Phormium\Model
{
    // Mapping meta-data
    protected static $_meta = array(
        'database' => 'testdb',
        'table' => 'person',
        'pk' => 'id'
    );

    // Table columns
    public $id;
    public $name;
    public $birthday;
    public $salary;
}
```

Public properties of the *Person* class match the column names of the *person* database table.

Additionaly, a protected static *$_meta* property is required which holds an array with the following values:

database Name of the database, as defined in the configuration.

table Name of the database table to which the model maps.

pk Name of the primary key column (or an array of names for composite primary keys). If not defined, will default to "id", if that column exists.

1.2.4 Try it out

Create a few test rows in the database table and run the following code to fetch them:

```php
require 'vendor/autoload.php';
require 'Person.php';

Phormium\Orm::configure('config.json');

$persons = Person::objects()->fetch();
```

Learn more about usage in the next chapter.

1.3 Configuration

Phormium uses a configuration array to configure the databases to which to connect. JSON and YAML files are also supported. To configure Phormium, pass the configuration array, or a path to the configuration file to `Phormium\Orm::configure()`.

The configuration array comprises of the following options:

databases Configuration for one or more databases to which you wish to connect, indexed by a database name which is used in the model to determine in which database the table is located.

1.3.1 Databases

Each entry in `databases` has the following configuration options:

dsn The Data Source Name, or DSN, contains the information required to connect to the database. See PDO documentation for more information.

username The username used to connect to the database.

password The username used to connect to the database.

attributes Associative array of PDO attributes with corresponding values to be set on the PDO connection after it has been created.

When using a configuration array PDO constants can be used directly (e.g. `PDO::ATTR_CASE`), whereas when using a config file, the constant can be given as a string (e.g. `"PDO::ATTR_CASE"`) instead.

For available attributes see the PDO attributes documentation.

1.3.2 Examples

PHP example

```
Phormium\Orm::configure([
    "databases" => [
        "db1" => [
            "dsn" => "mysql:host=localhost;dbname=db1",
            "username" => "myuser",
            "password" => "mypass",
            "attributes" => [
                PDO::ATTR_CASE => PDO::CASE_LOWER,
                PDO::ATTR_STRINGIFY_FETCHES => true
            ]
        ],
        "db2" => [
            "dsn" => "sqlite:/path/to/db2.sqlite"
        ]
    ]
]);
```

Note: Short array syntax *[...]* requires PHP 5.4+.

JSON example

This is the equivalent configuration in JSON.

```
{
    "databases": {
        "db1": {
            "dsn": "mysql:host=localhost;dbname=db1",
            "username": "myuser",
            "password": "mypass",
            "attributes": {
                "PDO::ATTR_CASE": "PDO::CASE_LOWER",
                "PDO::ATTR_STRINGIFY_FETCHES": true
            }
        },
        "db2": {
            "dsn": "sqlite:\/path\/to\/db2.sqlite"
```

```
        }
    }
}
```

```
Phormium\Orm::configure('/path/to/config.json');
```

YAML example

This is the equivalent configuration in YAML.

```
databases:
    db1:
        dsn: 'mysql:host=localhost;dbname=db1'
        username: myuser
        password: mypass
        attributes:
            'PDO::ATTR_CASE': 'PDO::CASE_LOWER'
            'PDO::ATTR_STRINGIFY_FETCHES': true
    db2:
        dsn: 'sqlite:/path/to/db2.sqlite'
```

```
Phormium\Orm::configure('/path/to/config.yaml');
```

1.4 Usage

Now that a database table and the corresponding PHP model are created, you can start using Phormium.

1.4.1 Bootstrap

If you installed Phormium via Composer, just include *vendor/autoload.php* in your application and Phormium will be autoloaded. Afterwards, you have to configure Phormium using your configuration file.

```
require 'vendor/autoload.php';

Phormium\Orm::configure('/path/to/config.json');
```

Alternatively, if you are not using Composer, Phormium has it's own autoloader:

```
require '/path/to/phormium/src/Phormium/Autoloader.php';

Phormium\Autoloader::register();
Phormium\Orm::configure('/path/to/config.json');
```

1.4.2 Querying individual records

Fetch a single record by primary key; throws exception if it doesn't exist.

```
Person::get(13);
```

Fetch a single record by primary key; return null if it doesn't exist.

```
Person::find(13);
```

Check if a record exists with the given primary key.

```
Person::exists(13);
```

Also works for composite primary keys:

```
Trade::get('2013-01-01', 123);
Trade::find('2013-01-01', 123);
Trade::exists('2013-01-01', 123);
```

Primary key can be given as an array:

```
$tradeID = array('2013-01-01', 123);
Trade::get($tradeID);
Trade::find($tradeID);
Trade::exists($tradeID);
```

1.4.3 Querying multiple records

To fetch all data from a table, run:

```
Person::all();
```

This is shorthand for calling:

```
Person::objects()->fetch();
```

The *objects()* method will return a *QuerySet* object which is used for querying data, and *fetch()* will form and execute the corresponding SQL query and return the results as an array of *Person* objects.

1.4.4 Filtering data

In order to retrieve only selected rows, *QuerySets* can be filtered. Filters are used to consturct a WHERE clause in the resulting SQL query.

Column filters

```
Person::objects()
    ->filter('birthday', '<' '2000-01-01')
    ->fetch();
```

This kind of filter is called a **column filter** since it acts on a single column of the SQL statement, and it will result in the following query:

```
SELECT ... FROM person WHERE birthday < ?;
```

Note: Since Phormium uses prepared statements, the values for each filter are given as *?* and are passed in separately when executing the query. This prevents any possibility of SQL injection.

Available column filters:

```
Person::objects()
    ->filter($column, '=',  $value)
    ->filter($column, '!=', $value)
    ->filter($column, '>',  $value)
    ->filter($column, '>=', $value)
    ->filter($column, '<',  $value)
    ->filter($column, '<=', $value)
    ->filter($column, 'IN', $array)
    ->filter($column, 'NOT IN', $array)
    ->filter($column, 'LIKE', $value)
    ->filter($column, 'ILIKE', $value)  // case insensitive like
    ->filter($column, 'NOT LIKE', $value)
    ->filter($column, 'BETWEEN', array($low, $high))
    ->filter($column, 'IS NULL')
    ->filter($column, 'NOT NULL')
```

You can also create a column filter using the *Filter::col()* factory method and pass the resulting ColumnFilter object to *QuerySet::filter()* as a single argument.

```
use Phormium\Filter\Filter;

$filter = Filter::col('birthday', '<' '2000-01-01');

Person::objects()
    ->filter($filter)
    ->fetch();
```

Filters can be chained; chanining multiple filters will AND them

```
Person::objects()
    ->filter('birthday', '<', '2000-01-01')
    ->filter('income', '>', 10000)
    ->fetch();
```

This will create:

```
SELECT ... FROM person WHERE birthday < ? AND income > ?;
```

Raw filters

New in version 0.6.

Sometimes column filters can be limiting, since they only allow operations on a single column. **Raw filters** allow usage of custom SQL code in your WHERE clause. They will pass any given SQL condition into the WHERE clause.

Raw filters can be created by passing a single string into *QuerySet::filter()*.

```
Table::objects()
    ->filter("col1 > col2")
    ->fetch();
```

This will produce:

```
SELECT ... FROM table WHERE col1 > col2;
```

Raw filters also work with arguments, for prepared queries:

```
PriceList::objects()
    ->filter('unit_price * quantity < ?', [100])
    ->fetch();
```

Which produces:

```
SELECT ... FROM price_list WHERE unit_price * quantity < ?;
```

Warning: Any string passed in as a raw filter is inserted into the resulting SQL query without any validation. This makes it easy to:

- break a query by passing in invalid SQL
- create queries which are platform dependent (e.g. by using database-specific functions)
- pass in unvalidated values (use arguments instead)

Be careful.

Alternative methods of creating raw filters:

```
use Phormium\Filter\RawFilter;

// Either by instantiating the RawFilter class directly
$filter = new RawFilter("col1 > col2");
$filter = new RawFilter("col1 > ?", [100]);

// Or using the raw() factory method
$filter = Filter::raw("col1 > col2");
$filter = Filter::raw("col1 > ?", [100]);

// And passing it into filter()
Table::objects()
    ->filter($filter)
    ->fetch();
```

Some use cases for raw filters:

```
// Conditions which don't involve any columns
Filter::raw("current_time < ?", ['16:00:00']);

// Mathematical expressions
Filter::raw("col1 * col2 / col3 < col4");

// Using SQL functions
Filter::raw("round(col1) < 0");
```

Composite filters

In order to create complex where clauses, Phormium provides composite filters. A *composite filter* is a collection of *column filters* joined by either **AND** or **OR** operator.

To make creating complex filters easier, two factory methods exist: *Filter::_and()* and *Filter::_or()*. These are prefixed by _ because *and* and *or* are PHP keywords and cannot be used as method names.

For example to find people younger than 10 and older than 20:

```
use Phormium\Filter\Filter;

$filter = Filter::_or(
    Filter::col('age', '<', 10),
```

```
    Filter::col('age', '>', 20),
);

Person::objects()
    ->filter($filter)
    ->fetch();
```

This will create:

```
SELECT ... FROM person WHERE age < ? OR age > ?;
```

To make composite filters less verbose, you can use the shorthand way and pass arrays to *Filter::_or()* and *Filter::_and()*.

```
use Phormium\Filter\Filter;

$filter = Filter::_or(
    array('age', '<', 10),
    array('age', '>', 20),
);
```

Additionally, you can use a class alias for *Phormium\Filter\Filter* to further shorten the syntax.

```
use Phormium\Filter\Filter as f;

$filter = f::_or(
    f::col('age', '<', 10),
    f::col('age', '>', 20),
);
```

Composite filters can be chained and combined. For example:

```
use Phormium\Filter\Filter as f;

Person::objects()->filter(
    f::_or(
        f::_and(
            f::col('id', '>=', 10),
            f::col('id', '<=', 20)
        ),
        f::_and(
            f::col('id', '>=', 50),
            f::col('id', '<=', 60)
        ),
        f::col('id', '>=', 100),
    )
)->fetch();
```

This will translate to:

```
SELECT
    ...
FROM
    person
WHERE ((
    (id >= ? AND id <= ?) OR
    (id >= ? AND id <= ?) OR
    id >= ?
));
```

1.4.5 Lazy execution

QuerySets are lazy - no queries will be executed on the database until one of the *fetch methods* are called.

QuerySets are immutable. Filtering and ordering of querysets produces a new instance, instead of changing the existing one.

Therefore watch out not to do this by accident:

```
$qs = Person::objects();
$qs->filter('id', '>', 10); // QUERYSET NOT CHANGED
$qs->fetch();
```

Instead do this:

```
$qs = Person::objects();
$qs = $qs->filter('id', '>', 10); // Better
$qs->fetch();
```

1.4.6 Ordering data

QuerySets can also be ordered to determine the order in which matching records will be returned.

To apply ordering:

```
Person::objects()
    ->orderBy('id', 'desc')
    ->fetch();
```

Ordering by multiple columns:

```
Person::objects()
    ->orderBy('id', 'desc')
    ->orderBy('name', 'asc')
    ->fetch();
```

1.4.7 Fetching data

There are several methods for fetching data. All these methods perform SQL queries on the database.

Table 1.1: Fetch methods

fetch()	Fetches records as objects.
single()	Fetches a single record as an object.
values()	Fetches records as associative arrays (for given columns).
valuesList()	Fetches records as number-indexed arrays (for given columns).
valuesFlat()	Fetches values from a single column.
count()	Returns the number of records matching the filter.
distinct()	Returns distinct values of given columns.

fetch()

Fetch all records matching the given filter and returns them as an array of Model objects.

```
Person::objects()
    ->filter('birthday', '<', '2000-01-01')
    ->filter('income', '>', 10000)
    ->fetch();
```

single()

Similar to *fetch()* but expects that the filter will match a single record. Returns just the single Model object, not an array.

This method will throw an exception if zero or multiple records are matched by the filter.

For example, to fetch the person with id = 13:

```
Person::objects()
    ->filter('id', '=', 13)
    ->single();
```

This can also be achieved by the *get()* shorthand method:

```
Person::get(13);
```

values()

Similar to fetch(), but returns records as associative arrays instead of objects.

Additionally, it's possible to specify which columns to fetch from the database:

```
Person::objects()->values('id', 'name');
```

This will return:

```
array(
    array('id' => '1', 'name' => 'Ivan'),
    array('id' => '1', 'name' => 'Marko'),
)
```

If no columns are specified, all columns in the model will be fetched.

valuesList()

Similar to fetch(), but returns records as number-indexed arrays instead of objects.

Additionally, it's possible to specify which columns to fetch from the database:

```
Person::objects()->valuesList('id', 'name');
```

This will return:

```
array(
    array('1', 'Ivan'),
    array('1', 'Marko'),
)
```

If no columns are specified, all columns in the model will be fetched.

valuesFlat()

Fetches values from a single column.

Similar to calling *values()* with a single column, but returns a 1D array, where *values()* would return a 2D array.

```
Person::objects()->valuesFlat('name');
```

This will return:

```
array(
    'Ivan',
    'Marko'
)
```

count()

Returns the number of records matching the given filter.

```
Person::objects()
    ->filter('income', '<', 10000)
    ->count();
```

This returns the number of Persons with income under 10k.

distinct()

Returns the distinct values in given columns matching the current filter.

```
Person::objects()
    ->filter('birthday', '>=', '2001-01-01')
    ->distinct('name');

Person::objects()
    ->filter('birthday', '>=', '2001-01-01')
    ->distinct('name', 'income');
```

The first query will return an array of distinct names for all people born in this millenium:

```
array('Ivan', 'Marko');
```

While the second returns the distinct combinations of name and income:

```
array(
    array(
        'name' => 'Ivan',
        'income' => '5000'
    ),
    array(
        'name' => 'Ivan',
        'income' => '7000'
    ),
    array(
        'name' => 'Marko',
        'income' => '3000'
    ),
)
```

Note that if a single column is requested, the method returns an array of values from the database, but when multiple columns are requested, then an array of associative arrays will be returned.

Aggregates

The following aggregate functions are implemented on the QuerySet object:

- *avg($column)*
- *min($column)*
- *max($column)*
- *sum($column)*

Aggregates are applied after filtering. For example:

```
Person::objects()
    ->filter('birthday', '<', '2000-01-01')
    ->avg('income');
```

Returns the average income of people born before year 2000.

1.4.8 Limited fetch

Limited fetch allows you to retrieve only a portion of results matched by a *QuerySet*. This will limit the data returned by *fetch()*, *values()* and *valuesList()* methods. *distinct()* is currently unaffected.

```
QuerySet::limit($limit, $offset)
```

If a *$limit* is given, that is the maximum number of records which will be returned by the fetch methods. It is possible fetch will return fewer records if the query itself yields less rows. Specifying NULL means without limit.

If *$offset* is given, that is the number of rows which will be skipped from the matched rows.

For example to return a maximum of 10 records:

```
Person::objects()
    ->limit(10)
    ->fetch();
```

It often makes sense to use *limit()* in conjunction with *orderBy()* because otherwise you will get un unpredictable set of rows, depending on how the database decides to order them.

```
Person::objects()
    ->orderBy('name')
    ->limit(10, 20)
    ->fetch();
```

This request returns a maximum of 10 rows, while skipping the first 20 records ordered by the *name* column.

1.4.9 Writing data

Creating records

To create a new record in *person*, just create a new *Person* object and *save()* it.

```
$person = new Person();
$person->name = "Frank Zappa";
$person->birthday = "1940-12-21";
$person->save();
```

If the primary key column is auto-incremented, it is not necessary to manually assign a value to it. The *save()* method will persist the object to the database and populate the primary key property of the Person object with the value assigned by the database.

It is also possible to create a model from data contained within an array (or object) by using the static *fromArray()* method.

```
// This is quivalent to the above example
$personData = array(
    "name" => "Frank Zappa",
    "birthday" => "1940-12-21"
);
Person::fromArray($personData)->save();
```

Updating records

To change an single existing record, fetch it from the database, make the required changes and call *save()*.

```
$person = Person::get(37);
$person->birthday = "1940-12-21";
$person->salary = 10000;
$person->save();
```

If you have an associative array (or object) containing the data which you want to modify in a model instance, you can use the *merge()* method.

```
// This is quivalent to the above example
$updates = array(
    "birthday" => "1940-12-21"
    "salary" => 10000
);

$person = Person::get(37);
$person->merge($updates);
$person->save();
```

To change multiple records at once, use the *QuerySet::update()* function. This function performs an update query on all records currently selected by the *QuerySet*.

```
$person = Person::objects()
    ->filter('name', 'like', 'X%')
    ->update([
        'name' => 'Xavier'
    ]);
```

This will update all Persons whose name starts with a X and set their name to 'Xavier'.

Deleting records

Similar for deleting records. To delete a single person:

```
Person::get(37)->delete();
```

To delete multiple records at once, use the *QuerySet::delete()* function. This will delete all records currently selected by the *QuerySet*.

```
$person = Person::objects()
    ->filter('salary', '>', 100000)
    ->delete();
```

This will delete all Persons whose salary is greater than 100k.

1.4.10 Custom queries

Every ORM has it's limits, and that goes double for Phormium. Sometime it's necessary to write the SQL by hand. This is done by fetching the desired *Connection* object and using provided methods.

execute()

```
Connection::execute($query)
```

Executes the given SQL without preparing it. Does not fetch. Useful for INSERT, UPDATE or DELETE queries which do not return data.

```
// Lowercase all names in the person table
$query = "UPDATE person SET name = LOWER(name);
$conn = Orm::database()->getConnection('myconn');
$numRows = $conn->execute($query);
```

Where *myconn* is a connection defined in the config file.

query()

```
Connection::query($query[, $fetchStyle[, $class]])
```

Executes the given SQL without preparing it. Fetches all rows returned by the query. Useful for SELECT queries without arguments.

- *$fetchStyle* can be set to one of PDO::FETCH_* constants, and it determines how data is returned to the user. This argument is optional and defaults to *PDO::FETCH_ASSOC*.
- *$class* is used in conjunction with PDO::FETCH_CLASS fetch style. Optional. If set, the records will be returned as instances of this class.

For more info, see PDOStatement documentation.

```
$query = "SELET * FROM x JOIN y ON x.pk = y.fk";
$conn = Orm::database()->getConnection('myconn');
$data = $conn->query($query);
```

preparedQuery()

```
Connection::preparedQuery($query[, $arguments[, $fetchType[, $class]]])
```

Prepares the given SQL query, and executes it using the provided arguments. Fetches and returns all data returned by the query. Useful for queries which have arguments.

- *$arguments* is an array of values with as many elements as there are bound parameters in the SQL statement being executed. Can be ommitted if no arguments are required.
- *$fetchStyle* and *$class* are the same as for *query()*.

The arguments can either be unnamed:

```
$query = "SELET * FROM x JOIN y ON x.pk = y.fk WHERE col1 > ? AND col2 < ?";
$arguments = array(10, 20);
$conn = Orm::database()->getConnection('myconn');
$data = $conn->preparedQuery($query, $arguments);
```

Or they can be named:

```
$query = "SELET * FROM x JOIN y ON x.pk = y.fk WHERE col1 > :val1 AND col2 < :val2";
$arguments = array(
    "val1" => 10,
    "val2" => 20
);
$conn = Orm::database()->getConnection('myconn');
$data = $conn->preparedQuery($query, $arguments);
```

Direct PDO access

If all else fails, you can fetch the underlying PDO connection object and work with it as you like.

```
$pdo = Orm::database()->getConnection('myconn')->getPDO();
$stmt = $pdo->prepare($query);
$stmt->execute($args);
$data = $stmt->fetchAll();
```

1.5 Transactions

Phormium has two ways of using transactions.

The transaction is global, meaning it will be started an all required database connections without the need to know which model is mapped to which database.

1.5.1 Callback transactions

By passing a callable to *Orm::transaction()*, the code within the callable will be executed within a transaction. If an exception is thrown within the callback, the transaction will be rolled back. Otherwise it will be commited once the callback is executed.

For example, if you wanted to increase the salary for several Persons, you might code it this way:

```
$ids = array(10, 20, 30);
$increment = 100;

Orm::transaction(function() use ($ids, $increment) {
    foreach ($ids as $id) {
        $p = Person::get($id);
        $p->income += $increment;
        $p->save();
    }
});
```

If any of the person IDs from *$ids* does not exist, *Person::get($id)* will raise an exception which will roll back any earlier changes done within the callback.

1.5.2 Manual transactions

It is also possible to control the transaction manaully, however this produces somewhat less readable code.

Equivalent to the callback example would look like:

```
$ids = array(10, 20, 30);
$increment = 100;

Orm::begin();

try {
    foreach ($ids as $id) {
        $p = Person::get($id);
        $p->income += $increment;
        $p->save();
    }
} catch (\Exception $ex) {
    Orm::rollback();
    throw new \Exception("Transaction failed. Rolled back.", 0, $ex);
}

Orm::commit();
```

1.6 Events

Phormium uses Événement as an event emitter. You can access it emitter by calling `Orm::emitter()`.

To subscribe to an event, call `on()` method on the event emitter with the event name as the first parameter, and the callback function as the second. The parameters of the callback function depend on the event and are documented below.

The `Event` class provides a catalogue of all available events.

```
Orm::emitter()->on('query.executing' function($query, $arguments, $connection) {
    // Do something
})
```

1.6.1 Query events

The following events are emitted when running a database query.

Event	Callback arguments	Description
Event::QUERY_STARTED	$query, $arguments, $connection	Before contacting the database.
Event::QUERY_PREPARING	$query, $arguments, $connection	Before preparing the query.
Event::QUERY_PREPARED	$query, $arguments, $connection	After preparing the query.
Event::QUERY_EXECUTING	$query, $arguments, $connection	Before executing the query.
Event::QUERY_EXECUTED	$query, $arguments, $connection	After executing the query.
Event::QUERY_FETCHING	$query, $arguments, $connection	Before fetching resulting data.
Event::QUERY_FETCHED	$query, $arguments, $connection, $data	After fetching resulting data.
Event::QUERY_COMPLETED	$query, $arguments, $connection, $data	On successful completion.
Event::QUERY_ERROR	$query, $arguments, $connection, $exception	On error.

Note that not all events are triggered for each query. Only prepared queries will trigger *preparing* and *prepared* events. Only queries which return data will trigger *fetching* and *fetched* events.

Event callback functions use the following arguments:

Name	Type	Description
$query	string	Query SQL code
$arguments	array	Query arguments
$connection	Connection	Connection on which the query is run
$data	array	The data fetched from the database.
$exception	Exception	Exception thrown on query failure

1.6.2 Transaction events

The following events are triggered when starting or ending a database transaction.

Event name	Description
Event::TRANSACTION_BEGIN	When starting a transaction.
Event::TRANSACTION_COMMIT	When committing a transaction.
Event::TRANSACTION_ROLLBACK	When rolling back a transaction.

Callbacks for these events have a single argument: the `Phormium\Database\Connection` on which the action is executed.

1.6.3 Examples

Logging

A simple logging example using Apache log4php.

```
use Logger;
use Phormium\Database\Connection;
use Phormium\Event;
use Phormium\Orm;

$log = Logger::getLogger('query');

Orm::emitter()->on(Event::QUERY_STARTED, function($query, $arguments) use ($log) {
    $log->info("Running query: $query");
});

Orm::emitter()->on(Event::QUERY_ERROR, function ($query, $arguments, Connection $connection, $ex) use
```

```
    $log->error("Query failed: $ex");
});
```

Collecting query statistics

Timing query execution for locating slow queries.

```
use Phormium\Event;
use Phormium\Orm;

class Stats
{
    private $active;

    private $stats = array();

    /** Hooks onto relevant events. */
    public function register()
    {
        Orm::emitter()->on(Event::QUERY_STARTED, array($this, 'started'));
        Orm::emitter()->on(Event::QUERY_COMPLETED, array($this, 'completed'));
    }

    /** Called when a query has started. */
    public function started($query, $arguments)
    {
        $this->active = array(
            'query' => $query,
            'arguments' => $arguments,
            'start' => microtime(true)
        );
    }

    /** Called when a query has completed. */
    public function completed($query)
    {
        $active = $this->active;

        $active['end'] = microtime(true);
        $active['duration'] = $active['end'] - $active['start'];

        $this->stats[] = $active;
        $this->active = null;
    }

    /** Returns the collected statistics. */
    public function getStats()
    {
        return $this->stats;
    }
}
```

And to start collecting stats:

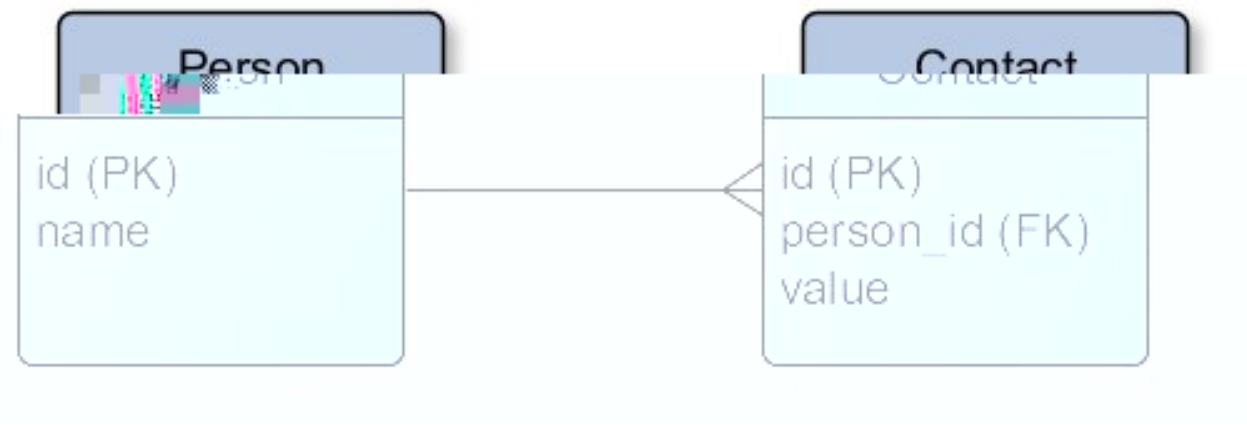
```
$stats = new Stats();
$stats->register();
```

Note that this example misses failed queries, which will never emit *query.completed*, but *query.error* instead.

1.7 Relations

Phormium allows you to define relations between models for tables which are linked via a foreign key.

Consider a Person and Contact tables like these:



The Contact table has a foreign key which references the Person table via the `person_id` field. This makes Person the parent table, and Contact the child table. Each Person record can have zero or more Contact records.

To keep things simple, relations are not defined in the model meta-data, but by invoking the following methods on the Model:

- `hasChildren()` method can be invoked on the parent model (Person), and will return a QuerySet for the child model (Contact) which is filtered to include all the child records linked to the parent model on which the method is executed. This QuerySet can contain zero or more records.
- `hasParent()` method can be invoked on the child model (Contact), and will return a QuerySet for the parent model (Person) which is filtered to include it's parent Person record.

1.7.1 Example

Models for these tables might look like this:

```
class Person extends Phormium\Model
{
    protected static $_meta = array(
        'database' => 'exampledb',
        'table' => 'person',
        'pk' => 'id'
    );

    public $id;

    public $name;

    public function contacts()
    {
        return $this->hasChildren("Contact");
    }
}
```

```
class Contact extends Phormium\Model
{
    protected static $_meta = array(
        'database' => 'exampledb',
        'table' => 'contact',
        'pk' => 'id'
    );

    public $id;

    public $person_id;

    public $value;

    public function person()
    {
        return $this->hasParent("Person");
    }
}
```

Note that these functions return a filtered QuerySet, so you need to call one of the fetching methods to fetch the data.

```
// Fetching person's contacts
$person = Person::get(1);
$contacts = $person->contacts()->fetch();

// Fetching contact's person
$contact = Contact::get(5);
$person = $contact->person()->single();
```

Returning a QuerySet allows you to further filter the result. For example, to return person's contact whose value is not null:

```
$person = Person::get(1);

$contacts = $person->contacts()
    ->filter('value', 'NOT NULL')
    ->fetch();
```

1.7.2 Overriding defaults

Phormium does it's best to guess the names of the foreign key column(s) in both tables. The guesswork, however depends on:

- Naming classes in CamelCase (e.g. `FooBar`)
- Naming tables in lowercase using underscores (e.g. `foo_bar`)
- Naming foreign keys which reference the `foo_bar` table `foo_bar_$id`, where `$id` is the name of the primary key column in `some_table`.

The following code:

```
$this->hasChildren("Contact");
```

is shorthand for:

```
$this->hasChildren("Contact", "person_id", "id");
```

where `person_id` is the name of the foreign key column in the child table (Contact), and `id` is the name of the referenced primary key column in the parent table (Person).

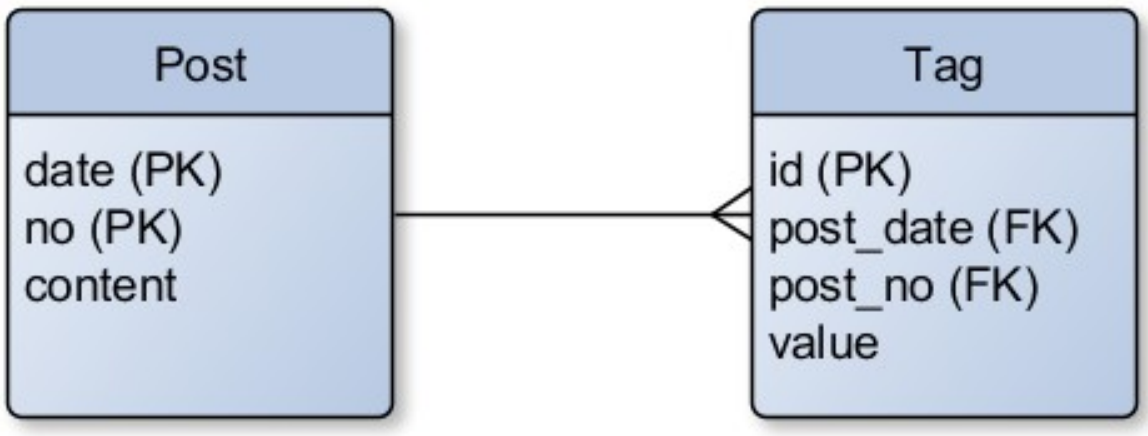
If your keys are named differently, you can override these settings. For example:

```
$this->hasChildren("Contact", "owner_id");
```

1.7.3 Composite keys

Relations also work for tables with composite primary/foreign keys.

For example, consider these tables:



Models for these tables can be implemented as:

```
class Post extends Phormium\Model
{
    protected static $_meta = array(
        'database' => 'exampledb',
        'table' => 'post',
        'pk' => ['date', 'no']
    );

    public $date;

    public $no;

    public $content;

    public function tags()
    {
        return $this->hasChildren("Tag");
    }
}
```

```
class Tag extends Phormium\Model
{
    protected static $_meta = array(
        'database' => 'exampledb',
        'table' => 'tag',
```

```
        'pk' => 'id'
    );

    public $id;

    public $post_date;

    public $post_no;

    public $value;

    public function post()
    {
        return $this->hasParent("Post");
    }
}
```

CHAPTER 2

License