DSQL Documentation

Release

Agile Toolkit

Apr 13, 2017

Contents

Contents:

CHAPTER 1

Overview

DSQL is a dynamic SQL query builder. You can write multi-vendor queries in PHP profiting from better security, clean syntax and most importantly – sub-query support. With DSQL you stay in control of when queries are executed and what data is transmitted. DSQL is easily composable – build one query and use it as a part of other query.

Goals of DSQL

- simple and consise syntax
- consistently scalable (e.g. 5 levels of sub-queries, 10 with joins and 15 parameters? no problem)
- "One Query" paradigm
- support for PDO vendors as well as NoSQL databases (with query language smilar to SQL)
- small code footprint (over 50% less than competing frameworks)
- free, licensed under MIT
- no dependencies
- **follows design paradigms:**
  - "PHP the Agile way"
  - "Functional ORM"
  - "Open to extend"
  - "Vendor Transparency"

DSQL by example

The simplest way to explain DSQL is by example:

```
$query = new atk4\dsql\Query();
$query  ->table('employees')
        ->where('birth_date','1961-05-02')
        ->field('count(*)')
        ;
echo "Employees born on May 2, 1961: ".$query->getOne();
```

The above code will execute the following query:

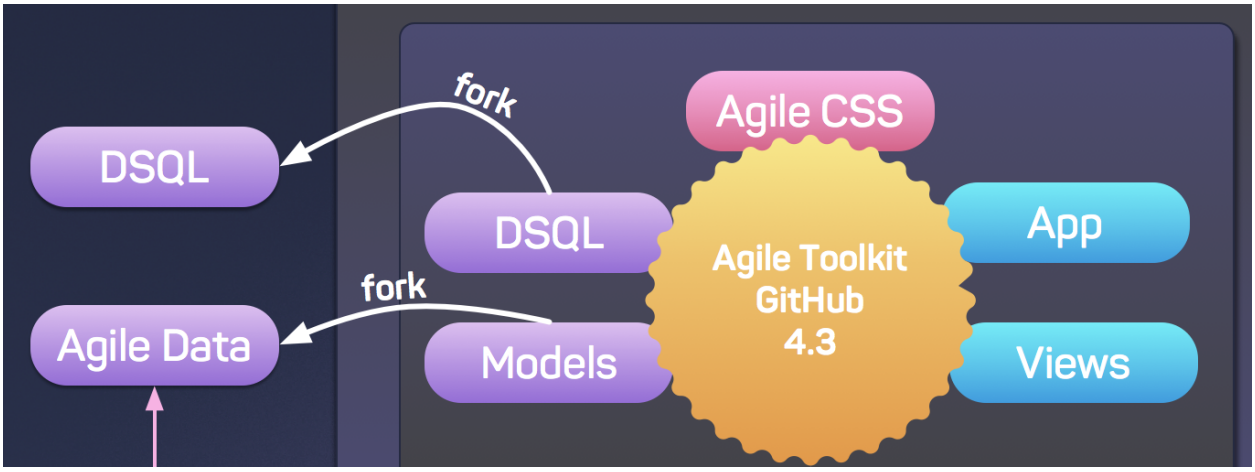
```
select count(*) from `salary` where `birth_date` = :a
    :a = "1961-05-02"
```

DSQL can also execute queries with multiple sub-queries, joins, expressions grouping, ordering, unions as well as queries on result-set.

- See *Quickstart* if you would like to start learning DSQL.
- See https://github.com/atk4/dsql-primer for various working examples of using DSQL with a real data-set.

DSQL is Part of Agile Toolkit

DSQL is a stand-alone and lightweight library with no dependencies and can be used in any PHP project, big or small.



DSQL is also a part of Agile Toolkit framework and works best with Agile Models. Your project may benefit from a higher-level data abstraction layer, so be sure to look at the rest of the suite.

Requirements

1. PHP 5.5 and above

Installation

The recommended way to install DSQL is with Composer. Composer is a dependency management tool for PHP that allows you to declare the dependencies your project has and it automatically installs them into your project.

```
# Install Composer
curl -sS https://getcomposer.org/installer | php
php composer.phar require atk4/dsql
```

You can specify DSQL as a project or module dependency in composer.json:

```
{
  "require": {
     "atk4/dsql": "*"
  }
}
```

After installing, you need to require Composer's autoloader in your PHP file:

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

You can find out more on how to install Composer, configure autoloading, and other best-practices for defining dependencies at getcomposer.org.

Getting Started

Continue reading *Quickstart* where you will learn about basics of DSQL and how to use it to it's full potential.

Contributing

Guidelines

1. DSQL utilizes PSR-1, PSR-2, PSR-4, and PSR-7.
2. DSQL is meant to be lean and fast with very few dependencies. This means that not every feature request will be accepted.
3. All pull requests must include unit tests to ensure the change works as expected and to prevent regressions.
4. All pull requests must include relevant documentation or amend the existing documentaion if necessary.

Review and Approval

1. All code must be submitted through pull requests on Github
2. Any of the project managers may Merge your pull request, but it must not be the same person who initiated the pull request.

Running the tests

In order to contribute, you'll need to checkout the source from GitHub and install DSQL dependencies using Composer:

```
git clone https://github.com/atk4/dsql.git
cd dsql && curl -s http://getcomposer.org/installer | php && ./composer.phar install -
↪-dev
```

DSQL is unit tested with PHPUnit. Run the tests using the Makefile:

```
make tests
```

There are also vendor-specific test-scripts which will require you to set database. To run them:

```
# All unit tests including SQLite database engine tests
phpunit --config phpunit.xml

# MySQL database engine tests
phpunit --config phpunit-mysql.xml
```

Look inside these the .xml files for further information and connection details.

Reporting a security vulnerability

We want to ensure that DSQL is a secure library for everyone. If you've discovered a security vulnerability in DSQL, we appreciate your help in disclosing it to us in a responsible manner.

Publicly disclosing a vulnerability can put the entire community at risk. If you've discovered a security concern, please email us at security@agiletoolkit.org. We'll work with you to make sure that we understand the scope of the issue, and that we fully address your concern. We consider correspondence sent to security@agiletoolkit.org our highest priority, and work to address any issues that arise as quickly as possible.

After a security vulnerability has been corrected, a security hotfix release will be deployed as soon as possible.

CHAPTER 2

Quickstart

When working with DSQL you need to understand the following basic concepts:

Basic Concepts

Expression (see expr) *Expression* object, represents a part of a SQL query. It can be used to express advanced logic in some part of a query, which *Query* itself might not support or can express a full statement Never try to look for "raw" queries, instead build expressions and think about escaping.

Query (see query) Object of a *Query* class can be used for building and executing valid SQL statements such as SELECT, INSERT, UPDATE, etc. After creating *Query* object you can call various methods to add "table", "where", "from" parts of your query.

Connection Represents a connection to the database. If you already have a PDO object you can feed it into *Expression* or *Query*, but for your comfort there is a *Connection* class with very little overhead.

Getting Started

We will start by looking at the *Query* building, because you do not need a database to create a query:

```
use atk4\dsql\Query;

$query = new Query(['connection' => $pdo]);
```

Once you have a query object, you can add parameters by calling some of it's methods:

```
$query
    ->table('employees')
    ->where('birth_date', '1961-05-02')
    ->field('count(*)')
    ;
```

Finally you can get the data:

```
$count = $query->getOne();
```

While DSQL is simple to use for basic queries, it also gives a huge power and consistency when you are building complex queries. Unlike other query builders that sometimes rely on "hacks" (such as method whereOr()) and claim to be useful for "most" database operations, with DSQL, you can use DSQL to build ALL of your database queries.

This is hugely benificial for frameworks and large applications, where various classes need to interact and inject more clauses/fields/joins into your SQL query.

DSQL does not resolve conflicts between similarly named tables, but it gives you all the options to use aliases.

The next example might be a bit too complex for you, but still read through and try to understand what each section does to your base query:

```
// Establish a query looking for a maximum salary
$salary = new Query(['connection'=>$pdo]);

// Create few expression objects
$e_ms = $salary->expr('max(salary)');
$e_df = $salary->expr('TimeStampDiff(month, from_date, to_date)');

// Configure our basic query
$salary
    ->table('salary')
    ->field(['emp_no', 'max_salary'=>$e_ms, 'months'=>$e_df])
    ->group('emp_no')
    ->order('-max_salary')

// Define sub-query for employee "id" with certain birth-date
$employees = $salary->dsql()
    ->table('employees')
    ->where('birth_date', '1961-05-02')
    ->field('emp_no')
    ;

// Use sub-select to condition salaries
$salary->where('emp_no', $employees);

// Join with another table for more data
$salary
    ->join('employees.emp_id', 'emp_id')
    ->field('employees.first_name');


// Finally, fetch result
foreach ($salary as $row) {
    echo "Data: ".json_encode($row)."\n";
}
```

The above query resulting code will look like this:

```
SELECT
    `emp_no`,
    max(salary) `max_salary`,
    TimeStampDiff(month, from_date, to_date) `months`
FROM
    `salary`
```

```
JOIN
    `employees` on `employees`.`emp_id` = `salary`.`emp_id`
WHERE
    `salary`.`emp_no` in (select `id` from `employees` where `birth_date` = :a)
GROUP BY `emp_no`
ORDER BY max_salary desc

:a = "1961-05-02"
```

Using DSQL in higher level ORM libraries and frameworks allows them to focus on defining the database logic, while DSQL can perform the heavy-lifting of query building and execution.

Creating Objects and PDO

DSQL classes does not need database connection for most of it's work. Once you create new instance of *Expression* or *Query* you can perform operation and finally call *Expression::render()* to get the final query string:

```
use atk4\dsql\Query;

$q = (new Query())->table('user')->where('id', 1)->field('name');
$query = $q->render();
$params = $q->params;
```

When used in application you would typically generate queries with the purpose of executing them, which makes it very useful to create a *Connection* object. The usage changes slightly:

```
$c = atk4\dsql\Connection::connect($dsn, $user, $password);
$q = $c->dsql()->table('user')->where('id', 1)->field('name');

$name = $q->getOne();
```

You no longer need "use" statement and *Connection* class will automatically do some of the hard work to adopt query building for your database vendor. There are more ways to create connection, see **'Advanced Connections'_** section.

The format of the `$dsn` is the same as with PDO class. If you need to execute query that is not supported by DSQL, you should always use expressions:

```
$tables = $c -> expr('show tables like []', [$like_str])->get();
```

DSQL classes are mindful about your SQL vendor and it's quirks, so when you're building sub-queries with *Query::dsql*, you can avoid some nasty problems:

```
$sqlite_c ->dsql()->table('user')->truncate();
```

The above code will work even though SQLite does not support truncate. That's because DSQL takes care of this.

Query Building

Each Query object represents a query to the database in-the-making. Calling methods such as *Query::table* or *Query::where* affect part of the query you're making. At any time you can either execute your query or use it inside another query.

Query supports majority of SQL syntax out of the box. Some unusual statements can be easily added by customizing template for specific query and we will look into examples in *Extending Query Class*

Query Mode

When you create a new *Query* object, it is going to be a *SELECT* query by default. If you wish to execute `update` operation instead, you simply call *Query::update*, for delete - *Query::delete* (etc). For more information see *Query Modes*. You can actually perform multiple operations:

```
$q = $c->dsql()->table('employee')->where('emp_no', 1234);
$backup_data = $q->get();
$q->delete();
```

A good practice is to re-use the same query object before you branch out and perform the action:

```
$q = $c->dsql()->table('employee')->where('emp_no', 1234);

if ($confirmed) {
    $q->delete();
} else {
    echo "Are you sure you want to delete ".$q->field('count(*)')." employees?";
}
```

Fetching Result

When you are selecting data from your database, DSQL will prepare and execute statement for you. Depending on the connection, there may be some magic involved, but once the query is executed, you can start streaming your data:

```
foreach ($query->table('employee')->where('dep_no',123) as $employee) {
    echo $employee['first_name']."\n";
}
```

In most cases, when iterating you'll have PDOStatement, however this may not always be the case, so be cautious. Remember that DQSL can support vendors that PDO does not support as well or can use *Proxy Connection*. In that case you may end up with other Generator/Iterator but regardless, *$employee* will always contain associative array representing one row of data. (See also **'Manual Query Execution'_**).

CHAPTER 3

Connection

DSQL supports various database vendors natively but also supports 3rd party extensions. For current status on database support see: *Vendor support and Extensions*

class Connection

Connection class is handy to have if you plan on building and executing queries in your application. It's more appropriate to store connection in a global variable or global class:

```
$app->db = atk4\dsql\Connection::connect($dsn, $user, $pass);
```

static Connection::**connect**(*$dsn*, *$user = null*, *$password = null*, *$args =*[ ])

Determine which Connection class should be used for specified $dsn, create new object of this connection class and return.

Parameters

- **$dsn** (*string*) – DSN, see http://php.net/manual/en/ref.pdo-mysql.connection.php
- **$user** (*string*) – username
- **$password** (*string*) – password
- **$args** (*array*) – Other default properties for connection class.

Returns new Connection

This should allow you to access this class from anywhere and generate either new Query or Expression class:

```
$query = $app->db->dsql();

// or

$expr = $app->db->expr('show tables');
```

Connection::**dsql**(*$args*)

Creates new Query class and sets Query::connection.

Parameters

- **$args** (*array*) – Other default properties for connection class.

Returns new Query

`Connection::`**expr**(*$template*, *$args*)

Creates new Expression class and sets `Expression::connection`.

Parameters

- **$args** (*array*) – Other default properties for connection class.
- **$args** – Other default properties for connection class.

Returns new Expression

Here is how you can use all of this together:

```
$dsn = 'mysql:host=localhost;port=3307;dbname=testdb';

$c = atk4\dsql\Connection::connect($dsn, 'root', 'root');
$expr = $c -> expr("select now()");

echo "Time now is : ". $expr;
```

connect will determine appropriate class that can be used for this DSN string. This can be a PDO class or it may try to use a 3rd party connection class.

Connection class is also responsible for executing queries. This is only used if you connect to vendor that does not use PDO.

`Connection::`**execute**(*Expression $expr*)

Creates new Expression class and sets `Expression::connection`.

Parameters

- **$expr** (`Expression`) – Expression (or query) to execute

Returns PDOStatement, Iterable object or Generator.

class `Expression`

CHAPTER 4

Expressions

Expression class implements a flexible way for you to define any custom expression then execute it as-is or as a part of another query or expression. Expression is supported anywhere in DSQL to allow you to express SQL syntax properly.

Quick Example:

```
$query -> where('time', $query->expr(
    'between "[]" and "[]"',
    [$from_time, $to_time]
));

// Produces: .. where `time` between :a and :b
```

Another use of expression is to supply field instead of value and vice versa:

```
$query -> where($query->expr(
    '[] between time_from and time_to',
    [$time]
));

// Produces: where :a between time_from and time_to
```

Yet another curious use for the DSQL library is if you have certain object in your ORM implementing `Expressionable` interface. Then you can also use it within expressions:

```
$query -> where($query->expr(
    '[] between [] and []',
    [$time, $model->getElement('time_form'), $model->getElement('time_to')]
));

// Produces: where :a between `time_from` and `time_to`
```

Another uses for expressions could be:

- Sub-Queries
- SQL functions, e.g. IF, CASE

- nested AND / OR clauses
- vendor-specific queries - "describe table"
- non-traditional constructions , UNIONS or SELECT INTO

Properties, Arguments, Parameters

Be careful when using those similar terms as they refer to different things:

- Properties refer to object properties, e.g. *$expr->template*, see *Other Properties*
- Arguments refer to template arguments, e.g. *select * from [table]*, see *Expression Template*
- Parameters refer to the way of passing user values within a query *where id=:a* and are further explained below.

Parameters

Because some values are un-safe to use in the query and can contain dangerous values they are kept outside of the SQL query string and are using PDO's bindParam instead. DSQL can consist of multiple objects and each object may have some parameters. During *rendering* those parameters are joined together to produce one complete query.

property `Expression::$params`
: This public property will contain the actual values of all the parameters. When multiple queries are merged together, their parameters are interlinked.

Creating Expression

```
use atk4\dsql\Expression;

$expr = new Expression("NOW()");
```

You can also use *`expr()`* method to create expression, in which case you do not have to define "use" block:

```
$query -> where('time', '>', $query->expr('NOW()'));

// Produces: .. where `time` > NOW()
```

You can specify some of the expression properties through first argument of the constructor:

```
$expr = new Expression(["NOW()", 'connection' => $pdo]);
```

Scroll down for full list of properties.

Expression Template

When you create a template the first argument is the template. It will be stored in `$template` property. Template string can contain arguments in a square brackets:

- `coalesce([], [])` is same as `coalesce([0], [1])`
- `coalesce([one], [two])`

Arguments can be specified immediatelly through an array as a second argument into constructor or you can specify arguments later:

```
$expr = new Expression(
    "coalesce([name], [surname])",
    ['name' => $name, 'surname' => $surname]
);

// is the same as

$expr = new Expression("coalesce([name], [surname])");
$expr['name'] = $name;
$expr['surname'] = $surname;
```

Nested expressions

Expressions can be nested several times:

```
$age = new Expression("coalesce([age], [default_age])");
$age['age'] = new Expression("year(now()) - year(birth_date)");
$age['default_age'] = 18;

$query -> table('user') -> field($age, 'calculated_age');

// select coalesce(year(now()) - year(birth_date), :a) `calculated_age` from `user`
```

When you include one query into another query, it will automatically take care of all user-defined parameters (such as value *18* above) which will make sure that SQL injections could not be introduced at any stage.

Rendering

An expression can be rendered into a valid SQL code by calling render() method. The method will return a string, however it will use references for *parameters*.

`Expression::render()`
: Converts *`Expression`* object to a string. Parameters are replaced with :a, :b, etc. Their original values can be found in *`params`*.

Executing Expressions

If your expression is a valid SQL query, (such as `` `show databases` ``) you might want to execute it. Expression class offers you various ways to execute your expression. Before you do, however, you need to have `$connection` property set. (See *Connecting to Database* on more details). In short the following code will connect your expression with the database:

```
$expr = new Expression('connection'=>$pdo_dbh);
```

If you are looking to use connection *`Query`* class, you may want to consider using a proper vendor-specific subclass:

```
$query = new Query_MySQL('connection'=>$pdo_dbh);
```

If your expression already exist and you wish to associate it with connection you can simply change the value of `$connection` property:

```
$expr -> connection = $pdo_dbh;
```

Finally, you can pass connection class into *execute* directly.

`Expression::`**`execute`**(*$connection = null*)
Executes expression using current database connection or the one you specify as the argument:

```
$stmt = $expr -> execute($pdo_dbh);
```

returns PDOStamement if you have used PDO class or ResultSet if you have used Connection.

`Expression::`**`expr`**(*$properties*, *$arguments*)
Creates a new *Expression* object that will inherit current `$connection` property. Also if you are creating a vendor-specific expression/query support, this method must return instance of your own version of Expression class.

The main principle here is that the new object must be capable of working with database connection.

`Expression::`**`get`**()
Executes expression and return whole result-set in form of array of hashes:

```
$data = new Expression([
        'connection' => $pdo_dbh,
        'template'   => 'show databases'
    ])->get();
echo json_encode($data);
```

The output would be

```
[
    { "Database": "mydb1" },
    { "Database": "mysql" },
    { "Database": "test" }
]
```

`Expression::`**`getRow`**()
Executes expression and returns first row of data from result-set as a hash:

```
$data = new Expression([
        'connection' => $pdo_dbh,
        'template'   => 'SELECT @@global.time_zone, @@session.time_zone'
    ])->getRow()

echo json_encode($data);
```

The output would be

```
{ "@@global.time_zone": "SYSTEM", "@@session.time_zone": "SYSTEM" }
```

`Expression::`**`getOne`**()
Executes expression and return first value of first row of data from result-set:

```
$time = new Expression([
        'connection' => $pdo_dbh,
        'template'   => 'now()'
    ])->getOne();
```

Magic an Debug Methods

`Expression::__toString()`
: You may use *Expression* or *Query* as a string. It will be automatically executed when being cast by executing *getOne*. Because the __toString() is not allowed to throw exceptions we encourage you not to use this format.

`Expression::__debugInfo()`
: This method is used to prepare a sensible information about your query when you are executing `var_dump($expr)`. The output will be HTML-safe.

`Expression::debug()`
: Calling this method will set *debug* into `true` and the further execution to *render* will also attempt to echo query.

`Expression::getDebugQuery(`*$html = false*`)`
: Outputs query as a string by placing parameters into their respective places. The parameters will be escaped, but you should still avoid using generated query as it can potentially make you vulnerable to SQL injection.

  This method will use HTML formatting if argument is passed.

In order for HTML parsing to work and to make your debug queries better formatted, install *sql-formatter*:

```
composer require jdorn/sql-formatter
```

Escaping Methods

The following methods are useful if you're building your own code for rendering parts of the query. You must not call them in normal circumstances.

`Expression::_consume(`*$sql_code*`)`
: Makes *$sql_code* part of *$this* expression. Argument may be either a string (which will be escaped) or another *Expression* or *Query*. If specified *Query* is in "select" mode, then it's automatically placed inside brackets:

```
$query->_consume('first_name');  // `first_name`
$query->_consume($other_query);  // will merge parameters and return string
```

`Expression::escape(`*$sql_code*`)`
: Creates new expression where $sql_code appears escaped. Use this method as a conventional means of specifying arguments when you think they might have a nasty back-ticks or commas in the field names. I generally **discourage** you from using this method. Example use would be:

```
$query->field('foo,bar');  // escapes and adds 2 fields to the query
$query->field($query->escape('foo,bar')); // adds field `foo,bar` to the query
$query->field(['foo,bar']);  // adds single field `foo,bar`

$query->order('foo desc');  // escapes and add `foo` desc to the query
$query->field($query->escape('foo desc')); // adds field `foo desc` to the query
$query->field(['foo desc']); // adds `foo` desc anyway
```

`Expression::_escape(`*$sql_code*`)`
: Always surrounds *$sql code* with back-ticks.

`Expression::_escapeSoft(`*$sql_code*`)`
: Surrounds *$sql code* with back-ticks.

It will smartly escape table.field type of strings resulting in *table*.‘field‘.

Will do nothing if it finds “*”, “‘” or “(” character in *$sql_code*:

```
$query->_escape('first_name');  // `first_name`
$query->_escape('first.name');  // `first`.`name`
$query->_escape('(2+2)');       // (2+2)
$query->_escape('*');           // *
```

`Expression::`**`_param`**`(`*$value*`)`

Converts value into parameter and returns reference. Used only during query rendering. Consider using *`_consume()`* instead, which will also handle nested expressions properly.

Other Properties

property `Expression::$`**`template`**

Template which is used when rendering. You can set this with either *new Expression(“show tables”)* or *new Expression([”show tables”])* or *new Expression([”template” => “show tables”])*.

property `Expression::$`**`connection`**

PDO connection object or any other DB connection object.

property `Expression::$`**`paramBase`**

Normally parameters are named :a, :b, :c. You can specify a different param base such as :param_00 and it will be automatically increased into :param_01 etc.

property `Expression::$`**`debug`**

If true, then next call of *`execute`* will `echo` results of *`getDebugQuery`*.

class **`Query`**

CHAPTER 5

Queries

Query class represents your SQL query in-the-making. Once you create object of the Query class, call some of the methods listed below to modify your query. To actually execute your query and start retrieving data, see fetching-result section.

You should use *Connection* if possible to create your query objects. All examples below are using *$c->dsql()* method which generates Query linked to your established database connection.

Once you have a query object you can execute modifier methods such as `field()` or `table()` which will change the way how your query will act.

Once the query is defined, you can either use it inside another query or expression or you can execute it in exchange for result set.

Quick Example:

```
$query = $c->dsql();

$query -> field('name');
$query -> where('id', 123);

$name = $query -> getOne();
```

Method invocation principles

Methods of Query are designed to be flexible and consise. Most methods have a variable number of arguments and some arguments can be skipped:

```
$query -> where('id', 123);
$query -> where('id', '=', 123);  // the same
```

Most methods will accept `Expression` or strings. Strings are escaped or quoted (depending on type of argument). By using `Expression` you can bypass the escaping.

There are 2 types of escaping:

- *Expression::_escape()*. Used for field and table names. Surrounds name with ‘.
- *Expression::_param()*. Will convert value into parameter and replace with *:a*

In the next example $a is escaped but $b is parametrised:

```
$query -> where('a', 'b');

// where `a` = "b"
```

If you want to switch places and execute *where “b” = ‘a‘*, then you can resort to Expressions:

```
$query -> where($c->expr('{} = []', ['b', 'a']));
```

Parameters which you specify into Expression will be preserved and linked into the *$query* properly.

Query Modes

When you create new Query it always start in “select” mode. You can switch query to a different mode using *mode*. Normally you shouldn’t bother calling this method and instead use one of the following methods. They will switch the query mode for you and execute query:

`Query::select()`
Switch back to “select” mode and execute *select* statement.

See *Modifying Select Query*.

`Query::insert()`
Switch to *insert* mode and execute statement.

See *Insert and Replace query*.

`Query::update()`
Switch to *update* mode and execute statement.

See *Update Query*.

`Query::replace()`
Switch to *replace* mode and execute statement.

See *Insert and Replace query*.

`Query::delete()`
Switch to *delete* mode and execute statement.

See *Delete Query*.

`Query::truncate()`
Switch to *truncate* mode and execute statement.

If you don’t switch the mode, your Query remains in select mode and you can fetch results from it anytime.

The pattern of defining arguments for your Query and then executing allow you to re-use your query efficiently:

```
$data = ['name'=>'John', 'surname'=>'Smith']

$query = $c->dsql();
$query
    -> where('id', 123)
    -> field('id')
    -> table('user')
```

```
    -> set($data)
    ;

$row = $query->getRow();

if ($row) {
    $query
        ->set('revision', $query->expr('revision + 1'))
        ->update()
        ;
} else {
    $query
        ->set('revision', 1)
        ->insert();
}
```

The example above will perform a select query first:

- *select id from user where id=123*

If a single row can be retrieved, then the update will be performed:

- *update user set name="John", surname="Smith", revision=revision+1 where id=123*

Otherwise an insert operation will be performed:

- *insert into user (name,surname,revision) values ("John", "Smith", 1)*

Chaining

Majority of methods return *$this* when called, which makes it pretty convenient for you to chain calls by using *->fx()* multiple times as illustrated in last example.

You can also combine creation of the object with method chaining:

```
$age = $c->dsql()->table('user')->where('id', 123)->field('age')->getOne();
```

Using query as expression

You can use query as expression where applicable. The query will get a special treatment where it will be surrounded in brackets. Here are few examples:

```
$q = $c->dsql()
    ->table('employee');

$q2 = $c->dsql()
    ->field('name')
    ->table($q);

$q->get();
```

This query will perform *select name from (select * from employee)*:

```
$q1 = $c->dsql()
    ->table('sales')
    ->field('date')
    ->field('amount', null, 'debit');

$q2 = $c->dsql()
    ->table('purchases')
    ->field('date')
    ->field('amount', null, 'credit');

$u = $c->dsql("[] union []", [$q1, $q2]);

$q = $c->dsql()
    ->field('date,debit,credit')
    ->table($u, 'derrivedTable')
    ;

$q->get();
```

This query will perform union between 2 table selects resulting in the following query:

```
select `date`,`debit`,`credit` from (
    (select `date`,`amount` `debit` from `sales`) union
    (select `date`,`amount` `credit` from `purchases`)
) `derrivedTable`
```

Modifying Select Query

Setting Table

`Query::table`(*$table*, *$alias*)
Specify a table to be used in a query.

Parameters

- **$table** (*mixed*) – table such as "employees"
- **$alias** (*mixed*) – alias of table

Returns $this

This method can be invoked using different combinations of arguments. Follow the principle of specifying the table first, and then optionally provide an alias. You can specify multiple tables at the same time by using comma or array (although you won't be able to use the alias there). Using keys in your array will also specify the aliases.

Basic Examples:

```
$c->dsql()->table('user');
    // SELECT * from `user`

$c->dsql()->table('user','u');
    // aliases table with "u"
    // SELECT * from `user` `u`

$c->dsql()->table('user')->table('salary');
    // specify multiple tables. Don't forget to link them by using "where"
    // SELECT * from `user`, `salary`
```

```
$c->dsql()->table(['user','salary']);
    // identical to previous example
    // SELECT * from `user`, `salary`

$c->dsql()->table(['u'=>'user','s'=>'salary']);
    // specify aliases for multiple tables
    // SELECT * from `user` `u`, `salary` `s`
```

Inside your query table names and aliases will always be surrounded by backticks. If you want to use a more complex expression, use *Expression* as table:

```
$c->dsql()->table(
    $c->expr('(SELECT id FROM user UNION select id from document)'),
    'tbl'
);
// SELECT * FROM (SELECT id FROM user UNION SELECT id FROM document) `tbl`
```

Finally, you can also specify a different query instead of table, by simply passing another *Query* object:

```
$sub_q = $c->dsql();
$sub_q -> table('employee');
$sub_q -> where('name', 'John');

$q = $c->dsql();
$q -> field('surname');
$q -> table($sub_q, 'sub');

// SELECT `surname` FROM (SELECT * FROM `employee` WHERE `name` = :a) `sub`
```

Method can be executed several times on the same Query object.

Setting Fields

Query::**field**(*$fields*, *$alias* = *null*)

Adds additional field that you would like to query. If never called, will default to *defaultField*, which normally is *.

This method has several call options. $field can be array of fields and also can be an *Expression* or *Query*

Parameters

- **$fields** (*string|array|object*) – Specify list of fields to fetch
- **$alias** (*string*) – Optionally specify alias of field in resulting query

Returns $this

Basic Examples:

```
$query = new Query();
$query->table('user');

$query->field('first_name');
    // SELECT `first_name` from `user`

$query->field('first_name,last_name');
    // SELECT `first_name`,`last_name` from `user`
```

```
$query->field('employee.first_name')
    // SELECT `employee`.`first_name` from `user`

$query->field('first_name','name')
    // SELECT `first_name` `name` from `user`

$query->field(['name'=>'first_name'])
    // SELECT `first_name` `name` from `user`

$query->field(['name'=>'employee.first_name']);
    // SELECT `employee`.`first_name` `name` from `user`
```

If the first parameter of field() method contains non-alphanumeric values such as spaces or brackets, then field() will assume that you're passing an expression:

```
$query->field('now()');

$query->field('now()', 'time_now');
```

You may also pass array as first argument. In such case array keys will be used as aliases (if they are specified):

```
$query->field(['time_now'=>'now()', 'time_created']);
    // SELECT now() `time_now`, `time_created` ...

$query->field($query->dsql()->table('user')->field('max(age)'), 'max_age');
    // SELECT (SELECT max(age) from user) `max_age` ...
```

Method can be executed several times on the same Query object.

Setting where and having clauses

`Query::`**`where`**(*$field*, *$operation*, *$value*)
Adds WHERE condition to your query.

Parameters

- **`$field`** (*mixed*) – field such as "name"
- **`$operation`** (*mixed*) – comparison operation such as ">" (optional)
- **`$value`** (*mixed*) – value or expression

Returns $this

`Query::`**`having`**(*$field*, *$operation*, *$value*)
Adds HAVING condition to your qurey.

Parameters

- **`$field`** (*mixed*) – field such as "name"
- **`$operation`** (*mixed*) – comparison operation such as ">" (optional)
- **`$value`** (*mixed*) – value or expression

Returns $this

Both methods use identical call interface. They support one, two or three argument calls.

Pass string (field name), *Expression* or even *Query* as first argument. If you are using string, you may end it with operation, such as "age>" or "parent_id is not" DSQL will recognize <, >, =, !=, <>, is, is not.

If you havent specified parameter as a part of $field, specify it through a second parameter - $operation. If unspecified, will default to '='.

Last argument is value. You can specify number, string, array, expression or even null (specifying null is not the same as omitting this argument). This argument will always be parameterised unless you pass expression. If you specify array, all elements will be parametrised individually.

Starting with the basic examples:

```
$q->where('id', 1);
$q->where('id', '=', 1); // same as above

$q->where('id>', 1);
$q->where('id', '>', 1); // same as above

$q->where('id', 'is', null);
$q->where('id', null);   // same as above

$q->where('now()', 1);   // will not use backticks
$q->where($c->expr('now()'),1);  // same as above

$q->where('id', [1,2]);  // renders as id in (1,2)
```

You may call where() multiple times, and conditions are always additive (uses AND). The easiest way to supply OR condition is to specify multiple conditions through array:

```
$q->where([['name', 'like', '%john%'], ['surname', 'like', '%john%']]);
    // .. WHERE `name` like '%john%' OR `surname` like '%john%'
```

You can also mix and match with expressions and strings:

```
$q->where([['name', 'like', '%john%'], 'surname is null']);
    // .. WHERE `name` like '%john%' AND `surname` is null

$q->where([['name', 'like', '%john%'], new Expression('surname is null')]);
    // .. WHERE `name` like '%john%' AND surname is null
```

There is a more flexible way to use OR arguments:

`Query::orExpr()`
: Returns new Query object with method "where()". When rendered all clauses are joined with "OR".

`Query::andExpr()`
: Returns new Query object with method "where()". When rendered all clauses are joined with "OR".

Here is a sophisticated example:

```
$q = $c->dsql();

$q->table('employee')->field('name');
$q->where('deleted', 0);
$q->where(
    $q
        ->orExpr()
        ->where('a', 1)
        ->where('b', 1)
        ->where(
```

```
            $q->andExpr()
                ->where('a', 2)
                ->where('b', 2)
        )
);
```

The above code will result in the following query:

```
select
    `name`
from
    `employee`
where
    deleted  = 0 and
    (`a` = :a or `b` = :b or (`a` = :c and `b` = :d))
```

Technically orExpr() generates a yet another object that is composed and renders its calls to where() method:

```
$q->having(
    $q
        ->orExpr()
        ->where('a', 1)
        ->where('b', 1)
);
```

```
having
    (`a` = :a or `b` = :b)
```

Grouping results by field

`Query::group`(*$field*)
Group by functionality. Simply pass either field name as string or *Expression* object.

Parameters

- **$field** (*mixed*) – field such as "name"

Returns $this

The "group by" clause in SQL query accepts one or several fields. It can also accept expressions. You can call *group()* with one or several comma-separated fields as a parameter or you can specify them in array. Additionally you can mix that with *Expression* or `Expressionable` objects.

Few examples:

```
$q->group('gender');

$q->group('gender,age');

$q->group(['gender', 'age']);

$q->group('gender')->group('age');

$q->group(new Expression('year(date)'));
```

Method can be executed several times on the same Query object.

Joining with other tables

`Query::join`(*$foreign_table*, *$master_field*, *$join_kind*)
Join results with additional table using "JOIN" statement in your query.

Parameters

- **$foreign_table** (*string|array*) – table to join (may include field and alias)
- **$master_field** (*mixed*) – main field (and table) to join on or Expression
- **$join_kind** (*string*) – 'left' (default), 'inner', 'right' etc - which join type to use

Returns $this

When joining with a different table, the results will be stacked by the SQL server so that fields from both tables are available. The first argument can specify the table to join, but may contain more information:

```
$q->join('address');            // address.id = address_id
    // JOIN `address` ON `address`.`id`=`address_id`

$q->join('address a');          // specifies alias for the table
    // JOIN `address` `a` ON `address`.`id`=`address_id`

$q->join('address.user_id');    // address.user_id = id
    // JOIN `address` ON `address`.`user_id`=`id`
```

You can also pass array as a first argument, to join multiple tables:

```
$q->table('user u');
$q->join(['a'=>'address', 'c'=>'credit_card', 'preferences']);
```

The above code will join 3 tables using the following query syntax:

```
join
    address as a on a.id = u.address_id
    credit_card as c on c.id = u.credit_card_id
    preferences on preferences.id = u.preferences_id
```

However normally you would have *user_id* field defined in your suplimentary tables so you need a different syntax:

```
$q->table('user u');
$q->join([
    'a'=>'address.user_id',
    'c'=>'credit_card.user_id',
    'preferences.user_id'
]);
```

The second argument to join specifies which existing table/field is used in *on* condition:

```
$q->table('user u');
$q->join('user boss', 'u.boss_user_id');
    // JOIN `user` `boss` ON `boss`.`id`=`u`.`boss_user_id`
```

By default the "on" field is defined as *$table."_id"*, as you have seen in the previous examples where join was done on "address_id", and "credit_card_id". If you have specified field explicitly in the foreign field, then the "on" field is set to "id", like in the example above.

You can specify both fields like this:

```
$q->table('employees');
$q->join('salaries.emp_no', 'emp_no');
```

If you only specify field like this, then it will be automatically prefixed with the name or alias of your main table. If you have specified multiple tables, this won't work and you'll have to define name of the table explicitly:

```
$q->table('user u');
$q->join('user boss', 'u.boss_user_id');
$q->join('user super_boss', 'boss.boss_user_id');
```

The third argument specifies type of join and defaults to "left" join. You can specify "inner", "straight" or any other join type that your database support.

Method can be executed several times on the same Query object.

Joining on expression

For a more complex join conditions, you can pass second argument as expression:

```
$q->table('user', 'u');
$q->join('address a', new Expression('a.name like u.pattern'));
```

Limiting result-set

`Query::limit` (*$cnt*, *$shift*)

Limit how many rows will be returned.

Parameters

- **$cnt** (*int*) – number of rows to return
- **$shift** (*int*) – offset, how many rows to skip

Returns $this

Use this to limit your *Query* result-set:

```
$q->limit(5, 10);
    // .. LIMIT 10, 5

$q->limit(5);
    // .. LIMIT 0, 5
```

Ordering result-set

`Query::order` (*$order*, *$desc*)

Orders query result-set in ascending or descending order by single or multiple fields.

Parameters

- **$order** (*int*) – one or more field names, expression etc.
- **$desc** (*int*) – pass true to sort descending

Returns $this

Use this to order your *Query* result-set:

```
$q->order('name');              // .. order by name
$q->order('name desc');         // .. order by name desc
$q->order('name desc, id asc')  // .. order by name desc, id asc
$q->order('name',true);         // .. order by name desc
```

Method can be executed several times on the same Query object.

Insert and Replace query

Set value to a field

`Query::set` (*$field*, *$value*)
Asigns value to the field during insert.

Parameters

- **$field** (*string*) – name of the field
- **$vlaue** (*mixed*) – value or expression

Returns $this

Example:

```
$q->table('user')->set('name', 'john')->insert();
    // insert into user (name) values (john)

$q->table('log')->set('date', $q->expr('now()'))->insert();
    // insert into log (date) values (now())
```

Method can be executed several times on the same Query object.

Set Insert Options

Update Query

Set Conditions

Same syntax as for Select Query.

Set value to a field

Same syntax as for Insert Query.

Other settings

Limit and Order are normally not included to avoid side-effects, but you can modify `$template_update` to include those tags.

Delete Query

Set Conditions

Same syntax as for Select Query.

Other settings

Limit and Order are normally not included to avoid side-effects, but you can modify `$template_update` to include those tags.

Dropping attributes

If you have called where() several times, there is a way to remove all the where clauses from the query and start from beginning:

`Query::reset` (*$tag*)

Parameters

- **$tag** (*string*) – part of the query to delete/reset.

Example:

```
$q
    ->table('user')
    ->where('name', 'John');
    ->reset('where')
    ->where('name', 'Peter');

// where name = 'Peter'
```

Other Methods

`Query::dsql` (*$properties*)
Use this instead of *new Query()* if you want to automatically bind query to the same connection as the parent.

`Query::option` (*$option*, *$mode*)
Use this to set additional options for particular query mode. For example:

```
$q
```

->table('test') ->field('name') ->set('name', 'John') ->option('calc_found_rows') // for default select mode ->option('ignore', 'insert') // for insert mode ;

$q->select(); // select calc_found_rows *name* from *test* $q->insert(); // insert ignore into *test* (*name*) values (*name* = 'John')

`Query::_set_args` (*$what*, *$alias*, *$value*)
Internal method which sets value in `Expression::args` array. It doesn't allow duplicate aliases and throws Exception in such case. Argument $what can be 'table' or 'field'.

Properties

property `Query::$mode`
Query will use one of the predefined "templates". The mode will contain name of template used. Basically it's array key of $templates property. See Query Modes.

property `Query::$defaultField`
If no fields are defined, this field is used.

property `Query::$template_select`
Template for SELECT query. See Query Modes.

property `Query::$template_insert`
Template for INSERT query. See Query Modes.

property `Query::$template_replace`
Template for REPLACE query. See Query Modes.

property `Query::$template_update`
Template for UPDATE query. See Query Modes.

property `Query::$template_delete`
Template for DELETE query. See Query Modes.

property `Query::$template_truncate`
Template for TRUNCATE query. See Query Modes.

CHAPTER 6

Results

When query is executed by `Connection` or PDO, it will return an object that can streem results back to you. The PDO class execution produces a PDOStatement object which you can iterate over.

If you are using a custom connection, you then will also need a custom object for streaming results.

The only requirement for such an object is that it has to be a Generator. In most cases developers will expect your generator to return sequence of id=>hash representing a key/value result set.

write more

CHAPTER 7

Transactions

When you work with the DSQL, you can work with transactions. There are 2 enhancements to the standard functionality of transactions in DSQL:

1. You can start nested transactions.
2. You can use *Connection::atomic()* which has a nicer syntax.

It is recommended to always use atomic() in your code.

class Connection

Connection::**atomic**(*$callback*)
: Execute callback within the SQL transaction. If callback encounters an exception, whole transaction will be automatically rolled back:

```
$c->atomic(function() use($c) {
    $c->dsql('user')->set('balance=balance+10')->where('id', 10)->update();
    $c->dsql('user')->set('balance=balance-10')->where('id', 14)->update();
});
```

atomic() can be nested. The successful completion of a top-most method will commit everything. Rollback of a top-most method will roll back everything.

Connection::**beginTransaction**()
: Start new transaction. If already started, will do nothing but will increase Connection::$transaction_depth.

Connection::**commit**()
: Will commit transaction, however if *Connection::beginTransaction* was executed more than once, will only decrease Connection::$transaction_depth.

Connection::**inTransaction**()
: Returns true if transaction is currently active. There is no need for you to ever use this method.

Connection::**rollBack**()
: Roll-back the transaction, however if *Connection::beginTransaction* was executed more than once, will only decrease Connection::$transaction_depth.

Warning: If you roll-back internal transaction and commit external transaction, then result might be unpredictable. Please discuss this https://github.com/atk4/dsql/issues/89

CHAPTER 8

Advanced Topics

DSQL has huge capabilities in terms of extending. This chapter explains just some of the ways how you can extend this already incredibly powerful library.

Advanced Connections

Connection is incredibly lightweight and powerful in DSQL. The class tries to get out of your way as much as possible.

Using DSQL without Connection

You can use *Query* and *Expression* without connection at all. Simply create expression:

```
$expr = new Expression('show tables like []', ['foo%']);
```

or query:

```
$query = (new Query())->table('user')->where('id', 1);
```

When it's time to execute you can specify your PDO manually:

```
$stmt = $expr->execute($pdo);
foreach($stmt as $row){
    echo json_encode($row)."\n";
}
```

With queries you might need to select mode first:

```
$stmt = $query->selectMode('delete')->execute($pdo);
```

The `Expresssion::execute` is a convenient way to prepare query, bind all parameters and get PDOStatement, but if you wish to do it manually, see *Manual Query Execution*.

Using in Existing Framework

If you use DSQL inside another framework, it's possible that there is already a PDO object which you can use. In Laravel you can optimise some of your queries by switching to DSQL:

```
$pdo = DB::connection()->getPdo();
$c = new Connection(['connection'=>$pdo]);

$user_ids = $c->dsql()->table('expired_users')->field('user_id');
$c->dsql()->table('user')->where('id', 'in', $user_ids)->set('active', 0)->update();

// Native Laravel Database Query Builder
// $user_ids = DB::table('expired_users')->lists('user_id');
// DB::table('user')->whereIn('id', $user_ids)->update(['active', 0]);
```

The native query builder in the example above populates $user_id with array from *expired_users* table, then creates second query, which is an update. With DSQL we have accomplished same thing with a single query and without fetching results too.

```
UPDATE
    user
SET
    active = 0
WHERE
    id in (SELECT user_id from expired_users)
```

If you are creating *Connection* through constructor, you may have to explicitly specify property `Connection::query_class`:

```
$c = new Connection(['connection'=>$pdo, 'query_class'=>'atk4\dsql\Query_SQLite']);
```

This is also useful, if you have created your own Query class in a different namespace and wish to use it.

Using Dumper and Counter

DSQL comes with two nice features - "dumper" and "counter". Dumper will output all the executed queries and how much time each query took and Counter will record how many queries were executed and how many rows you have fetched through DSQL.

In order to enable those extensions you can simply change your DSN from:

```
"mysql:host=localhost;port=3307;dbname=testdb"
```

to:

```
"dumper:mysql:host=localhost;port=3307;dbname=testdb"
"counter:mysql:host=localhost;port=3307;dbname=testdb"
"dumper:counter:mysql:host=localhost;port=3307;dbname=testdb"
```

When this DSN is passed into *Connection::connect*, it will return a proxy connection object that will collect the necessary statistics and "echo" them out.

If you would like to do something else with these statistics, you can set a callback. For Dumper:

```
$c->callback = function($expression, $time) {
    ...
}
```

and for Counter:

```
$c->callback = function($queries, $selects, $rows, $expressions) {
    ...
}
```

If you have used "dumper:counter:", then use this:

```
$c->callback = function($expression, $time) {
    ...
}

$c->connection()->callback = function($queries, $selects, $rows, $expressions) {
    ...
}
```

Proxy Connection

Connection class is designed to create instances of *Expression*, *Query* as well as executing queries. A standard *Connection* class with the use of PDO will do nothing inside its execute() because *Expression::execute* would handle all the work.

However if `Connection::connection` is NOT PDO, then *Expression* will not know how to execute query and will simply call:

```
return $connection->execute($this);
```

`Connection_Proxy` class would re-execute the query with a different connection class. In other words `Connection_Proxy` allows you to "wrap" your actual connection class. As a benefit you get to extend `Proxy` class implementing some unified features that would work with any other connection class. Often this will require you to know externals, but let's build a proxy class that will add "DELAYED" options for all INSERT operations:

```
class Connection_DelayInserts extends \atk4\dsql\Connection_Proxy
{
    function execute(\atk4\dsql\Expression $expr)
    {
        if ($expr instanceof \atk4\dsql\Query) {

            if ($expr->mode == 'insert') {
                $expr->insertOption('delayed');
            }

        }
        return parent::execute($expr);
    }
}
```

Next we need to use this proxy class instead of the normal one. Frankly, that's quite simple to do:

```
$c = \atk4\dsql\Connection::connect($dsn, $user, $pass);

$c = new Connection_DelayInserts(['connection'=>$c]);

// use the new $c
```

`Connection_Proxy` can be used for many different things.

Extending Query Class

You can add support for new database vendors by creating your own *Query* class. Let's say you want to add support for new SQL vendor:

```
class Query_MyVendor extends atk4\dsql\Query
{
    // truncate is done differently by this vendor
    protected $template_truncate = 'delete [from] [table]';

    // also join is not supported
    public function join($foreign_table, $master_field = null, $join_kind = null, $_
↪foreign_alias = null)
    {
        throw new atk4\dsql\Exception("Join is not supported by the database");
    }
}
```

Now that our custom query class is complete, we would like to use it by default on the connection:

```
$c = \atk4\dsql\Connection::connect($dsn, $user, $pass, ['query_class'=>'Query_
↪MyVendor']);
```

Adding new vendor support through extension

If you think that more people can benefit from your custom query class, you can create a separate add-on with it's own namespace. Let's say you have created *myname/dsql-myvendor*.

1. Create your own Query_* class inside your library. If necessary create your own Connection_* class too.
2. Make use of composer and add dependency to DSQL.
3. Add a nice README file explaining all the quirks or extensions. Provide install instructions.
4. Fork DSQL library.
5. Modify *Connection::connect* to recognize your database identifier and refer to your namespace.
6. Modify docs/extensions.rst to list name of your database and link to your repository / composer requirement.
7. Copy phpunit-mysql.xml into phpunit-myvendor.xml and make sure that dsql/tests/db/* works with your database.

Finally:

- Submit pull request for only the Connection class and docs/extensions.rst.

If you would like that your vendor support be bundled with DSQL, you should contact copyright@agiletoolkit.org after your external class has been around and received some traction.

Adding New Query Modes

By Default DSQL comes with the following *Query Modes*:

- select
- delete
- insert

- replace
- update
- truncate

You can add new mode if you wish. Let's look at how to add a MySQL specific query "LOAD DATA INFILE":

1. Define new property inside your *Query* class $template_load_data.
2. Add public method allowing to specify necessary parameters.
3. Re-use existing methods/template tags if you can.
4. Create _render method if your tag rendering is complex.

So to implement our task, you might need a class like this:

```
use \atk4\dsql\Exception;
class Query_MySQL extends \atk4\dsql\Query_MySQL
{
    protected $template_load_data = 'load data local infile [file] into table [table]
↪';

    public function file($file)
    {
        if (!is_readable($file)) {
            throw Exception(['File is not readable', 'file'=>$file]);
        }
        $this['file'] = $file;
    }

    public function loadData()
    {
        return $this->mode('load_data')->execute();
    }
}
```

Then to use your new statement, you can do:

```
$c->dsql()->file('abc.csv')->loadData();
```

Manual Query Execution

If you are not satisfied with *Expression::execute* you can execute query yourself.

1. *Expression::render* query, then send it into PDO::prepare();
2. use new $statement to bindValue with the contents of `Expression::params`;
3. set result fetch mode and parameters;
4. execute() your statement

Exception Class

DSQL slightly extends and improves *Exception* class

class Exception

The main goal of the new exception is to be able to accept additional information in addition to the message. We realize that often $e->getMessage() will be localized, but if you stick some variables in there, this will no longger be possible. You also risk injection or expose some sensitive data to the user.

Exception::**__construct**(*$message*, *$code*)
Create new exception

Parameters

- **$message** (*string|array*) – Describes the problem
- **$code** (*int*) – Error code

Usage:

```
throw new atk4\dsql\Exception('Hello');

throw new atk4\dsql\Exception(['File is not readable', 'file'=>$file]);
```

When displayed to the user the exception will hide parameter for $file, but you still can get it if you really need it:

Exception::**getParams**()
Return additional parameters, that might be helpful to find error.

Returns array

Any DSQL-related code must always throw atk4dsqlException. Query-related errors will generate PDO exceptions. If you use a custom connection and doing some vendor-specific operations, you may also throw other vendor-specific exceptions.

CHAPTER 9

Vendor support and Extensions

Vendor	Support	PDO	Dependency
MySQL	Full	mysql:	native, PDO
SQLite	Full	sqlite:	native, PDO
PostgreSQL	Untested	pgsql:	native, PDO
MSSQL	Untested	mssql:	native, PDO

Note: Most PDO vendors should work out of the box

Other Interesting Drivers

Class	Support	PDO	Dependency
Connection_Dumper	Full	dumper:	native, Proxy
Connection_Counter	Full	counter:	native, Proxy

3rd party vendor support

Class	Support	PDO	Dependency
Connection_MyVendor	Full	myvendor:	http://github/test/myvendor

See *Adding new vendor support through extension* for more details on how to add support for your driver.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index

R

S

T

U

W