SQL

The SQL class provides a lightweight, consistent interface for accessing SQL databases in PHP. It is a superset of the php PDO class (http://www.php.net/manual/en/class.pdo.php).

Namespace: `\DB`
File location: `lib/db/sql.php`

Constructor

```
$db = new \DB\SQL ( string $dsn [, string $user = NULL [, string $pw = NULL [, array
$options = NULL ]]] );
```

For example, to connect to a MySQL database, the syntax looks like:

```
$db=new \DB\SQL('mysql:host=localhost;port=3306;dbname=mysqldb','username','passwor
d');
```

Connecting to a SQLite database would look like:

```
$db=new \DB\SQL('sqlite:/path/to/db.sqlite');
```

The 4th parameter is an array of options you can use to set additional PDO attributes:

```
$options = array(
    \PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION, // generic attribute
    \PDO::ATTR_PERSISTENT => TRUE,  // we want to use persistent connections
    \PDO::MYSQL_ATTR_COMPRESS => TRUE, // MySQL-specific attribute
);
$db = new \DB\SQL('mysql:host=localhost;port=3306;dbname=mysqldb','username','passwor
d', $options);
```

Here is a list of links to DSN connection details for all currently supported engines in the SQL layer:

- mysql (http://www.php.net/manual/en/ref.pdo-mysql.php): MySQL 5.x
- sqlite (http://www.php.net/manual/en/ref.pdo-sqlite.connection.php): SQLite 3 and SQLite 2
- pgsql (http://www.php.net/manual/en/ref.pdo-pgsql.connection.php): PostgreSQL
- sqlsrv (http://www.php.net/manual/en/ref.pdo-sqlsrv.connection.php): Microsoft SQL Server / SQL Azure
- mssql, dblib, sybase (http://www.php.net/manual/en/ref.pdo-dblib.connection.php): FreeTDS / Microsoft SQL Server / Sybase
- odbc (http://www.php.net/manual/en/ref.pdo-odbc.connection.php): ODBC v3
- oci (http://www.php.net/manual/en/ref.pdo-oci.connection.php): Oracle

Methods

driver

Return the SQL driver name

```
echo $db->driver(); // mysql
```

version

Return the server version

```
echo $db->version(); // 5.1.51
```

name

Return the database name

```
echo $db->name(); // mysqldb
```

schema

Retrieve schema of SQL table

```
array|FALSE schema ( string $table [, array|string $fields = NULL [, int $ttl = 0 ]]
)
```

This function allows you to retrieve the schema of a given SQL table.

`$fields` is either an array or a list (according to the F3 function split (base#split)) of the names of columns to include in the returned schema. Defaulted to all fields.

When specified, `$ttl` will trigger a cache check for previous schema results and if not found or expired, will save the actual result to the cache backend, provided a CACHE (quick-reference#cache) system is activated.

Example of use:

```
$db->exec("CREATE TABLE IF NOT EXISTS mytable
          (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
           name varchar(128) NULL DEFAULT 'anonymous',
           age SMALLINT UNSIGNED NOT NULL,
           birth DATE NULL
          )"
);
$columns = $db->schema('mytable', 'name;age'); // only interested in these 2 columns
var_dump($columns);
// outputs
array (size=2) {
    'name' => array (size=5) {
        'type' => string 'varchar(128)' (length=12)
        'pdo_type' => int 2  // \PDO::PARAM_STR
        'default' => string 'anonymous' (length=9)
        'nullable' => boolean true
        'pkey' => boolean false
    }
    'age' => array (size=5) {
        'type' => string 'SMALLINT UNSIGNED' (length=17)
        'pdo_type' => int 1 // \PDO::PARAM_INT
        'default' => null
        'nullable' => boolean false
        'pkey' => boolean false
    }
}
```

MySQL Hint:
You can improve InnoDB performance on MySQL with `SET GLOBAL innodb_stats_on_metadata=0;` *! This requires SUPER privilege!*

exec

Execute a SQL command

```
array|int|FALSE exec ( string|array $commands [, string|array $args = NULL [, int $tt
l = 0 [, bool $log=TRUE ]]] )
```

This method allows you to execute one or more given `$commands` SQL statements and returns either the resulting rows (for SELECT, CALL, EXPLAIN, PRAGMA & SHOW statements) or the number of affected rows (for INSERT, DELETE & UPDATE statements) or `FALSE` on failure.

When specified, `$args` allows you to apply specific arguments to the SQL commands.

The `$ttl` argument, when specified, will trigger a cache check for previous command and if not found or expired, will save the actual result to the cache backend, provided a CACHE (quick-reference#cache) system is activated.

The `$log` is a toggle switch for suppressing or enabling the log of executed commands. You can use it as a profiler as the processing time, in milliseconds, of every SQL command is logged as well.

For example, consider the following table `mytable`:

id	name
1	Joe
2	William
3	Jack
4	Averell

a SELECT statement would return an array of rows:

```
$rows=$db->exec('SELECT id,name FROM mytable ORDER BY id DESC');
echo count($rows); // outputs 4
foreach($rows as $row)
  echo $row['name'];
// outputs 'Averell,Jack,William,Joe,'
```

while an UPDATE statement would return the number of updated rows:

```
echo $db->exec('UPDATE mytable SET id=id+10'); // outputs 4
```

Parameterized queries

The `exec()` method's 2nd argument is there to pass arguments safely (cf. Parameterized Queries (databases#parameterized-queries)).

For example, instead of writing:

```
$db->exec('INSERT INTO mytable VALUES(5,\'Jim\')')
```

it is highly encouraged to write:

```
$db->exec('INSERT INTO mytable VALUES(:id,:name)',array(':id'=>5,':name'=>'Jim'))
```

Here's the equivalent syntax with unnamed placeholders:

```
$db->exec('INSERT INTO mytable VALUES(?,?)',array(5,'Jim'))
```

The short syntax for single placeholders looks like the following:

```
$db->exec('INSERT INTO mytable(name) VALUES(?)','Jim');
```

Notice:
Prior to Fat-Free Framework 3.5.1, parameters had to be provided as 1-based arrays for unnamed placeholders. Otherwise, an `Invalid parameter number` error was returned. See fatfree#853 (https://github.com/bcosca/fatfree/issues/853) for more details. The parameters can be provided as 0-based or 1-based arrays.

```
$db->exec('INSERT INTO mytable VALUES(?,?)',array(1=>5,2=>'Jim'))
```

Query caching

The 3rd argument `$ttl` is used to enable query caching. Set it to your desired time-to-live in seconds and make sure you have a CACHE activated (quick-reference#cache). This way you can speed up your application when processing data that does not change very frequently.

The 4th argument `$log` is a toggle switch for suppressing or enabling the log of executed commands. You can use it as a profiler as the processing time, in milliseconds, of every SQL command is logged as well.

Transaction

Several SQL statements can be executed at once, if providing an array of statements. F3 will execute them as transaction, so if one statement fails, the whole query stack is rolled back.

> Be aware that the return value refers to the last executed statement only.

```
$result=$db->exec(array(
    'INSERT INTO mytable VALUES(:id,:name)',
    'DELETE FROM mytable'
  ),
  array(
    array(':id'=>6,':name'=>'Bill'),
    NULL
  )
);
echo $result; // outputs 6 (deleted rows)
```

If you need a return value for each statement, then you have to explicitly define a transaction and use `exec()` for each statement (cf. below).

begin, rollback & commit

Start, abort or end a SQL transaction

```
// state 1: empty table
$db->begin();
$db->exec('INSERT INTO mytable(name) VALUES(?)','Alfred');
$db->exec('INSERT INTO mytable(name) VALUES(?)','Bonnie');
// state 2: table contains 2 rows
$db->rollback();
// state 3: back to state 1
$db->exec('INSERT INTO mytable(name) VALUES(?)','Clyde');
// state 4: table contains 1 row
$db->commit();
// state 5: changes commited
// end of transaction: only Clyde has been inserted to database
```

trans

Return TRUE if a SQL transaction is currently active

```
if (!$db->trans())
    $db->begin();
```

count

Return the number of rows affected by the last query

```
$db->exec('INSERT INTO mytable(name) VALUES(?)','Alfred');
echo $db->count(); // outputs 1
$db->exec('INSERT INTO mytable(name) VALUES(?)','Bonnie');
echo $db->count(); // outputs 1
$db->exec('SELECT * FROM mytable');
echo $db->count(); // outputs 2
```

log

Return the SQL profiler results

```
$db->exec('INSERT INTO mytable(name) VALUES(?)','Clyde');
$db->exec('INSERT INTO mytable(name) VALUES(?)','Don');
$db->exec('INSERT INTO mytable(name) VALUES(?)','Elliott');

echo $db->log();

// outputs:
Mon, 27 Dec 2013 12:26:05 +0100 (2.0ms) INSERT INTO mytable(name) VALUES('Clyde')
Mon, 27 Dec 2013 12:26:05 +0100 (3.6ms) INSERT INTO mytable(name) VALUES('Don')
Mon, 27 Dec 2013 12:26:05 +0100 (1.3ms) INSERT INTO mytable(name) VALUES('Elliott')
```

uuid

Return unique connection identifier hash

```
string uuid ( )
```

This function returns the hash (base#hash) of the `$dsn` DSN passed to the __constructor (sql#constructor)

type

Map the data type of an argument to its reciprocal PDO constant

```
int type ( scalar $val )
```

This function allows you to retrieve the PDO constant corresponding to the php type of the provided `$val` as follow:

- the `NULL` php type will return `\PDO::PARAM_NULL`
- the `boolean` php type will return `\PDO::PARAM_BOOL` (int 5)
- the `integer` php type will return `\PDO::PARAM_INT` (int 1)

- and any other php type will return `\PDO::PARAM_STR` (int 2)

quote

Quote string

```
string quote ( string $val [, int $type = \PDO::PARAM_STR ] )
```

quotekey

Return quoted identifier name

```
string quotekey ( string $key )
```

This function quotes a table or column key name according to the requirements and syntax of the current database engine. E.g will quote `page to "page"` for SQLite and Oracle; while it will quote `page to `page`` for a MySQL based request.

pdo

Return PDO object

```
\PDO pdo ( )
```