SQL Mapper

The SQL Object-Relational-Mapper is an implementation of the abstract Active Record Cursor class (cursor).

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

Instantiation

To use the SQL ORM, create a valid SQL DB Connection (sql#constructor) and follow this example:

```
$mapper = new \DB\SQL\Mapper(\DB\SQL $db, string $table [, array|string $fields = NUL
L [, int $ttl = 60 ]] )
```

The `$fields` argument allows you to specify only the fields you need to map. `$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 mapper. Defaulted to all columns.

The 4th argument `$ttl` is a TTL to give the schema detector a hint about how often a SHOW COLUMNS call is issued by the mapper. When `$ttl` != 0, a cache check for previous schema is triggered and if expired or not found, the actual result is saved to the cache backend, provided a CACHE (quick-reference#cache) system is activated.

Now, if you'd like to create a model class, you might like to wrap it up:

```
$f3->set('DB',new DB\SQL('sqlite:db/database.sqlite'));

class User extends \DB\SQL\Mapper {
    public function __construct() {
        parent::__construct( \Base::instance()->get('DB'), 'users' );
    }
}

$user = new User();
$user->load('id = 1');
// etc.
```

Syntax

$filter

The `$filter` argument for SQL accepts the following structure:

- string value for simple where strings

```
string $whereClause
```

- array value for parameterized queries

```
array ( string $whereClause [, string $bindValue1 [, string $bindValue2 [, ...
]]] )
```

Parameterized Queries

It is recommended to use parameterized queries for all `where` conditions that may include user input data.

An example with question mark positional parameters:

```
$mapper->load(array('username = ? and password = ? and deleted = 0','John','acbd18db4
cc2f85cedef654fccc4a4d8'));
```

And with named parameters:

```
$mapper->load(array(
    'username = :user and password = :pass and deleted = 0',
    ':user'=>'John',':pass'=>'acbd18db4cc2f85cedef654fccc4a4d8'
));
```

> Workaround: Due to a PDO limitation, you cannot use a named parameter more than once in a query. You need to create e.g. 2 parameters `:user1` and `:user2` and pass them the same value.

> Make Your Choice: You cannot use both named and question mark positional parameter markers within the same SQL statement; pick one or the other parameter style but don't mix.

User-specified data type

Usually the data type is auto-detected, but to force a bind value to be a specific PDO type, use the following syntax:

```
array(
    'prize > :prize and active = 1',
    ':prize' => array(123, \PDO::PARAM_INT)
)
```

Search

When you use a `LIKE` operator in your `where` condition, notice that the `%` wildcards do not belong into the `where` criteria, but goes into the bind parameter like this:

```
$user->find(array('email LIKE ?','%gmail%')); // returns all users with an email addr
ess at GMAIL
```

Full-text search with MATCH

If you'd like to do a full-text search for a keywords against one or multiple fields in MySQL, you can use the `MATCH AGAINST` feature, which looks like this:

```
$text = 'some text';
$mapper->find(["MATCH (name,code) AGAINST (:search IN BOOLEAN MODE)", ':search' => $t
ext ]);
```

If you also want to sort the results by relevance, you need to add this match expression as adhoc field like this:

```
$mapper->relevance = "MATCH (name,code) AGAINST (:search1 IN BOOLEAN MODE)";
$mapper->find(["MATCH (name,code) AGAINST (:search2 IN BOOLEAN MODE)", ':search1' =>
$text, ':search2' => $text ], ['order'=>'relevance desc']);
```

And in case you want to match against multiple keywords at once, you need to wrap and **escape** the keywords

```
$mapper->find(["(MATCH(fieldA,fieldB) AGAINST('(".implode('") ("',$keywords).")' IN B
OOLEAN MODE )"]);
```

or insert multiple placeholders:

```
$mapper->find(["(MATCH(fieldA,fieldB) AGAINST('(?) (?) (?)' IN BOOLEAN MODE )", $a,
$b, $c]);
```

$option

The `$option` argument for SQL accepts the following structure:

```
array(
    'order' => string $orderClause,
    'group' => string $groupClause,
    'limit' => integer $limit,
    'offset' => integer $offset
)
```

i.e:

```
array(
    'order' => 'score DESC, team_name ASC',
    'group' => 'score, player',
    'limit' => 20,
    'offset' => 0
)
```

Methods

table

Return the name of the mapped table

```
string table()
```

exists

Return TRUE if a given field is defined

```
bool exists( string $key )
```

changed

Return TRUE if any/specified field value has changed

```
bool changed ( [ string $key = NULL ] )
```

set

Assign a given value to a field

```
scalar set( string $key, scalar $val )
```

This class takes advantage of the Magic class (magic) and ArrayAccess interface. It means you can set and get variables with direct access like this:

```
$mapper->foo = 'bar';
$mapper['foo'] = 'bar';
```

Virtual Fields

If you set a new value to an empty / not hydrated mapper, you create a virtual field on it. This way you can add some aggregate functions to your query:

```
$scores = new Scores();
$scores->sum_score = 'SUM(score)';
$scores->avg_score = 'AVG(score)';
$scores->load(null,array('group'=>'player_id'));
echo $scores->sum_score; // returns the sum of all scores made by player_id
echo $scores->avg_score; // returns the avarage score of that player
```

get

Retrieve value of a field

```
scalar get( string $key )
```

To get the ID of the last inserted row or the last value from a sequence object, you must use the reserved `$key` value = `'_id'`:

```
$lastInsertedID = $mapper->get('_id'); // get the ID of the last inserted row or the
last value from a sequence object
```

clear

Clear value of a field

```
NULL clear( string $key )
```

type

Get the name of the PHP type equivalent of a PDO constant

```
string type( string $pdo )
```

Basically, this method converts a given PDO types constants to the equivalent named PHP types as follow:

```
switch ($pdo) {
    case \PDO::PARAM_NULL:
        return 'unset';
    case \PDO::PARAM_INT:
        return 'int';
    case \PDO::PARAM_BOOL:
        return 'bool';
    case \PDO::PARAM_STR:
        return 'string';
}
```

value

Cast value to a PHP type

```
scalar value( string $type, scalar $val );
```

This method allows you to cast the value from a DB to a PHP type. Basically, this method converts PDO types to equivalent PHP types as follow:

```
switch ($type) {
    case \PDO::PARAM_NULL:
        return (unset)$val;
    case \PDO::PARAM_INT:
        return (int)$val;
    case \PDO::PARAM_BOOL:
        return (bool)$val;
    case \PDO::PARAM_STR:
        return (string)$val
```

cast

Return the fields of the mapper object as an associative array

```
array cast( [ object $obj = NULL ] );
```

select

Build a query string and execute it

```
array select( string $fields [, string|array $filter = NULL [, array $options = NULL
[, int $ttl = 0 ]]] );
```

find

Return records that match a given criteria

```
array find( [ string|array $filter = NULL [, array $options = NULL [, int $ttl = 0
]]] );
```

count

Count records that match a given criteria

```
int count( [ string|array $filter = NULL [, $ttl=0 ]] )
```

insert

Insert a new record

```
object insert()
```

update

Update the current record

```
object update()
```

save

Update an existing record, or insert a new one

```
object save()
```

If one or more records have been loaded into the mapper, `save` will use the `update` method. If there are no records currently loaded in the mapper, `save` will use the `insert` method. The `save` method is often used in conjunction with the `copyfrom` method.

skip

Return the record at the specified offset using the same criteria as previous load() call and make it active

```
array skip( [ int $ofs = 1 ] )
```

erase

Delete the current record

```
int erase( [ string|array $filter = NULL ] )
```

This deletes the current mapped record. If a `$filter` is given, it performs a SQL `DELETE FROM $this->table WHERE $filter` on the table specified when instanciating the mapper. Notice that erase event hooks are skipped when `$filter` is present.

reset

Reset the cursor

```
NULL reset( )
```

All underlying values are set to `NULL`.

copyfrom

Hydrate the mapper object using an array

```
NULL copyfrom( array | string $var [, callback $func = NULL ] )
```

This function allows you to hydrate the mapper using an array (or the name of a hive variable containing an array).

`$func` is a callback function you can apply to the hive *array* variable. As explained in the Databases User Guide (databases#beyond-crud), the array keys must have names identical to the mapper object properties. It allows for example to hydrate the mapper object with the fields of a POSTed form:

```
$f3->get('user')->copyfrom('POST'); // F3 synch the 'POST' hive array variable with t
he $_POST array
```

Danger By default, `copyfrom` takes the whole array provided; in our example above, the whole `POST` from the <form>. So if somebody modifies or forges your form by adding some extra <input> fields in your DOM with tools like e.g. firebug, it's possible to overwrite e.g. the ID of the record, the permission role, or what ever… Pretty huge *security leak*. Fortunately, F3 offers you a versatile solution through a callback function you can use to apply any pre-processing on the hive *array* variable, such as normalizing the values and/or filtering and limiting the fields to copy from. Your callback function will receive the hive *array* variable and must similarly return an array of keys/values pairs: the fields to pass to the mapper object.

Ok, let's do it. For example, let's use a callback filter function retaining only the fields 'name' & 'age':

```
$db = new DB\SQL('sqlite:db/ent.sqlite');
$f3->set('user',new DB\SQL\Mapper($db,'users'));
$f3->get('user')->copyfrom('POST',function($val) {
    // the 'POST' array is passed to our callback function
    return array_intersect_key($val, array_flip(array('name','age')));
});
$f3->get('user')->save();
});
```

That's it! As F3 sanitizes the values, with such an extra filtering, your DB is safe from injections.

copyto

Populate hive array variable with mapper fields

```
NULL copyto( string $key )
```

fields

Return field names

```
array fields( bool $adhoc = TRUE )
```

This method returns all available fields for this mapper. The `$adhoc` argument controls if the adhoc virtual fields are returned as well. This method will respect any limitations on fields that were set during instantiation (sql-mapper#Instantiation).

schema

Returns the table schema

```
array schema( array $fields = null )
```

When `$fields` argument is provided, it's used to update the underlaying field schema. See Retrieve schema of SQL table (sql#schema) for additional information.

required

Return TRUE if field is not nullable

```
bool required( string $field )
```

factory

Convert an array to a mapper object

```
protected object factory( array $row )
```

This *protected* method is used internally by the `select` method.