Cursor

This Cursor class is an abstract foundation of an Active Record (https://en.wikipedia.org/wiki/Active_Record) implementation, that is used by all F3 Data Mappers.

Have a look at the SQL Mapper (sql-mapper), Mongo Mapper (mongo-mapper) or JIG Mapper (jig-mapper) page to get to know about how to create and use them with their own functions and the following described below.

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

Cursor Methods

The following methods are available to all Data Mappers. Even if the filter and option syntax may differ from one mapper to another, the behaviour of these functions are the same. So assume you have this set of data in your DB table when you'll be reading the examples below.

ID	title	text	author
1	F3 for the win	Use It Now!	49
2	Once upon a time	there was a dragon.	2
3	Barbar the Foo	foo bar	25
4	untitled	lorem ipsum	8

load

Map to first record that matches criteria

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

The `load` method hydrates the mapper object with records. You can define a `$filter` to load only records that matches your criteria.

It uses find() (cursor#find) to load records and makes the first record (mapper object) that matches criteria the active record. Additional records that match the same criteria can be made the active record by moving the cursor pointer (i.e. with skip() (cursor#skip)).

You can find detailed descriptions about the `$filter` and `$option` syntax on the mapper specific pages: Jig Mapper (jig-mapper#$filter), Mongo Mapper (mongo-mapper#$filter) and SQL Mapper (sql-mapper#$filter).

The `$ttl` argument, when specified in seconds, allows you to cache the result of the mapper load, provided a CACHE (quick-reference#cache) system is activated.

Let's start with a simple example where we do not specify any filter, so the first record will be loaded:

```
$mapper->load();  // by default, loads the 1st record and makes it active
echo $mapper->title; // displays 'F3 for the win'
```

Important:
Make sure there is enough memory for the result set returned by this function. Attempting to `load` (without filtering) from a huge database might go beyond PHP's `memory_limit` .

The Cursor class extends the Magic class (magic). This means you are able to access your data fields like you do it with any object properties and array keys:

```
echo $mapper->title; // 'F3 for the win'
echo $mapper['text']; // 'Use It Now!'
echo $mapper->get('author'); // 49
```

next

Map to next record

```
mixed next ( )
```

When a mapper object is hydrated, its internal cursor pointer points to a single record. To move the pointer forward to the next record, you use this method.

```
$mapper->load();
echo $mapper->title; // 'F3 for the win'
$mapper->next();
echo $mapper->title; // 'Once upon a time'
```

Internally, the `Cursor` class fetches all records from the underlying database matching the `$filter` specified when calling `load()` . Any attempt to navigate to the next, previous, first or last record will just move the internal pointer, thereby reducing disk I/O.

prev

Map to previous record

```
mixed prev ( )
```

first

Map to first record in cursor

```
mixed first ( )
```

last

Map to last record in cursor

```
mixed last ( )
```

skip

Map to n-th record relative to current cursor position

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

dry

Return `TRUE` if the current cursor position is not mapped to any record

```
bool dry ( )
```

This is some kind of "empty" function. It returns `TRUE` if the cursor is not mapped to any database record, even if records have been `load` ed.

The next example snippet loops through all loaded records and stops when the current cursor pointer is dry (not hydrated) :

```
$mapper->load();  // by default, loads the 1st record

while ( !$mapper->dry() ) {  // gets dry when we passed the last record
    echo $mapper->title;
    // moves forward even when the internal pointer is on last record
    $mapper->next();
}
```

findone

Return first record (mapper object) that matches criteria

```
object|FALSE findone ( [ string|array $filter = NULL [, array $options = NULL [, int
$ttl = 0 ]]] )
```

Use this method if you only want to process a single entity in your business logic. It is helpful to only `load` that single record.

See the load() method (cursor#load) for details regarding the parameters.

paginate

Return a subset of records with additional pagination information

```
array paginate ( [ int $pos = 0 [, int $size = 10 [, string|array $filter = NULL [, a
rray $options = NULL [, int $ttl = 0]]]]] )
```

This method returns an array containing a subset of records that are matching the `$filter` criterias, the total number of records found, the specified limit `$size`, the number of subsets available and the actual subset position.

For example:

```
// page 1
$result = $mapper->paginate(0, 3);
/*
array(4) {
    ["subset"] => array(3) {
        [0] => mapper object, #ID: 1, title: F3 for the win
        [1] => mapper object, #ID: 2, title: Once upon a time
        [2] => mapper object, #ID: 3, title: Barbar the Foo
    }
    ["total"] => int(4)
    ["limit"] => int(3)
    ["count"] => float(2)
    ["pos"] => int(0)
}
*/

// page 2
$result = $mapper->paginate(1, 3);
/*
array(4) {
    ["subset"] => array(1) {
        [0] => mapper object, #ID: 4, title: untitled
    }
    ["total"] => int(4)
    ["limit"] => int(3)
    ["count"] => float(2)
    ["pos"] => int(1)
}
*/
```

The `subset` key contains an array of mapper objects returned from find(). `total` is the sum of all records for all pages. `limit` holds the same value as the `size` input parameter to the call. `count` is the number of subsets/pages available. `pos` gives you the current subset cursor position (it's the page number - 1).

loaded

Return the count of records loaded

```
int loaded ( )
```

save

Save mapped record

```
mixed save ( )
```

This method saves data to the database. The `Cursor` class automatically determines if the record should be updated (cursor#update) or inserted (cursor#insert) as a new entry, based on the return value of `dry`.

erase

Delete current record

```
int|bool erase ( )
```

reset

Reset/dehydrate the cursor

```
NULL reset ( )
```

Event handlers

All Cursor derivatives offer the possibility to intercept the processing on certain events through custom functions. This gives you the opportunity to add some validation, sanitation or more complex business logic. See the following described methods to get an overview of possible events you may hook into.

Notice: If you extend the mapper and try to read or modify mapper fields (especially those that also exist as class property) with magic getter/setter within those event handlers, you will get unpredictable behaviours or run into errors.

The simple solution to this is to stick to the usage of the `get` or `set` methods from within the child class. These are safe to use and won't override protected properties.

```
class Model extends \DB\SQL\Mapper {
    function __construct(){
        parent::__construct(\Base::instance()->get('DB'),'test_model');
        $this->beforeinsert(function($self){
            $self->source='bar'; // fails due to protected property named 'source'
            $self->set('source','bar'); // works - just use the setter directly
        });
}}
```

Refer to the FatFree GitHub Issue #697 (https://github.com/bcosca/fatfree/issues/697) for more details.

onload

Define a hook to the `onload` event

The hook will be executed everytime the mapper is loaded. For example, it will be executed once after a call to `load()` , `next()` or `prev()` , but also will be executed for every mapper returned by `find()` .

```
callable onload ( callable $func )
```

The hook function takes 1 argument : the mapper object. E.g:

```
$mapper->onload(function($self){
  //do something
});
```

beforeinsert, afterinsert

Define hooks to the `beforeinsert and afterinsert events`

The `beforeinsert` hook will be executed just before any INSERT statement and is able to abort the event by returning `false` .

The `afterinsert` hook will be executed right after any issued INSERT statement.

```
callable beforeinsert ( callable $func )
callable afterinsert ( callable $func )
```

Each hook function takes 2 arguments : the mapper object and the primary key(s) value(s). E.g:

```
$mapper->beforeinsert(function($self,$pkeys){
  //do something before inserting
});
$mapper->afterinsert(function($self,$pkeys){
  //do something after inserting
});
```

beforeupdate, afterupdate

Define hooks to the `beforeupdate and afterupdate events`

The `beforeupdate` hook will be executed just before any UPDATE statement and is able to abort the event by returning `false`.

The `afterupdate` hook will be executed right after any issued UPDATE statement.

```
callable beforeupdate ( callable $func )
callable afterupdate ( callable $func )
```

Each hook function takes 2 arguments : the mapper object and the primary key(s) value(s). E.g:

```
$mapper->beforeupdate(function($self,$pkeys){
  //do something before updating
});
$mapper->afterupdate(function($self,$pkeys){
  //do something after updating
});
```

beforesave, aftersave

Define hooks to the `beforeinsert`, `beforeupdate`, `afterinsert` and `afterupdate` events

The `beforesave` hook will be executed just before any INSERT or UPDATE statement and is able to abort the event by returning `false`.

The `aftersave` hook will be executed right after any issued INSERT or UPDATE statement.

```
callable beforesave ( callable $func )
callable aftersave ( callable $func )
```

Each hook function takes 2 arguments : the mapper object and the primary key(s) value(s). E.g:

```
$mapper->beforesave(function($self,$pkeys){
  //do something before inserting or updating
});
$mapper->aftersave(function($self,$pkeys){
  //do something after inserting or updating
});
```

Notice:
Either define a hook to the `save` event or to the underlying `insert` / `update` events, but not to both. Behind the scenes, a `save` hook is translated into both an `insert` *and* an `update` hook and that's why a `save` hook will overwrite any existing `insert` / `update` hooks (and vice versa).

beforeerase, aftererase

Define hooks to the `beforeerase` and `aftererase` events

The `beforerase` hook will be executed just before any DELETE statement and is able to abort the event by returning `false`.

The `aftererase` hook will be executed right after any issued DELETE statement.

```
callable beforeerase ( callable $func )
callable aftererase ( callable $func )
```

Each hook function takes 2 arguments : the mapper object and the primary key(s) value(s). E.g:

```
$mapper->beforeerase(function($self,$pkeys){
  //do something before deleting
});
$mapper->aftererase(function($self,$pkeys){
  //do something after deleting
  echo 'the id of the deleted row was '.$pkeys['id'];
});
```

Notice:
Please notice that the erase hooks can only be executed when using `$mapper->erase();` on a loaded/hydrated mapper. If you are going to erase a whole set by providing a `$filter` to the erase method, these event hooks will be skipped.

oninsert, onupdate, onerase

Provided for backwards compatibility, they are aliases to respectively `afterinsert()` , `afterupdate()` and `aftererase()` .

Abstract Methods

The following methods must be implemented by all extending mapper classes to work properly.

find

Return records (array of mapper objects) that match criteria

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

See the load() method (cursor#load) for details regarding the parameters.

insert

Insert a new record

```
array insert ( )
```

update

Update the current record

```
array update ( )
```