









 ikkez /  
**f3-cortex**



 **Code**  **Issues** **14**  **Pull requests**  **Actions**  **Projects**  **Security** 




A multi-engine ORM / ODM for the PHP Fat-Free Framework

 GPL-3.0 license

 **114 stars**  **19 forks**  **24 watching**  **Activity**

 Public repository

 **master** ▾

...

 **Branches**  **Tags**



ikkez ...

on Jan 24 

[View code](#)

# CORTEX

## A general purpose Data-Mapper for the PHP Fat-Free Framework

Cortex is a multi-engine ActiveRecord ORM / ODM that offers easy object persistence. Some of its main features are:

- It handles SQL, Jig and MongoDB database engines
- Write queries in well-known SQL Syntax, they can be translated to Jig and Mongo
- automated SQL table creation and column extension from defined schema configurations
- Easy prototyping with the SQL Fluid Mode, which makes your RDBMS schema-less and adds new table columns automatically
- Support for models and collections

- Relationships: link multiple models together to one-to-one, one-to-many and many-to-many associations
- smart-loading of related models (intelligent lazy and eager-loading with zero configuration)
- useful methods for nested filtering through relations
- lots of event handlers and custom setter / getter preprocessors for all fields
- define default values and nullable fields for NoSQL
- additional [validation plugin](#) available

With Cortex you can create generic apps, that work with any DB of the users choice, no matter if it's SQLite, PostgreSQL, MongoDB or even none. You can also mash-up multiple engines or use them simultaneously.

It's great for fast and easy data abstraction and offers a bunch of useful filter possibilities.

---

## Table of Contents

---

1. [Quick Start](#)
2. [SQL Fluid Mode](#)
3. [Cortex Models](#)
  - i. [Configuration](#)
    - a. [Additional Data Types](#)
    - b. [Alternative Configuration](#)
    - c. [Blacklist Fields](#)
  - ii. [Setup](#)
  - iii. [Setdown](#)
4. [Relations](#)
  - i. [Setup the linkage](#)
  - ii. [Working with Relations](#)
    - a. [One-To-One](#)
    - b. [Many-To-Many, bidirectional](#)
    - c. [Many-To-Many, unidirectional](#)
    - d. [Many-To-Many, self-referencing](#)
5. [Event Handlers](#)
  - i. [Custom Field Handler](#)
6. [Filter Query Syntax](#)
  - i. [Operators](#)
  - ii. [Options Array](#)

## 7. Advanced Filter Techniques

- i. [has](#)
- ii. [orHas](#)
- iii. [filter](#)

## 8. Insight into aggregation

- i. [Counting Relations](#)
- ii. [Virtual Fields](#)

## 9. Mapper API

## 10. Collection API

## 11. Additional notes

## 12. Known Issues

## 13. Roadmap

## 14. License

# Quick Start

---

## System Requirements

Cortex requires at least Fat-Free v3.4 and PHP 5.4. For some of the features, it also requires the F3 [SQL Schema Plugin](#).

## Install

To install Cortex, just copy the `/lib/db/cortex.php` file into your libs. For the SQL Schema Plugin, copy `lib/db/sql/schema.php` as well.

If you use **composer**, all you need is to run `composer require ikkez/f3-cortex:1.*` and it'll include Cortex and its dependencies into your package.

## Setup a DB

Create a DB object of your choice. You can choose between [SQL](#), [Jig](#) or [MongoDB](#). Here are some examples:

```
// SQL - MySQL
$db = new \DB\SQL('mysql:host=localhost;port=3306;dbname=MyAppDB','user','pw')
// SQL - SQLite
$db = new \DB\SQL('sqlite:db/database.sqlite');
// SQL - PostgreSQL
$db = new \DB\SQL('pgsql:host=localhost;dbname=MyAppDB','user','pw');
// SQL - SQL Server
$db = new \DB\SQL('sqlsrv:SERVER=LOCALHOST\SQLEXPRESS2012;Database=MyAppDB','u
// Jig
```

```
$db = new \DB\Jig('data/');
// Mongo
$db = new \DB\Mongo('mongodb://localhost:27017', 'testdb');
```

## Let's get it rolling

If you are familiar with F3's own Data-Mappers, you already know all about the basic CRUD operations you can do with Cortex too. It implements the ActiveRecord [Cursor Class](#) with all its methods. So you can use Cortex as a **drop-in replacement** of the F3 mappers and it's basic usage will stay that simple:

```
$user = new \DB\Cortex($db, 'users');
$user->name = 'Jack Ripper';
$user->mail = 'jacky@email.com';
$user->save();
```

Alright, that wasn't very impressive. But now let's find this guy again:

```
$user->load(['mail = ?', 'jacky@email.com']);
echo $user->name; // shouts out: Jack Ripper
```

As you can see, the filter array is pure SQL syntax, that you would already use with the F3 SQL Mapper. In Cortex this will work with all 3 DB engines. Here is a little more complex `where` criteria:

```
$user->load(['name like ? AND (deleted = 0 OR rights > ?]', 'Jack%', 3));
```

No need for complex criteria objects or confusing Mongo where-array constructions. It's just as simple as you're used to. Using a Jig DB will automatically translate that query into the appropriate Jig filter:

```
Array (
    [0] => (isset(@name) && preg_match(?,@name)) AND ( (isset(@deleted) && (@d
    [1] => /^Jack/
    [2] => 3
)
```

And for MongoDB it translates into this:

```
Array (
    [$and] => Array (
        [0] => Array (
```



```
// file at app/model/user.php
namespace Model;

class User extends \DB\Cortex {
    protected
        $db = 'AppDB1',      // F3 hive key of a valid DB object
        $table = 'users';    // the DB table to work on
}
```

Now you can create your mapper object that easy:

```
$user = new \Model\Users();
```

This is the minimal model configuration. Cortex needs at least a working DB object. You can also pass this through the constructor ( `new \Model\Users($db);` ) and drop it in the setup. `$db` must be a string of a hive key, where the DB object is stored *OR* the DB object itself. If no `$table` is provided, Cortex will use the class name as table name.

## Configuration

Cortex does not need that much configuration. But at least it would be useful to have setup the field configuration. This way it's able to follow a defined schema of your data entity and enables you to use some auto-installation routines (see [setup](#)). It looks like this:

```
// file at app/model/user.php
namespace Model;

class User extends \DB\Cortex {

    protected
        $fieldConf = [
            'name' => [
                'type' => 'VARCHAR256',
                'nullable' => false,
            ],
            'mail' => [
                'type' => 'VARCHAR128',
                'index' => true,
                'unique' => true,
            ],
            'website' => [
                'type' => 'VARCHAR256'
            ],
            'rights_level' => [
```

```

        'type' => 'TINYINT',
        'default' => 3,
    ],
],
$db = 'DB',
$table = 'users',
$primary = 'id';    // name of the primary key (auto-created), default: id
}

```

In the `$fieldConf` array, you can set data types ( `type` ), nullable flags and default values for your columns. With `index` and `unique` , you can even setup an index for

≡ [readme.md](#)

makes your models easy interchangeable along various databases using this loosely coupled field definitions.

**You don't need to configure all fields this way.** If you're working with existing tables, the underlying SQL Mapper exposes the existing table schema. So if you don't need that auto-installer feature, you can just skip the configuration for those fields, or just setup only those you need (i.e. for fields with relations).

Because column data types are currently only needed for setting up the tables in SQL, it follows that [SQL Data Types Table](#) from the required [SQL Schema Plugin](#).

You may also extend this config array to have a place for own validation rules or whatever you like.

The data type values are defined constants from the Schema Plugin. If you like to use some auto-completion in your IDE to find the right values, type in the longer path to the constants:

```

'type' => \DB\SQL\Schema::DT_TIMESTAMP,
'default' => \DB\SQL\Schema::DF_CURRENT_TIMESTAMP,

```

## Additional Data Types

Cortex comes with two own data types for handling array values in fields. Even when Jig and Mongo support them naturally, most SQL engines do not yet. Therefore Cortex introduces:

- `DT_SERIALIZED`
- `DT_JSON`

In example:

```
'colors' => [
    'type' => self::DT_JSON,
],
```

Now you're able to save array data in your model field, which is json\_encoded into a `text` field behind the scene (when using a SQL backend).

```
$mapper->colors = ['red', 'blue', 'green'];
```

## Alternative Configuration

In case you need some more flexible configuration and don't want to hard-wire it, you can overload the Model class constructor to load its config from an `ini`-file or elsewhere. In example:

```
class User extends \DB\Cortex {

    function __construct() {
        // get the DB from elsewhere
        $this->db = \Registry::get('DB');
        $f3 = \Base::instance();
        // load fields from .ini file
        if (!$f3->exists('usermodel'))
            $f3->config('app/models/usermodel.ini');
        foreach ($f3->get('usermodel') as $key => $val)
            $this->{$key} = $val;
        parent::__construct();
    }
}
```

And in your `usermodel.ini` file:

```
[usermodel]
table = users

[usermodel.fieldConf]
name.type = VARCHAR256
name.nullable = FALSE
mail.type = VARCHAR128
website.type = VARCHAR256
rights_level.type = TINYINT
rights_level.default = 3
```

## Blacklist Fields



The `fields()` method can be used to return the available fields on the current model. If called with one simple array argument like `$news->fields(['title']);`, it'll apply the provided elements as a whitelist to the whole mapper. For the rest of its lifetime it'll only hydrate the fields you permitted here. If called with a 2nd argument like `$news->fields(['author'], true);`, the array is going to be used as a blacklist instead, and restrict the access to the provided fields. You can also define deep nested fields using a **dot** as separator: `$news->fields(['tags.title']);` will only hydrate the tag title in your news model and won't load or save any other field that exists in your tag model. Subsequent calls to the `fields` method will merge with all already defined blacklist/whitelist definitions.

## Set up

This method creates the SQL DB tables you need to run your Cortex model. **It also adds just missing fields to already existing tables.**

If your Model has a valid field configuration, you are able to run this installation method:

```
\Model\User::setup();
```

If you have no model class, you need to provide all of the setup method's parameters.

```
$fields = [  
    'name' => ['type' => \DB\SQL\Schema::DT_TEXT],  
    'mail' => ['type' => \DB\SQL\Schema::DT_INT4],  
    'website' => ['type' => \DB\SQL\Schema::DT_INT4],  
];  
\DB\Cortex::setup($db, 'users', $fields);
```

## Set down

This method completely removes the specified table from the used database. So handle with care.

```
// With Model class  
\Model\User::setdown();  
  
// Without Model class  
\DB\Cortex::setdown($db, 'users');
```

# Relations

With Cortex you can create associations between multiple models. By linking them together, you can create all common relationships you need for smart and easy persistence.

## Setup the linkage

To make relations work, you need to use a model class with field configuration. Cortex offers the following types of associations, that mostly **must be defined in both classes** of a relation:

| Type | Model A         | Direction | Model B  |
|------|-----------------|-----------|----------|
| 1:1  | belongs-to-one  | <- ->     | has-one  |
| 1:m  | belongs-to-one  | <- ->     | has-many |
| m:m  | has-many        | <- ->     | has-many |
| m:m  | belongs-to-many | --->      |          |

This is how a field config looks with a relation:

```

1 <?php
2
3 class NewsModel extends \DB\Cortex {
4
5     protected
6     $fieldConf = array(
7         'title' => array(
8             'type' => \DB\SQL\Schema::DT_VARCHAR128
9         ),
10        'text' => array(
11            'type' => \DB\SQL\Schema::DT_TEXT
12        ),
13        'author' => array(
14            'belongs-to-one' => '\AuthorModel',
15        ),
16        'tags' => array(
17            'belongs-to-many' => '\TagModel',
18        ),
19        'tags2' => array(
20            'has-many' => array('\TagModel', 'news'),
21        ),
22    ),
23    $stable = 'news',
24    $db = 'DB';
25 }
  
```

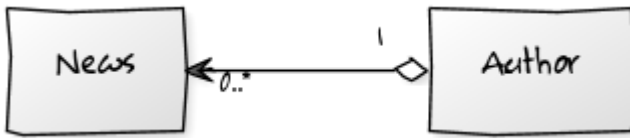
```

1 <?php
2
3 class AuthorModel extends \DB\Cortex {
4
5     protected
6     $fieldConf = array(
7         'name' => array(
8             'type' => \DB\SQL\Schema::DT_VARCHAR256,
9         ),
10        'mail' => array(
11            'type' => \DB\SQL\Schema::DT_VARCHAR128
12        ),
13        'website' => array(
14            'type' => \DB\SQL\Schema::DT_VARCHAR256
15        ),
16        'rights_level' => array(
17            'type' => \DB\SQL\Schema::DT_INT1,
18        ),
19        'news' => array(
20            'has-many' => array('\NewsModel', 'author'),
21        ),
22        'profile' => array(
23            'has-one' => array('\ProfileModel', 'author'),
24        ),
25    ),
26    $stable = 'author',
27    $db = 'DB';
28 }
29
  
```

This creates an aggregation between Author and News, so

- One News belongs to one Author.

- One Author has written many News.



As a side note: `belongs-to-*` definitions will create a new column in that table, that is used to save the id of the counterpart model (foreign key field). Whereas `has-*` definitions are just virtual fields which are going to query the linked models by their own id (the inverse way). This leads us to the following configuration schema:

For **belongs-to-one** and **belongs-to-many**

```

'realTableField' => [
    'relationType' => '\Namespace\ClassName',
],

```

Defining a foreign key for `belongs-to-*` is optional. The default way is to use the identifier field. For SQL engines this is either the default primary key `id` or the custom primary key that can be set with the `$primary` class property. NoSQL engines will use `_id`. If you need to define another non-primary field to join with, use `['\Namespace\ClassName', 'cKey']`.

For **has-one** and **has-many**

```

'virtualField' => [
    'relationType' => ['\Namespace\ClassName', 'foreignKey'],
],

```

The foreign key is the field name you used in the counterpart model to define the `belongs-to-one` connection.

## many-to-many

There is one special case for many-to-many relations: here you use a `has-many` type on both models, which implies that there must be a 3rd pivot table that will be used for keeping the foreign keys that binds everything together. Usually Cortex will auto-create that table upon `setup` method, using an auto-generated table name. If you like to use a custom name for that joining-table, add a 3rd parameter to the config array of *both* models, i.e.:

```

'tags' => [
    'has-many' => [\Model\Tag::class, 'news', 'news_tags'],
],

```

By default the primary key is used as reference for the record in the pivot table. In case you need to use a different field for the primary key, so can set a custom `localKey`.

```
'tag_key' => [
    'type' => \DB\SQL\Schema::DT_VARCHAR128,
],
'tags' => [
    'has-many' => [\Model\Tag::class, 'news', 'news_tags',
        'localKey' => 'tag_key'
    ],
],
```

For a custom relation key (foreign key) use `relPK`:

```
'tags' => [
    'has-many' => [\Model\Tag::class, 'news', 'news_tags',
        'relPK' => 'news_id'
    ],
],
```

### Custom pivot column names

If you're working with an existing database table, or want to use own field names for the column in the pivot table, you can set those up with the `relField` option:

I.e. in the news model:

```
'tags' => [
    'has-many' => [\Model\Tag::class, 'news', 'news_tags',
        'relField' => 'news_id'
    ],
],
```

and in the tag model:

```
'news' => [
    'has-many' => [\Model\News::class, 'tags', 'news_tags',
        'relField' => 'tag_id'
    ],
],
```

That means that the 3rd pivot table contains `news_id` and `tag_id` fields.

## Working with Relations

Okay finally we come to the cool part. When configuration is done and setup executed, you're ready to go.

## one-to-one

To create a new relation:

```
// load a specific author
$author = new \AuthorModel();
$author->load(['_id = ?', 2]);

// create a new profile
$profile = new ProfileModel();
$profile->status_message = 'Hello World';

// link author and profile together, just set the foreign model to the desired
$profile->author = $author;

// OR you can also just put in the id instead of the whole object here
// (means you don't need to load the author model upfront at all)
$profile->author = 23;

$profile->save();
```

You can of course do it the other way around, starting from the author model:

```
// create a new profile
$profile = new ProfileModel();
$profile->status_message = 'Hello World';
$profile->save();

// load a specific author and add that profile
$author = new \AuthorModel();
$author->load(['_id = ?', 2]);
$author->profile = $profile;
$author->save();
```

and to load it again:

```
$author->load(['_id = ?', 23]);
echo $author->profile->status_message; // Hello World

$profile->load(['_id = ?', 1]);
echo $profile->author->name; // Johnny English
```

## one-to-many, many-to-one

Save an author to a news record.

```
$author->load(['name = ?', 'Johnny English']);  
$news->load(['_id = ?', 42]);  
$news->author = $author; // set the object or the raw id  
$news->save();
```

now you can get:

```
echo $news->author->name; // 'Johnny English'
```

The field `author` now holds the whole mapper object of the AuthorModel. So you can also update, delete or cast it.

The getting all news by an author in the counterpart looks like this:

```
$author->load(['_id = ?', 42]);  
$author->news; // is now an array of NewsModel objects  
  
// if you like to cast them all you can use  
$allNewsByAuthorX = $author->castField('news'); // is now a multi-dimensional
```

## many-to-many, bidirectional

When both models of a relation has a `has-many` configuration on their linkage fields, Cortex create a new reference table in setup, where the foreign keys of both models are linked together. This way you can query model A for related models of B and vice versa.

To save many collections to a model you've got several ways:

```
$news->load(['_id = ?', 1]);  
  
// array of IDs from TagModel  
$news->tags = [12, 5];  
// OR a split-able string  
$news->tags = '12;5;3;9'; // delimiter: [,;|]  
// OR an array of single mapper objects  
$news->tags = [$tag, $tag2, $tag3];  
// OR a hydrated mapper that may contain multiple results  
$tag->load(['_id != ?', 42]);  
$news->tags = $tag;  
  
// you can also add a single tag to your existing tags  
$tag->load(['_id = ?', 23]);
```

```
$news->tags[] = $tag;  
  
$news->save();
```

Now you can get all tags of a news entry:

```
$news->load(['_id = ?', 1]);  
echo $news->tags[0]['title']; // Web Design  
echo $news->tags[1]['title']; // Responsive
```

And all news that are tagged with *Responsive*:

```
$tags->load(['title = ?', 'Responsive']);  
echo $tags->news[0]->title; // '10 Responsive Images Plugins'
```

This example shows the inverse way of querying (using the TagModel to find the corresponding news). Of course the can also use a more direct way that offers even more possibilities, therefore check the [has\(\)](#) method.

### many-to-many, unidirectional

You can use a `belongs-to-many` field config to define a one-way m:m relation. This is a special type for many-to-many as it will not use a 3rd table for reference and just puts a list of IDs into the table field, as commonly practiced in NoSQL solutions. This is an unidirectional binding, because the counterpart wont know anything about its relation and it's harder to query the reserve way, but it's still a lightweight and useful solution in some cases.

Saving works the same way like the other m:m type described above

```
$news->tags = [4, 7]; // IDs of TagModel  
$news->save();
```

and get them back:

```
$news->load(['_id = ?', 77]);  
echo $news->tags[0]->title; // Web Design  
echo $news->tags[1]->title; // Responsive
```

### many-to-many, self-referencing

In case you want to bind a many-to-many relation to itself, meaning that you'd like to add it to the own property of the same model, you can do this too, since these are detected as self-referenced fields now.

| Type | Model A  | Direction | Model A  |
|------|----------|-----------|----------|
| m:m  | has-many | <-->      | has-many |

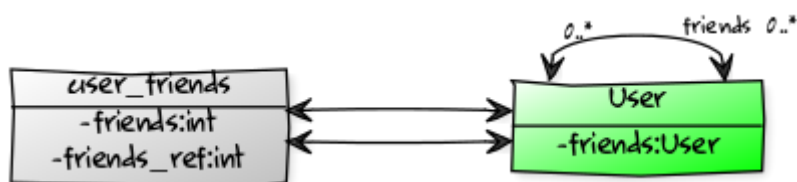
A common scenario is where a `User` has friends and that relation target is also `User`. So it would bind the relation to itself:

```
namespace Model;
class User extends \DB\Cortex {
    protected $fieldConf = [
        'friends' => [
            'has-many' => [\Model\User::class, 'friends']
        ]
    ];
}
```

To use a different field name in the pivot table for the reference field, use `selfRefField` option:

```
'friends' => [
    'has-many' => [\Model\User::class, 'friends',
        'selfRefField' => 'friends_ref'
    ]
]
```

Because this is also a many to many relation, a pivot table is needed too. Its name is generated based on the table and fields name, but can also be defined as 3rd array parameter, i.e. `['\Model\User', 'friends', 'user_friends']`.



Usually, this is a bidirectional relation, meaning that you would get a direct linkage to your friends ( `friends` ), and another one to the inverse linkage (friends with me, `friends_ref` ). As this is pretty inconvenient for further working and filtering on those, both fields are linked together internally and will always represent **all** relations, whether the relation was added from UserA or UserB.



```

$userA = new \Model\User();
$userA->load(['_id = ?', 1]);

$userB = new \Model\User();
$userB->load(['_id = ?', 2]);

$userC = new \Model\User();
$userC->load(['_id = ?', 3]);

if ($userA->friends)
    $userA[] = $userB;
else
    $userA = [$userB];

$userA->save();

$userC->friends = [$userA, $userB];
$userC->save();

```

The only exception is, that the current record itself is always excluded, so you wont get UserA as friend of UserA:

```

$userA->load(['_id = ?', 1]);
$userA->friends->getAll('_id'); // [2,3]
$userB->load(['_id = ?', 2]);
$userB->friends->getAll('_id'); // [1,3]
$userC->load(['_id = ?', 3]);
$userC->friends->getAll('_id'); // [1,2]

```

## Event Handlers

Cortex inherits all setters form the [Cursor Event Handlers](#) and additionally adds custom field handlers (setter/getter). These can be used to execute some extra code right before or after doing something. This could be useful for validation directly in your Model, or some extended save, load or delete cascades.

The following events are supported:

- `onload`
- `onset`
- `onget`
- `beforeerase`
- `aftererase`
- `beforeinsert`

- `afterinsert`
- `beforeupdate`
- `afterupdate`

You can setup own handlers to this events like this:

```
$mapper->onload(function($self){
    // custom code
});
// or
$mapper->onload('App/Foo/Bar::doSomething');
```

You can provide anything that is accepted by the `Base->call` method as handler function. Notice to use the `$self->set('field', 'val')` instead of `$self->field=val`, if you define a handler within a child class of Cortex (i.e. an extended `__construct` in your own model class).

If any `before*` event returns a `false` result, the action that is going to be performed will be aborted, and the `after*` events are skipped.

## Custom Field Handler

The `onset` and `onget` events have slightly different parameters:

```
$mapper->onset('field',function($self, $val){
    return md5($val);
});
```

You can also define these custom field preprocessors as a method within the class, named `set_*` or `get_*`, where `*` is the name of your field. In example:

```
class User extends \DB\Cortex {
    // [...]

    // validate email address
    public function set_mail($value) {
        if (\Audit::instance()->email($value) == false) {
            // no valid email address
            // throw exception or set an error var and display a flash message
            $value = null;
        }
        return $value;
    }

    // hash a password before saving
    public function set_password($value) {
```

```
        return \Bcrypt::instance()->hash($value);
    }
    public function get_name($value) {
        return ucfirst($value);
    }
}
```

So setting these fields in your Model, like:

```
$user->password = 'secret';
$user->mail = 'foo@bar.com';
```

will now trigger your custom setters, doing anything you like.

## Filter Query Syntax

---

Well basically the `$filter` syntax for writing cortex queries is simple SQL. But there are some slightly modifications you should have read in these additional notes.

### Operators

These common filter operators are supported:

- relational operators: `<`, `>`, `<=`, `>=`, `==`, `=`, `!=`, `<>`
- search operators: `LIKE`, `NOT LIKE`, `IN`, `NOT IN` (not case-sensitive)
- logical operators: `( , )`, `AND`, `OR`, `&&`, `( ||` only mysql and jig)

### Comparison

With comparison operators, you can do the following things:

- compare fields against other fields:

```
['foo < bar']
```

- compare fields against values:

```
['foo >= 1'] or ['foo == \'bar\']
```

Especially for value comparison, it's **highly recommended** to use placeholders in your filter and bind their values accordingly. This ensures that the data mapper uses parameterized queries for better security. Placeholders go like this:

- positional bind-parameters:

```
['foo = ?', 1] or ['foo = ? AND bar < ?', 'baz', 7]
```

- named bind-parameters:

```
['foo = :foo', ':foo'=>1]
```

```
['foo = :foo AND bar < :bar', ':foo'=>'hallo', ':bar'=>7]
```

## Sugar

- what's a special sugar in Cortex is, that you can also mix both types together:

```
['foo = ? AND bar < :bar', 'bar', ':bar'=>7]
```

- and you can also reuse named parameter (not possible in raw PDO):

```
['min > :num AND max < :num', ':num' => 7]
```

- comparison with `NULL` (nullable fields) works this easy:

```
['foo = ?', NULL] or ['foo != ?', NULL]
```

## Search

- The `LIKE` operator works the same way like the [F3 SQL search syntax](#). The search wildcard ( `%` ) belongs into the bind value, not the query string.

```
['title LIKE ?', '%castle%'] or ['email NOT LIKE ?', '%gmail.com']
```

- The `IN` operator usually needs multiple placeholders in raw PDO (like `foo IN (?, ?, ?)`). In Cortex queries you simply use an array for this, the QueryParser does the rest.

```
['foo IN ?', [1,2,3]]
```

You can also use a `CortexCollection` as bind parameter. In that case, the primary keys are automatically used for matching:

```
$fruits = $fruitModel->find(['taste = ?', 'sweet']);
$result = $userModel->find(['favorite_fruit IN ?', $fruits])
```

## Options

The `$options` array for load operations respects the following keys:

- order
- limit

- offset

Use `DESC` and `ASC` flags for sorting fields, just like in SQL. Additional `group` settings are currently just bypassed to the underlying mapper and should work dependant on the selected db engine. Any unification on that might be handled in a future version.

## Relational sorting

NB: This is currently experimental as of v1.7

For 1-n relations, you can apply a sorting rule, based on a field of a relation to your order option. You need to prefix the field you used in your `$fieldConf` for the relation with an `@` in your order definition:

Given the following field configuration:

```
// Contracts fieldConf:
'user' => ['belongs-to-one' => UserModel::class]

// User fieldConf:
'contracts' => ['has-many' => [ContractsModel::class, 'user']]
```

This example will paginate through all contracts records that are sorted by the relational user name:

```
$contracts = new Contracts();
$res = $contracts->paginate(0,10,null, ['order'=>'@user.name ASC']);
```

## Advanced Filter Techniques

When your application reaches the point where all basic CRUD operations are working, you probably need some more control about finding your records based on conditions for relations. Here comes the `has()` and `filter()` methods into play:

### has

The `has` method adds some conditions to a related field, that must be fulfilled in addition, when the `next` `find()` or `load()` method of its parent is fired. So this is meant for limiting the main results.

In other words: Let's find all news records that are tagged by "Responsive".

```
$news->has('tags', ['title = ?', 'Responsive']);
$results = $news->find();
```

```
echo $results[0]->title; // '10 Responsive Images Plugins'
```

Of course you can also use the inverse way of querying, using the `TagModel`, load them by title and access the shared `$tags->news` property to find your records. The advantage of the "has" method is that you can also add a condition to the parent as well. This way you could edit the load line into something like this: `$news->find(['published = ?', 1]);`. Now you can limit your results based on two different models - you only load *published* news which were tagged "Responsive".

You can also add multiple has-conditions to different relations:

```
$news->has('tags', ['title = ?', 'Responsive']);
$news->has('author', ['username = ?', 'ikkez']);
$results = $news->find(['published = ?', 1], ['limit'=>3, 'order'=>'date DESC']
```

Now you only load the last 3 published news written by me, which were tagged "Responsive", sorted by release date. ;)

If you like, you can also call them in a fluent style: `$news->has(...)->load(...);`.

## orHas

Similar to has method, but adds the has condition with an OR operator.

## filter

The filter method is meant for limiting the results of relations. In example: load author x and only his news from 2014.

```
$author->filter('news', ['date > ?', '2014-01-01']);
$author->load(['username = ?', 'ikkez']);
```

The same way like the `has()` method does, you can add multiple filter conditions. You can mix filter and has conditions too. Once a `load` or `find` function is executed, the filter (and has) conditions are cleared for the next upcoming query.

Filter conditions are currently not inherited. That means if you recursively access the fields of a relation (`$author->news[0]->author->news`) they get not filtered, but fully lazy loaded again.

## Propagation

It is also possible to filter deep nested relations using the `.` dot style syntax. The following example finds all authors and only loads its news that are tagged with "Responsive":

```
$author->filter('news.tags', ['title = ?', 'Responsive']);  
$author->find();
```

The same applies for the `has` filter. The next example is similar to the previous one, but this time, instead of finding all authors, it only returns authors that have written a news entry that was tagged with "Responsive":

```
$author->has('news.tags', ['title = ?', 'Responsive']);  
$author->find();
```

**Notice:** These nested filter techniques are still experimental, so please handle with care and test your application well.

## Insight into aggregation

---

Cortex comes with some handy shortcuts that could be used for essential field aggregation.

### Counting relations

Sometimes you need to know how many relations a record has - i.e. for some stats or sorting for top 10 list views.

Therefore have a look at the `countRel` method, which let you setup a new adhoc field to the resulting records that counts the related records on `has-many` fields.

```
// find all tags with the sum of all news that uses the tag, ordered by the to  
$tag = new \Model\Tag();  
$tag->countRel('news');  
$result = $tag->find(null, ['order'=>'count_news DESC, title']);
```

The new field is named like `count_{key}`, but you can also set a custom alias. As you can see, you can also use that field for additional sorting of your results. You can also combine this with the `has()` and `filter()` methods and set relation counters to nested relations with the `.` separator. Notice that `countRel()` only applies to the next called `find()` operation. Currently, you cannot use those virtual count field in a `$filter` query.

## Virtual fields

Cortex has some abilities for own custom virtual fields. These might be useful to add additional fields that may contain data that is not stored in the real db table or computes its value out of other fields or functions, similar to the [custom field setters and getters](#).

```
// just set a simple value
$user->virtual('is_online', TRUE);
// or use a callback function
$user->virtual('full_name', function($this) {
    return $this->name.' '.$this->surname;
});
```

You can also use this to count or sum fields together and even reorder your collection on this fields using `$collection->orderBy('foo DESC, bar ASC')`. Keep in mind that these virtual fields only applies to your final received collection - you cannot use these fields in your filter query or sort condition before the actual find.

But if you use a SQL engine, you can use the underlying mapper abilities of virtual adhoc fields - just set this before any load or find operation is made:

```
$mapper->newField = 'SQL EXPRESSION';
```

## Mapper API

---

### \$db

#### DB object

Can be an object of `\DB\SQL`, `\DB\Jig` or `\DB\Mongo`, OR a string containing a HIVE key where the actual database object is stored at.

### \$table

#### table to work with, string

If the table is not set, Cortex will use the `strtolower(get_class($this))` as table name.

### \$fluid

#### trigger SQL fluid schema mode, boolean = false



## \$fieldConf

field configuration, array

The array scheme is:

```
protected $fieldConf = [  
    'fieldName' => [  
        'type' => string  
        'nullable' => bool  
        'default' => mixed  
        'index' => bool  
        'unique' => bool  
    ]  
]
```

Get the whole list of possible types from the [Data Types Table](#).

*NB:* You can also add `'passThrough' => true` in order to use the raw value in *type* as data type in case you need a custom type which is not available in the data types table.

## \$ttl

default mapper schema ttl, int = 60

This only affects the schema caching of the SQL mapper.

## \$rel\_ttl

default mapper rel ttl, int = 0

This setting in your model will add a caching to all relational queries

## \$primary

SQL table primary key, string

Defines the used primary key of the table. Default is `id` for SQL engine, and *always* `_id` for JIG and Mongo engines. The setup method respects this value for creating new SQL tables in your database and has to be an integer column.

## load

Retrieve first object that satisfies criteria

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

Simple sample to load a user:

```
$user->load(['username = ?', 'jacky']);
if (!$user->dry()) {
    // user was found and loaded
    echo $user->username;
} else {
    // user was not found
}
```

When called without any parameter, it loads the first record from the database. The method returns `TRUE` if the load action was successful.

## loaded

Count records that are currently loaded

```
int loaded()
```

Sample:

```
$user->load(['last_name = ?', 'Johnson']);
echo $user->loaded(); // 3
```

## first, last, next, prev, skip

Methods to navigate the cursor position and map a record

See <http://fatfreeframework.com/cursor#CursorMethods>.

```
$user->load(['last_name = ?', 'Johnson']);
echo $user->loaded(); // 3
echo $user->_id; // 1
$user->last();
echo $user->_id; // 3
echo $user->prev()->_id; // 2
echo $user->first()->_id; // 1
echo $user->skip(2)->_id; // 3
```

## cast

## Return fields of mapper object as an associative array

```
array cast ([ Cortex $obj = NULL [, int $rel_depths = 1]])
```

### Field masks

NB: Since configuring *relations depths* seems more and more less practical, a new way of casting relations was introduced: "Field masks". This is the way to go and will replace the legacy "relations depths configuration" in a future release.

You can also use `$rel_depths` for defining a mask to mappers, so you can restrict the fields returned from a cast:

```
$data = $item->cast(null, [
    '_id',
    'order.number',
    'product._id',
    'product.title',
    'product.features._id',
    'product.features.title',
    'product.features.icon',
]);
```

### relation depths (old way)

A simple cast sample. If the model contains relations, they are also casted for 1 level depth by default:

```
$user->load(['_id = ?', 3]);
var_dump($user->cast());
/* Array (
    [_id] => 3
    [first_name] => Steve
    [last_name] => Johnson
    [comments] => Array(
        [1] => Array (
            [_id] => 23
            [post] => 2
            [message] => Foo Bar
        ),
        [2] => Array (
            [_id] => 28
            [post] => 3
            [message] => Lorem Ipsum
        )
    )
)*/
```

If you increase the `$rel_depths` value, you can also resolve further relations down the road:

```
var_dump($user->cast(NULL, 2));
/* Array (
    ...
    [comments] => Array(
        [1] => Array (
            [_id] = 23
            [post] => Array(
                [_id] => 2
                [title] => Kittenz
                [text] => ...
            )
            [message] => Foo Bar
        ),
        ...
    )
)*/
```

## relation depths configuration

If you only want particular relation fields to be resolved, you can set an array to the `$rel_depths` parameter, with the following schema:

```
$user->cast(NULL, [
    '*' => 0,      // cast all own relations to the given depth,
                  // 0 doesn't cast any relation (default if this key is missing)
    'modelA' => 0, // if a relation key is defined here, modelA is being loaded a
                  // but not its own relations, because the depth is 0 for it
    'modelB' => 1, // modelB and all its 1st level relations are loaded and caste
    'modelC' => [...], // you can recursively extend this cast array scheme
]);

// simple sample: only cast yourself and the author model without its childs
$news->cast(NULL, [
    '*'=>0,
    'author'=>0
]);

// nested sample: only cast yourself,
// your own author relation with its profile and all profile relations
$news->cast(NULL, [
    '*'=>0,
    'author'=>[
        '*'=>0,
        'profile'=>1
    ]
]);
```

```
]
]);
```

If you don't want any relation to be resolved and casted, just set `$rel_depths` to `0`. Any one-to-many relation field then just contains the `_id` (or any other custom field binding from `$fieldConf`) of the foreign record, many-to-one and many-to-many fields are just empty.

## castField

### Cast a related collection of mappers

```
array|null castField( string $key [, int $rel_depths = 0 ])
```

## find

### Return a collection of objects matching criteria

```
CortexCollection|false find([ array $filter = NULL [, array $options = NULL [,
```

The resulting `CortexCollection` implements the `ArrayIterator` and can be treated like a usual array. All filters and counters which were set before are used once `find` is called:

```
// find published #web-design news, sorted by approved user comments
$news->has('tags', ['slug = ?', 'web-design']);
$news->filter('comments', ['approved = ?', 1]);
$news->countRel('comments');
$records = $news->find(
    ['publish_date <= ? and published = ?', date('Y-m-d'), true],
    ['order' => 'count_comments desc']
);
```

## findByRawSQL

### Use a raw SQL query to find results and factory them into models

```
CortexCollection findByRawSQL( string|array $query [, array $args = NULL [, in
```

In case you want to write your own SQL query and factory the results into the appropriate model, you can use this method. I.e.:

```
$news_records = $news->findByRawSQL('SELECT * from news where foo <= ? and act
```

## findone

Return first record (mapper object) that matches criteria

```
Cortex|false findone([ array $filter = NULL [, array $options = NULL [, int $t
```

This method is inherited from the [Cursor](#) class.

## afind

Return an array of result arrays matching criteria

```
array|null find([ array $filter = NULL [, array $options = NULL [, int $ttl =
```

Finds a whole collection, matching the criteria and casts all mappers into an array, based on the `$rel_depths` configuration.

## addToCollection

Give this model a reference to the collection it is part of

```
null addToCollection( CortexCollection $cx )
```

**onload, aftererase, afterinsert, aftersave, afterupdate, beforeerase, beforeinsert, beforesave, beforeupdate**

Define an event trigger

```
callback onload( callback $func )
```

See the guide about [Event Handlers](#) for more details.

## onget, onset

## Define a custom field getter/setter

```
callback onget( string $field, callback $func )
```

See the guide about [Custom Field Handler](#) for more details.

## clear

### Clear any mapper field or relation

```
null clear( string $key )
```

## cleared

### Returns whether the field was cleared or not

```
mixed initial( string $key )
```

If the field initially had data, but the data was cleared from the field, it returns that old cleared data. If no initial data was present or the field has not changed (cleared)

`FALSE` is returned.

## clearFilter

### Removes one or all relation filter

```
null clearFilter([ string $key = null ])
```

Removes only the given `$key` filter or all, if none was given.

## compare

### Compare new data against existing initial values of certain fields

```
null compare( array $fields, callback $new [, callback $old = null ])
```

This method compares new data in form of an assoc array of [field => value] against the initial field values and calls a callback functions for *\$new* and *\$old* values, which can be used to prepare new / cleanup old data.

Updated fields are set, the *\$new* callback MUST return a value.

```

$uploads=[
    'profile_image' => 'temp_uploads/thats_me.jpg',
    'pictures' => ['7bbn4ksw8m5', 'temp_uploads/random_pic.jpg']
];
$this->model->compare($uploads,function($filepath) {
    // new files
    return $this->handleFileUpload($filepath);
}, function($fileId){
    // old files
    $this->deleteFile($fileId);
});

```

In the example above, we handle multiple fields and compare their values with an incoming array for new data. For each new field value or changed / added array item value, the `$new` function is called. For existing data, that's not present in the new data anymore, the `$old` function is called.

## copyfrom

Hydrate the mapper from hive key or given array

```

null copyfrom( string|array $key [, callback|array|string $fields = null ])

```

Use this method to set multiple values to the mapper at once. The `$key` parameter must be an array or a string of a hive key, where the actual array can be found.

The `$fields` parameter can be a splittable string:

```

$news->copyfrom('POST','title;text');

```

Or an array:

```

$news->copyfrom('POST',['title','text']);

```

Or a callback function, which is used to filter the input array:

```

$news->copyfrom('POST',function($fields) {
    return array_intersect_key($fields,array_flip(['title','text']));
});

```

## copyto

Copy mapper values into hive key



```
null copyto( string $key [, array|string $relDepth = 0 ] )
```

## copyto\_flat

Copy mapper values to hive key with relations being simple arrays of keys

```
null copyto_flat( string $key )
```

All `has-many` relations are being returned as simple array lists of their primary keys.

## count

Count records that match criteria

```
null count([ array $filter [, array $options = NULL [, int $ttl = 60 ]]])
```

Just like `find()` but it only executes a count query instead of the real select.

## countRel

add a virtual field that counts occurring relations

```
null countRel( string $key [, string $alias [, array $filter [, array $option]
```

The `$key` parameter must be an existing relation field name. This adds a virtual counter field to your result, which contains the count/sum of the matching relations to the current record, which is named `count_{$key}`, unless you define a custom `$alias` for it.

It's also possible to define a `$filter` and `$options` to the query that's used for counting the relations.

You can also use this counter for sorting, like in this tag-cloud sample:

```
$tags = new \Model\Tag();
$tags->filter('news',['published = ? and publish_date <= ?', true, date('Y-m-d
$tags->countRel('news');
$result = $tags->find(['deleted = ?',0], ['order'=>'count_news desc']);
```

This method also supports propagation, so you can define counters on nested relations pretty straightforward:

```
// fetch all posts, with comments and count its likes (reactions of type "like")
$post->countRel('comments.reaction', 'count_likes', ['type = ?', 'like']);
$results = $post->find();
```

## dbtype

Returns the currently used db type

```
string dbtype()
```

The type is `SQL`, `Mongo` or `Jig`.

## defaults

Return default values from schema configuration

```
array defaults([ bool $set = FALSE ])
```

Returns a `$key => $value` array of fields that has a default value different than `NULL`.

## dry

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

```
bool dry()
```

Sample:

```
$mapper->load(['_id = ?', '234']);
if ($mapper->dry()) {
    // not found
} else {
    // record was loaded
}
```

## erase

## Delete object/s and reset ORM

```
null erase([ array $filter = null ])
```

When a `$filter` parameter is set, it deletes all matching records:

```
$user->erase(['deleted = ?', 1]);
```

It deletes the loaded record when called on a hydrated mapper without `$filter` parameter:

```
$user->load(['_id = ?', 6]);  
$user->erase();
```

This also calls the `beforeerase` and `aftererase` events.

## exists

Check if a certain field exists in the mapper or is a virtual relation field

```
bool exists( string $key [, bool $relField = false ])
```

If `$relField` is true, it also checks the `$fieldConf` for defined relational fields.

## fields

get fields or set whitelist / blacklist of fields

```
array fields([ array $fields = [] [, bool $exclude = false ])
```

When you call this method without any parameter, it returns a list of available fields from the schema.

```
var_dump( $user->fields() );  
/* Array(  
    '_id'  
    'username'  
    'password'  
    'email'  
    'active'  
    'deleted'  
)*/
```

If you set a `$fields` array, it'll enable the field whitelisting, and put the given fields to that whitelist. All non-whitelisted fields on loaded records are not available, visible nor accessible anymore. This is useful when you don't want certain fields in a returned casted array.

```
$user->fields(['username','email']); // only those fields
$user->load();
var_dump($user->cast());
/* Array(
    '_id' => 5
    'username' => joe358
    'email' => joe@domain.com
)*/
```

Calling this method will re-initialize the mapper and takes effect on any further load or find action, so run this first of all.

If you set the `$exclude` parameter to `true`, it'll also enable the whitelisting, but set all available fields, without the given, to the whitelist. In other words, the given `$fields` become blacklisted, the only the remaining fields stay visible.

```
$user->fields(['email'], true); // all fields, but not these
$user->load();
var_dump($user->cast());
/* Array(
    '_id' => 5
    'username' => joe358
    'password' => $18m$fsk555a3f2f08ff28
    'active' => 1
    'deleted' => 0
)*/
```

In case you have relational fields configured on the model, you can also prohibit access for the fields of that relations. For that use the dot-notation:

```
$comments->fields(['user.password'], true); // exclude the password field in u
$comments->load();
var_dump($comments->cast());
/* Array(
    '_id' => 53
    'message' => ....
    'user' => Array(
        '_id' => 5
        'username' => joe358
        'active' => 1
        'deleted' => 0
    )
)
```

```
)
)* /
```

You can call this method multiple times in conjunction. It'll always merge with your previously set white and blacklisted fields. `_id` is always present.

## filter

### Add filter for loading related models

```
Cortex filter( string $key [, array $filter = null [, array $options = null ]]
```

See [Advanced Filter Techniques](#).

## get

### Retrieve contents of key

```
mixed get( string $key [, bool $raw = false ])
```

If `$raw` is `true`, it'll return the raw data from a field as is. No further processing, no relation is resolved, no get-event fired. Useful if you only want the raw foreign key value of a relational field.

## getRaw

### Retrieve raw contents of key

```
mixed getRaw( string $key )
```

This is a shortcut method to `$mapper->get( 'key', TRUE )`.

## getFieldConfiguration

### Returns model \$fieldConf array

```
array|null getFieldConfiguration()
```

## getTable

### returns model table name

```
string getTable()
```

If no table was defined, it uses the current class name to lowercase as table name.

## has

### Add has-conditional filter to next find call

```
Cortex has( string $key [, array $filter = null [, array $options = null ]])
```

See [Advanced Filter Techniques](#).

## orHas

### Add has-conditional filter with OR operator to previous condition

Same as has filter, but chains with a logical OR to the previous condition.

## initial

### Return initial field value

```
mixed initial( string $key )
```

Returns the initial data from a field, like it was fetched from the database, even if the field as changed afterwards. Array fields are decoded / unserialized properly before it's returned.

## mergeFilter

### Glue multiple filter arrays together into one

```
array mergeFilter( array $filters [, string $glue = 'and' ])
```

This is useful when you want to add more conditions to your filter array or want to merge multiple filter arrays together, i.e. when you assemble the filter for a complex search functionality which is based on conditions. Use the `$glue` parameter to define the part that is used to merge two filters together (usually `AND` or `OR` ).

```
$filter1 = ['_id = ?', 999];  
$filter2 = ['published = ? or active = ?', true, false];
```

```
$new_filter = $mapper->mergeFilter([$filter1, $filter2]);
// array('(_id = ?) and (published = ? or active = ?)', 999, true, false)
```

## paginate

Return array containing subset of records matching criteria

```
array paginate([ int $pos = 0 [, int $size = 10 [, array $filter = NULL [, arr
```

See [Cursor->paginate](#). Any *has* and *filter* filters can be used in conjunction with `paginate` as well.

## rel

returns a clean/dry model from a relation

```
Cortex rel( string $key )
```

For instance, if `comments` is a one-to-many relation to `\Model\Comment` :

```
$user->load();
var_dump($user->comments); // array of comments
$new_comment = $user->rel('comments');
// $new_comment is a new empty \Model\Comment
```

## reset

reset and re-initialize the mapper

```
null reset([ bool $mapper = true ])
```

If `$mapper` is *false*, it only reset filter, default values and internal caches of the mapper, but leaves the hydrates record untouched.

## resetFields

reset only specific fields and return to their default values

```
null resetFields( array $fields )
```

If any field doesn't have a default value, it's reset to `NULL`.

## resolveConfiguration

### kick start mapper to fetch its config

```
array resolveConfiguration()
```

Returns an array that exposes a mapper configuration. The array includes:

- table
- fieldConf
- db
- fluid
- primary

## save

### Save mapped record

It is recommended to always use the save method. It'll automatically see if you want to save a new record or update an existing, loaded record.

```
$user->username = 'admin'  
$user->email = 'admin@domain.com';  
$user->save(); // insert  
  
$user->reset();  
$user->load(['username = ?', 'admin']);  
$user->email = 'webmaster@domain.com';  
$user->save(); // update
```

The save method also fires the `beforeinsert`, `beforeupdate`, `afterinsert` and `afterupdate` events. There are also `insert` and `update` method, but using that methods directly, will skip the events and any cascading actions.

## set

### Bind value to key

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

## setdown

### erase all model data, handle with care



```

null setdown([ object|string $db = null [, string $table = null ]])

```

This method completely drops the own table, and used many-to-many pivot-tables from the database.

## setFieldConfiguration

### set model definition

```

null setFieldConfiguration( array $config )

```

Used to set the **\$fieldConf** array.

## setup

### setup / update table schema

```

bool setup([ object|string $db = null [, string $table = null [, array $fields

```

This method creates the needed tables for the model itself and additionally required pivot tables. It uses the internal model properties *\$db*, *\$table* and *fieldConf*, but can also be fed with method parameters which would take precedence.

## touch

### update a given date or time field with the current time

```

null touch( string $key [, int $timestamp = NULL ])

```

If **\$key** is a defined field in the *\$fieldConf* array, and is a type of date, datetime or timestamp, this method updates the field to the current time/date in the appropriate format.

If a **\$timestamp** is given, that value is used instead of the current time.

## valid

Return whether current iterator position is valid.

```

bool valid()

```

It's the counterpart to [dry\(\)](#).

```
$mapper->load(['_id = ?', '234']);  
if ($mapper->valid()) {  
    // record was loaded  
} else {  
    // not found  
}
```

## virtual

### virtual mapper field setter

```
null virtual( string $key, mixed $val )
```

This sets a custom virtual field to the mapper. Useful for some on-demand operations:

```
$user->virtual('pw_unsecure', function($this) {  
    return \Bcrypt::instance()->needs_rehash($this->password, 10);  
});
```

It is possible to use the virtual fields for a post-sorting on a selected collection, see [virtual fields](#).

## Collection API

---

Whenever you use the `find` method, it will return an instance of the new `CortexCollection` class. This way we are able determine the whole collection from the inside of every single mapper in the results, and that gives us some more advanced features, like the [smart-loading of relations](#). The `CortexCollection` implements the `ArrayIterator` interface, so it is accessible like an usual array. Here are some of the most useful methods the Cortex Collection offers:

### add

#### add single model to collection

```
null add( Cortex $model )
```

It's also possible to use the array notation to add models:

```
$news->load();
$new_comment = $news->rel('comments');
$new_comment->text = 'Foo Bar';
$news->comments[] = $new_comment;
$news->save();
```

## castAll

### cast all contained mappers to a nested array

```
array castAll([ $reldepths = 1 ])
```

Similar to the `cortex->cast` method for a single mapper, this automatically casts all containing mappers to a simple nested array.

```
$result = $news->find(['published = ?', true]);
if ($result)
    $json = json_encode($result->castAll());
```

Use the `$reldepths` parameter to define what to cast, see [cast](#) method for details.

## compare

### compare collection with a given ID stack

```
array compare( array|CortexCollection $stack [, string $cpm_key = '_id'])
```

This method is useful to compare the current collection with another collection or a list of values that is checked for existence in the collection records.

In example you got a relation collection that is about to be updated and you want to know which records are going to be removed or would be new in the collection:

```
$res = $user->friends->compare($newFriendIds);
if (isset($res['old'])) {
    // removed friends
}
if (isset($res['new'])) {
    // added friends
    foreach($res['new'] as $userId) {
        // do something with $userId
    }
}
```

The compare result `$res` is an array that can contain the array keys `old` and `new`, which both represent an array of `$cpm_key` values.

NB: This is just a comparison - it actually does not update any of the collections. Add a simple `$user->friends = $newFriendIds;` after comparison to update the collection.

## contains

check if the collection contains a record with the given key-val set

```
bool contains( mixed|Cortex $val [, string $key = '_id' ])
```

This method can come handy to check if a collections contains a given record, or has a record with a given value:

```
if ($user->friends && $user->friends->contains($newFriend)) {
    $f3->error(400, 'This user is already your friend');
    return;
}
```

With custom compare key:

```
if ($user->blocked_users->contains($currentUserId, 'target')) {
    // this user has blocked you
}
```

## expose

return internal array representation

```
array expose()
```

## factory

create a new collection instance from given records

```
CortexCollection factory( array $records )
```

`$records` must be an array, containing Cortex mapper objects.

## getAll

## returns all values of a specified property from all models

```
array getAll( string $prop [, bool $raw = false ])
```

You can fetch all values of a certain key from all containing mappers using `getAll()`. Set the 2nd argument to `true` to get only the raw DB results instead of resolved mappers on fields that are configured as a relation.

```
$users = $user->find(['active = ?',1]);  
$mails = $users->getAll('email');  
/* Array(  
    'user1@domain.com',  
    'user2@domain.com',  
    'user3@domain.com'  
)*/
```

## getBy

```
array getBy( string $index [, bool $nested = false ])
```

You can transpose the results by a defined key using `getBy()`. Therefore you need to provide an existing field in the mapper, like this;

```
$pages = $page->find();  
$pages_by_slug = $pages->getBy('slug');
```

This will resort the resulting array by the email field of each mapper, which gives you a result array like `array("foo@domain.com"=>array(...))`. If you provide `true` as 2nd argument, the records are ordered into another array depth, to keep track of multiple results per key.

## hasChanged

returns true if any model was modified after it was added to the collection

```
bool hasChanged()
```

## orderBy

re-assort the current collection using a sql-like syntax

```
null orderBy( string $cond )
```

If you need to re-sort a result collection once more to another key, use this method like `$results->orderBy( 'name DESC' );`. This also works with multiple sort keys.

## setModels

### set a collection of models

```
array setModels( array $models [, bool $init = true ])
```

This adds multiple Cortex objects to the own collection. When `$init` is `true`, added models with this method won't effect the **changed** state.

## slice

### slice the collection

```
null slice( int $offset [, int $limit = null ])
```

This removes a part from the collection.

## Additional notes

- To release any relation, just set the field to `NULL` and save the mapper.
- All relations are lazy loaded to save performance. That means they won't get loaded until you access them by the linked property or cast the whole parent model.
- lazy loading within a result collection will **automatically** invoke the eager loading of that property **to the whole set**. The results are saved to an [Identity Map](#) to relieve the strain on further calls. I called this *smart loading* and is used to get around the [1+N query problem](#) with no need for extra configuration.
- If you need to use a primary key in SQL that is different from `id` (for any legacy reason), you can use the `$primary` class property to set it to something else. You should use the new custom pkey in your queries now. Doing so will limit your app to SQL engines.
- to get the id of any record use `$user->_id;`. This even works if you have setup a custom primary key.

- To find any record by its **id** use the field `_id` in your filter array, like `['_id = ?', 123]`.
- primary fields should not be included in the `$fieldConf` array. They could interfere with the [setup](#) routine.
- There are some little behaviours of Cortex you can control by these hive keys:
  - `CORTEX.queryParserCache` : if `TRUE` all query strings are going to be cached too (may add a lot of cache entries). Default: `FALSE`
  - `CORTEX.smartLoading` : triggers the intelligent-lazy-eager-loading. Default is `TRUE`, but turn it off if you think something works wrong. Could cause a lot of extra queries send to your DB, if deactivated.
  - `CORTEX.standardiseID` : Default `TRUE`. This moves any defined primary key into the `_id` field on returned arrays.
  - `CORTEX.quoteConditions` : Default `TRUE`. By default, all field names in where conditions are quoted automatically according to the used database engine. This helps to work around reserved names in SQL. However the detection of fields isn't perfect yet, so in case you want to add the correct backticks or other quotation yourself, set this to `FALSE`.

## Known Issues

---

- Not really a bug, but returned collections (from relations, *find*, or *paginate* method) are not cloneable because they need to keep a unique references to the identity map of its relations. This leads to the point that all containing mappers are not automatically escaped in templates, regardless of the `ESCAPE` setting. Keep in mind to add the `| esc` filter to your tokens.

If you find any issues or bugs, please file a [new Issue](#) on github or write a mail. Thanks.

## Roadmap

---

If you have any ideas, suggestions or improvements, feel free to add an issue for this on github.

Cortex currently only reflects to the most common use cases. If you need more extensive control over your queries or the DB, you may consider to use the underlying mapper or DB directly. This could be done in custom methods or field preprocessors in your Model classes.

Anyways, I hope you find this useful. If you like this plugin, why not make a donation?



If you like to see Cortex in action, have a look at [fabulog](#).

## License

---

GPLv3

---

### Releases 11



+ 10 releases

---

### Packages

No packages published

---

### Contributors 6



---

### Languages

---

● PHP 100.0%