codeigniter-base-model

My CodeIgniter Base Model is an extended CI_Model class to use in your CodeIgniter applications. It provides a full CRUD base to make developing database interactions easier and quicker, as well as an event-based observer system, in-model data validation, intelligent table name guessing and soft delete.

Synopsis

```php class Post_model extends MY_Model { }

$this->load->model('post_model', 'post');

$this->post->get_all();

$this->post->get(1); $this->post->get_by('title', 'Pigs CAN Fly!'); $this->post->get_many_by('status', 'open');

$this->post->insert(array( 'status' => 'open', 'title' => "I'm too sexy for my shirt" ));

$this->post->update(1, array( 'status' => 'closed' ));

$this->post->delete(1); ```

## Installation/Usage

Download and drag the MY_Model.php file into your *application/core* folder. CodeIgniter will load and initialise this class automatically for you.

Extend your model classes from `MY_Model` and all the functionality will be baked in automatically.

## Naming Conventions

This class will try to guess the name of the table to use, by finding the plural of the class name.

For instance:

```
class Post_model extends MY_Model { }
```

...will guess a table name of `posts`. It also works with `_m`:

```
class Book_m extends MY_Model { }
```

...will guess `books`.
```

If you need to set it to something else, you can declare the *$_table* instance variable and set it to the table name:

```
class Post_model extends MY_Model
{
    public $_table = 'blogposts';
}
```

Some of the CRUD functions also assume that your primary key ID column is called '*id*'. You can overwrite this functionality by setting the *$primary_key* instance variable:

```
class Post_model extends MY_Model
{
    public $primary_key = 'post_id';
}
```

Callbacks/Observers

There are many times when you'll need to alter your model data before it's inserted or returned. This could be adding timestamps, pulling in relationships or deleting dependent rows. The MVC pattern states that these sorts of operations need to go in the model. In order to facilitate this, **MY_Model** contains a series of callbacks/observers -- methods that will be called at certain points.

The full list of observers are as follows:

- $before_create
- $after_create
- $before_update
- $after_update
- $before_get
- $after_get
- $before_delete
- $after_delete

These are instance variables usually defined at the class level. They are arrays of methods on this class to be called at certain points. An example:

```php class Book_model extends MY_Model { public $before_create = array( 'timestamps' );

```
protected function timestamps($book)
{
    $book['created_at'] = $book['updated_at'] = date('Y-m-d H:i:s');
    return $book;
}
```
```

}```

Remember to always always always return the `$row` object you're passed. Each observer overwrites its predecesor's data, sequentially, in the order they're defined.

Observers can also take parameters in their name, much like CodeIgniter's Form Validation library. Parameters are then accessed in `$this->callback_parameters`:

```
public $before_create = array( 'data_process(name)' );
public $before_update = array( 'data_process(date)' );

protected function data_process($row)
{
    $row[$this->callback_parameters[0]] = $this->_process($row[$this-
>callback_parameters[0]]);

    return $row;
}
```

Validation

MY_Model uses CodeIgniter's built in form validation to validate data on insert.

You can enable validation by setting the `$validate` instance to the usual form validation library rules array:

```
class User_model extends MY_Model
{
    public $validate = array(
        array( 'field' => 'email',
               'label' => 'email',
               'rules' => 'required|valid_email|is_unique[users.email]' ),
        array( 'field' => 'password',
               'label' => 'password',
               'rules' => 'required' ),
        array( 'field' => 'password_confirmation',
               'label' => 'confirm password',
               'rules' => 'required|matches[password]' ),
    );
}
```

Anything valid in the form validation library can be used here. To find out more about the rules array, please view the library's documentation.

With this array set, each call to `insert()` or `update()` will validate the data before allowing the query to be run. **Unlike the CodeIgniter validation library, this won't validate the POST data, rather, it validates the data passed directly through.**

You can skip the validation with `skip_validation()`:

```
$this->user_model->skip_validation();
$this->user_model->insert(array( 'email' => 'blah' ));
```

Alternatively, pass through a `TRUE` to `insert()`:

```
$this->user_model->insert(array( 'email' => 'blah' ), TRUE);
```

Under the hood, this calls `validate()`.

Protected Attributes

If you're lazy like me, you'll be grabbing the data from the form and throwing it straight into the model. While some of the pitfalls of this can be avoided with validation, it's a very dangerous way of entering data; any attribute on the model (any column in the table) could be modified, including the ID.

To prevent this from happening, MY_Model supports protected attributes. These are columns of data that cannot be modified.

You can set protected attributes with the `$protected_attributes` array:

```
class Post_model extends MY_Model
{
    public $protected_attributes = array( 'id', 'hash' );
}
```

Now, when `insert` or `update` is called, the attributes will automatically be removed from the array, and, thus, protected:

```
$this->post_model->insert(array(
    'id' => 2,
    'hash' => 'aqe3fwrga23fw243fWE',
    'title' => 'A new post'
));

// SQL: INSERT INTO posts (title) VALUES ('A new post')
```

Relationships

MY_Model now has support for basic *belongs_to* and has_many relationships. These relationships are easy to define:

```
class Post_model extends MY_Model
{
    public $belongs_to = array( 'author' );
    public $has_many = array( 'comments' );
}
```

It will assume that a MY_Model API-compatible model with the singular relationship's name has been defined. By default, this will be `relationship_model` . The above example, for instance, would require two other models:

```
class Author_model extends MY_Model { }
class Comment_model extends MY_Model { }
```

If you'd like to customise this, you can pass through the model name as a parameter:

```
class Post_model extends MY_Model
{
    public $belongs_to = array( 'author' => array( 'model' => 'author_m' )
);
    public $has_many = array( 'comments' => array( 'model' =>
'model_comments' ) );
}
```

You can then access your related data using the `with()` method:

```
$post = $this->post_model->with('author')
                         ->with('comments')
                         ->get(1);
```

The related data will be embedded in the returned value from `get` :

```
echo $post->author->name;

foreach ($post->comments as $comment)
{
    echo $message;
}
```

Separate queries will be run to select the data, so where performance is important, a separate JOIN and SELECT call is recommended.

The primary key can also be configured. For *belongs_to* calls, the related key is on the

current object, not the foreign one. Pseudocode:

```
SELECT * FROM authors WHERE id = $post->author_id
```

...and for a *has_many* call:

```
SELECT * FROM comments WHERE post_id = $post->id
```

To change this, use the `primary_key` value when configuring:

```
class Post_model extends MY_Model
{
    public $belongs_to = array( 'author' => array( 'primary_key' =>
'post_author_id' ) );
    public $has_many = array( 'comments' => array( 'primary_key' =>
'parent_post_id' ) );
}
```

Arrays vs Objects

By default, MY_Model is setup to return objects using CodeIgniter's QB's `row()` and `result()` methods. If you'd like to use their array counterparts, there are a couple of ways of customising the model.

If you'd like all your calls to use the array methods, you can set the `$return_type` variable to `array`.

```
class Book_model extends MY_Model
{
    protected $return_type = 'array';
}
```

If you'd like just your *next* call to return a specific type, there are two scoping methods you can use:

```
$this->book_model->as_array()
                 ->get(1);
$this->book_model->as_object()
                 ->get_by('column', 'value');
```

Soft Delete

By default, the delete mechanism works with an SQL `DELETE` statement. However, you might not want to destroy the data, you might instead want to perform a 'soft delete'.

If you enable soft deleting, the deleted row will be marked as `deleted` rather than actually being removed from the database.

Take, for example, a `Book_model` :

```
class Book_model extends MY_Model { }
```

We can enable soft delete by setting the `$this->soft_delete` key:

```
class Book_model extends MY_Model
{
    protected $soft_delete = TRUE;
}
```

By default, MY_Model expects a `TINYINT` or `INT` column named `deleted` . If you'd like to customise this, you can set `$soft_delete_key` :

```
class Book_model extends MY_Model
{
    protected $soft_delete = TRUE;
    protected $soft_delete_key = 'book_deleted_status';
}
```

Now, when you make a call to any of the `get_` methods, a constraint will be added to not withdraw deleted columns:

```
=> $this->book_model->get_by('user_id', 1);
-> SELECT * FROM books WHERE user_id = 1 AND deleted = 0
```

If you'd like to include deleted columns, you can use the `with_deleted()` scope:

```
=> $this->book_model->with_deleted()->get_by('user_id', 1);
-> SELECT * FROM books WHERE user_id = 1
```

Built-in Observers

MY_Model contains a few built-in observers for things I've found I've added to most of my models.

The timestamps (MySQL compatible) `created_at` and `updated_at` are now available as

built-in observers:

```
class Post_model extends MY_Model
{
    public $before_create = array( 'created_at', 'updated_at' );
    public $before_update = array( 'updated_at' );
}
```

MY_Model also contains serialisation observers for serialising and unserialising native PHP objects. This allows you to pass complex structures like arrays and objects into rows and have it be serialised automatically in the background. Call the `serialize` and `unserialize` observers with the column name(s) as a parameter:

```
class Event_model extends MY_Model
{
    public $before_create = array( 'serialize(seat_types)' );
    public $before_update = array( 'serialize(seat_types)' );
    public $after_get = array( 'unserialize(seat_types)' );
}
```

Unit Tests

MY_Model contains a robust set of unit tests to ensure that the system works as planned.

Install the testing framework (PHPUnit) with Composer:

```
$ curl -s https://getcomposer.org/installer | php
$ php composer.phar install
```

You can then run the tests using the `vendor/bin/phpunit` binary and specify the tests file:

```
$ vendor/bin/phpunit tests/MY_Model_test.php
```

Contributing to MY_Model

If you find a bug or want to add a feature to MY_Model, great! In order to make it easier and quicker for me to verify and merge changes in, it would be amazing if you could follow these few basic steps:

1. Fork the project.
2. **Branch out into a new branch. `git checkout -b name_of_new_feature_or_bug`**
3. Make your feature addition or bug fix.

4. **Add tests for it. This is important so I don't break it in a future version unintentionally.**
5. Commit.
6. Send me a pull request!

Other Documentation

- My book, The CodeIgniter Handbook, talks about the techniques used in MY_Model and lots of other interesting useful stuff. Get a copy now.
- Jeff Madsen has written an excellent tutorial about the basics (and triggered me updating the documentation here). Read it now, you lovely people.
- Rob Allport wrote a post about MY_Model and his experiences with it. Check it out!
- I've written a write up of the new 2.0.0 features over at my blog, Jamie On Software.

Changelog

Version 2.0.0 * Added support for soft deletes * Removed Composer support. Great system, CI makes it difficult to use for MY_ classes * Fixed up all problems with callbacks and consolidated into single `trigger` method * Added support for relationships * Added built-in timestamp observers * The DB connection can now be manually set with `$this->_db`, rather than relying on the `$active_group` * Callbacks can also now take parameters when setting in callback array * Added support for column serialisation * Added support for protected attributes * Added a `truncate()` method

Version 1.3.0 * Added support for array return types using `$return_type` variable and `as_array()` and `as_object()` methods * Added PHP5.3 support for the test suite * Removed the deprecated `MY_Model()` constructor * Fixed an issue with after_create callbacks (thanks zbrox!) * Composer package will now autoload the file * Fixed the callback example by returning the given/modified data (thanks druu!) * Change order of operations in `_fetch_table()` (thanks JustinBusschau!)

Version 1.2.0 * Bugfix to `update_many()` * Added getters for table name and skip validation * Fix to callback functionality (thanks titosemi!) * Vastly improved documentation * Added a `get_next_id()` method (thanks gbaldera!) * Added a set of unit tests * Added support for Composer

Version 1.0.0 - 1.1.0 * Initial Releases