Idiorm Documentation

Release 1.2.3

Jamie Matthews, Simon Holywell, Durham Hale

February 28, 2013

CONTENTS

Contents:

ONE

PHILOSOPHY

The Pareto Principle states that *roughly 80% of the effects come from 20% of the causes.* In software development terms, this could be translated into something along the lines of *80% of the results come from 20% of the complexity*. In other words, you can get pretty far by being pretty stupid.

Idiorm is deliberately simple. Where other ORMs consist of dozens of classes with complex inheritance hierarchies, Idiorm has only one class, ORM, which functions as both a fluent SELECT query API and a simple CRUD model class. If my hunch is correct, this should be quite enough for many real-world applications. Let's face it: most of us aren't building Facebook. We're working on small-to-medium-sized projects, where the emphasis is on simplicity and rapid development rather than infinite flexibility and features.

You might think of **Idiorm** as a *micro-ORM*. It could, perhaps, be "the tie to go along with Slim's tux" (to borrow a turn of phrase from DocumentCloud). Or it could be an effective bit of spring cleaning for one of those horrendous SQL-littered legacy PHP apps you have to support.

Idiorm might also provide a good base upon which to build higher-level, more complex database abstractions. For example, Paris is an implementation of the Active Record pattern built on top of Idiorm.

INSTALLATION

2.1 Packagist

This library is available through Packagist with the vendor and package identifier of `j4mie/idiorm`

Please see the Packagist documentation for further information.

2.2 Download

You can clone the git repository, download idiorm.php or a release tag and then drop the idiorm.php file in the vendors/3rd party/libs directory of your project.

CONFIGURATION

The first thing you need to know about Idiorm is that *you don't need to define any model classes to use it*. With almost every other ORM, the first thing to do is set up your models and map them to database tables (through configuration variables, XML files or similar). With Idiorm, you can start using the ORM straight away.

3.1 Setup

First, `require` the Idiorm source file:

```
require_once 'idiorm.php';
```

Then, pass a *Data Source Name* connection string to the `configure` method of the ORM class. This is used by PDO to connect to your database. For more information, see the PDO documentation.

```
ORM::configure('sqlite:./example.db');
```

You may also need to pass a username and password to your database driver, using the `username` and `password` configuration options. For example, if you are using MySQL:

```
ORM::configure('mysql:host=localhost;dbname=my_database');
ORM::configure('username', 'database_user');
ORM::configure('password', 'top_secret');
```

Also see "Configuration" section below.

3.2 Configuration

Other than setting the DSN string for the database connection (see above), the `configure` method can be used to set some other simple options on the ORM class. Modifying settings involves passing a key/value pair to the `configure` method, representing the setting you wish to modify and the value you wish to set it to.

```
ORM::configure('setting_name', 'value_for_setting');
```

A shortcut is provided to allow passing multiple key/value pairs at once.

```
ORM::configure(array(
    'setting_name_1' => 'value_for_setting_1',
    'setting_name_2' => 'value_for_setting_2',
    'etc' => 'etc'
));
```

3.2.1 Database authentication details

Settings: `username` and `password`

Some database adapters (such as MySQL) require a username and password to be supplied separately to the DSN string. These settings allow you to provide these values. A typical MySQL connection setup might look like this:

```
ORM::configure('mysql:host=localhost;dbname=my_database');
ORM::configure('username', 'database_user');
ORM::configure('password', 'top_secret');
```

Or you can combine the connection setup into a single line using the configuration array shortcut:

```
ORM::configure(array(
    'mysql:host=localhost;dbname=my_database',
    'username' => 'database_user',
    'password' => 'top_secret'
));
```

3.2.2 Result sets

Setting: `return_result_sets`

Collections of results can be returned as an array (default) or as a result set. See the *find_result_set()* documentation for more information.

```
ORM::configure('return_result_sets', true); // returns result sets
```

Note: It is recommended that you setup your projects to use result sets as they are more flexible.

3.2.3 PDO Driver Options

Setting: `driver_options`

Some database adapters require (or allow) an array of driver-specific configuration options. This setting allows you to pass these options through to the PDO constructor. For more information, see the PDO documentation. For example, to force the MySQL driver to use UTF-8 for the connection:

```
ORM::configure('driver_options', array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8'));
```

3.2.4 PDO Error Mode

Setting: `error_mode`

This can be used to set the `PDO::ATTR_ERRMODE` setting on the database connection class used by Idiorm. It should be passed one of the class constants defined by PDO. For example:

```
ORM::configure('error_mode', PDO::ERRMODE_WARNING);
```

The default setting is `PDO::ERRMODE_EXCEPTION`. For full details of the error modes available, see the PDO set attribute documentation.

3.2.5 PDO object access

Should it ever be necessary, the PDO object used by Idiorm may be accessed directly through `ORM::get_db()`, or set directly via `ORM::set_db()`. This should be an unusual occurance.

After a statement has been executed by any means, such as `::save()` or `::raw_execute()`, the `PDOStatement` instance used may be accessed via `ORM::get_last_statement()`. This may be useful in order to access `PDOStatement::errorCode()`, if PDO exceptions are turned off, or to access the `PDOStatement::rowCount()` method, which returns differing results based on the underlying database. For more information, see the PDOStatement documentation.

3.2.6 Identifier quote character

Setting: `identifier_quote_character`

Set the character used to quote identifiers (eg table name, column name). If this is not set, it will be autodetected based on the database driver being used by PDO.

3.2.7 ID Column

By default, the ORM assumes that all your tables have a primary key column called `id`. There are two ways to override this: for all tables in the database, or on a per-table basis.

Setting: `id_column`

This setting is used to configure the name of the primary key column for all tables. If your ID column is called `primary_key`, use:

```
ORM::configure('id_column', 'primary_key');
```

Setting: `id_column_overrides`

This setting is used to specify the primary key column name for each table separately. It takes an associative array mapping table names to column names. If, for example, your ID column names include the name of the table, you can use the following configuration:

```
ORM::configure('id_column_overrides', array(
    'person' => 'person_id',
    'role' => 'role_id',
));
```

3.2.8 Query logging

Setting: `logging`

Idiorm can log all queries it executes. To enable query logging, set the `logging` option to `true` (it is `false` by default).

When query logging is enabled, you can use two static methods to access the log. `ORM::get_last_query()` returns the most recent query executed. `ORM::get_query_log()` returns an array of all queries executed.

3.2.9 Query caching

Setting: `caching`

Idiorm can cache the queries it executes during a request. To enable query caching, set the `caching` option to `true` (it is `false` by default).

When query caching is enabled, Idiorm will cache the results of every `SELECT` query it executes. If Idiorm encounters a query that has already been run, it will fetch the results directly from its cache and not perform a database query.

Warnings and gotchas

- Note that this is an in-memory cache that only persists data for the duration of a single request. This is *not* a replacement for a persistent cache such as Memcached.

- Idiorm's cache is very simple, and does not attempt to invalidate itself when data changes. This means that if you run a query to retrieve some data, modify and save it, and then run the same query again, the results will be stale (ie, they will not reflect your modifications). This could potentially cause subtle bugs in your application. If you have caching enabled and you are experiencing odd behaviour, disable it and try again. If you do need to perform such operations but still wish to use the cache, you can call the `ORM::clear_cache()` to clear all existing cached queries.

- Enabling the cache will increase the memory usage of your application, as all database rows that are fetched during each request are held in memory. If you are working with large quantities of data, you may wish to disable the cache.

QUERYING

Idiorm provides a *fluent interface* to enable simple queries to be built without writing a single character of SQL. If you've used jQuery at all, you'll be familiar with the concept of a fluent interface. It just means that you can *chain* method calls together, one after another. This can make your code more readable, as the method calls strung together in order can start to look a bit like a sentence.

All Idiorm queries start with a call to the `for_table` static method on the ORM class. This tells the ORM which table to use when making the query.

Note that this method **does not* escape its query parameter and so the table name should **not** be passed directly from user input.*

Method calls which add filters and constraints to your query are then strung together. Finally, the chain is finished by calling either `find_one()` or `find_many()`, which executes the query and returns the result.

Let's start with a simple example. Say we have a table called `person` which contains the columns `id` (the primary key of the record - Idiorm assumes the primary key column is called `id` but this is configurable, see below), `name`, `age` and `gender`.

4.1 Single records

Any method chain that ends in `find_one()` will return either a *single* instance of the ORM class representing the database row you requested, or `false` if no matching record was found.

To find a single record where the `name` column has the value "Fred Bloggs":

```
$person = ORM::for_table('person')->where('name', 'Fred Bloggs')->find_one();
```

This roughly translates into the following SQL: `SELECT * FROM person WHERE name = "Fred Bloggs"`

To find a single record by ID, you can pass the ID directly to the `find_one` method:

```
$person = ORM::for_table('person')->find_one(5);
```

4.2 Multiple records

Note: It is recommended that you use results sets over arrays - see *As a result set* below.

Any method chain that ends in `find_many()` will return an *array* of ORM class instances, one for each row matched by your query. If no rows were found, an empty array will be returned.

To find all records in the table:

```
$people = ORM::for_table('person')->find_many();
```

To find all records where the `gender` is `female`:

```
$females = ORM::for_table('person')->where('gender', 'female')->find_many();
```

4.2.1 As a result set

Note: There is a configuration setting `return_result_sets` that will cause `find_many()` to return result sets by default. It is recommended that you turn this setting on:

```
ORM::configure('return_result_sets', true);
```

You can also find many records as a result set instead of an array of Idiorm instances. This gives you the advantage that you can run batch operations on a set of results.

So for example instead of running this:

```
$people = ORM::for_table('person')->find_many();
foreach ($people as $person) {
    $person->age = 50;
    $person->save();
}
```

You can simple do this instead:

```
ORM::for_table('person')->find_result_set()
->set('age', 50)
->save();
```

To do this substitute any call to `find_many()` with `find_result_set()`.

A result set will also behave like an array so you can *count()* it and *foreach* over it just like an array.

```
foreach(ORM::for_table('person')->find_result_set() as $record) {
    echo $person->name;
}

echo count(ORM::for_table('person')->find_result_set());
```

Note: For deleting many records it is recommended that you use *delete_many()* as it is more efficient than calling *delete()* on a result set.

4.2.2 As an associative array

You can also find many records as an associative array instead of Idiorm instances. To do this substitute any call to `find_many()` with `find_array()`.

```
$females = ORM::for_table('person')->where('gender', 'female')->find_array();
```

This is useful if you need to serialise the the query output into a format like JSON and you do not need the ability to update the returned records.

4.3 Counting results

To return a count of the number of rows that would be returned by a query, call the `count()` method.

```
$number_of_people = ORM::for_table('person')->count();
```

4.4 Filtering results

Idiorm provides a family of methods to extract only records which satisfy some condition or conditions. These methods may be called multiple times to build up your query, and Idiorm's fluent interface allows method calls to be *chained* to create readable and simple-to-understand queries.

4.4.1 *Caveats*

Only a subset of the available conditions supported by SQL are available when using Idiorm. Additionally, all the `WHERE` clauses will be `AND`ed together when the query is run. Support for `OR`ing `WHERE` clauses is not currently present.

These limits are deliberate: these are by far the most commonly used criteria, and by avoiding support for very complex queries, the Idiorm codebase can remain small and simple.

Some support for more complex conditions and queries is provided by the `where_raw` and `raw_query` methods (see below). If you find yourself regularly requiring more functionality than Idiorm can provide, it may be time to consider using a more full-featured ORM.

4.4.2 Equality: `where, where_equal, where_not_equal`

By default, calling `where` with two parameters (the column name and the value) will combine them using an equals operator (=). For example, calling `where('name', 'Fred')` will result in the clause `WHERE name = "Fred"`.

If your coding style favours clarity over brevity, you may prefer to use the `where_equal` method: this is identical to `where`.

The `where_not_equal` method adds a `WHERE column != "value"` clause to your query.

4.4.3 Shortcut: `where_id_is`

This is a simple helper method to query the table by primary key. Respects the ID column specified in the config.

4.4.4 Less than / greater than: `where_lt, where_gt, where_lte, where_gte`

There are four methods available for inequalities:

- Less than: `$people = ORM::for_table('person')->where_lt('age', 10)->find_many();`

- Greater than: `$people = ORM::for_table('person')->where_gt('age', 5)->find_many();`

- Less than or equal: `$people = ORM::for_table('person')->where_lte('age', 10)->find_many();`

- Greater than or equal: `$people = ORM::for_table('person')->where_gte('age', 5)->find_many();`

4.4.5 String comparision: `where_like` and `where_not_like`

To add a `WHERE ... LIKE` clause, use:

```
$people = ORM::for_table('person')->where_like('name', '%fred%')->find_many();
```

Similarly, to add a `WHERE ... NOT LIKE` clause, use:

```
$people = ORM::for_table('person')->where_not_like('name', '%bob%')->find_many();
```

4.4.6 Set membership: `where_in` and `where_not_in`

To add a `WHERE ... IN ()` or `WHERE ... NOT IN ()` clause, use the `where_in` and `where_not_in` methods respectively.

Both methods accept two arguments. The first is the column name to compare against. The second is an *array* of possible values.

```
$people = ORM::for_table('person')->where_in('name', array('Fred', 'Joe', 'John'))->find_many();
```

4.4.7 Working with `NULL` values: `where_null` and `where_not_null`

To add a `WHERE column IS NULL` or `WHERE column IS NOT NULL` clause, use the `where_null` and `where_not_null` methods respectively. Both methods accept a single parameter: the column name to test.

4.4.8 Raw WHERE clauses

If you require a more complex query, you can use the `where_raw` method to specify the SQL fragment for the WHERE clause exactly. This method takes two arguments: the string to add to the query, and an (optional) array of parameters which will be bound to the string. If parameters are supplied, the string should contain question mark characters (`?`) to represent the values to be bound, and the parameter array should contain the values to be substituted into the string in the correct order.

This method may be used in a method chain alongside other `where_*` methods as well as methods such as `offset`, `limit` and `order_by_*`. The contents of the string you supply will be connected with preceding and following WHERE clauses with AND.

```
$people = ORM::for_table('person')
            ->where('name', 'Fred')
            ->where_raw('(`age` = ? OR `age` = ?)', array(20, 25))
            ->order_by_asc('name')
            ->find_many();

// Creates SQL:
SELECT * FROM `person` WHERE `name` = "Fred" AND (`age` = 20 OR `age` = 25) ORDER BY `name` ASC;
```

Note that this method only supports "question mark placeholder" syntax, and NOT "named placeholder" syntax. This is because PDO does not allow queries that contain a mixture of placeholder types. Also, you should ensure that the number of question mark placeholders in the string exactly matches the number of elements in the array.

If you require yet more flexibility, you can manually specify the entire query. See *Raw queries* below.

4.4.9 Limits and offsets

Note that these methods **do not* escape their query parameters and so these should **not** be passed directly from user input.*

The `limit` and `offset` methods map pretty closely to their SQL equivalents.

```
$people = ORM::for_table('person')->where('gender', 'female')->limit(5)->offset(10)->find_many();
```

4.4.10 Ordering

Note that these methods **do not* escape their query parameters and so these should **not** be passed directly from user input.*

Two methods are provided to add `ORDER BY` clauses to your query. These are `order_by_desc` and `order_by_asc`, each of which takes a column name to sort by. The column names will be quoted.

```
$people = ORM::for_table('person')->order_by_asc('gender')->order_by_desc('name')->find_many();
```

If you want to order by something other than a column name, then use the `order_by_expr` method to add an unquoted SQL expression as an `ORDER BY` clause.

```
$people = ORM::for_table('person')->order_by_expr('SOUNDEX(`name`)')->find_many();
```

4.5 Grouping

Note that this method **does not* escape it query parameter and so this should **not** by passed directly from user input.*

To add a `GROUP BY` clause to your query, call the `group_by` method, passing in the column name. You can call this method multiple times to add further columns.

```
$people = ORM::for_table('person')->where('gender', 'female')->group_by('name')->find_many();
```

It is also possible to `GROUP BY` a database expression:

```
$people = ORM::for_table('person')->where('gender', 'female')->group_by_expr("FROM_UNIXTIME(`time`, '
```

4.6 Having

When using aggregate functions in combination with a `GROUP BY` you can use `HAVING` to filter based on those values.

`HAVING` works in exactly the same way as all of the `where*` functions in Idiorm. Substitute `where_` for `having_` to make use of these functions.

For example:

```
$people = ORM::for_table('person')->group_by('name')->having_not_like('name', '%bob%')->find_many();
```

4.7 Result columns

By default, all columns in the `SELECT` statement are returned from your query. That is, calling:

```
$people = ORM::for_table('person')->find_many();
```

Will result in the query:

```
SELECT * FROM `person`;
```

The `select` method gives you control over which columns are returned. Call `select` multiple times to specify columns to return or use `select_many <#shortcuts-for-specifying-many-columns>`_ to specify many columns at once.

```
$people = ORM::for_table('person')->select('name')->select('age')->find_many();
```

Will result in the query:

```
SELECT `name`, `age` FROM `person`;
```

Optionally, you may also supply a second argument to `select` to specify an alias for the column:

```
$people = ORM::for_table('person')->select('name', 'person_name')->find_many();
```

Will result in the query:

```
SELECT `name` AS `person_name` FROM `person`;
```

Column names passed to `select` are quoted automatically, even if they contain `table.column`-style identifiers:

```
$people = ORM::for_table('person')->select('person.name', 'person_name')->find_many();
```

Will result in the query:

```
SELECT `person`.`name` AS `person_name` FROM `person`;
```

If you wish to override this behaviour (for example, to supply a database expression) you should instead use the `select_expr` method. Again, this takes the alias as an optional second argument. You can specify multiple expressions by calling `select_expr` multiple times or use `select_many_expr <#shortcuts-for-specifying-many-columns>`_ to specify many expressions at once.

```
// NOTE: For illustrative purposes only. To perform a count query, use the count() method.
$people_count = ORM::for_table('person')->select_expr('COUNT(*)', 'count')->find_many();
```

Will result in the query:

```
SELECT COUNT(*) AS `count` FROM `person`;
```

4.7.1 Shortcuts for specifying many columns

`select_many` and `select_many_expr` are very similar, but they allow you to specify more than one column at once. For example:

```
$people = ORM::for_table('person')->select_many('name', 'age')->find_many();
```

Will result in the query:

```
SELECT `name`, `age` FROM `person`;
```

To specify aliases you need to pass in an array (aliases are set as the key in an associative array):

```
$people = ORM::for_table('person')->select_many(array('first_name' => 'name', 'age'), 'height')->find
```

Will result in the query:

```
SELECT `name` AS `first_name`, `age`, `height` FROM `person`;
```

You can pass the the following styles into `select_many` and `select_many_expr` by mixing and matching arrays and parameters:

```
select_many(array('alias' => 'column', 'column2', 'alias2' => 'column3'), 'column4', 'column5')
select_many('column', 'column2', 'column3')
select_many(array('column', 'column2', 'column3'), 'column4', 'column5')
```

All the select methods can also be chained with each other so you could do the following to get a neat select query including an expression:

```
$people = ORM::for_table('person')->select_many('name', 'age', 'height')->select_expr('NOW()', 'times
```

Will result in the query:

```
SELECT `name`, `age`, `height`, NOW() AS `timestamp` FROM `person`;
```

4.8 DISTINCT

To add a `DISTINCT` keyword before the list of result columns in your query, add a call to `distinct()` to your query chain.

```
$distinct_names = ORM::for_table('person')->distinct()->select('name')->find_many();
```

This will result in the query:

```
SELECT DISTINCT `name` FROM `person`;
```

4.9 Joins

Idiorm has a family of methods for adding different types of `JOIN`s to the queries it constructs:

Methods: `join`, `inner_join`, `left_outer_join`, `right_outer_join`, `full_outer_join`.

Each of these methods takes the same set of arguments. The following description will use the basic `join` method as an example, but the same applies to each method.

The first two arguments are mandatory. The first is the name of the table to join, and the second supplies the conditions for the join. The recommended way to specify the conditions is as an *array* containing three components: the first column, the operator, and the second column. The table and column names will be automatically quoted. For example:

```
$results = ORM::for_table('person')->join('person_profile', array('person.id', '=', 'person_profile.p
```

It is also possible to specify the condition as a string, which will be inserted as-is into the query. However, in this case the column names will **not** be escaped, and so this method should be used with caution.

```
// Not recommended because the join condition will not be escaped.
$results = ORM::for_table('person')->join('person_profile', 'person.id = person_profile.person_id')->
```

The `join` methods also take an optional third parameter, which is an `alias` for the table in the query. This is useful if you wish to join the table to *itself* to create a hierarchical structure. In this case, it is best combined with the `table_alias` method, which will add an alias to the *main* table associated with the ORM, and the `select` method to control which columns get returned.

```
$results = ORM::for_table('person')
    ->table_alias('p1')
    ->select('p1.*')
    ->select('p2.name', 'parent_name')
    ->join('person', array('p1.parent', '=', 'p2.id'), 'p2')
    ->find_many();
```

4.10 Aggregate functions

There is support for `MIN`, `AVG`, `MAX` and `SUM` in addition to `COUNT` (documented earlier).

To return a minimum value of column, call the `min()` method.

```
$min = ORM::for_table('person')->min('height');
```

The other functions (`AVG`, `MAX` and `SUM`) work in exactly the same manner. Supply a column name to perform the aggregate function on and it will return an integer.

4.11 Raw queries

If you need to perform more complex queries, you can completely specify the query to execute by using the `raw_query` method. This method takes a string and optionally an array of parameters. The string can contain placeholders, either in question mark or named placeholder syntax, which will be used to bind the parameters to the query.

```
$people = ORM::for_table('person')->raw_query('SELECT p.* FROM person p JOIN role r ON p.role_id = r.
```

The ORM class instance(s) returned will contain data for all the columns returned by the query. Note that you still must call `for_table` to bind the instances to a particular table, even though there is nothing to stop you from specifying a completely different table in the query. This is because if you wish to later called `save`, the ORM will need to know which table to update.

Note that using `raw_query` is advanced and possibly dangerous, and Idiorm does not make any attempt to protect you from making errors when using this method. If you find yourself calling `raw_query` often, you may have

misunderstood the purpose of using an ORM, or your application may be too complex for Idiorm. Consider using a more full-featured database abstraction system.

MODELS

5.1 Getting data from objects

Once you've got a set of records (objects) back from a query, you can access properties on those objects (the values stored in the columns in its corresponding table) in two ways: by using the `get` method, or simply by accessing the property on the object directly:

```
$person = ORM::for_table('person')->find_one(5);

// The following two forms are equivalent
$name = $person->get('name');
$name = $person->name;
```

You can also get the all the data wrapped by an ORM instance using the `as_array` method. This will return an associative array mapping column names (keys) to their values.

The `as_array` method takes column names as optional arguments. If one or more of these arguments is supplied, only matching column names will be returned.

```
$person = ORM::for_table('person')->create();

$person->first_name = 'Fred';
$person->surname = 'Bloggs';
$person->age = 50;

// Returns array('first_name' => 'Fred', 'surname' => 'Bloggs', 'age' => 50)
$data = $person->as_array();

// Returns array('first_name' => 'Fred', 'age' => 50)
$data = $person->as_array('first_name', 'age');
```

5.2 Updating records

To update the database, change one or more of the properties of the object, then call the `save` method to commit the changes to the database. Again, you can change the values of the object's properties either by using the `set` method or by setting the value of the property directly. By using the `set` method it is also possible to update multiple properties at once, by passing in an associative array:

```
$person = ORM::for_table('person')->find_one(5);

// The following two forms are equivalent
```

```
$person->set('name', 'Bob Smith');
$person->age = 20;

// This is equivalent to the above two assignments
$person->set(array(
    'name' => 'Bob Smith',
    'age'  => 20
));

// Syncronise the object with the database
$person->save();
```

5.2.1 Properties containing expressions

It is possible to set properties on the model that contain database expressions using the `set_expr` method.

```
$person = ORM::for_table('person')->find_one(5);
$person->set('name', 'Bob Smith');
$person->age = 20;
$person->set_expr('updated', 'NOW()');
$person->save();
```

The `updated` column's value will be inserted into query in its raw form therefore allowing the database to execute any functions referenced - such as `NOW()` in this case.

5.3 Creating new records

To add a new record, you need to first create an "empty" object instance. You then set values on the object as normal, and save it.

```
$person = ORM::for_table('person')->create();

$person->name = 'Joe Bloggs';
$person->age = 40;

$person->save();
```

After the object has been saved, you can call its `id()` method to find the autogenerated primary key value that the database assigned to it.

5.3.1 Properties containing expressions

It is possible to set properties on the model that contain database expressions using the `set_expr` method.

```
$person = ORM::for_table('person')->create();
$person->set('name', 'Bob Smith');
$person->age = 20;
$person->set_expr('added', 'NOW()');
$person->save();
```

The `added` column's value will be inserted into query in its raw form therefore allowing the database to execute any functions referenced - such as `NOW()` in this case.

5.4 Checking whether a property has been modified

To check whether a property has been changed since the object was created (or last saved), call the `is_dirty` method:

```
$name_has_changed = $person->is_dirty('name'); // Returns true or false
```

5.5 Deleting records

To delete an object from the database, simply call its `delete` method.

```
$person = ORM::for_table('person')->find_one(5);
$person->delete();
```

To delete more than one object from the database, build a query:

```
$person = ORM::for_table('person')
    ->where_equal('zipcode', 55555)
    ->delete_many();
```

TRANSACTIONS

Idiorm doesn't supply any extra methods to deal with transactions, but it's very easy to use PDO's built-in methods:

```
// Start a transaction
ORM::get_db()->beginTransaction();

// Commit a transaction
ORM::get_db()->commit();

// Roll back a transaction
ORM::get_db()->rollBack();
```

For more details, see the PDO documentation on Transactions.

MULITPLE CONNECTIONS

Idiorm now works with multiple conections. Most of the static functions work with an optional connection name as an extra parameter. For the `ORM::configure` method, this means that when passing connection strings for a new connection, the second parameter, which is typically omitted, should be `null`. In all cases, if a connection name is not provided, it defaults to `ORM::DEFAULT_CONNECTION`.

When chaining, once `for_table()` has been used in the chain, remaining calls in the chain use the correct connection.

```
// Default connection
ORM::configure('sqlite:./example.db');

// A named connection, where 'remote' is an arbitrary key name
ORM::configure('mysql:host=localhost;dbname=my_database', null, 'remote');
ORM::configure('username', 'database_user', 'remote');
ORM::configure('password', 'top_secret', 'remote');

// Using default connection
$person = ORM::for_table('person')->find_one(5);

// Using default connection, explicitly
$person = ORM::for_table('person', ORM::DEFAULT_CONNECTION)->find_one(5);

// Using named connection
$person = ORM::for_table('different_person', 'remote')->find_one(5);
```

7.1 Supported Methods

In each of these cases, the `$connection_name` parameter is optional, and is an arbitrary key identifying the named connection.

- `ORM::configure($key, $value, $connection_name)`
- `ORM::for_table($table_name, $connection_name)`
- `ORM::set_db($pdo, $connection_name)`
- `ORM::get_db($connection_name)`
- `ORM::raw_execute($query, $parameters, $connection_name)`
- `ORM::get_last_query($connection_name)`
- `ORM::get_query_log($connection_name)`

Of these methods, only ORM::get_last_query($connection_name) does *not* fallback to the default connection when no connection name is passed. Instead, passing no connection name (or null) returns the most recent query on *any* connection.

```
// Using default connection, explicitly
$person = ORM::for_table('person')->find_one(5);

// Using named connection
$person = ORM::for_table('different_person', 'remote')->find_one(5);

// Last query on *any* connection
ORM::get_last_query(); // returns query on 'different_person' using 'remote'

// returns query on 'person' using default by passing in the connection name
ORM::get_last_query(ORM::DEFAULT_CONNECTION);
```

7.1.1 Notes

- **There is no support for joins across connections**

- Multiple connections do not share configuration settings. This means if one connection has logging set to true and the other does not, only queries from the logged connection will be available via ORM::get_last_query() and ORM::get_query_log().

- A new method has been added, ORM::get_connection_names(), which returns an array of connection names.

- Caching *should* work with multiple connections (remember to turn caching on for each connection), but the unit tests are not robust. Please report any errors.

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*