Titi Documentation

Jim Winstead Jr.

Aug 22, 2023

Contents

Contents:

CHAPTER 1

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.

Titi is deliberately simple. Where other ORMs consist of dozens of classes with complex inheritance hierarchies, Titi has only one class at its base, `Titi\ORM`, which functions as both a fluent `SELECT` query API and a simple CRUD model class. 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 **Titi** 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.

Titi might also provide a good base upon which to build higher-level, more complex database abstractions. For example, it includes an implementation of the Active Record pattern built on top of that base `Titi\ORM` class.

CHAPTER 2

Installation

2.1 Packagist

This library is available through Packagist with the vendor and package identifier of `jimwins/titi`

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.

CHAPTER 3

Configuration

The first thing you need to know about Titi 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 Titi, you can start using the ORM straight away.

3.1 Setup

If you have used Composer to include Titi in your project, you can use the autoloader to provide access to the Titi namespace:

```php
<?php
require_once __DIR__ . '/vendor/autoload.php';
```

You can also use `use` to pull in the ORM and/or Model from the Titi namespace (for an easy upgrade path from Idiorm and Paris).

```php
<?php
require_once __DIR__ . '/vendor/autoload.php';

use \Titi\ORM, \Titi\Model;
```

(Further examples will assume that you have done this so we don't need to keep explicitly referencing the Titi namespace.)

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.

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

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.

```php
<?php
ORM::configure('setting_name', 'value_for_setting');
```

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

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

Use the `get_config` method to read current settings.

```php
<?php
$isLoggingEnabled = ORM::get_config('logging');
ORM::configure('logging', false);
try {
  // some crazy loop we don't want to log
} finally {
  ORM::configure('logging', $isLoggingEnabled);
}
```

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. (Although this is no longer true as of PHP 7.4, where you can now specify these in the DSN for more adapters.) These settings allow you to provide these values. A typical MySQL connection setup might look like this:

```php
<?php
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:

```php
<?php
ORM::configure(array(
    'connection_string' => '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.

```php
<?php
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:

```php
<?php
ORM::configure('driver_options', array(PDO::MYSQL_ATTR_LOCAL_INFILE => true));
```

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 Titi. It should be passed one of the class constants defined by PDO. For example:

```php
<?php
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 Titi 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:

```php
<?php
ORM::configure('id_column', 'primary_key');
```

You can specify a compound primary key using an array:

```php
<?php
ORM::configure('id_column', array('pk_1', 'pk_2'));
```

Note: If you use a auto-increment column in the compound primary key then it should be the first one defined into the array.

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:

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

As with `id_column` setting, you can specify a compound primary key using an array.

3.2.8 Limit clause style

Setting: `limit_clause_style`

You can specify the limit clause style in the configuration. This is to facilitate a MS SQL style limit clause that uses the `TOP` syntax.

Acceptable values are `ORM::LIMIT_STYLE_TOP_N` and `ORM::LIMIT_STYLE_LIMIT`.

Note: If the PDO driver you are using is one of sqlsrv, dblib or mssql then Titi will automatically select the `ORM::LIMIT_STYLE_TOP_N` for you unless you override the setting.

3.2.9 Query logging

Setting: `logging`

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

3.3 Model prefixing

Setting: `Model::$auto_prefix_models`

To save having type out model class name prefixes whenever code utilises `Model::for_table()` it is possible to specify a prefix that will be prepended onto the class name.

The model prefix is treated the same way as any other class name when the Model attempts to convert it to a table name. This is documented in the *Models* section of the documentation.

Here is a namespaced example to make it clearer:

```
<?php
Model::$auto_prefix_models = '\\Tests\\';
Model::factory('Simple')->find_many(); // SQL executed: SELECT * FROM `tests_simple`
Model::factory('SimpleUser')->find_many(); // SQL executed: SELECT * FROM `tests_
↪simple_user`
```

Model prefixes are only compatible with the `Model::factory()` methods described above. Where the shorter `SimpleUser::find_many()` style syntax is used, the addition of a Model prefix will cause `Class not found` errors.

Note: Model class property `$_table` sets an explicit table name, ignoring the `$auto_prefix_models` property in your individual model classes. See documentation in the *Models* section of the documentation.

3.4 Model namespaces

Setting: `Model::$short_table_names`

Set as `true` to disregard namespace information when computing table names from class names.

By default the class `\Models\CarTyre` expects the table name `models_car_tyre`. With `Model::$short_table_names = true` the class `\Models\CarTyre` expects the table name `car_tyre`.

```
<?php

Model::$short_table_names = true;
Model::factory('CarTyre')->find_many(); // SQL executed: SELECT * FROM `car_tyre`

namespace Models {
    class CarTyre extends Model {

    }
}
```

3.5 Further Configuration

The only other configuration options provided by Titi Models are the `$_table` and `$_id_column` static properties on model classes. To configure the database connection, you should use the ORM's configuration system via the `ORM::configure` method.

If you are using multiple connections, the optional *$_connection_key* static property may also be used to provide a default string key indicating which database connection in *ORM* should be used.

3.6 Query logging

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

```php
<?php
ORM::configure('logging', true);
```

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.

Note: The code that does the query log is an approximation of that provided by PDO/the database (see the ORM source code for detail). The actual query isn't even available to log as the database/PDO handles the binding and doesn't pass it back.

This means that you might come across some inconsistencies between what is logged and what is actually run. In these case you'll need to look at the query log provided by your database vendor (eg. MySQL).

3.6.1 Query logger

Setting: `logger`

Note: You must enable `logging` for this setting to have any effect.

It is possible to supply a `callable` to this configuration setting, which will be executed for every query that idiorm executes. In PHP a `callable` is anything that can be executed as if it were a function. Most commonly this will take the form of a anonymous function.

This setting is useful if you wish to log queries with an external library as it allows you too whatever you would like from inside the callback function.

```php
<?php
ORM::configure('logger', function($log_string, $query_time) {
    echo $log_string . ' in ' . $query_time;
});
```

3.6.2 Query caching

Setting: `caching`

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

```php
<?php
ORM::configure('caching', true);
```

Setting: `caching_auto_clear`

Titi's cache is never cleared by default. If you wish to automatically clear it on save, set `caching_auto_clear` to `true`

```php
<?php
ORM::configure('caching_auto_clear', true);
```

When query caching is enabled, Titi will cache the results of every `SELECT` query it executes. If Titi 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.
- Titi'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.

Custom caching

If you wish to use custom caching functions, you can set them from the configure options.

```php
<?php
$my_cache = array();
ORM::configure('cache_query_result', function ($cache_key, $value, $table_name,
↪$connection_name) use (&$my_cache) {
    $my_cache[$cache_key] = $value;
});
ORM::configure('check_query_cache', function ($cache_key, $table_name, $connection_
↪name) use (&$my_cache) {
    if(isset($my_cache[$cache_key])){
       return $my_cache[$cache_key];
    } else {
    return false;
    }
});
ORM::configure('clear_cache', function ($table_name, $connection_name) use (&$my_
↪cache) {
     $my_cache = array();
});

ORM::configure('create_cache_key', function ($query, $parameters, $table_name,
↪$connection_name) {
    $parameter_string = join(',', $parameters);
    $key = $query . ':' . $parameter_string;
    $my_key = 'my-prefix'.crc32($key);
    return $my_key;
});
```

CHAPTER 4

Querying

Titi 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 Titi 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 - the ORM assumes the primary key column is called `id` but this is configurable, see below), `name`, `age` and `gender`.

4.1 A note on PSR-1 and camelCase

All the methods detailed in the documentation can also be called in a PSR-1 way: underscores (_) become camelCase. Here follows an example of one query chain being converted to a PSR-1 compliant style.

```php
<?php
// documented and default style
$person = ORM::for_table('person')->where('name', 'Fred Bloggs')->find_one();

// PSR-1 compliant style
$person = ORM::forTable('person')->where('name', 'Fred Bloggs')->findOne();
```

As you can see any method can be changed from the documented underscore (_) format to that of a camelCase method name.

Note: In the background the PSR-1 compliant style uses the *__call()* and *__callStatic()* magic methods to map the camelCase method name you supply to the original underscore method name. It then uses *call_user_func_array()* to apply the arguments to the method. If this minimal overhead is too great then you can simply revert to using the underscore methods to avoid it. In general this will not be a bottle neck in any application however and should be considered a micro-optimisation.

As *__callStatic()* was added in PHP 5.3.0 you will need at least that version of PHP to use this feature in any meaningful way.

4.2 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":

```
<?php
$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:

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

If you are using a compound primary key, you can find the records using an array as the parameter:

```
<?php
$person = ORM::for_table('user_role')->find_one(array(
    'user_id' => 34,
    'role_id' => 10
));
```

4.3 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:

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

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

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

4.3.1 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 ORM instances. This gives you the advantage that you can run batch operations on a set of results.

So for example instead of running this:

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

You can simply do this instead:

```
<?php
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.

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

```
<?php
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.

4.3.2 As an associative array

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

```
<?php
$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.

4.4 Counting results

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

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

4.5 Filtering results

Titi 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 Titi's fluent interface allows method calls to be *chained* to create readable and simple-to-understand queries.

4.5.1 *Caveats*

Only a subset of the available conditions supported by SQL are available when using Titi. 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 Titi 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 Titi can provide, it may be time to consider using a more full-featured ORM.

4.5.2 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.

You can specify multiple columns and their values in the same call. In this case you should pass an associative array as the first parameter. The array notation uses keys as column names.

```php
<?php
$people = ORM::for_table('person')
            ->where(array(
                'name' => 'Fred',
                'age' => 20
            ))
            ->find_many();

// Creates SQL:
SELECT * FROM `person` WHERE `name` = "Fred" AND `age` = "20";
```

4.5.3 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. If you are using a compound primary key, you must pass an array where the key is the column name. Columns that don't belong to the key will be ignored.

4.5.4 Shortcut: `where_id_in`

This helper method is similar to "where_id_is', but it expects an array of primary keys to be selected. It is compound primary keys aware.

4.5.5 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();`

4.5.6 String comparision: `where_like` and `where_not_like`

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

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

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

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

4.5.7 Multiple OR'ed conditions

You can add simple OR'ed conditions to the same WHERE clause using `where_any_is`. You should specify multiple conditions using an array of items. Each item will be an associative array that contains a multiple conditions.

```php
<?php
$people = ORM::for_table('person')
            ->where_any_is(array(
                array('name' => 'Joe', 'age' => 10),
                array('name' => 'Fred', 'age' => 20)))
            ->find_many();

// Creates SQL:
SELECT * FROM `widget` WHERE (( `name` = 'Joe' AND `age` = '10' ) OR ( `name` = 'Fred
↪' AND `age` = '20' ));
```

(continues on next page)

(continued from previous page)

By default, it uses the equal operator for every column, but it can be overriden for any column using a second parameter:

```php
<?php
$people = ORM::for_table('person')
            ->where_any_is(array(
                array('name' => 'Joe', 'age' => 10),
                array('name' => 'Fred', 'age' => 20)), array('age' => '>'))
            ->find_many();

// Creates SQL:
SELECT * FROM `widget` WHERE (( `name` = 'Joe' AND `age` = '10' ) OR ( `name` = 'Fred
↪' AND `age` > '20' ));
```

If you want to set the default operator for all the columns, just pass it as the second parameter:

```php
<?php
$people = ORM::for_table('person')
            ->where_any_is(array(
                array('score' => '5', 'age' => 10),
                array('score' => '15', 'age' => 20)), '>')
            ->find_many();

// Creates SQL:
SELECT * FROM `widget` WHERE (( `score` > '5' AND `age` > '10' ) OR ( `score` > '15'␣
↪AND `age` > '20' ));
```

4.5.8 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. As all the `where_` methods, you can specify multiple columns using an associative *array* as the only parameter.

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

4.5.9 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.

4.5.10 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.

```
<?php
$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: You must wrap your expression in parentheses when using any of `ALL`, `ANY`, `BETWEEN`, `IN`, `LIKE`, `OR` and `SOME`. Otherwise the precedence of `AND` will bind stronger and in the above example you would effectively get `` WHERE (`name` = "Fred" AND `age` = 20) OR `age` = 25 ``

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.

4.5.11 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.

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

4.5.12 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.

```
<?php
$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.

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

4.6 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.

```
<?php
$people = ORM::for_table('person')->where('gender', 'female')->group_by('name')->find_
↪many();
```

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

```
<?php
$people = ORM::for_table('person')->where('gender', 'female')->group_by_expr("FROM_
↪UNIXTIME(`time`, '%Y-%m')")->find_many();
```

4.7 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 Titi. Substitute `where_` for `having_` to make use of these functions.

For example:

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

4.8 Result columns

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

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

Will result in the query:

```
<?php
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.

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

Will result in the query:

```
<?php
SELECT `name`, `age` FROM `person`;
```

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

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

Will result in the query:

```
<?php
SELECT `name` AS `person_name` FROM `person`;
```

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

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

Will result in the query:

```
<?php
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.

```
<?php
// 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:

```
<?php
SELECT COUNT(*) AS `count` FROM `person`;
```

4.8.1 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:

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

Will result in the query:

```
<?php
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):

```
<?php
$people = ORM::for_table('person')->select_many(array('first_name' => 'name'), 'age',
↪'height')->find_many();
```

Will result in the query:

```
<?php
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:

```
<?php
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:

```
<?php
$people = ORM::for_table('person')->select_many('name', 'age', 'height')->select_expr(
↪'NOW()', 'timestamp')->find_many();
```

Will result in the query:

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

4.9 DISTINCT

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

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

This will result in the query:

```
<?php
SELECT DISTINCT `name` FROM `person`;
```

4.10 Joins

Titi 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:

```
<?php

$results = ORM::for_table('person')->join('person_profile', array('person.id', '=',
↪'person_profile.person_id'))->find_many();
```

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.

```
<?php
// Not recommended because the join condition will not be escaped.
$results = ORM::for_table('person')->join('person_profile', 'person.id = person_
↪profile.person_id')->find_many();
```

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.

```
<?php
$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();
```

4.10.1 Raw JOIN clauses

If you need to construct a more complex query, you can use the `raw_join` method to specify the SQL fragment for the JOIN clause exactly. This method takes four required arguments: the string to add to the query, the conditions is as an *array* containing three components: the first column, the operator, and the second column, the table alias and (optional) the parameters array. 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 `*_join` 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 JOIN clauses.

```
<?php
$people = ORM::for_table('person')
            ->raw_join(
                'JOIN (SELECT * FROM role WHERE role.name = ?)',
                array('person.role_id', '=', 'role.id'),
                'role',
                array('role' => 'janitor'))
            ->order_by_asc('person.name')
            ->find_many();
```

(continues on next page)

(continued from previous page)

```
// Creates SQL:
SELECT * FROM `person` JOIN (SELECT * FROM role WHERE role.name = 'janitor') `role`␣
↪ON `person`.`role_id` = `role`.`id` ORDER BY `person`.`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.

4.11 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.

```php
<?php
$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.

4.12 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.

```php
<?php
$people = ORM::for_table('person')->raw_query('SELECT p.* FROM person p JOIN role r␣
↪ON p.role_id = r.id WHERE r.name = :role', array('role' => 'janitor'))->find_many();
```

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: Using `raw_query` is advanced and possibly dangerous, and Titi 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 Titi. Consider using a more full-featured database abstraction system.

4.12.1 Raw SQL execution using PDO

Warning: By using this function you're dropping down to PHPs PDO directly. Titi does not make any attempt to protect you from making errors when using this method.

You're essentially just using Titi to manage the connection and configuration when you implement `raw_execute()`.

It can be handy, in some instances, to make use of the PDO instance underneath Titi to make advanced queries. These can be things like dropping a table from the database that Titi doesn't support and will not support in the future. These are operations that fall outside the 80/20 philosophy of Titi. That said there is a lot of interest in this function and quite a lot of support requests related to it.

This method directly maps to PDOStatement::execute() underneath so please familiarise yourself with it's documentation.

Dropping tables

This can be done very simply using `raw_execute()`.

```
<?php
if (ORM::raw_execute('DROP TABLE my_table')) {
    echo "Table dropped";
} else {
    echo "Drop query failed";
}
```

Selecting rows

Warning: You really, should not be doing this, use Titi with `raw_query()` instead where possible.

Here is a simple query implemented using `raw_execute()` - note the call to `ORM::get_last_statement()` as `raw_execute()` returns a boolean as per the PDOStatement::execute() underneath.

```
<?php
$res = ORM::raw_execute('SHOW TABLES');
$statement = ORM::get_last_statement();
$rows = array();
while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
    var_dump($row);
}
```

It is also worth noting that `$statement` is a `PDOStatement` instance so calling its `fetch()` method is the same as if you had called against PDO without Titi being involved.

4.12.2 Getting the PDO instance

Warning: By using this function you're dropping down to PHP's PDO directly. Titi does not make any attempt to protect you from making errors when using this method.

You're essentially just using Titi to manage the connection and configuration when you implement against `get_db()`.

If none of the preceeding methods suit your purposes then you can also get direct access to the PDO instance underneath Titi using `ORM::get_db()`. This will return a configured instance of PDO.

```php
<?php
$pdo = ORM::get_db();
foreach($pdo->query('SHOW TABLES') as $row) {
    var_dump($row);
}
```

4.13 Querying with Models

Querying allows you to select data from your database and populate instances of your model classes. Queries start with a call to a static *factory method* on the base `Model` class that takes a single argument: the name of the model class you wish to use for your query. This factory method is then used as the start of a *method chain* which gives you full access to Titi's fluent query API.

```php
<?php
$users = Model::factory('User')
    ->where('name', 'Fred')
    ->where_gte('age', 20)
    ->find_many();
```

You can also use the same shortcut provided by Titi when looking up a record by its primary key ID:

```php
<?php
$user = Model::factory('User')->find_one($id);
```

If you are using PHP 5.3+ you can also do the following:

```php
<?php
$users = User::where('name', 'Fred')
    ->where_gte('age', 20)
    ->find_many();
```

This does the same as the example above but is shorter and more readable.

The only differences between using the ORM class and the Model classes for querying are as follows:

1. You do not need to call the `for_table` method to specify the database table to use. The Model class will do this automatically based on the class name (or the `$_table` static property, if present).
2. The `find_one` and `find_many` methods will return instances of *your model subclass*, instead of the base `ORM` class. Like ORM, `find_one` will return a single instance or `false` if no rows matched your query, while `find_many` will return an array of instances, which may be empty if no rows matched.
3. Custom filtering, see next section.

You may also retrieve a count of the number of rows returned by your query. This method behaves exactly like ORM's `count` method:

```php
<?php
$count = Model::factory('User')->where_lt('age', 20)->count();
```

The model instances returned by your queries now behave exactly as if they were instances of Titi's raw `ORM` class.

You can access data:

```php
<?php
$user = Model::factory('User')->find_one($id);
echo $user->name;
```

Update data and save the instance:

```php
<?php
$user = Model::factory('User')->find_one($id);
$user->name = 'Paris';
$user->save();
```

To create a new (empty) instance, use the `create` method:

```php
<?php
$user = Model::factory('User')->create();
$user->name = 'Paris';
$user->save();
```

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

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

You can also use database expressions when setting values on your model:

```php
<?php
$user = Model::factory('User')->find_one($id);
$user->name = 'Paris';
$user->set_expr('last_logged_in', 'NOW()');
$user->save();
```

Of course, because these objects are instances of your base model classes, you can also call methods that you have defined on them:

```php
<?php
class User extends Model {
    public function full_name() {
        return $this->first_name . ' ' . $this->last_name;
    }
}

$user = Model::factory('User')->find_one($id);
echo $user->full_name();
```

To delete the database row associated with an instance of your model, call its `delete` method:

```php
<?php
$user = Model::factory('User')->find_one($id);
$user->delete();
```

You can also get the all the data wrapped by a model subclass 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.

```php
<?php
class Person extends Model {
}

$person = Model::factory('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');
```

CHAPTER 5

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:

```php
<?php
$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.

```php
<?php
$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:

```php
<?php
$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.

```php
<?php
$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.

```php
<?php
$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.

```
<?php
$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:

```
<?php
$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.

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

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

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

5.6 Model classes

You can also create a model class for each entity in your application. For example, if you are building an application that requires users, you should create a `User` class. Your model classes should extend the base `Model` class:

```
<?php
class User extends Model {
}
```

The base class takes care of creating instances of your model classes, and populating them with *data* from the database. You can then add *behaviour* to this class in the form of public methods which implement your application logic. This combination of data and behaviour is the essence of the Active Record pattern.

5.6.1 IDE Auto-complete

As the model does not require you to specify a method/function per database column it can be difficult to know what properties are available on a particular model. Due to the magic nature of PHP's __get() method it is impossible for an IDE to give you autocomplete hints as well.

To work around this you can use PHPDoc comment blocks to list the properties of the model. These properties should mirror the names of your database tables columns.

```php
<?php
/**
 * @property int $id
 * @property string $first_name
 * @property string $last_name
 * @property string $email
 */
class User extends Model {
}
```

For more information please see the PHPDoc manual @property documentation.

5.7 Database tables

Your `User` class should have a corresponding `user` table in your database to store its data.

By default, models assume your class names are in *CapWords* style, and your table names are in *lowercase_with_underscores* style. It will convert between the two automatically. For example, if your class is called `CarTyre`, the model will look for a table named `car_tyre`.

If you are using namespaces then they will be converted to a table name in a similar way. For example `\Models\CarTyre` would be converted to `models_car_tyre`. Note here that backslashes are replaced with underscores in addition to the *CapWords* replacement discussed in the previous paragraph.

To disregard namespace information when calculating the table name, set `Model::$short_table_names = true;`. Optionally this may be set or overridden at class level with the **public static** property `$_table_use_short_name`. The

`$_table_use_short_name` takes precedence over `Model::$short_table_names` unless `$_table_use_short_name` is `null` (default).

Either setting results in `\Models\CarTyre` being converted to `car_tyre`.

```php
<?php
class User extends Model {
    public static $_table_use_short_name = true;
}
```

To override the default naming behaviour and directly specify a table name, add a **public static** property to your class called `$_table`:

```php
<?php
class User extends Model {
    public static $_table = 'my_user_table';
}
```

5.7.1 Auto prefixing

To save having type out model class name prefixes whenever code utilises `Model::for_table()` it is possible to specify a prefix that will be prepended onto the class name.

See the *Configuration* documentation for more details.

5.8 ID column

Models require that your database tables have a unique primary key column. By default, the model will use a column called `id`. To override this default behaviour, add a **public static** property to your class called `$_id_column`:

```
<?php
class User extends Model {
    public static $_id_column = 'my_id_column';
}
```

Note - The Model class has its *own* default ID column name mechanism, and does not respect column names specified in ORM's configuration.

CHAPTER 6

Associations

Titi Models provide a simple API for one-to-one, one-to-many and many-to-many relationships (associations) between models. It takes a different approach to many other ORMs, which use associative arrays to add configuration metadata about relationships to model classes. These arrays can often be deeply nested and complex, and are therefore quite error-prone.

Instead, these models treat the act of querying across a relationship as a *behaviour*, and supplies a family of helper methods to help generate such queries. These helper methods should be called from within *methods* on your model classes which are named to describe the relationship. These methods return ORM instances (rather than actual Model instances) and so, if necessary, the relationship query can be modified and added to before it is run.

6.1 Summary

The following list summarises the associations provided by models, and explains which helper method supports each type of association:

6.1.1 One-to-one

Use `has_one` in the base, and `belongs_to` in the associated model.

6.1.2 One-to-many

Use `has_many` in the base, and `belongs_to` in the associated model.

6.1.3 Many-to-many

Use `has_many_through` in both the base and associated models.

Below, each association helper method is discussed in detail.

6.2 Has-one

One-to-one relationships are implemented using the `has_one` method. For example, say we have a `User` model. Each user has a single `Profile`, and so the `user` table should be associated with the `profile` table. To be able to find the profile for a particular user, we should add a method called `profile` to the `User` class (note that the method name here is arbitrary, but should describe the relationship). This method calls the protected `has_one` method provided by the model, passing in the class name of the related object. The `profile` method should return an ORM instance ready for (optional) further filtering.

```php
<?php
class Profile extends Model {
}

class User extends Model {
    public function profile() {
        return $this->has_one('Profile');
    }
}
```

The API for this method works as follows:

```php
<?php
// Select a particular user from the database
$user = Model::factory('User')->find_one($user_id);

// Find the profile associated with the user
$profile = $user->profile()->find_one();
```

By default, models assume that the foreign key column on the related table has the same name as the current (base) table, with `_id` appended. In the example above, the model will look for a foreign key column called `user_id` on the table used by the `Profile` class. To override this behaviour, add a second argument to your `has_one` call, passing the name of the column to use.

In addition, models assume that the foreign key column in the current (base) table is the primary key column of the base table. In the example above, the model will use the column called `user_id` (assuming `user_id` is the primary key for the user table) in the base table (in this case the user table) as the foreign key column in the base table. To override this behaviour, add a third argument to your `has_one call`, passing the name of the column you intend to use as the foreign key column in the base table.

6.3 Has many

One-to-many relationships are implemented using the `has_many` method. For example, say we have a `User` model. Each user has several `Post` objects. The `user` table should be associated with the `post` table. To be able to find the posts for a particular user, we should add a method called `posts` to the `User` class (note that the method name here is arbitrary, but should describe the relationship). This method calls the protected `has_many` method provided by the model, passing in the class name of the related objects. **Pass the model class name literally, not a pluralised version**. The `posts` method should return an ORM instance ready for (optional) further filtering.

```php
<?php
class Post extends Model {
}

class User extends Model {
    public function posts() {
```

(continues on next page)

(continued from previous page)

```
        return $this->has_many('Post'); // Note we use the model name literally - not␣
↪a pluralised version
    }
}
```

The API for this method works as follows:

```
<?php
// Select a particular user from the database
$user = Model::factory('User')->find_one($user_id);

// Find the posts associated with the user
$posts = $user->posts()->find_many();
```

By default, models assume that the foreign key column on the related table has the same name as the current (base) table, with `_id` appended. In the example above, the model will look for a foreign key column called `user_id` on the table used by the `Post` class. To override this behaviour, add a second argument to your `has_many` call, passing the name of the column to use.

In addition, models assume that the foreign key column in the current (base) table is the primary key column of the base table. In the example above, the model will use the column called `user_id` (assuming `user_id` is the primary key for the user table) in the base table (in this case the user table) as the foreign key column in the base table. To override this behaviour, add a third argument to your `has_many call`, passing the name of the column you intend to use as the foreign key column in the base table.

6.4 Belongs to

The 'other side' of `has_one` and `has_many` is `belongs_to`. This method call takes identical parameters as these methods, but assumes the foreign key is on the *current* (base) table, not the related table.

```
<?php
class Profile extends Model {
    public function user() {
        return $this->belongs_to('User');
    }
}

class User extends Model {
}
```

The API for this method works as follows:

```
<?php
// Select a particular profile from the database
$profile = Model::factory('Profile')->find_one($profile_id);

// Find the user associated with the profile
$user = $profile->user()->find_one();
```

Again, models make an assumption that the foreign key on the current (base) table has the same name as the related table with `_id` appended. In the example above, the model will look for a column named `user_id`. To override this behaviour, pass a second argument to the `belongs_to` method, specifying the name of the column on the current (base) table to use.

Models also make an assumption that the foreign key in the associated (related) table is the primary key column of the related table. In the example above, the model will look for a column named `user_id` in the user table (the related table in this example). To override this behaviour, pass a third argument to the belongs_to method, specifying the name of the column in the related table to use as the foreign key column in the related table.

6.5 Has many through

Many-to-many relationships are implemented using the `has_many_through` method. This method has only one required argument: the name of the related model. Supplying further arguments allows us to override default behaviour of the method.

For example, say we have a `Book` model. Each `Book` may have several `Author` objects, and each `Author` may have written several `Books`. To be able to find the authors for a particular book, we should first create an intermediary model. The name for this model should be constructed by concatenating the names of the two related classes, in alphabetical order. In this case, our classes are called `Author` and `Book`, so the intermediate model should be called `AuthorBook`.

We should then add a method called `authors` to the `Book` class (note that the method name here is arbitrary, but should describe the relationship). This method calls the protected `has_many_through` method provided by the model, passing in the class name of the related objects. **Pass the model class name literally, not a pluralised version**. The `authors` method should return an ORM instance ready for (optional) further filtering.

```php
<?php
class Author extends Model {
    public function books() {
        return $this->has_many_through('Book');
    }
}

class Book extends Model {
    public function authors() {
        return $this->has_many_through('Author');
    }
}

class AuthorBook extends Model {
}
```

The API for this method works as follows:

```php
<?php
// Select a particular book from the database
$book = Model::factory('Book')->find_one($book_id);

// Find the authors associated with the book
$authors = $book->authors()->find_many();

// Get the first author
$first_author = $authors[0];

// Find all the books written by this author
$first_author_books = $first_author->books()->find_many();
```

6.5.1 Overriding defaults

The `has_many_through` method takes up to six arguments, which allow us to progressively override default assumptions made by the method.

First argument: associated model name - this is mandatory and should be the name of the model we wish to select across the association.

Second argument: intermediate model name - this is optional and defaults to the names of the two associated models, sorted alphabetically and concatenated.

Third argument: custom key to base table on intermediate table - this is optional, and defaults to the name of the base table with `_id` appended.

Fourth argument: custom key to associated table on intermediate table - this is optional, and defaults to the name of the associated table with `_id` appended.

Fifth argument: foreign key column in the base table - this is optional, and defaults to the name of the primary key column in the base table.

Sixth argument: foreign key column in the associated table - this is optional, and defaults to the name of the primary key column in the associated table.

CHAPTER 7

Filters

It is often desirable to create reusable queries that can be used to extract particular subsets of data without repeating large sections of code. Models allow this by providing a method called `filter` which can be chained in queries alongside the existing ORM query API. The filter method takes the name of a **public static** method on the current Model subclass as an argument. The supplied method will be called at the point in the chain where `filter` is called, and will be passed the `ORM` object as the first parameter. It should return the ORM object after calling one or more query methods on it. The method chain can then be continued if necessary.

It is easiest to illustrate this with an example. Imagine an application in which users can be assigned a role, which controls their access to certain pieces of functionality. In this situation, you may often wish to retrieve a list of users with the role 'admin'. To do this, add a static method called (for example) `admins` to your Model class:

```php
<?php
class User extends Model {
    public static function admins($orm) {
        return $orm->where('role', 'admin');
    }
}
```

You can then use this filter in your queries:

```php
<?php
$admin_users = Model::factory('User')->filter('admins')->find_many();
```

You can also chain it with other methods as normal:

```php
<?php
$young_admins = Model::factory('User')
                    ->filter('admins')
                    ->where_lt('age', 18)
                    ->find_many();
```

7.1 Filters with arguments

You can also pass arguments to custom filters. Any additional arguments passed to the `filter` method (after the name of the filter to apply) will be passed through to your custom filter as additional arguments (after the ORM instance).

For example, let's say you wish to generalise your role filter (see above) to allow you to retrieve users with any role. You can pass the role name to the filter as an argument:

```php
<?php
class User extends Model {
    public static function has_role($orm, $role) {
        return $orm->where('role', $role);
    }
}

$admin_users = Model::factory('User')->filter('has_role', 'admin')->find_many();
$guest_users = Model::factory('User')->filter('has_role', 'guest')->find_many();
```

These examples may seem simple (`filter('has_role', 'admin')` could just as easily be achieved using `where('role', 'admin')`), but remember that filters can contain arbitrarily complex code - adding `raw_where` clauses or even complete `raw_query` calls to perform joins, etc. Filters provide a powerful mechanism to hide complexity in your model's query API.

CHAPTER 8

Transactions

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

```
<?php
// 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.

CHAPTER 9

A word on validation

It's generally considered a good idea to centralise your data validation in a single place, and a good place to do this is inside your model classes. This is preferable to handling validation alongside form handling code, for example. Placing validation code inside models means that if you extend your application in the future to update your model via an alternative route (say a REST API rather than a form) you can re-use the same validation code.

Despite this, Titi doesn't provide any built-in support for validation. This is because validation is potentially quite complex, and often very application-specific. Titi is deliberately quite ignorant about your actual data - it simply executes queries, and gives you the responsibility of making sure the data inside your models is valid and correct. Adding a full validation framework to Titi would probably require more code than Titi itself!

However, there are several simple ways that you could add validation to your models without any help from Titi. You could override the `save()` method, check the data is valid, and return `false` on failure, or call `parent::save()` on success. You could create your own subclass of the `Model` base class and add your own generic validation methods. Or you could write your own external validation framework which you pass model instances to for checking. Choose whichever approach is most suitable for your own requirements.

CHAPTER 10

Multiple Connections

Titi can work 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.

The connection to use can be specified in two separate ways. To indicate a default connection key for a subclass of `Model`, create a public static property in your model class called `$_connection_name`.

```php
<?php
// 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);

// A named connection, where 'alternate' is an arbitray key name
ORM::configure('sqlite:./example2.db', null, 'alternate');

class SomeClass extends Model
{
```

(continues on next page)

(continued from previous page)

```
    public static $_connection_name = 'alternate';
}
```

The connection to use can also be specified as an optional additional parameter to `OrmWrapper::for_table()`, or to `Model::factory()`. This will override the default setting (if any) found in the `$_connection_name` static property.

```
<?php
$person = Model::factory('Author', 'alternate')->find_one(1);  // Uses connection␣
↪named 'alternate'
```

The connection can be changed after a model is populated, should that be necessary:

```
<?php

$person = Model::factory('Author')->find_one(1);     // Uses default connection
$person->orm = Model::factory('Author', 'alternate');  // Switches to connection␣
↪named 'alternate'
$person->name = 'Foo';
$person->save();                                     // *Should* now save through the␣
↪updated connection
```

10.1 Notes

- **There is no support for joins across connections**
- As the Model methods `has_one`, `has_many` and `belongs_to` don't require joins, these *should* work as expected, even when the objects on opposite sides of the relation belong to diffrent connections. The `has_many_through` relationship requires joins, and so will not reliably work across different 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()`.
- `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.

10.1.1 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.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search