Paris Documentation

Release

Jamie Matthews and Simon Holywell

Mar 21, 2017

Contents

Contents:

CHAPTER 1

Philosophy

Paris is built with the same *less is more* philosophy as Idiorm.

CHAPTER 2

Installation

Packagist

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

Please see the Packagist documentation for further information.

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

Setup

Paris requires Idiorm. Install Idiorm and Paris somewhere in your project directory, and `require` both.

```
<?php
require_once 'your/path/to/idiorm.php';
require_once 'your/path/to/paris.php';
```

Then, you need to tell Idiorm how to connect to your database. **For full details of how to do this, see 'Idiorm's documentation'_.**

Briefly, you need to pass a *Data Source Name* connection string to the `configure` method of the ORM class.

```
<?php
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:

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

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 Paris 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.

Model prefixing

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 {

    }
}
```

Further Configuration

The only other configuration options provided by Paris itself are the `$_table` and `$_id_column` static properties on model classes. To configure the database connection, you should use Idiorm'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.

See 'Idiorm's documentation'_ for full details.

Query logging

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

```
<?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.

CHAPTER 4

Models

Model classes

You should 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 {
}
```

Paris 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.

IDE Auto-complete

As Paris 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
/**
 * @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.

Database tables

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

By default, Paris assumes 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`, Paris 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
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
class User extends Model {
    public static $_table = 'my_user_table';
}
```

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.

ID column

Paris requires that your database tables have a unique primary key column. By default, Paris 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 - Paris has its *own* default ID column name mechanism, and does not respect column names specified in Idiorm's configuration.

CHAPTER 5

Associations

Paris provides 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, Paris treats 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.

Summary

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

One-to-one

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

One-to-many

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

Many-to-many

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

Below, each association helper method is discussed in detail.

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 Paris, passing in the class name of the related object. The `profile` method should return an ORM instance ready for (optional) further filtering.

```
<?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
// 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, Paris assumes 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, Paris 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, Paris assumes that the foreign key column in the current (base) table is the primary key column of the base table. In the example above,

Paris 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.

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 Paris, 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
class Post extends Model {
}

class User extends Model {
```

```
    public function posts() {
        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, Paris assumes 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, Paris 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, Paris assumes that the foreign key column in the current (base) table is the primary key column of the base table. In the example above, Paris 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.

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, Paris makes 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, Paris 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.

Paris also makes an assumption that the foreign key in the associated (related) table is the primary key column of the related table. In the example above, Paris 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.

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 Paris, 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
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
// 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();
```

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 6

Querying

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 Idiorm's fluent query API. **See Idiorm's documentation for details of this API.**

For example:

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

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

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

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

```
<?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 Idiorm and using Paris for querying are as follows:

1. You do not need to call the `for_table` method to specify the database table to use. Paris will supply 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 Idiorm, `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 Idiorm's `count` method:

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

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
// documented and default style
$count = Model::factory('User')->where_lt('age', 20)->find_one();

// PSR-1 compliant style
$count = Model::factory('User')->whereLt('age', 20)->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.

Getting data from objects, updating and inserting data

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

You can access data:

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

Update data and save the instance:

```
<?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 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. Paris allows this by providing a method called `filter` which can be chained in queries alongside the existing Idiorm 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
class User extends Model {
    public static function admins($orm) {
        return $orm->where('role', 'admin');
    }
}
```

You can then use this filter in your queries:

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

You can also chain it with other methods as normal:

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

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
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

Paris (or Idiorm) 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, Paris doesn't provide any built-in support for validation. This is because validation is potentially quite complex, and often very application-specific. Paris 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 Paris would probably require more code than Paris itself!

However, there are several simple ways that you could add validation to your models without any help from Paris. 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

Migrations

Paris does not have native support for migrations, but some work has been done to integrate PHPMig. If you want to have migrations in your project then this is recommended route as Paris will never have migrations directly implemented in the core. Please refer to the Paris and Idiorm Philosophy for reasons why.

To integrate Paris with PHPMig you will need to follow their installation instructions and then configure it to use the Paris PDO instance:

```
<?php
$container['db'] = $container->share(function(){
    return ORM::get_db();
});
$container['phpmig.adapter'] = $container->share(function() use ($container) {
    return new Adapter\PDO\Sql($container['db'], 'migrations');
});
```

CHAPTER 11

Multiple Connections

Paris now works with multiple database conections (and necessarily relies on an updated version of Idiorm that also supports multiple connections). Database connections are identified by a string name, and default to `OrmWrapper::DEFAULT_CONNECTION` (which is really `ORM::DEFAULT_CONNECTION`).

See Idiorm's documentation for information about configuring multiple connections.

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

class SomeClass extends Model
{
    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
```

Queries across multiple connections are not supported. However, as the Paris 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.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search