Pheasant Documentation

Release 1.1.0

Lachlan Donald, Richard Bone

May 19, 2016

Contents

Pheasant is a bare-bones data mapper written specifically for PHP 5.3.1+ and MySQL. Rather than focus on abstraction and rich feature sets, Pheasant focuses on a performance and simplicity.

Despite it's small foot-print (<5k LOC), Pheasant packs a punch and offers:

- Easy persistence for objects: create, update and delete
- Property mapping for common types
- Relationships: One-to-one, One-to-many
- Fluid querying
- Events system for extension

Pheasant is Open Source and licensed under the BSD License.

CHAPTER 1

Contents

1.1 Getting Started

This document will get you up and running with a basic Pheasant environment. You'll need:

- MySQL 5.1+
- PHP 5.3.1+
- Composer

1.1.1 Installing

The easiest way to install Pheasant in your project is to use Composer and include the following in your *composer.json* file:

```
{
    "require": {
        "lox/pheasant": "1.1.*"
    }
}
```

After adding pheasant as a dependency of your project, update with *composer update* and you'll have a working pheasant environment.

See Packagist for more details.

1.1.2 Loading and Connecting

Pheasant uses the class loading mechanism in Composer, which is basically a *PSR-0* loader.

```
<?php

// use composer's autoload mechanism
require_once('vendor/autoload.php');

// setup database connections
Pheasant::setup('mysql://user:pass@localhost:3306/mydb');
```

1.1.3 Defining Objects

Anything mapped to the database is referred to as a *Domain Object* in Pheasant. The easiest way to define properties on a domain object is to simply extend *PheasantDomainObject* and implement a *properties()* method.

In the absence of an explicit definition, the tablename is inferred from the classname: *post*.

```php
<?php

use \Pheasant\Types;

class Post extends \Pheasant\DomainObject
{
  public function properties()
  {
    return array(
      'postid'    => new Types\IntegerType(11, 'primary auto_increment'),
      'title'     => new Types\StringType(255, 'required'),
      'type'      => new Types\StringType(128, 'default=blog'),
      'timestamp' => new Types\DateTimeType(),
    );
  }
}
```

Properties are mapped to the same named column in the database. See `mapping/types` for more about what types are available and their parameters.

1.1.4 Creating Tables

Pheasant tends to stay out of the way when it comes to creating and altering tables, but it does provide a minimal helper for the creation of tables:

```php
<?php

$migrator = new \Pheasant\Migrate\Migrator();
$migrator->create('post', Post::schema());
```

Alternately, DomainObjects provide a tableName() method that returns the table name as a string.

1.1.5 Saving and Updating

Whilst Pheasant uses the data mapper pattern, for convenience domain objects have activerecord-like helpers:

```php
<?php

$post = new Post();
$post->title = "The joys of llama farming";
$post->timestamp = new \DateTime('2013-01-01');
$post->save();
```

Simple as that. Subsequent changes will update the record with whatever columns have been changed.

1.1.6 Finding

The core of Pheasant's finder capability is based around *find()* and *one()*. Find returns a *Collection*, where one returns a single object.

See `finding` for more details.

```
<?php

// by identifier
$post = Post::byId(1);

// using a magic finder method
$posts = Post::findByTitleAndTimestamp('The joys of llama farming', '2013-01-01');
$posts = Post::findByType(array('blog', 'article'));

// by insertion order
$post = Post::find()->latest();

// paged, 1 - 100
$post = Post::find()->limit(1,100);
```

If you prefer direct sql, that works too and correctly handles binding *null* and *array* parameter. Note that *?* is used for variable interpolation:

```
<?php

// using SQL interpolation
$post = Post::find('title LIKE ?', '%llama%');

// automatic IN binding
$posts = Post::find('type IN ?', array('blog', 'article'));
```

1.1.7 Relationships

An object defines what objects relate to it in the *relationships()* method.

See `relationships` for more details.

```
<?php

use \Pheasant;
use \Pheasant\Types;

class Post extends DomainObject
{
    public function properties()
    {
        return array(
            'postid'    => new Types\IntegerType(11, 'primary auto_increment'),
            'title'     => new Types\StringType(255, 'required'),
            'type'      => new Types\StringType(128, 'default=blog'),
            'timestamp' => new Types\DateTimeType(),
            'authorid'  => new Types\IntegerType(11)
        );
    }

    public function relationships()
    {
        return array(
            'Author' => Author::hasOne('authorid');
            );
    }
}

class Author extends DomainObject
{
    public function properties()
    {
        return array(
            'authorid' => new Types\IntegerType(11, 'primary auto_increment'),
            'fullname' => new Types\StringType(255, 'required')
            );
    }

    public function relationships()
    {
        return array(
            'Posts' => Post::hasOne('authorid')
            );
    }
}

// create some objects
$author = new Author(array('fullname'=>'Lachlan'));
$post = new Post(array('title'=>'My Post', 'author'=>$author));

// save objects
$author->save();
$post->save();

echo $post->title; // returns 'My Post'
echo $post->Author->fullname; // returns 'Lachlan'
```

Pheasant supports one-to-one, and one-to-many relationship types.

1.2 Configuration

1.2.1 Autoloading

Pheasant relies on a *PSR-0* class-loader, such as the one in Composer.

```
<?php

// use composer's autoload mechanism
require_once('vendor/autoload.php');
```

1.2.2 MySQL Connection

The simplest way to configure Pheasant is to use static *setup()* method:

```
<?php

Pheasant::setup('mysql://user:pass@localhost:3306/mydb');
```

Note: It's possible to pass in query parameters to this DSN for setting connection parameters:

```php
<?php

Pheasant::setup('mysql://user:pass@localhost:3306/mydb?charset=utf8&strict=true');
```

Supported parameters are:

charset The character set to set for the connection, defaults to utf8

timezone Any mysql timezone string, defaults to UTC

strict Set strict mode on the connection. This is generally not a good idea, as it creates issues with binding.

Multiple Connections

At present domain objects can only use the *default* connection, but you can setup and access other connections via the connection manager:

```php
<?php

$pheasant = Pheasant::instance();

// define the connections
$pheasant->connections()->addConnection('mydb1', 'mysql://user:pass@localhost:3306/mydb');
$pheasant->connections()->addConnection('mydb2', 'mysql://user:pass@localhost:3306/another');

// look them up
$pheasant->connection('mydb1')->execute('SELECT * FROM mytable');
```

Allowing Pheasant objects to use different connections is a feature that is under development.

PSR-0 A standard for for interoperable PHP autoloaders See https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md

1.3 Domain Objects

The domain object represents an object that can be persisted in Pheasant.

1.3.1 Initialization

A domain object has a *Schema* instance that defines everything about it including:

- Properties
- Relationships
- Custom getter/setters
- Events

The top-level *Pheasant* instance stores a mapping of class names to *Schema* instances. Whenever a domain object is instantiated or a static-method is called on one, the schema instance is checked. If one doesn't exist, it's initialized.

The initialization happens in *DomainObject::initialize*, for which the default implementation is to look for several template methods in the object.

properties() A map of column names to Type objects (see *mapping/types*)

relationships() A map of keys to RelationType objects representing 1-n or 1-1 relationships (see *mapping/relationships*)

tableName() The database table name to map to, defaults to the name of the class

mapper() The mapper instance to use, defaults to the *mapping/rowmapper*.

1.3.2 Property Access

Once properties have been defined in an objects schema, they are available via property access for read and write.

```php
<?php

$post = new Post();
$post->title = 'Test Post';
$post->save();

echo $post->title; // shows 'Test Post'

$post->title = 'Updated Title';
$post->save();
```

Note: You can see which properties have been changed on a domain object in-between saves using the *changes()* method.

Calling save on an unchanged object won't do anything.

1.3.3 Identity

A domain object has some sort of primary key. This is exposed within the domain object as an *Identity*. This object can be easily converted into a *Criteria* object for locating the object.

Any property that is either a *Sequence* or is defined with the *primary* option is considered part of the *Identity*. Composite keys are supported.

1.3.4 Constructors

The default constructor for a domain object allows for an array of key/values to be passed in:

```php
<?php

$post = new Post(array('title'=>'Test Post'));
$post->save();
```

If you want to have a different constructor for your domain object, you must override the *construct()* method, as the actual *__construct()* method is final to ensure it's always available.

```php
<?php

class Post extends DomainObject
{
    public function construct($title)
    {
      $this->set('title', $title);
    }
}

$post = new Post('Test Post');
echo $post->title; // shows 'Test Post'
```

1.3.5 Inheritance

Inheritance and extending domain objects isn't something that has any explicit support, although it would certainly be possible to override the *properties* method and extend it.

Index