PowerOrm Documentation

Release 1.0-beta.1

Eddilbert Macharia

Nov 05, 2018

Contents

A set of Components that easily integrate to you php project.

Note: You can get a Demo App that show this componets in use.

Visit a Live Demo of the App that shows this componets in use.

CHAPTER 1

The Orm

Powerorm is an object-relational mapper in which you describe your database layout in PHP code.

CHAPTER 2

Form

Dealing with HTML forms is one of the most common - and challenging - tasks for a web developer. *Powerform* component makes dealing with forms easy.

CHAPTER 3

Debugbar

Debugbar Displays a debug bar in the browser with information from php.

CHAPTER 4

Data Faker

Data Faker Generate dummy data for powerorm models.

CHAPTER 5

PhpGis

PhpGis Makes the powerorm work with gis data.

CHAPTER 6

Extra Information

6.1 PowerOrm User Guide

Note: This documentation is mostly for PowerOrm(1.1.0).

You can get a Demo App that show this componets in use.

Visit a Live Demo of the App that shows this componets in use.

For assistance post your questions here Powerorm Help

6.1.1 Overview

Introduction

Powerorm at a glance

This is an informal overview of how to write a database-driven Web app with Powerorm. The goal of this document is to give you enough technical specifics to understand how Powerorm works.

This assumes you have already *Installed* the orm

Design your model

Powerorm is an object-relational mapper in which you describe your database layout in PHP code.

The data-model syntax offers many rich ways of representing your models – so far.

What all this means is if an app requires a database table called role with two columns name and code. Instead of creating this table manually on database server. The orm can do this for us.

We tell the orm of our intentions by creating a *Model* which outlines the table, columns to be on that table and anyother information needed on the table.

For the table role outlined above, The model for that table would look as below

```
// application/models/Role.php

use Eddmash\PowerOrm\Model\Model;

class Role extends Model
{
    public function unboundFields() {
        return [
            'name' => Model::CharField(['maxLength' => 40, 'dbIndex' => true]),
            'code' => Model::CharField(['maxLength' => 10, 'dbIndex' => true]),
        ];
    }

}
```

Create Table

So far we have created a model that represents a role table, but the actual table does not exist yet.

Note: The advantage of having the orm create tables for is that it will keep track of any changes made to the models, hence we can always undo or redo any changes we have made to the models reflected onto the database. This is made possible by *Migration*

Now to create the table that represents the Role model on the database.

Run this command to have orm keep track of the models state by detecting any changes made.

```
$ php pmanager.php makemigrations
```

Run this *command* to have the orm apply the changes on the database.

```
$ php pmanager.php migrate
```

The migrate *command* looks at all your available models and creates tables in your database for whichever tables don't already exist, as well as optionally providing much richer schema control.

If your project is based on an framework *Integrations* for how to access Powerorm *command* line utility.

Persisting Objects to the Database

Now that you have mapped the Role model to its corresponding role table, you're ready to persist Role objects to the database.

```
$role = new Role();
$role->name = "test role";
$role->code = "test_role";
$role->save();
var_dump("saved ".$role->id);
```

Fetching Objects from the Database

Fetching an object back out of the database is even easier.

When you query for a particular type of object, you always use a "manager". You can think of a manager as a PHP class whose only job is to help you fetch models of a certain class. You can access the manager object for a model class via the objects() method:

```
// fetch select * from role
$roles = Role::objects()->all();

foreach ($roles as $role) :
    echo $role->name."===>".$role->code."<br>";
endforeach;

// fetch select * from role where code = 'Et qui qui'
$roles = Role::objects()->filter(['code'=>"Et qui qui"]);

foreach ($roles as $role) :
    echo $role->name."===>".$role->code."<br>";
endforeach;

// fetch SELECT * FROM testing_role WHERE (code = a) AND (code = qu)
$roles = Role::objects()->filter(['code'=>"qu"], ["code"=>"a"]);

foreach ($roles as $role) :
    echo $role->name."===>".$role->code."<br>";
endforeach;

// fetch SELECT * FROM testing_role WHERE (code LIKE %v) AND (code LIKE e%)
$roles = Role::objects()->filter(['code__startswith'=>"e", "code__endswith"=>"v"]);

foreach ($roles as $role) :
    echo $role->name."===>".$role->code."<br>";
endforeach;

// fetch SELECT * FROM testing_role WHERE (code LIKE a%) OR (code LIKE %qu)
$roles = Role::objects()->filter(['code__endswith'=>"qu", "~code__startswith"=>"a"]);

foreach ($roles as $role) :
    echo $role->name."===>".$role->code."<br>";
endforeach;

// fetch select * from role where id in (1,2,3)
$roles = Role::objects()->filter(['id__in'=>[1,2,3]]);

foreach ($roles as $role) :
    echo $role->name."===>".$role->code."<br>";
endforeach;
```

Note: The filter method can take one/multiple arrays that contain the conditions to use when filtering.

Since its not possible to have the same array key repeated on the same array, use a second array to add more conditions for the same key.

For example to query roles based on the value of the **code** field.

This will work fine since the keys are different

```
Role::objects()->filter(["code"=>"admin_role", "~code"=>"user_role"])
```

This wont work as expected since the keys are the same

```
Role::objects()->filter(["code"=>"admin_role", "code"=>"user_role"])
```

Solve this by making another array.

```
Role::objects()->filter(["code"=>"admin_role"], ["code"=>"user_role"])
```

Setup Powerorm

Install

We assume your project structure looks like this

```
app_base
    ----vendor/
    ----composer.json
    ----config.php
    ----src
        ---- Models
        ---- Migrations
```

Via composer **(recommended)**:

```
composer require eddmash/powerorm
```

Or add this to the composer.json file:

```
"eddmash/powerorm": "1.1" (check latest version)
```

You could also Download or Clone package from github.

To rest of this guide assumes a plain php based project, no-frameworks used.

If you wish to see how to setup on specific frameworks please visit *Frameworks Integrations*

Create configuration

The orm requires a few *Configurations*, things like the database settings.

This will be added in the `config.php`

```
return [
    'database' => [
        'host' => '127.0.0.1',
        'dbname' => 'tester',
        'user' => 'admin',
        'password' => 'admin',
        'driver' => 'pdo_pgsql',
    ],
    'dbPrefix' => 'demo_',
    'charset' => 'utf-8',
];
```

Load the orm

To load powerorm use the following code and pass the Configs needed for powerorm to work.

```
$configs = require_once 'config.php';
\Eddmash\PowerOrm\Loader::webRun($configs);
```

This should be load as early as possible, Place it a the beginning of your script.

Create application

So far we have not told the orm anything about our project, to do this We need to create an *application class* the will be used by the orm to get information about your php project.

If your project is namespaced as App; Create a class the extends the `Eddmash\\PowerOrm\\Components\\AppComponent`. This class should be placed on the same level as your models, migration folders.

```
namespace App;

use Eddmash\PowerOrm\BaseOrm;
use Eddmash\PowerOrm\Components\AppComponent;

class App extends AppComponent
{

    public function ready(BaseOrm $baseOrm)
    {
    }

    public function getDbPrefix()
    {
        return "php_app";
    }
}
```

Technically this file can be place anywhere in your project tree, To get this flexibility you need to override :

- *Application::getMigrationsPath()* to tell the the orm where to find the models files and
- *Application::getMigrationsPath()* to tell the orm where to place generated migrations files.

Register Application

Once we have the projects application class, we need to register it with the orm.

To register we add the App class we have created above into our configurations under the *component configuration* setting as shown below.

```
$config = [
    'database' => [
        'host' => '127.0.0.1',
        'dbname' => 'tester',
        'user' => 'root',
        'password' => 'root1.',
        'driver' => 'pdo_mysql',
    ],
    'dbPrefix' => 'demo_',
    'charset' => 'utf-8',
    'timezone' => 'Africa/Nairobi',
    'components' => [
        App::class,
    ]
];
```

Create enable Powerorm on the command line

To be able to use the orm on the command line create a file on the same level as the **composer.json** file.

create file named **pmanager.php** and add the following .

```
use Eddmash\PowerOrm\Loader;
require_once 'vendor/autoload.php';
$configs = require_once 'config.php';
Loader::consoleRun($configs);
```

See all the available commands *commands*.

With that you ready.

Enjoy !

Configuration

The ORM takes several configurations

- **database** This are the database configurations the ORM will use to connect to a database.

  The orm uses Doctrine Dbal to connect to database.

  This means the orm will support most if not all the databases suported by Doctrine dbal.

  To learn more about the database drivers supported and the different configurations requried for each databases please view Database Configs

  An example of database configuration for mysql is presented below.

- **charset**

  The charset used when working with strings.

- **timezone**

  Default: uses timezone of the current php installation.

  A string representing the time zone .

- **dbPrefix**

  This is the prefix to use in all tables created by the ORM. e.g.

  if

```
dbPrefix = 'testing'
```

  all tables created will prefixed with testing so instead of the table *user* it will becomes *testing_user*.

- **components**

  This configuration serves two purposes:

  - Registering *applications*. This are applications/project that the orm will be used to manage models, migrations and perform queries.
  - Registering *components* that extend the orm good examples of this are :
    - *Faker* which is used to generate dummy data for the orm.
    - *Debuging Toolbar* A tool bar to help in development to view things like what sql the orm ran.
    - *PhpGis* Makes the orm work with gis data.

  see *Components* for more.

Sample Configuration file.

A sample configurations.

```
$config = [
    'database' => [
        'host' => '127.0.0.1',
        'dbname' => 'tester',
        'user' => 'root',
        'password' => 'root1.',
        'driver' => 'pdo_mysql',
    ],
    'dbPrefix' => 'demo_',
    'charset' => 'utf-8',
    'timezone' => 'Africa/Nairobi',
    'components' => [
        App::class,
        PhpGis::class,
        Toolbar::class,
        Faker::class,
    ],
    'signalManager' => function (BaseOrm $orm) {

        return new SignalManager();
```

(continues on next page)

(continued from previous page)

```
    },
];
```

Components

Basics

A component in powerorm is any project that needs to extend the orm.

Basically its a class that implements **Eddmash\PowerOrm\Components\ComponentInterface** interface .

An example of a component class can be found at *Faker* , which is the component class for the faker library.

This library extends the orm to add the **generatedata** command which is used to generate dummy data.

```
namespace Eddmash\PowerOrmFaker;


use Eddmash\PowerOrm\BaseOrm;
use Eddmash\PowerOrm\Components\Component;
use Eddmash\PowerOrmFaker\Commands\Generatedata;

class Faker extends Component
{

    function ready(BaseOrm $baseOrm)
    {
    }

    /**
     * Command classes
     * @return array
     * @since 1.1.0
     *
     * @author Eddilbert Macharia (http://eddmash.com) <edd.cowan@gmail.com>
     */
    function getCommands()
    {
        return [
            Generatedata::class
        ];
    }
}
```

Project Application

An application in powerorm is a form of a *component* with the difference being an application provides more information that determines things like where

- the orm should look for models,
- where it should place the generated migrations

For a php project to use powerorm, an application class needs to be created for that project.

If you have a project with the namespace **App**. Create a class that extends the **Eddmash\PowerOrm\Components\AppComponent**. This class should be placed on the same level as your models, migration folders.

```
namespace App;


use Eddmash\PowerOrm\BaseOrm;
use Eddmash\PowerOrm\Components\AppComponent;

class App extends AppComponent
{

    public function ready(BaseOrm $baseOrm)
    {
    }

}
```

Technically this file can be placed anywhere on your project tree, To get this flexibility you need to override :

- *Application::getMigrationsPath()* to tell the the orm where to find the models files and
- *Application::getMigrationsPath()* to tell the orm where to place generated migrations files.

Class Reference

Component

\Eddmash\PowerOrm\Components\Component

ready()

This method is invoked after the orm registry is ready . This means the models can be accessed within this model without any issues.

isQueryable()
: true if it this component is accessible as an attribute of the orm.

getInstance()
: Instance to return if the component is queryable..

getCommands()
: An array of Command classes that this component provides.

getName()
: Name to use when querying this component, ensure its unique.

AppComponent

\Eddmash\PowerOrm\Components\AppComponent

ready()

This method is invoked after the orm registry is ready . This means the models can be accessed within this model without any issues.

getMigrationsPath()
This is location where the ORM will use to store migrations files.

getModelsPath()
This is location where the ORM will expect to find the model files.

getDbPrefix()
This is the prefix to use in all tables created by the ORM for this project.e.g. if

```
dbPrefix = 'testing'
```

all tables created for this project will prefixed with testing so instead of the table *user* it will becomes *testing_user*.

Dependencies

PowerOrm relies on :

- Doctrine Dbal.
- Symfony console component.
- Symfony polyfill-mbstring component.

Features

- Allows to fully think of the database and its table in an object oriented manner i.e. table are represented by model and columns are represented by fields.
- Create automatic migrations.
- Create forms automatically based on models.
- All fields visible on the model, no need to look at the database table when you want to interact with the database.
- Provides database interaction methods

Credits

The following frameworks assisted me greatly in creating this project :

- Django framework
- FuelPHP framework
- Yii2 framework
- CakePHP framework

- Laravel framework
- Symfony2 framework
- Codeigniter 3/4 framework

6.1.2 Model

Model

A model is the single, definitive source of information about your data. It

contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

A model is a PHP class that subclasses `Eddmash\PowerOrm\Model\Model`.

Quick example

This example model defines a User, which has a first_name and last_name:

```
// models/User.php
use Eddmash\PowerOrm\Model\Model;

class User extends Model
{
     public function unboundFields(){
          return [
               'first_name'=>Model::CharField(['maxLength'=>30]),
               'last_name'=>Model::CharField(['maxLength'=>30]),
          ];
     }
}
```

first_name and last_name are fields of the model and each attribute maps to a database column.

Model Fields

The most important part of a model – and the only required part of a model – is the list of database fields it defines.

Note: The name `unboundFields` was choosen for the method because at the time of declaring this fields the have not been attached to class in any way.

Model fields are defined on `unboundFields` method which should return an associative array whose :

- **keys** are the names of the fields.Be careful not to choose field names that conflict with the models API like clean, save, or delete. The know should be an acceptable php variable name.
- **values** are instances of one of the subclasses of the `Eddmash\PowerOrm\Model\Field` class. see *Fields*.

All the field subclasses can be accessed from the `Eddmash\PowerOrm\Model\Model`, via static methods whose name matches that of the subclass. e.g To use the `Eddmash\PowerOrm\Model\Field\CharField` subclass from the `Eddmash\PowerOrm\Model\Model`; use the `Eddmash\PowerOrm\Model\Model::CharField()`.

PowerOrm ships with dozens of built in field types, a complete list can be found at *model field reference*, its uses this models fields to determine things like

- database column type (e.g. INTEGER, VARCHAR).
- how to perform queries e.g. if its a relationship field when to add joins

The above User model would create a database table like this:

```
CREATE TABLE user (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Note:

- The name of the table, user, is automatically derived from the name of model but can be overridden. See *Table names for more details*.
- An **id** field is added automatically, but this behavior can be overridden. See *Automatic primary key fields*.
- The CREATE TABLE SQL in this example is formatted using PostgreSQL syntax, but it's worth noting PowerOrm uses SQL tailored to the database backend specified in your configurations file.

Automatic primary key fields

By default, PowerOrm gives each model the following field:

```
id =  Eddmash\PowerOrm\Model\Model::AutoField(['primaryKey'=>true])
```

This is an auto-incrementing primary key.

If you would like to specify a custom primary key, just specify `primarykey=true` on one of your fields. If PowerOrm sees you've explicitly set Field->primaryKey, it won't add the automatic id column.

Each model requires exactly one field to have `primaryKey=true` (either explicitly declared or automatically added).

Verbose field names

All fields accept a verboseName argument.

If the verbose name isn't given, PowerOrm will automatically create it using the field's attribute name, converting underscores to spaces.

In this example, the verbose name is "person's first name":

```
first_name = Eddmash\PowerOrm\Model\Model::CharField(['verboseName'=>"person's first␣
↪name", 'maxLength'=30])
```

In this example, the verbose name is "first name":

```
first_name = Eddmash\PowerOrm\Model\Model::CharField(['maxLength'=30])
```

Relationships

The power of relational databases lies in relating tables to each other. PowerOrm offers ways to define the three most common types of database relationships: many-to-one, many-to-many and one-to-one.

In all the relationships types a *recursive relationship* can be defined.

Many-to-one relationships

To define a many-to-one relationship, use `Eddmash\PowerOrm\Model\Model::ForeignKey`. You use it just like any other Field type: by including it on the `unboundFields` method of your model.

ForeignKey requires a `to` argument, which specifies the class to which the model is related.

For example, if a Car model has a Manufacturer – that is, a Manufacturer makes multiple cars but each Car only has one Manufacturer – use the following definitions:

```
// models/Car.php
use Eddmash\PowerOrm\Model\Model;
class Car extends Model{
    public function unboundFields()
    {
        return [
            'manufacturer' => Model::ForeignKey(['to' => Manufacturer::class])
        ];
    }
}

// models/Manufacturer.php
use Eddmash\PowerOrm\Model\Model;
class Manufacturer extends Model
{

    public function unboundFields(){
        return [];
    }
}
```

It's suggested, but not required, that the name of a ForeignKey field (manufacturer in the example above) be the name of the model, lowercase.You can, of course, call the field whatever you want.

Many-to-many relationships

To define a many-to-many relationship, use `Eddmash\PowerOrm\Model\Model::ManyToManyField`. You use it just like any other Field type: by including it on the `unboundFields` method of your model.

ManyToManyField requires a `to` argument, which specifies the class to which the model is related.

For example, if a Pizza has multiple Topping objects – that is, a Topping can be on multiple pizzas and each Pizza has multiple toppings – here's how you'd represent that:

```
// models/Topping.php
use Eddmash\PowerOrm\Model\Model;
class Topping extends Model
{

    public function unboundFields(){
        return [
            'name'=> Model::CharField(['maxLength'=>50])
        ];
    }
}

// models/Pizza.php
use Eddmash\PowerOrm\Model\Model;
class Pizza extends Model{
    public function unboundFields()
    {
        return [
            'toppings' => Model::ManyToManyField(['to' => Topping::class])
        ];
    }
}
```

It's suggested, but not required, that the name of a ManyToManyField (toppings in the example above) be a plural describing the set of related model objects.

It doesn't matter which model has the ManyToManyField, but you should only put it in one of the models – not both.

Generally, ManyToManyField instances should go in the object that's going to be edited on a form. In the above example, toppings is in Pizza (rather than Topping having a pizzas ManyToManyField) because it's more natural to think about a pizza having toppings than a topping being on multiple pizzas. The way it's set up above, the Pizza form would let users select the toppings.

Extra fields on many-to-many relationships

When you're only dealing with simple many-to-many relationships such as mixing and matching pizzas and toppings, a standard ManyToManyField is all you need. However, sometimes you may need to associate data with the relationship between two models.

For example, consider the case of an application tracking the musical groups which musicians belong to. There is a many-to-many relationship between a person and the groups of which they are a member, so you could use a ManyToManyField to represent this relationship. However, there is a lot of detail about the membership that you might want to collect, such as the date at which the person joined the group.

For these situations, PowerOrm allows you to specify the model that will be used to govern the many-to-many relationship. You can then put extra fields on the intermediate model. The intermediate model is associated with the ManyToManyField using the *through* argument to point to the model that will act as an intermediary. For our musician example, the code would look something like this:

```
// models/Pesron.php
use Eddmash\PowerOrm\Model\Model;
class Person extends Model
{

    public function unboundFields(){
        return [
            'name'=> Model::CharField(['maxLength'=>50])
        ];
    }
}

// models/Group.php
use Eddmash\PowerOrm\Model\Model;
class Group extends Model{
    public function unboundFields()
    {
        return [
            'name'=> Model::CharField(['maxLength'=>50]),
            'members' => Model::ManyToManyField(['to' => 'Person', 'through'=>
↪Membership::class])
        ];
    }
}

// models/Membership.php
use Eddmash\PowerOrm\Model\Model;
class Membership extends Model{
    public function unboundFields()
    {
        return [
            'person' => Model::ForeignKey(['to' => 'Person']),
            'group' => Model::ForeignKey(['to' => 'Group']),
            'invite_reason'=>Model::CharField(['maxLength'=>65])
        ];
    }
}
```

One-to-one relationships

To define a one-to-one relationship, use `Eddmash\PowerOrm\Model\Model::OneToOneField`. You use it just like any other Field type: by including it on the `unboundFields` method of your model.

This is most useful on the primary key of an object when that object "extends" another object in some way.

OneToOneField requires a `to` argument, which specifies the class to which the model is related.

For example, if you were building a database of "places", you would build pretty standard stuff such as address, phone number, etc. in the database. Then, if you wanted to build a database of restaurants on top of the places, instead of repeating yourself and replicating those fields in the Restaurant model, you could make Restaurant have a OneToOneField to Place (because a restaurant "is a" place; in fact, to handle this you'd typically use inheritance, which involves an implicit one-to-one relation).

Meta Settings

Give your model metadata by return an array of model meta setting from the method `getMetaSettings`, like so:

```
// models/Group.php
use Eddmash\PowerOrm\Model\Model;
class User extends Model
{

    public function unboundFields(){
        return [
            'first_name'=>Model::CharField(['maxLength'=>30]),
            'last_name'=>Model::CharField(['maxLength'=>30]),
        ];
    }

    public function getMetaSettings(){
        return [
            'dbTable'=>"local_user",
            'verboseName'=>"Local Users",
        ];
    }
}
```

Model metadata is 'anything that's not a field' such as database table name (db_table) or human-readable . None are required, and overriding the getMetaSettings method is completely optional.

A complete list of all possible Meta options can be found in the *model option reference*.

Model inheritance

Model inheritance in Powerorm works almost identically to the way normal class inheritance works in PHP, That means the base class should subclass EddmashPowerOrmModelModel;.

The only decision you have to make is whether you want the parent models to be models in their own right (with their own database tables), or if the parents are just holders of common information that will only be visible through the child models.

There are three styles of inheritance that are possible in PowerOrm:

- *Abstract base* Often, you will just want to use the parent class to hold information that you don't want to have to type out for each child model. This class isn't going to ever be used in isolation, so Abstract base classes are what you're after.
- *Multi-table inheritance* If you're subclassing an existing model and want each model to have its own database table, Multi-table inheritance is the way to go.
- *Proxy models* Finally, if you only want to modify the PHP-level behavior of a model, without changing the models fields in any way, you can use Proxy models.

Multi-table inheritance

The second type of model inheritance supported by Powerorm is when each model in the hierarchy is a table all by itself. i.e. each model corresponds to its own database table and can be queried and created individually.

The inheritance relationship introduces links between the child model and its parents (via an automatically-created OneToOne).

For example:

```
use Eddmash\PowerOrm\Model\Model;

class Place extends Model
{
    public function unboundFields()
    {
        return [
            'name' => Model::CharField(['maxLength' => 100]),
            'address' => Model::CharField(['maxLength' => 80])
        ];
    }
}

class Restaurant extends Place
{
    public function unboundFields()
    {
        return [
            'serves_hot_dogs' => Model::BooleanField(['default' => false]),
            'serves_pizza' => Model::BooleanField(['default' => false])
        ];
    }
}
```

All of the fields of Place will also be available in Restaurant, although the data will reside in a different database table. So these are both possible:

```
Restaurant::objects()->filter([name=>"Bob's Cafe"]);
Place::objects()->filter([name=>"Bob's Cafe"]);
```

If you have a Place that is also a Restaurant, you can get from the Place object to the Restaurant object by using the lower-case version of the model name:

```
$p = Place::objects()->get(['id'=>12]);
// If p is a Restaurant object, this will give the child class:
$p->restaurant
```

However, if *$p* in the above example was not a Restaurant (it had been created directly as a Place object or was the parent of some other class), referring to p.restaurant would throw an exception.

In reality the orm creates the base model table as expected in the database i.e with all the field the model specifies but for the child model it creates the a table with fields that have been specfied in the child model and create a one-to-one connection to the base models' table.

Abstract base classes

Abstract base classes are useful when you want to put some common information into a number of other models. You create an Abstract base class by simply creating a normal php abstract base class.

This model will then not be used to create any database table.

Instead, when it is used as a base class for other models, its fields will be added to those of the child class.

Any fields defined in the Abstract that are again defined in the Child class will be over written by those in the child class.:

```
use Eddmash\PowerOrm\Model\Model;

abstract class CommonInfo extends Model
{
    public function unboundFields()
    {
        return [
            'name' => Model::CharField(['maxLength' => 100]),
            'age' => Model::IntegerField()
        ];
    }
}

class Student extends CommonInfo
{
    public function unboundFields()
    {
        return [
            'home_group' => Model::CharField(['maxLength' => 5])
        ];
    }
}
```

The **Student** model will have three fields: name, age and home_group.

The **CommonInfo** model cannot be used as a normal model, since it is an abstract base class. It does not generate a database table, and cannot be instantiated or saved directly.

For many uses, this type of model inheritance will be exactly what you want.

It provides a way to factor out common information at the php level, while still only creating one database table per child model at the database level.

Meta inheritance

When inheriting, Some attributes will need to be overridden in child classes, since it doesn't make sense to set them in the base class.

For example, setting `dbtable` would mean that all the child classes (the ones that don't specify *dbtable* explicitly) would use the same database table,which is almost certainly not what you want.

Proxy models

When using multi-table inheritance, a new database table is created for each subclass of a model.

This is usually the desired behavior, since the subclass needs a place to store any additional data fields that are not present on the base class.

Sometimes, however, you only want to change the php behavior of a model – perhaps to add a new method.

This is what **proxy model** inheritance is for:

- creating a proxy for the original model. You can create, delete and update instances of the proxy model and all the data will be saved as if you were using the original (non-proxied) model.
- The difference is that you can change things like the default model ordering in the proxy, without having to alter the original.

Proxy models are declared like normal models. You tell PowerOrm that it's a proxy model by setting the *proxy* meta setting of the class to True.

```
use Eddmash\PowerOrm\Model\Model;

class Employee extends Model
{
    public function unboundFields()
    {
        return [
            'name' => Model::CharField(['maxLength' => 100]),
            'age' => Model::IntegerField()
        ];
    }
}

class Auditor extends Employee
{

    public function pricePerAuditJob($employee)
    {
        // logic
    }

    public function getMetaSettings()
    {
        return [
            'proxy' => true
        ];
    }
}
```

The Auditor class operates on the same database table as its parent Employee class.

In particular, any new instances of Employee will also be accessible through Auditor, and vice-versa:

```
$employee = new Employee();
$employee->name = 'foobar';
$employee->save();

Auditor::objects()->get(['name'='foobar']);
```

Learn more on *Querying Models*

Note: QuerySets still return the model that was requested

There is no way to have Powerorm return, say, a Auditor object whenever you query for Employee objects. A queryset for Employee objects will return those employee objects. The whole point of proxy objects is that code relying on the original Employee will use employee objects and your own code can use the extensions you included (that no other code is relying on anyway). It is not a way to replace the Employee (or any other) model everywhere with something of your own creation.

Model Meta Settings

This document explains all the possible metadata options that you can give your model in its `getMetaSettings` method. see example below

dbTable

The name of the database table to use for the model:

```
dbTable = 'inventory_user'
```

Table names

To save you time, PowerOrm automatically derives the name of the database table from the name of your model class.

For example, if you have, a model defined as class Book will have a database table named *book*.

To override the database table name, use the `dbTable` parameter in `getMetaSettings` model method.

managed

Default True,

When managed is true PowerOrm will create the appropriate database tables in migrate or as part of migrations and remove them as part of a flush management command. That is, PowerOrm manages the database tables' lifecycles.

If false, no database table creation or deletion operations will be performed for this model. This is useful if the model represents an existing table or a database view that has been created by some other means. This is the only difference when managed=false. All other aspects of model handling are exactly the same as normal.

This includes:

- Adding an automatic primary key field to the model if you don't declare it. To avoid confusion for later code readers, it's recommended to specify all the columns from the database table you are modeling when using unmanaged models.
- If a model with managed=false contains a ManyToManyField that points to another unmanaged model, then the intermediate table for the many-to-many join will also not be created. However, the intermediary table between one managed and one unmanaged model will be created.

  If you need to change this default behavior, create the intermediary table as an explicit model (with managed set as needed) and use the ManyToManyField **through** attribute to make the relation use your custom model. See *Through model*

If you are interested in changing the PHP-level behavior of a model class, you could set managed=false and create a copy of an existing model. However, there's a better approach for that situation: *Proxy models*.

proxy

default false

If proxy = true, a model which subclasses another model will be treated as a *Proxy models*.

verboseName

A human-readable name for the object, singular:

```
verboseName = "pizza"
```

If this is not given, PowerOrm will use a munged version of the class name: CamelCase becomes camel case.

defaultRelatedName

The name that will be used by default for the relation from a related object back to this one. The default is *<model_name>_set*.

This option also sets <related_query_name>relatedQueryQame.

As the reverse name for a field should be unique, be careful if you intend to subclass your model. To work around name collisions, part of the name should contain '%s', which are replaced respectively by the name of the model it defined in, both lowercased and any '' in namespace replaced with '_'. See the paragraph on related names for abstract models.

Example

```
use Eddmash\PowerOrm\Model\Model;

class User extends Model
{

    public function unboundFields()
    {
        return [
          'username'=> Model::CharField(['maxLength'=>25])
        ];
    }

    public function getMetaSettings()
    {
        return [
            'proxy'=>false,
            'managed'=>true,
            'verbose'=> "Local Users",
            'dbTable'=> 'demo_user'
        ];
    }
}
```

Model Fields

This document contains all the API references of Field including the field options and field types PowerOrm offers.

Field options

The following arguments are available to all field types. All are optional.

null

If True, PowerOrm will store empty values as NULL in the database. Default is False.

blank

If True, the field is allowed to be blank. Default is False.

Note that this is different than null. null is purely database-related, whereas blank is validation-related. If a field has blank=True, form validation will allow entry of an empty value. If a field has blank=False, the field will be required.

choices

An array consisting itself of associative arrays (e.g. `[['f'=>'female'], ['m'=>'male', ...]]`) to use as choices for this field.

If this is given, the default form widget will be a select box with these choices instead of the standard text field.

The key element in each associative array is the actual value to be set on the model, and the second element is the human-readable name. For example:

```
$gender_choices = [
    'm'=>'Male',
    'f'=>'Female',
];

$gender = Eddmash\PowerOrm\Model\Model::CharField(['maxLength'=>2, 'choices'=>$gender_
↪choices])
```

dbColumn

The name of the database column to use for this field. If this isn't given, PowerOrm will use the field's name.

dbIndex

If True, this field will be indexed.

default

The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

primaryKey

If True, this field is the primary key for the model.

unique

If True, this field must be unique throughout the table.

verboseName

A human-readable name for the field. If the verbose name isn't given, PowerOrm will automatically create it using the field's attribute name, converting underscores to spaces. See *Verbose field names*

helpText

Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.

validators

A list of validators to run for this field. See the *validators documentation* for more information.

Field types

AutoField

An IntegerField that automatically increments according to available IDs. You usually won't need to use this directly; a primary key field will automatically be added to your model if you don't specify otherwise. See *Automatic primary key fields*

CharField

A string field, for small- to large-sized strings.

For large amounts of text, use TextField.

The default form widget for this field is a TextInput.

CharField has one extra required argument:

- **maxLength :** The maximum length (in characters) of the field. The maxLength is enforced at the database level and in PowerOrm's validation.

EmailField

maxLength default is 254.

A CharField that checks that the value is a valid email address. It uses EmailValidator to validate the input.

BooleanField

A true/false field.

The default form widget for this field is a CheckboxInput.

IntegerField

An integer.

The default form widget for this field is a TextInput.

TextField

A large text field.

The default form widget for this field is a Textarea.

If you specify a **maxLength** attribute, it will be reflected in the Textarea widget of the auto-generated form field. However it is not enforced at the model or database level. Use a CharField for that.

URLField

A CharField for a URL.

maxLength default is 200.

The default form widget for this field is a TextInput.

Like all CharField subclasses, URLField takes the optional maxLength argument.

If you don't specify maxLength, a default of 200 is used.

SlugField

Slug is a newspaper term. A slug is a short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs.

Like a *CharField*, you can specify **maxLength**. If **maxLength** is not specified, Powerorm will use a default length of 50.

Implies setting Field.dbIndex to **true**.

Relationship fields

PowerOrm also defines a set of fields that represent relations.

ForeignKey

A many-to-one relationship. Requires a `to` argument: the class to which the model is related.

```
// model/Car.php
use Eddmash\PowerOrm\Model\Model;

class Car extends Model{
    public function unboundFields()
    {
        return [
            'manufacturer' => Model::ForeignKey(['to' => Manufacturer::class])
```

(continues on next page)

(continued from previous page)

```
        ];
    }
}

// model/Manufacturer.php
use Eddmash\PowerOrm\Model\Model;

class Manufacturer extends Model
{

    public function unboundFields(){
        return [];
    }
}
```

A database index is automatically created on the `ForeignKey`. You can disable this by setting *dbIndex* to `false`. You may want to avoid the overhead of an index if you are creating a foreign key for consistency rather than joins, or if you will be creating an alternative index like a partial or multiple column index.

relatedName

The name to use for the relation from the related object back to this one. It's also the default value for <_related_query_name>relatedQueryName (the name to use for the reverse filter name from the target model). See the <backwards_related_objects>related objects documentation for a full explanation and example. Note that you must set this value when defining relations on *abstract models* and when you do so some <abstract_related_name>special syntax is available.

If you'd prefer powerorm not to create a backwards relation, set related_name to '+' or end it with '+'. For example, this will ensure that the User model won't have a backwards relation to this model:

relatedQueryName

The name to use for the reverse filter name from the target model. It defaults to the value of <_related_name>relatedName or <default_related_name>defaultRelatedName if set, otherwise it defaults to the name of the model.

Like <_related_name>relatedName, <default_related_name>defaultRelatedName supports app label and class interpolation via some <abstract_related_name>special syntax.

Recursive relationship

Recursive relationship is when an object that has a many-to-one relationship with itself.

To create a recursive relationship set the `to` argument to the constant `Model::SELF` or the name of the model like we have done in for foreign keys.

```
Eddmash\PowerOrm\Model\Model::ForeignKey(['to'=>Model::SELF])
```

ManyToManyField

A many-to-many relationship. Requires a 'to' argument: the class to which the model is related, which works exactly the same as it does for ForeignKey.

Through Model

OneToOneField

A one-to-one relationship. Conceptually, this is similar to a ForeignKey with unique=True, but the "reverse" side of the relation will directly return a single object.

Model Instance

Saving objects

To save an object back to the database, call **save()**:

6.1.3 Queries

Making queries

Once you've created your data models, Powerorm automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects.

This document explains how to use this API.

Throughout this guide (and in the reference), we'll refer to the following models, which comprise a Weblog application:

```
// models/Blog

namespace App\Models;

use Eddmash\PowerOrm\Model\Model;

class Blog extends Model
{

    public function unboundFields()
    {
        return [
            'name'=>Model::CharField(['maxLength'=>100]),
            'tagline'=>Model::TextField()
        ];
    }
}

// models/Author

namespace App\Models;
```

(continues on next page)

(continued from previous page)

```
use Eddmash\PowerOrm\Model\Model;

class Author extends Model
{


    public function unboundFields()
    {
        return [
            'name'=>Model::CharField(['maxLength'=>200]),
            'email'=>Model::EmailField()
        ];
    }
}


// models/Entry
namespace App\Models;

use Eddmash\PowerOrm\Model\Model;

/**
* Class Entry
*/
class Entry extends Model
{
    public function unboundFields()
    {
        return [
            'blog'=>Model::ForeignKey(['to'=>Blog::class]),
            'headline'=>Model::CharField(['maxLength'=>255]),
            'blog_text'=>Model::TextField(),
            'authors'=>Model::ManyToManyField(['to'=>Author::class]),
            'n_comments'=>Model::IntegerField(),
            'n_pingbacks'=>Model::IntegerField(),
            'ratings'=>Model::IntegerField(),
        ];
    }

}
```

Creating objects

To represent database-table data in PHP objects, Powerorm uses an intuitive system:

- A model class represents a database table, and
- an instance of that class represents a particular record in the database table.

To create an object, instantiate it using an associative array(*keys* are name of a model field, *values* are the value to assign to the fields) to the model class, then call *save()* to save it to the database.

Using the block model we created above :

```
$blog = new \App\Models\Blog();
$blog->name = "Beatles Blog";
```

(continues on next page)

(continued from previous page)

```
$blog->tagline='All the latest Beatles news.';
$blog->save();
```

This performs an **INSERT** SQL statement behind the scenes. Powerorm doesn't hit the database until you explicitly call *save()*.

The *save()* method has no return value.

Saving changes to objects

To save changes to an object that's already in the database, use *save()*.

Given a Blog instance `$blog` that has already been saved to the database, this example changes its name and updates its record in the database:

```
$blog->name = 'New name';
$blog->save();
```

This performs an UPDATE SQL statement behind the scenes. Powerorm doesn't hit the database until you explicitly call *save()*.

Saving ForeignKey and ManyToManyField fields

Updating a ForeignKey field works exactly the same way as saving a normal field – simply assign an object of the right type to the field in question. This example updates the blog attribute of an Entry instance entry, assuming appropriate instances of Entry and Blog are already saved to the database (so we can retrieve them below):

```
$en = \App\Models\Entry::objects()->get(['pk' => 1]);
$en->blog = \App\Models\Blog::objects()->get(['pk'=>4]);
$en->headline = "Filtered :doc:`QuerySet <queryset>`s are unique";
$en->blog_text = "These three :doc:`QuerySet <queryset>`s are separate.";
$en->save();
```

Updating a *ManyToManyField* works a little differently – use the add() method on the field to add a record to the relation. This example adds the Author instance joe to the entry object:

```
$en = \App\Models\Entry::objects()->get(['pk' => 1]);
$en->authors->add(\App\Models\Author::objects()->get(['name'=>'joe']));
```

To add multiple records to a *ManyToManyField* in one go, include multiple arguments in the call to add(), like this:

```
$en = \App\Models\Entry::objects()->get(['pk' => 1]);

$paul = \App\Models\Author::objects()->get(['name'=>'paul']);
$john = \App\Models\Author::objects()->get(['name'=>'john']);
$george = \App\Models\Author::objects()->get(['name'=>'george']);
$joe = \App\Models\Author::objects()->get(['name'=>'joe']);
$en->authors->add($paul, $john, $george, $joe);
```

Note: Powerorm will complain if you try to assign or add an object of the wrong type.

Retrieving objects

To retrieve objects from your database, construct a *QuerySet* via a *Manager* on your model class.

A *QuerySet* represents a collection of objects from your database. It can have zero, one or many filters. Filters narrow down the query results based on the given parameters.

In SQL terms, a *QuerySet* equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT.

You get a *QuerySet* by using your model's *Manager*. Each model has at least one Manager, which is accessed via the static method object(). Access it directly via the model class, like so:

```
\App\Models\Blog::objects()
```

The *Manager* is the main source of a *QuerySet* for a model.

For example, `App\Models\Blog->objects->all()` returns a *QuerySet* that contains all Blog objects in the database.

Retrieving all objects

The simplest way to retrieve objects from a table is to get all of them. To do this, use the *all()* method on a *Manager*:

```
\App\Models\Blog::objects()->all()
```

The *all()* method returns a *QuerySet* of all the objects in the database.

Retrieving specific objects with filters

The *QuerySet* returned by *all()* describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects.

To create such a subset, you refine the initial *QuerySet*, adding filter conditions. The two most common ways to refine a *QuerySet* are:

filter()

Returns a new *QuerySet* containing objects that match the given lookup parameters.

exclude()

Returns a new *QuerySet* containing objects that do not match the given lookup parameters.

This method take an associative array of parameters.

For example, to get a *QuerySet* of Authors who have the letter 'joe' in there name, use *filter()* like so:

```
\App\Models\Author::objects()->filter(['name__contains'=>'joe'])
```

With the default manager class, it is the same as:

```
\App\Models\Author::objects()->all()->filter(['name__contains'=>'joe'])
```

Chaining filters

The result of refining a *QuerySet* is itself a *QuerySet*, so it's possible to chain refinements together. For example:

```
\App\Models\Entry::objects()
    ->filter(['headline__startswith'=>'what'])
    ->exclude(['rating__lte'=>3])
    ->filter(['blog_text__contains'=>'kenya']);
```

This takes the initial *QuerySet* of all entries in the database, adds a filter, then an exclusion, then another filter. The final result is a *QuerySet* containing all entries with a headline that starts with "What", excluding any entries with a rating of 3 or less and the blog_text contains the word 'kenya'.

Filtered QuerySets are unique

Each time you refine a *QuerySet*, you get a brand-new *QuerySet* that is in no way bound to the previous *QuerySet*.

Each refinement creates a separate and distinct *QuerySet* that can be stored, used and reused.

Example:

```
$qs1 = \App\Models\Entry::objects()->filter(['headline__startswith' => 'what']);
$qs2 = $qs1->exclude(['rating__gte' => 3]);
$qs3 = $qs1->filter(['blog_text__contains' => 'kenya']);
```

These three *QuerySet* are separate.

- The first is a base *QuerySet* containing all entries that contain a headline starting with "What".
- The second is a subset of the first, with an additional criteria that excludes any entries with a rating of 3 or less.
- The third is a subset of the first, with an additional criteria that selects only the records whose the blog_text contains the word 'kenya'.

The initial *QuerySet* ($q1) is unaffected by the refinement process.

QuerySets are lazy

QuerySet are lazy – the act of creating a *QuerySet* doesn't involve any database activity. You can stack filters together all day long, and Powerorm won't actually run the query until the *QuerySet* is evaluated.

Take a look at this example:

```
$qs = \App\Models\Entry::objects()->filter(['headline__startswith' => 'what']);
$qs = $qs->exclude(['ratings' => 3]);
$qs = $qs->filter(['blog_text__contains' => 'kenya']);
var_dump($qs);
```

Though this looks like three database hits, in fact it hits the database only once, at the last line (var_dump($qs)). In general, the results of a *QuerySet* aren't fetched from the database until you "ask" for them. When you do, the *QuerySet* is evaluated by accessing the database.

For more details on exactly when evaluation takes place, see *When QuerySets are evaluated*.

Retrieving a single object with get()

filter() will always give you a *QuerySet*, even if only a single object matches the query - in this case, it will be a *QuerySet* containing a single element.

If you know there is only one object that matches your query, you can use the *get()* method on a *Manager* which returns the object directly:

```
$autor = \App\Models\Author::objects()->get(['pk' => 1]);
```

You can use any query expression with *get()*, just like with *filter()* - again, see Field lookups below.

Note: There is a difference between using *get()*, and using *filter()* with a *limit()*. If there are no results that match the query, *filter()* will raise a **DoesNotExist** exception. so in the code above, if there is no Author object with a primary key of 1, Powerorm will raise **DoesNotExist**.

Similarly, Powerorm will complain if more than one item matches the *filter()* query. In this case, it will raise **MultipleObjectsReturned**.

Other QuerySet methods

Most of the time you'll use *all()*, *get()*, *filter()* and *exclude()* when you need to look up objects from the database. However, that's far from all there is; see the *QuerySet API Reference* for a complete list of all the various QuerySet methods.

Limiting QuerySets

To *limit()* your QuerySet to a certain number of results. This is the equivalent of SQL's LIMIT and OFFSET clauses.

For example, this returns the first 5 objects (LIMIT 5):

```
var_dump(\App\Models\Entry::objects()->all()->limit(null,5));
```

This returns the sixth through tenth objects (OFFSET 5 LIMIT 5):

```
var_dump(\App\Models\Entry::objects()->all()->limit(5,5));
```

Limiting a *QuerySet* returns a new *QuerySet*.

Field lookups

Field lookups are how you specify the meat of an SQL **WHERE** clause. They're specified as an associative array to the *QuerySet* methods *filter()*, *exclude()* and *get()*.

Basic lookups keyword arguments take the form **[field__lookuptype=> value]**. (That's a double-underscore). For example:

```
\App\Models\Blog::objects()->filter(['name__istartswith'=>"a"])
```

translates (roughly) into the following SQL:

```
SELECT * FROM blog  WHERE name LIKE 'a%'
```

The field specified in a lookup has to be the name of a model field. There's one exception though, in case of a *ForeignKey* you can specify the field name suffixed with _id. In this case, the value parameter is expected to contain the raw value of the foreign model's primary key. For example:

```
\App\Models\Entry::objects()->filter(['blog_id'=>1]);
```

Lookups that span relationships

Powerorm offers a powerful and intuitive way to "follow" relationships in lookups, taking care of the SQL **JOINs** for you automatically, behind the scenes. To span a relationship, just use the field name of related fields across models, separated by double underscores, until you get to the field you want.

This example retrieves all **Entry** objects with a **Blog** whose name is **'Beatles Blog'**:

```
\App\Models\Entry::objects()->filter(['blog__name'=>"Beatles Blog"]);
```

This spanning can be as deep as you'd like.

It works backwards, too. To refer to a "reverse" relationship, just use the lowercase name of the model.

This example retrieves all Blog objects which have at least one Entry whose headline contains 'Lennon':

QuerySet API reference

This document describes the details of the QuerySet API.

It builds on the material presented in the model and database query guides, so you"ll probably want to read and understand those documents before reading this one.

Throughout this reference we'll use the example Weblog models presented in the *database query guide*.

- *icontains*
- *in*
- *gt*
- *gte*
- *lt*
- *lte*
- *istartswith*
- *iendswith*
- *range*

When QuerySets are evaluated

Internally, a QuerySet can be constructed, filtered, and generally passed around without actually hitting the database. No database activity actually occurs until you do something to evaluate the queryset.

You can evaluate a QuerySet in the following ways:

- **Iteration.** A QuerySet is iterable, and it executes its database query the first time you iterate over it. For example, this will print the headline of all entries in the database:

```
$entries = \App\Models\Entry::objects()->all();
foreach ($entries as $entry) :
    echo $entry->headline . "<br>";
endforeach;
```

- **dumping the queryset.** A QuerySet is evaluated when you call var_dump(), print_r() or symfony's dump() on it. This is for convenience so you can immediately see your results when using the API.

```
var_dump(\App\Models\Entry::objects()->all());
```

- **count().** A QuerySet is evaluated when you call count() on it. This, as you might expect, returns the length of the result list.

> **Note:** If you only need to determine the number of records in the set (and don't need the actual objects), it's much more efficient to handle a count at the database level using SQL's SELECT COUNT(*). Powerorm provides a *count()* method for precisely this reason.

Methods that return new QuerySets

Powerorm provides a range of QuerySet refinement methods that modify either the types of results returned by the QuerySet or the way its SQL query is executed.

filter()

Returns a new QuerySet containing objects that match the given lookup parameters. Multiple parameters are joined via AND in the underlying SQL statement.

exclude()

Returns a new QuerySet containing objects that do not match the given lookup parameters.

all()

Returns a copy of the current QuerySet (or QuerySet subclass). This can be useful in situations where you might want to pass in either a model manager or a QuerySet and do further filtering on the result. After calling all() on either object, you.ll definitely have a QuerySet to work with.

When a QuerySet is *evaluated*, it typically caches its results. If the data in the database might have changed since a QuerySet was evaluated, you can get updated results for the same query by calling all() on a previously evaluated QuerySet.

limit(offset, limit)

`options`

- offset - the position to start fetching the record from, if *null* start fetching for the first record in the table.
- limit - the number of records to fetch

To limit QuerySet to a certain number of results. This is the equivalent of SQL's LIMIT and OFFSET clauses.

For example, this returns the first 5 objects (LIMIT 5):

```
var_dump(\App\Models\Entry::objects()->all()->limit(null,5));
```

This returns the sixth through tenth objects (OFFSET 5 LIMIT 5):

```
var_dump(\App\Models\Entry::objects()->all()->limit(5,5));
```

Limiting a QuerySet returns a new QuerySet.

Methods that do not return QuerySets

The following QuerySet methods evaluate the QuerySet and return something other than a QuerySet.

get()

Returns the object matching the given lookup parameters, which should be in the format described in Field lookups.

get() raises **MultipleObjectsReturned** if more than one object was found.

get() raises a **DoesNotExist exception** if an object wasn't found for the given parameters.

count()

Returns an integer representing the number of objects in the database matching the QuerySet. The count() method never raises exceptions.

Example:

Returns the total number of entries in the database.

```
// Returns the total number of entries in the database.
echo \App\Models\Entry::objects()->count();

// Returns the number of entries whose headline starts with 'what'
echo \App\Models\Entry::objects()->filter(['headline__startswith' => 'what'])->
↪count();
```

A count() call performs a SELECT COUNT(*) behind the scenes, so you should always use count() rather than loading all of the record into PHP objects and calling count() on the result (unless you need to load the objects into memory anyway, in which case count() will be faster).

Note that if you want the number of items in a QuerySet and are also retrieving model instances from it (for example, by iterating over it), it's probably more efficient to use count(queryset) which won't cause an extra database query like Queryset::count() would.

exists()

Returns True if the QuerySet contains any results, and False if not.

This tries to perform the query in the simplest and fastest way possible, but it does execute nearly the same query as a normal QuerySet query.

exists() is useful for searches relating to both object membership in a QuerySet and to the existence of any objects in a QuerySet, particularly in the context of a large QuerySet.

The most efficient method of finding whether a model with a unique field (e.g. primary_key) is a member of a QuerySet is:

```
if(Entry::objects()->filter(['pk'=>123])->exists()):
    ... code
endif;
```

Field lookups

Field lookups are how you specify the meat of an SQL WHERE clause. They're specified as keyword arguments to the QuerySet methods *filter()*, *exclude()* and *get()*.

For an introduction, see *models and database queries documentation*.

Powerorms' built-in lookups are listed below. It is also possible to write *custom lookups* for model fields.

As a convenience when no lookup type is provided (like in `Entry::objects()->get(['id'=>14])`) the lookup type is assumed to be *exact*.

exact

Exact match. If the value provided for comparison is **null**, it will be interpreted as an SQL NULL (see *isnull* for more details).

Examples:

```
Entry::objects()->filter(['pk__exact'=>14])
Entry::objects()->filter(['pk__exact'=>null])
```

SQL equivalents:

```
SELECT ... WHERE id = 14;
SELECT ... WHERE id IS NULL;
```

isnull

Takes either **true** or **false**, which correspond to SQL queries of **IS NULL** and **IS NOT NULL**, respectively.

Example:

```
Entry::objects()->filter(['pk__isnull'=>true])
```

SQL equivalent:

```
SELECT ... WHERE id IS NULL;
```

icontains

Case-insensitive containment test.

Example:

```
Entry::objects()->get(['blog_text__icontains'=>'sequi']);
```

SQL equivalent:

```
SELECT ... WHERE blog_text LIKE '%sequi%';
```

Note this will match the blog_text 'Sequi honored today' and 'sequi honored today'.

in

In a given list.

Example:

```
Entry::objects()->filter(['pk__in'=>[2,5,3]]);
```

SQL equivalent:

```
SELECT ... WHERE id IN (2,5,3);
```

You can also use a queryset to dynamically evaluate the list of values instead of providing a list of literal values:

```
inner_qs = Blog::objects()->filter(['name__icontains'=>'dolor']);
entries = Entry::objects()->filter(['blog__in'=>inner_qs)
```

This queryset will be evaluated as subselect statement:

```
SELECT ... WHERE blog.id IN (SELECT id FROM ... WHERE NAME LIKE '%dolor%')
```

gt

Greater than.

Example:

```
Entry::objects()->filter(['pk__gt'=>1])
```

SQL equivalent:

```
SELECT ... WHERE id > 4;
```

gte

Greater than or equal to.

lt

Less than.

lte

Less than or equal to.

istartswith

Case-insensitive starts-with.

Example:

```
Entry::objects()->filter(['headline__iendswith'=>'Will'])
```

SQL equivalent:

```
SELECT ... WHERE headline ILIKE 'Will%';
```

iendswith

Case-insensitive ends-with.

Example:

```
Entry::objects()->filter(['headline__iendswith'=>'Will'])
```

SQL equivalent:

```
SELECT ... WHERE headline ILIKE '%will'
```

range

Range test (inclusive).

Example:

```
$date = new \DateTime('2005-01-01');
$date2 = new \DateTime('2005-05-01');
Entry::objects()->filter(['pub_date__range'=>[$date, $date2]]);
```

SQL equivalent:

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01' and '2005-05-01';
```

Custom Lookups

Custom Lookups

6.1.4 Migration

Migration

Migrations are PowerOrm's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, but you'll need to know when to make migrations, when to run them, and the common problems you might run into.

The Commands

There are several commands which you will use to interact with migrations and PowerOrm's handling of database schema:

- *migrate*, which is responsible for applying migrations, as well as unapplying and listing their status.
- *makemigrations*, which is responsible for creating new migrations based on the changes you have made to your models.

You should think of migrations as a version control system for your database schema. **makemigrations** is responsible for packaging up your model changes into individual migration files - analogous to commits - and **migrate** is responsible for applying those to your database.

The migration files for each app live in a "migrations" directory inside of that app, and are designed to be committed to, and distributed as part of, its codebase. You should be making them once on your development machine and then running the same migrations on your colleagues' machines, your staging machines, and eventually your production machines.

Migrations will run the same way on the same dataset and produce consistent results, meaning that what you see in development and staging is, under the same circumstances, exactly what will happen in production.

PowerOrm will make migrations for any change to your models or fields - even options that don't affect the database - as the only way it can reconstruct a field correctly is to have all the changes in the history, and you might need those options in some data migrations later on (for example, if you've set custom validators).

Workflow

Working with migrations is simple. Make changes to your models - say, add a field and remove a model - and then run makemigrations

```
$ php pmanager.php makemigrations
      Creating Migrations :
        m0002_Auto_20161023_1010.php
         - Create Model Car
         - Create Model Manufacturer
         - Add Field Manufacturer To Car
```

Your models will be scanned and compared to the versions currently contained in your migration files, and then a new set of migrations will be written out. Make sure to read the output to see what makemigrations thinks you have changed - it's not perfect, and for complex changes it might not be detecting what you expect.

Once you have your new migration files, you should apply them to your database to make sure they work as expected:

```
$ php pmanager.php migrate
 Perfoming system checks ...
 System check identified 0 issues
 Running migrations:
 Applying \app\migrations\m0001_Initial...OK
```

The command runs in two stages;

- first, it synchronizes unmigrated models, and
- then it runs any migrations that have not yet been applied.

Once the migration is applied, commit the migration and the models change to your version control system as a single commit - that way, when other developers (or your production servers) check out the code, they'll get both the changes to your models and the accompanying migration at the same time.

Migration files

Migrations are stored as an on-disk format, referred to here as "migration files". These files are actually just normal PHP files with an agreed-upon object layout, written in a declarative style.

A basic migration file looks like this:

```php
/**Migration file generated by PowerOrm(1.1.0-pre-alpha)*/

namespace app\migrations;

use Eddmash\PowerOrm\Migration\Migration;
use Eddmash\PowerOrm\Migration\Operation\Model as modelOperation;
use Eddmash\PowerOrm\Model\Field as modelField;

class m0001_Initial extends Migration{

    public function getDependency(){
```

(continues on next page)

(continued from previous page)

```
        return [];
    }

    public function getOperations(){
        return [
            modelOperation\CreateModel::createObject(
                [
                    'name'=> 'User',
                    'fields'=>[
                        'firstName'=> modelField\CharField::createObject(['maxLength
↪'=> 30]),
                        'lastName'=> modelField\CharField::createObject(['maxLength'=>
↪ 30]),
                        'id'=> modelField\AutoField::createObject(['primaryKey'=>␣
↪true, 'autoCreated'=> true]),
                    ],
                    'meta'=>[
                        'dbTable'=> 'local_user',
                        'verboseName'=> 'Local Users',
                    ],
                ]
            ),
        ] ;
    }

}
```

What PowerOrm looks for when it loads a migration file is a subclass of `Eddmash\PowerOrm\Migration\Migration` called Migration. It then inspects this object for four attributes, only two of which are used most of the time:

- dependencies, a list of migrations this one depends on.
- operations, a list of Operation classes that define what this migration does.

The operations are the key; they are a set of declarative instructions which tell PowerOrm what schema changes need to be made. PowerOrm scans them and builds an in-memory representation of all of the schema changes to all apps, and uses this to generate the SQL which makes the schema changes.

That in-memory structure is also used to work out what the differences are between your models and the current state of your migrations; PowerOrm runs through all the changes, in order, on an in-memory set of models to come up with the state of your models last time you ran makemigrations. It then uses these models to compare against the ones in your models directory to work out what you have changed.

You should rarely, if ever, need to edit migration files by hand, but it's entirely possible to write them manually if you need to. Some of the more complex operations are not autodetectable and are only available via a hand-written migration, so don't be scared about editing them if you have to.

6.1.5 Integrations

Integrating Powerorm

This is guide in setting up and using Powerorm with frameworks :

Integrating with Codeigniter 4

This is recipe for using Powerorm with codeigniter. Make sure to install powerorm via composer.

```
composer require eddmash/powerorm
```

Codeigniter 4

Note: Before its officially released ensure you have latest version as per github commits

For Codeigniter 4 and any other projects that use namespace (see *Laravel*) you just need to ensure the orm is loaded early enough.

To integrate the orm into CI4, It boils down to the following steps:

- *create application class*
- *create a service*
- *Load Orm*
- *Create Powerorm Command line*

Create Application Class

The orm requires application to register there information with it for it to work. some of the information the application needs to know about an application are where to find the models, where to place migrations..visit *Components* to learn more.

Powerorm needs some *configurations* for it to work e.g. the database settings.

We are creating this file inside the *application* folder, on the same level as the views folder.

```
namespace App;

use Eddmash\PowerOrm\BaseOrm;
use Eddmash\PowerOrm\Components\Application;

class Powerorm extends Application
{
    public static function configs()
    {
        return [
            'database' => [
                'host' => '127.0.0.1',
                'dbname' => 'tester',
                'user' => 'root',
                'password' => '',
                'driver' => 'pdo_mysql',
            ],
            'components' => [
                'app' => static::class,
            ],
            'dbPrefix' => 'test_',
            'charset' => 'utf-8',
```

(continues on next page)

(continued from previous page)

```
        ];

    }

    /**
     * @inheritdoc
     */
    public function ready(BaseOrm $baseOrm)
    {
    }
}
```

Create Service

We need to create an orm service which we can use to access the orm across the application. If an instance does not exist one will be created. We use a *getSharedInstance* to always get the same instance of the orm.

Add this method to the Service class at *application/Config/Services.php*

```
/**
 * @param bool $getShared
 * @return \Eddmash\PowerOrm\BaseOrm
 */
public static function orm($getShared = true)
{
    if ($getShared):
        return self::getSharedInstance('orm');
    endif;

    return \Eddmash\PowerOrm\Loader::webRun(\Config\Powerorm::asArray());
}
```

Load the Orm

To load the orm we listen for the **pre_system** and call the orm service. This is a shared service hence we only get the same instance of the orm through out the application.

Add this to *application/Config/Events.php*

```
Events::on('pre_system', function (){
    Services::orm();
});
```

Create Powerorm Command File

To be able to run *commands* provided by powerorm, we need to create a codeigniter 4 command that will enable us interact with powerorm.

create the file `application/Commands/Powerorm.php` and add the following content.

```php
namespace App\Commands;

use CodeIgniter\CLI\BaseCommand;
use Eddmash\PowerOrm\Console\Manager;
use Symfony\Component\Console\Input\ArgvInput;

class Powerorm extends BaseCommand
{
    protected $group = 'Powerorm';
    protected $name  = 'powerorm:pmanager';
    protected $description = 'Displays powerorm commands.';


    public function run(array $params)
    {
        // remove the 'ci4.php' from the arguments
        $input = new ArgvInput(array_slice($_SERVER['argv'], 1));

        // launch powerorm console
        Manager::run(true, $input);
    }
}
```

With that you can run all the *commands* that powerorm provides as follows:

```
php spark powerorm:pmanager
php spark powerorm:pmanager makemigrations
php spark powerorm:pmanager makemigrations --dry-run
php spark powerorm:pmanager makemigrations --dry-run -vvv
php spark powerorm:pmanager makemigrations -h
php spark powerorm:pmanager migrate
php spark powerorm:pmanager migrate zero
php spark powerorm:pmanager robot
```

See *commands* for all the available commands.

Integrating with Laravel

This is recipe for using Powerorm with laravel. Make sure to install powerorm via composer.

```
composer require eddmash/powerorm
```

Create Powerorm Service Provider

Make a Powerorm service provider that is both a wrapper and a bootstrap for Powerorm.

```
php artisan make:provider PowerormServiceProvider
```

Register the service provider in the `config/app.php` configuration file. This file contains a `providers` array where you can list the class names of your service providers.

To register `PowerormServiceProvider`, simply add it to the array:

```
'providers' => [
    // Other Service Providers
```

(continues on next page)

(continued from previous page)

```
    App\Providers\PowerormServiceProvider::class,
],
```

Make sure it looks like the one below.

```
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class PowerormServiceProvider extends ServiceProvider
{

    /**
     * Bootstrap the application services.
     *
     * @return void
     */
    public function boot()
    {
        \Eddmash\PowerOrm\Loader::webRun(config('powerorm'));
    }

    /**
     * Register the application services.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(\Eddmash\PowerOrm\BaseOrm::class);

    }
}
```

Create Application Class

The orm requires application to register there information with it for it to work. some of the information the application needs to know about an application are where to find the models, where to place migrations..visit *Components* to learn more.

Powerorm needs some *configurations* for it to work e.g. the database settings.

We create this class inside the *app* folder on the same level as providers folder.

```
namespace App;

use Eddmash\PowerOrm\BaseOrm;
use Eddmash\PowerOrm\Components\Application;

class Powerorm extends Application
{
    public static function configs()
    {
        return [
```

(continues on next page)

(continued from previous page)

```
            'database' => [
                'host' => '127.0.0.1',
                'dbname' => 'tester',
                'user' => 'root',
                'password' => '',
                'driver' => 'pdo_mysql',
            ],
            'components' => [
                'app' => static::class,
            ],
            'dbPrefix' => 'test_',
            'charset' => 'utf-8',
        ];

    }

    /**
     * @inheritdoc
     */
    public function ready(BaseOrm $baseOrm)
    {
    }
}
```

Create Laravel Command

To be able to run *commands* provided by powerorm, we need to create a laravel command that will enable us interact with powerorm.

Create a powerom command using artisan this will be placed at `app/Console/Commands` as show below.

```
php artisan make:command Powerorm
```

Register the new command with laravel, This is done on the file `app/Console/Kernel.php` as shown below

```
protected $commands = [
    //
    Powerorm::class
];
```

Make powerorm command look like the one below `app/Console/Commands/Powerorm.php`

```
namespace App\Console\Commands;

use Eddmash\PowerOrm\Console\Manager;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\ArgvInput;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class Powerorm extends Command
{
    /**
     * The name and signature of the console command.
     *
```

(continues on next page)

(continued from previous page)

```
     * @var string
     */
    protected $signature = 'powerorm:pmanager';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Display commands provided by powerorm.';

    /**
     * We stop laravel from running the command and pass control to powerorm
     * {@inheritdoc}
     */
    public function run(InputInterface $input, OutputInterface $output)
    {
        // remove the 'artisan' from the arguments
        $input = new ArgvInput(array_slice($_SERVER['argv'], 1));

        // launch powerorm console
        Manager::run(true, $input);
    }
}
```

With that you can run all the *commands* that powerorm provides as follows:

```
php artisan powerorm:pmanager
php artisan powerorm:pmanager makemigrations
php artisan powerorm:pmanager makemigrations --dry-run
php artisan powerorm:pmanager makemigrations --dry-run -vvv
php artisan powerorm:pmanager makemigrations -h
php artisan powerorm:pmanager migrate
php artisan powerorm:pmanager migrate zero
php artisan powerorm:pmanager robot
```

See *commands* for all the availabel commands.

6.1.6 Extra Information

Development Tools

Powerorm Management Commands

pmanager.php

Is Powerorm's command-line utility for administrative tasks. This document outlines all it can do.

Usage:

```
php pmanager.php <command> [options]
```

Available Commands

help

Displays help for a command

list

Lists commands

makemigrations

Creates new migrations based on the changes detected to your models. Migrations, their relationship with apps and more are covered in depth in the *migrations documentation*.

–dry-run

Shows what migrations would be made without actually writing any migrations files to disk. Using this option along with -vvv will also show the complete migrations files that would be written.

checks

Inspects the models(as per now) in the project for common problems.

–list-tags

Lists all available tags.

–tag TAGS, -t TAGS

As of now this command runs checks on models but in future this command might perfom different types of checks that are categorized with tags.

You can use these tags to restrict the checks performed to just those in a particular category.

For example, to perform only models and compatibility checks, run:

```
php pmanager.php check -t MODEL
```

–fail-level {CRITICAL,ERROR,WARNING,INFO,DEBUG}

Specifies the message level that will cause the command to exit with a non-zero status. Default is ERROR.

migrate

Synchronizes the database state with the current set of models and migrations. Migrations, their relationship with apps and more are covered in depth in the *migrations documentation*.

The behavior of this command changes depending on the arguments provided:

- No arguments: All models have all of their migrations run.
- **<migrationname>**: Brings the database schema to a state where the named migration is applied,but no later migrations in the same app are applied. This may involve unapplying migrations if you have previously migrated past the named migration. Use the name **zero** to unapply all migrations for an app.

–fake

Tells Powerorm to mark the migrations as having been applied or unapplied, but without actually running the SQL to change your database schema.

This is intended for advanced users to manipulate the current migration state directly if they're manually applying changes; be warned that using **–fake** runs the risk of putting the migration state table into a state where manual recovery will be needed to make migrations run correctly.

showmigrations

Shows all migrations in a project. i.e. lists all the migrations available, and whether or not each migration is applied (marked by an (applied) next to the migration name).

makemodel

Generate a model class.

```
php pmanager.php makemodel 'App\Models\Author' -p application/Models
```

<model_name>

The name of the model to generate. Use the name "zero" to unapply all migrations.

-p, –path

The location the generated model will be place relative to vendor folder. defaults to the same level as the vendor folder.

any path provided should be relative to the vendor folder e.g.

```
-p app/models
```

will look for directory name *app* on the same level as vendor directory.

-f, –force

Force overwrite if model already exists. if this option is not available the command will through an `CommandError` if the model already exists.

robot

A little fun is good for the soul, draws a robot because... well, why not ?

6.2 PowerForm User Guide

The Form component is a tool to help you solve the problem of allowing end-users to interact with the data and modify the data in your application. And though traditionally this has been through HTML forms, the component focuses on processing data to and from your client and application, whether that data be from a normal form post or from an API.

6.2.1 Installation

Via composer **(recommended)**:

```
composer require eddmash/powerform
```

Or add this to the composer.json file:

```
{
    "require": {
        "eddmash/powerform": "1.1"(check latest version)
    }
}
```

You could also Download or Clone package from github.

Then, require the vendor/autoload.php file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of Powerform.

Working with forms

Handling forms is a complex business. where numerous items of data of several different types may need to be:

- prepared for display in a form,
- rendered as HTML,
- edited using a convenient interface,
- returned to the server,
- validated and cleaned up, and then saved or passed on for further processing.

Powerform functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Powerform handles three distinct parts of the work involved in forms:

- preparing and restructuring data to make it ready for rendering.
- creating HTML forms for the data
- receiving and processing submitted forms and data from the client.

It is possible to write code that does all of this manually, but Powerform can take care of it all for you.

Building a form

The work that needs to be done

Suppose you want to create a simple form on your website, in order to obtain the user's name. You'd need something like this in your template:

```
<form action="/your-name/" method="post">

    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="">

    <input type="submit" value="OK">

</form>
```

This tells the browser to return the form data to the URL **/your-name/**, using the **POST** method. It will display a text field, labeled "Your name:", and a button marked "OK".

When the form is submitted, the POST request which is sent to the server will contain the form data.

Now you'll also need a view corresponding to that **/your-name/** URL which will find the appropriate key/value pairs in the request, and then process them.

This is a very simple form. In practice, a form might contain dozens or hundreds of fields, many of which might need to be pre-populated, and we might expect the user to work through the edit-submit cycle several times before concluding the operation.

We might require some validation to occur in the browser, even before the form is submitted; we might want to use much more complex fields, that allow the user to do things like pick dates from a calendar and so on.

At this point it's much easier to get Powerform to do most of this work for us.

Form class

At the heart of this system of components is Powerforms' *Form* class. In much the same way that a Powerorm model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a *Form* class describes a form and determines how it works and appears.

In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form <input> elements. (A *Model Form* maps a model class's fields to HTML form <input> elements via a Form)

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A *DateField* and a *FieldField* handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML "widget" - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required.

Building a form in Powerform

The Form class

We already know what we want our HTML form to look like. Our starting point for it in Powerform is this:

```
namespace App\Forms;


use Eddmash\PowerOrm\Form\Form;

class CommentForm extends Form
{
    /**
     * @inheritDoc
     */
    public function fields()
    {

        return [
            'your_name' => Form::CharField(['label'=>'Your name', 'maxLength'=>100]),
        ];
    }

}
```

This defines a *Form* class with a field (your_name). We've applied a human-friendly label to the field, which will appear in the <label> when it's rendered (although in this case, the label we specified is actually the same one that would be generated automatically if we had omitted it).

The field's maximum allowable length is defined by *maxLength*. This does two things.:

- It puts a **maxlength="100"** on the HTML **<input>** (so the browser should prevent the user from entering more than that number of characters in the first place).
- It also means that when Powerform receives the form back from the browser, it will validate the length of the data.

A *Form* instance has an *isValid()* method, which runs validation routines for all its fields. When this method is called, if all fields contain valid data, it will:

- return **true**
- place the form's data in its *cleanedData* attribute.

The whole form, when rendered for the first time, will look like:

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100" required />
```

Note that it does not include the <form> tags, or a submit button. We'll have to provide those ourselves in the template.

The Logic

Form data is sent back to your controller, generally the same controller that published the form. This allows us to reuse some of the same logic.

To handle the form we need to instantiate it in the controller for the URL where we want it to be published.

```
public function commentform()
{
    if ($_SERVER['REQUEST_METHOD'] === "POST"):

        $form = new CommentForm(['data' => $_POST]);
        if ($form->isValid()):
            // process the data in form.cleaned_data as required
            // ...
            // redirect to a new URL:
        endif;
    else:
        $form = new CommentForm();
    endif;

    return render('create.html', ['form' => $form]);
}
```

If we arrive at this controller with a **GET** request, it will create an empty form instance and pass it in to the template for rendering. This is what we can expect to happen the first time we visit the URL.

If the form is submitted using a **POST** request, the controller will once again create a form instance and populate it with data from the request: `$form = new CommentForm(['data' => $_POST])`. This is called "binding data to the form" (it is now a bound form).

We call the form's *isValid()* method; if it's not **true**, we go back to the template with the form. This time the form is no longer empty (unbound) so the HTML form will be populated with the data previously submitted, where it can be edited and corrected as required.

If *isValid()* is **true**, we'll now be able to find all the validated form data in its *cleanedData* attribute. We can use this data to update the database or do other processing before sending an HTTP redirect to the browser telling it where to go next.

We don't need to do much in our **create.html** template. The simplest example is:

```
<form method="post" novalidate>

    <?php echo $form;?>

    <input type="submit" value="Send" name="Send">
</form>
```

All the form's fields and their attributes will be unpacked into HTML markup from that `echo $form;`

Note: HTML5 input types and browser validation

If your form includes a *URLField*, an *EmailField* or any integer field type, Powerform will use the url, email and number HTML5 input types. By default, browsers may apply their own validation on these fields, which may be stricter than Powerforms's validation. If you would like to disable this behavior, set the **novalidate** attribute on the form tag, or specify a different widget on the field, like TextInput.

That's all you need to get started, but the forms puts a lot more at your fingertips. Once you understand the basics of the process described above, you should be prepared to understand other features of the forms system and ready to learn a bit more about the underlying machinery.

More about Powerform classes

All form classes are created as subclasses of `\Eddmash\PowerOrm\Form\Form`, including the *ModelForm*.

Note: Models and Forms

In fact if your form is going to be used to directly add or edit a Powerorm model, a *ModelForm* can save you a great deal of time, effort, and code, because it will build a form, along with the appropriate fields and their attributes, from a **Model** class.

Bound and unbound form instances

The distinction between *Bound and unbound* forms is important:

- An **unbound form** has no data associated with it. When rendered to the user, it will be empty or will contain default values.
- A **bound form** has submitted data, and hence can be used to tell if that data is valid. If an invalid bound form is rendered, it can include inline error messages telling the user what data to correct.

The form's *isBound* attribute will tell you whether a form has data bound to it or not.

More on fields

Consider a more useful form than our minimal example above, which we could use to implement "contact me" functionality on a personal website:

```
namespace App\Forms;

use Eddmash\PowerOrm\Form\Form;

class ContactForm extends Form
{
    /**
     * @inheritDoc
     */
    public function fields()
    {
        return [
            'subject' => Form::CharField(['maxLength'=>100]),
            'message' => Form::CharField(['widget'=>Form::TextArea()]),
            'email' => Form::EmailField(),
            'cc_myself'=>Form::BooleanField(['required'=>false])
```

(continues on next page)

(continued from previous page)

```
        ];
    }

}
```

Our earlier form used a single field, **your_name**, a *CharField*. In this case, our form has four fields: **subject**, **message**, **sender** and **cc_myself**. *CharField*, *EmailField* and *BooleanField* are just three of the available field types; a full list can be found in *Form fields*.

Widgets

Each *Form fields* has a corresponding *Widget class*, which in turn corresponds to an HTML form widget such as **<input type="text">**.

In most cases, the field will have a sensible default widget. For example, by default, a *CharField* will have a *TextInput* widget, that produces an **<input type="text">** in the HTML. If you needed **<textarea>** instead, you'd specify the appropriate widget when defining your form field, as we have done for the message field.

Field data

Whatever the data submitted with a form, once it has been successfully validated by calling **isValid()** (and **isValid()** has returned **true**), the validated form data will be in the *cleanedData* associative array.

This data will have been nicely converted into Php types for you.

Note: You can still access the unvalidated data directly from **$_POST** at this point, but the validated data is better.

In the contact form example above, cc_myself will be a **boolean** value. Likewise, fields such as **IntegerField** and **DecimalField** convert values to a Php **int** and **float** respectively.

Here's how the form data could be processed in the view that handles this form:

```
// on your controller
public function contactform()
{
    if (($_SERVER['REQUEST_METHOD'] === "POST"):

        $form = new ContactForm(['data' => $_POST]);
        if ($form->isValid()):
            $subject = $form->cleanedData['subject'];
            $email = $form->cleanedData['email'];
            $message = $form->cleanedData['message'];
            $cc_myself = $form->cleanedData['cc_myself'];

            // more code
        endif;
    else:
        $form = new ContactForm();
    endif;

    return render('form', ['form' => $form]);
}
```

Some field types need some extra handling. For example, files that are uploaded using a form need to be handled differently (they can be retrieved from **$_FILES**, rather than **$_POST**).

For details of how to handle file uploads with your form, see *Binding uploaded files* to a form.

Working with form templates

All you need to do to display your form, is to create an instance of the form and **echo** it out.

```
echo $form;
```

This will render its <label> and <input> elements appropriately.

Form rendering options

Note: Additional form template furniture

Don't forget that a form's output does not include the surrounding <form> tags, or the form's submit control. You will have to provide these yourself.

There are other output options though for the **<label>/<input>** pairs:

- **asTable()** will render them as table cells wrapped in **<tr>** tags
- **asParagraph()** will render them wrapped in **<p>** tags
- **asUl()** will render them wrapped in **<li>** tags

Note that you'll have to provide the surrounding **<table>** or **<ul>** elements yourself.

Here's the output of **asParagraph()** for our ContactForm instance:

```
echo $form->asParagraph();
```

```
<p>
    <label for="id_subject">Subject</label>
    <input maxlength="100" type="text" name="subject" id="id_subject"> <br>

</p>
<p>
    <label for="id_message">Message</label> <br>
    <textarea name="message"id="id_message"></textarea>
    <br>
</p>
<p>
    <label for="id_email">Email</label> <br>
    <input type="email" name="email" id="id_email"> <br>
</p>
<p>
    <label for="id_cc_myself">Cc myself</label> <br>
    <input type="checkbox" name="cc_myself" id="id_cc_myself">
</p>
```

Note that each form field has an ID attribute set to **id_<field-name>**, which is referenced by the accompanying label tag. This is important in ensuring that forms are accessible to assistive technology such as screen reader software. You can also *customize the way in which labels and ids are generated*.

See *Outputting forms as HTML* for more on this.

Rendering fields manually

We can do it manually if we like (allowing us to reorder the fields, for example). Each field is available as an attribute of the form using

```
echo $form->{name_of_field}
```

For example:

```
<form method="post" novalidate>

    <?= $form->nonFieldErrors(); ?>

    <div class="fieldWrapper">
        <?= $form->subject->getErrors(); ?>
        <label for="<?= $form->subject->getIdForLabel(); ?>">Email subject:</label>
        <?= $form->subject; ?>
    </div>

    <div class="fieldWrapper">
        <?= $form->message->getErrors(); ?>
        <label for="<?= $form->message->getIdForLabel(); ?>">Message:</label>
        <?= $form->message; ?>
    </div>

    <div class="fieldWrapper">
        <?= $form->email->getErrors(); ?>
        <label for="<?= $form->email->getIdForLabel(); ?>">Your email address:</label>
        <?= $form->email; ?>
    </div>

    <div class="fieldWrapper">
        <?= $form->cc_myself->getErrors(); ?>
        <label for="<?= $form->cc_myself->getIdForLabel(); ?>">CC yourself?:</label>
        <?= $form->cc_myself; ?>
    </div>

    <input type="submit" value="Send" name="Send">
</form>
```

Complete **<label>** elements can also be generated using the **labelTag()**. For example:

```
<div class="fieldWrapper">
    <?= $form->cc_myself->getErrors(); ?>
    <?= $form->cc_myself->labelTag(); ?>
    <?= $form->cc_myself; ?>
</div>
```

Rendering form error messages

Of course, the price of this flexibility is more work. Until now we haven't had to worry about how to display form errors, because that's taken care of for us. In this example we have had to make sure we take care of any errors for

each field and any errors for the form as a whole. Note **nonFieldErrors()** at the top of the form and the **getErrors()** on each field.

Using `$form->field_name->getErrors();` displays a list of form errors, rendered as an unordered list.

This might look like:

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

The list has a CSS class of **errorlist** to allow you to style its appearance. If you wish to further customize the display of errors you can do so by looping over them:

```
<div class="fieldWrapper">
    <ol>
        <?php foreach ($form->subject->getErrors() as $error) : ?>
            <?= $error; ?>
        <?php endforeach; ?>
    </ol>
    <label for="<?= $form->subject->getIdForLabel(); ?>">Email subject:</label>
    <?= $form->subject; ?>
</div>
```

Non-field errors (and/or hidden field errors that are rendered at the top of the form when using helpers like **form.asParagraph()**) will be rendered with an additional class of nonfield to help distinguish them from field-specific errors.

Looping over the form's fields

If you're using the same HTML for each of your form fields, you can reduce duplicate code by looping through each field in turn using a **foreach** loop:

```
<?php foreach ($form as $field):?>
    <div class="fieldWrapper">
        <ol>
            <?php foreach ($field->getErrors() as $error) : ?>
                <?= $error; ?>
            <?php endforeach; ?>
        </ol>
        <label for="<?= $field->getIdForLabel(); ?>"><?=$field->getLabelName()?></
↪label>
        <?= $field; ?>
    </div>
<?php endforeach; ?>
```

Useful attributes and methods on **Field** include:

- **getLabelName()**

  The label of the field, e.g. Email address.

- **labelTag()** The field's label wrapped in the appropriate HTML <label> tag. This includes the form's label_suffix. For example, the default label_suffix is a colon:

  ```
  <label for="id_email">Email address:</label>
  ```

- **getIdForLabel()**

The ID that will be used for this field (id_email in the example above). If you are constructing the label manually, you may want to use this in lieu of **labelTag()**. It's also useful, for example, if you have some inline JavaScript and want to avoid hardcoding the field's ID.

- **value()**

  The value of the field. e.g someone@example.com.

- **getHtmlName()**

  The name of the field that will be used in the input element's name field. This takes the form prefix into account, if it has been set.

- **getHelpText()**

  Any help text that has been associated with the field.

- **getErrors()**

  Outputs a <ul class="errorlist"> containing any validation errors corresponding to this field. You can customize the presentation of the errors with a **foreach** loop as shown above. In this case, each object in the loop is a simple string containing the error message.

- **isHidden()**

  This method is **true** if the form field is a **hidden** field and **false** otherwise.

```
foreach ($form as $field):
    if($field->isHidden()):
        // do something
    endif;
endforeach;
```

Looping over hidden and visible fields

If you're manually laying out a form, you might want to treat **<input type="hidden">** fields differently from non-hidden fields.

For example, because hidden fields don't display anything, putting error messages "next to" the field could cause confusion for your users – so errors for those fields should be handled differently.

Powerform provides two methods on a form that allow you to loop over the hidden and visible fields independently:

- **hiddenFields()** and
- **visibleFields()**.

Here's a modification of an earlier example that uses these two methods:

```
// display hidden fields
<?php foreach ($form->hiddenFields() as $field): ?>
    <?= $field; ?>
<?php endforeach; ?>

// display visible fields
<?php foreach ($form->visibleFields() as $field): ?>

    <div class="fieldWrapper">
        <?=$field->getErrors()?>
        <?=$field->labelTag()?>
        <?= $field; ?>
```

(continues on next page)

(continued from previous page)

```
    </div>
<?php endforeach; ?>
```

This example does not handle any errors in the hidden fields. Usually, an error in a hidden field is a sign of form tampering, since normal form interaction won't alter them. However, you could easily insert some error displays for those form errors, as well.

The Forms API

Bound and unbound forms

A Form instance is either bound to a set of data, or unbound.

If it's bound to a set of data, it's capable of validating that data and rendering the form as HTML with the data displayed in the HTML.

If it's unbound, it cannot do validation (because there's no data to validate!), but it can still render the blank form as HTML.

class Form

To create an unbound Form instance, simply instantiate the class:

```
$form = new AuthorForm();
```

The form constructor accepts an associative as argument.

To bind data to a form, pass the data as a associative array with the key data to your Form class constructor:

```
$data =[
    "name" => "rrrr"
    "email" => "edd.cowan@gmail.com"
];
$form = new AuthorForm(['data'=>$data]);
```

In this associative array, the keys are the field names, which correspond to the attributes in your Form class. The values are the data you're trying to validate. These will usually be strings, but there's no requirement that they be strings; the type of data you pass depends on the Field, as we'll see in a moment.

Form.isBound

If you need to distinguish between bound and unbound form instances at runtime, check the value of the form's isBound attribute:

```
$form = new AuthorForm();
var_dump($form->isBound); // false

$form = new AuthorForm(['data'=>$data]);
var_dump($form->isBound); // true
```

Note that passing an empty associative array creates a bound form with empty data:

```
$form = new AuthorForm([]);
var_dump($form->isBound); // true
```

If you have a bound Form instance and want to change the data somehow, or if you want to bind an unbound Form instance to some data, create another Form instance. There is no way to change data in a Form instance.

Once a Form instance has been created, you should consider its data immutable, whether it has data or not.

Using forms to validate data

Form.clean()

Implement a clean() method on your Form when you must add custom validation for fields that are interdependent. See *Cleaning and validating fields that depend on each other* for example usage.

Form.isValid()

The primary task of a Form object is to validate data. With a bound Form instance, call the isValid() method to run validation and return a boolean designating whether the data was valid:

```
$data =[
    "name" => "rrrr"
    "email" => "edd.cowan@gmail.com"
];
$form = new AuthorForm(['data'=>$data]);
var_dump($form->isValid()); // true
```

Let's try with some invalid data. In this case, subject is blank (an error, because all fields are required by default) and email is not a valid email address:

```
$data =[
    "name" => "rrrr"
    "email" => "edd.gmail.com"
];
$form = new AuthorForm(['data'=>$data]);
var_dump($form->isValid()); // false
```

Form.errors()

Access the errors method to get a associative array of error messages:

```
var_dump($form->errors());

[
  "name" => [
    ValidationError { }
  ]
  "email" => [
    ValidationError { }
  ]
]
```

In this associative array, the keys are the field names, and the values are an array of strings representing the error messages. The error messages are stored in an array because a field can have multiple error messages.

You can access errors without having to call *isValid()* first. The form's data will be validated the first time either you call *isValid()* or access errors.

The validation routines will only get called once, regardless of how many times you access errors or call *isValid()*. This means that if validation has side effects, those side effects will only be triggered once.

Form.errors()->asData()

Access the errors method to get a associative array of error messages:

```
var_dump($form->errors());

[
  "name" => [
    ValidationError { }
  ]
  "email" => [
    ValidationError { }
  ]
]
```

Returns an associative array of fields to their original ValidationError instances.

Form.addError($field, $error)

This method allows adding errors to specific fields from within the **Form.clean()** method, or from outside the form altogether; for instance from a controller.

The **field** argument is the name of the field to which the errors should be added. If its value is None the error will be treated as a non-field error as returned by *Form.nonFieldErrors()*.

The error argument can be a simple string, or preferably an instance of ValidationError. See *Raising ValidationError* for best practices when defining form errors.

Note that **Form.addError()** automatically removes the relevant field from ****cleanedData****.

Form.hasError($field, $code=null)

This method returns a boolean designating whether a field has an error with a specific error **code**. If **code** is **null**, it will return **true** if the field contains any errors at all.

To check for non-field errors use *NON_FIELD_ERRORS* as the field parameter.

Form.nonFieldErrors()

This method returns the list of errors from *Form.errors()* that aren't associated with a particular field. This includes ValidationErrors that are raised in *Form.clean()* and errors added using *Form.addError(null, "...")*.

Dynamic initial values

Form.initial

Use **initial** to declare the initial value of form fields at runtime. For example, you might want to fill in a username field with the username of the current session.

To accomplish this, use the initial argument to a Form. This argument, if given, should be a associative array mapping field names to initial values. Only include the fields for which you're specifying an **initial** value; it's not necessary to include every field in your form. For example:

```
$data = []; // that the form is validated against.mostly will be from post
$initial = ['subject'=>"yello there"];
$form = ContactForm(['data'=>$data, 'initial'=>$initial])
```

These values are only displayed for unbound forms, and they're not used as fallback values if a particular value isn't provided.

If a Field defines initial and you include initial when instantiating the Form, then the latter **initial** will have precedence. In this example, **initial** is provided both at the field level and at the form instance level, and the latter gets precedence:

```
class ContactForm extends Form
{
    public function fields()
    {
        return [
            'subject' => Form::CharField(['maxLength' => 100, 'initial'=>'welcome']),
            'recipients' => MultiEmailField::instance(),
            'cc_myself' => Form::BooleanField(['required' => false]),
        ];
    }
}
```

```
<input maxlength="100" name="subject" id="id_subject" value="yello there" type="text">
```

Form.getInitialForField($field, $name)

Use **getInitialForField()** to retrieve initial data for a form field. It retrieves data from **Form.initial** and **Field.initial**, in that order, and evaluates any callable initial values.

Accessing the fields from the form

Form.getFields()

You can access the fields of Form instance from its getFields() method:

```
var_dump($form->getFields());

[
  "subject" => CharField { }
  "recipients" => MultiEmailField { }
  "cc_myself" => BooleanField { }
]
```

Accessing "clean" data

Form.cleanedData

Each field in a Form class is responsible not only for validating data, but also for "cleaning" it – normalizing it to a consistent format. This is a nice feature, because it allows data for a particular field to be input in a variety of ways, always resulting in consistent output.

For example, DateField normalizes input into a PhP DateTime object. Regardless of whether you pass it a string in the format '1994-07-15', a DateTime object, or a number of other formats, DateField will always normalize it to a DateTime object as long as it's valid.

Once you've created a Form instance with a set of data and validated it, you can access the clean data via its cleanedData attribute:

```
$data = [
  "subject" => "help yo",
  "recipients" => "fred@example.com,edd@gmail.com",
  "cc_myself" => true
];

$form = new ContactForm(['data'=>$data]);
$form->isValid();
var_dump($form->cleanedData);

[
  "subject" => "help yo"
  "recipients" => [
    "fred@example.com"
    "edd@gmail.com"
  ]
  "cc_myself" => true
]
```

If your data does not validate, the **cleanedData** associative array contains only the valid fields:

```
$data = [
  "subject" => "help yo",
  "recipients" => "invalid email",
  "cc_myself" => true
];

$form = new ContactForm(['data'=>$data]);
$form->isValid();
var_dump($form->cleanedData);

[
  "subject" => "help yo",
  "cc_myself" => true
]
```

cleanedData will always only contain a key for fields defined in the Form, even if you pass extra data when you define the Form. In this example, we pass a bunch of extra fields to the ContactForm constructor, but **cleanedData** contains only the form's fields:

```
$data = [
  "subject" => "help yo"
  "recipients" => "invalid email"
  "cc_myself" => "on"
  "Send" => "Send"
]

$form = new ContactForm(['data'=>$data]);
$form->isValid();
var_dump($form->cleanedData);
```

(continues on next page)

(continued from previous page)

```
[
  "subject" => "help yo"
  "cc_myself" => true
]
```

When the Form is valid, **cleanedData** will include a key and value for all its fields, even if the data didn't include a value for some optional fields. In this example, the data associative array doesn't include a value for the **box** field, but **cleanedData** includes it, with an empty value:

```
$data = [
  "subject" => "help there"
  "recipients" => "fred@example.com"
  "cc_myself" => "on"
  "Send" => "Send"
];

$form = new ContactForm(['data'=>$data]);
$form->isValid();
var_dump($form->cleanedData);

[
  "subject" => "help there"
  "recipients" => []
  "cc_myself" => true
  "box" => ""
];
```

In this above example, the **cleanedData** value for **box** is set to an empty string, because **box** is **CharField**, and **CharFields** treat empty values as an empty string. Each field type knows what its "blank" value is – e.g., for DateField, it's null instead of the empty string. For full details on each field's behavior in this case, see the "Empty value" note for each field in the "Built-in Field classes" section below.

You can write code to perform validation for particular form fields (based on their name) or for the form as a whole (considering combinations of various fields). More information about this is in *Form and field validation*.

Outputting forms as HTML

Form.asParagraph()

asParagraph() renders the form as a series of <p> tags, with each <p> containing one field:

```
echo $form->asParagraph();
```

```
<p>
    <label for="id_subject">Subject</label>
    <input maxlength="100" type="text" name="subject" id="id_subject"> <br>

</p>
<p>
    <label for="id_message">Message</label> <br>
    <textarea name="message"id="id_message"></textarea>
    <br>
</p>
<p>
```

(continues on next page)

(continued from previous page)

```
    <label for="id_email">Email</label> <br>
    <input type="email" name="email" id="id_email"> <br>
</p>
<p>
    <label for="id_cc_myself">Cc myself</label> <br>
    <input type="checkbox" name="cc_myself" id="id_cc_myself">
</p>
```

Form.asUl()

asUl() renders the form as a series of <li> tags, with each <li> containing one field. It does not include the <ul> or , so that you can specify any HTML attributes on the <ul> for flexibility:

```
echo $form->asUl();
```

```
<li>
    <label for="id_mo-subject"> Subject</label>
    <input maxlength="100" type="text" name="mo-subject" id="id_mo-subject">
</li>
<li>
    <label for="id_mo-message"> Message</label>
    <textarea name="mo-message" id="id_mo-message"></textarea>
</li>
<li>
    <label for="id_mo-cc_myself"> Cc myself</label>
    <input type="checkbox" name="mo-cc_myself" id="id_mo-cc_myself">
</li>
```

Form.asTable()

Finally, **asTable()** outputs the form as an HTML **<table>**. :

```
<tr>
    <th><label for="id_mo-subject"> Subject</label></th>
    <td><input maxlength="100" type="text" name="mo-subject" id="id_mo-subject"></td>
</tr>
<tr>
    <th><label for="id_mo-message"> Message</label></th>
    <td><textarea name="mo-message" id="id_mo-message"></textarea><br><span class=
↪"helptext">messages</span></td>
</tr>
<tr>
    <th><label for="id_mo-cc_myself"> Cc myself</label></th>
    <td><input type="checkbox" name="mo-cc_myself" id="id_mo-cc_myself"></td>
</tr>
```

Configuring form elements' HTML id attributes and <label> tags

autoId

By default, the form rendering methods include:

- HTML id attributes on the form elements.
- The corresponding **<label>** tags around the labels. An HTML **<label>** tag designates which label text is associated with which form element. This small enhancement makes forms more usable and more accessible to assistive devices. It's always a good idea to use **<label>** tags.

The **id** attribute values are generated by prepending **id_** to the form field names. This behavior is configurable, though, if you want to change the id convention or remove HTML **id** attributes and **<label>** tags entirely.

Use the **autoId** argument to the Form constructor to control the **id** and label behavior. This argument must be **true**, **false** or a **string**.

- If **autoId** is **false**, then the form output will not include **<label>** tags nor **id** attributes.

```
$f = new ContactForm(['autoId'=>false]);

echo $f->asTable ();

<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100"␣
↪required /></td></tr>
<tr><th>Message:</th><td><input type="text" name="message" required /></td></tr>
<tr><th>Sender:</th><td><input type="email" name="sender" required /></td></tr>
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>

echo $f->asUl ();

<li>Subject: <input type="text" name="subject" maxlength="100" required /></li>
<li>Message: <input type="text" name="message" required /></li>
<li>Sender: <input type="email" name="sender" required /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>

echo $f->asParagraph();

<p>Subject: <input type="text" name="subject" maxlength="100" required /></p>
<p>Message: <input type="text" name="message" required /></p>
<p>Sender: <input type="email" name="sender" required /></p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

- If **autoId** is set to **true**, then the form output will include **<label>** tags and will simply use the field name as its id for each form field:

```
$f = new ContactForm(['autoId'=>true]);

echo $f->asTable ();

<tr><th><label for="subject">Subject:</label></th><td><input id="subject"␣
↪type="text" name="subject" maxlength="100" required /></td></tr>
<tr><th><label for="message">Message:</label></th><td><input type="text"␣
↪name="message" id="message" required /></td></tr>
<tr><th><label for="sender">Sender:</label></th><td><input type="email" name=
↪"sender" id="sender" required /></td></tr>
<tr><th><label for="cc_myself">Cc myself:</label></th><td><input type=
↪"checkbox" name="cc_myself" id="cc_myself" /></td></tr>

echo $f->asUl ();

<li><label for="subject">Subject:</label> <input id="subject" type="text"␣
↪name="subject" maxlength="100" required /></li>
<li><label for="message">Message:</label> <input type="text" name="message"␣
↪id="message" required /></li>
```

(continues on next page)

(continued from previous page)

```
<li><label for="sender">Sender:</label> <input type="email" name="sender" id=
↪"sender" required /></li>
<li><label for="cc_myself">Cc myself:</label> <input type="checkbox" name=
↪"cc_myself" id="cc_myself" /></li>

echo $f->asParagraph();

<p><label for="subject">Subject:</label> <input id="subject" type="text"␣
↪name="subject" maxlength="100" required /></p>
<p><label for="message">Message:</label> <input type="text" name="message"␣
↪id="message" required /></p>
<p><label for="sender">Sender:</label> <input type="email" name="sender" id=
↪"sender" required /></p>
<p><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_
↪myself" id="cc_myself" /></p>
```

- If **autoId** is set to a string containing the format character **'%s'**, then the form output will include **<label>** tags, and will generate **id** attributes based on the format string. For example, for a format string **'field_%s'**, a field named subject will get the id value **'field_subject'**.

```
$f = new ContactForm(autoId=['autoId'=>'id_for_%s']);

echo $f->asTable ();

<tr><th><label for="id_for_subject">Subject:</label></th><td><input id="id_for_
↪subject" type="text" name="subject" maxlength="100" required /></td></tr>
<tr><th><label for="id_for_message">Message:</label></th><td><input type="text"␣
↪name="message" id="id_for_message" required /></td></tr>
<tr><th><label for="id_for_sender">Sender:</label></th><td><input type="email"␣
↪name="sender" id="id_for_sender" required /></td></tr>
<tr><th><label for="id_for_cc_myself">Cc myself:</label></th><td><input type=
↪"checkbox" name="cc_myself" id="id_for_cc_myself" /></td></tr>

echo $f->asUl ();

<li><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type=
↪"text" name="subject" maxlength="100" required /></li>
<li><label for="id_for_message">Message:</label> <input type="text" name="message
↪" id="id_for_message" required /></li>
<li><label for="id_for_sender">Sender:</label> <input type="email" name="sender"␣
↪id="id_for_sender" required /></li>
<li><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name=
↪"cc_myself" id="id_for_cc_myself" /></li>

echo $f->asParagraph();

<p><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type=
↪"text" name="subject" maxlength="100" required /></p>
<p><label for="id_for_message">Message:</label> <input type="text" name="message"␣
↪id="id_for_message" required /></p>
<p><label for="id_for_sender">Sender:</label> <input type="email" name="sender"␣
↪id="id_for_sender" required /></p>
<p><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name=
↪"cc_myself" id="id_for_cc_myself" /></p>
```

If **autoId** is set to any other true value – such as a string that doesn't include **%s** – then the library will act as if **autoId** is **true**.

By default, **autoId** is set to the string **'id_%s'**.

Form.labelSuffix

A translatable string (defaults to a colon (:) in English) that will be appended after any label name when a form is rendered.

It's possible to customize that character, or omit it entirely, using the **labelSuffix** parameter:

```
$f = new ContactForm([autoId=>'id_for_%s', labelSuffix=>'']);

echo $f->asUl();

<li><label for="id_for_subject">Subject</label> <input id="id_for_subject" type="text
↪" name="subject" maxlength="100" required /></li>
<li><label for="id_for_message">Message</label> <input type="text" name="message" id=
↪"id_for_message" required /></li>
<li><label for="id_for_sender">Sender</label> <input type="email" name="sender" id=
↪"id_for_sender" required /></li>
<li><label for="id_for_cc_myself">Cc myself</label> <input type="checkbox" name="cc_
↪myself" id="id_for_cc_myself" /></li>
$f = new ContactForm(auto_id='id_for_%s', label_suffix=' ->')

echo $f->asUl();

<li><label for="id_for_subject">Subject -></label> <input id="id_for_subject" type=
↪"text" name="subject" maxlength="100" required /></li>
<li><label for="id_for_message">Message -></label> <input type="text" name="message"␣
↪id="id_for_message" required /></li>
<li><label for="id_for_sender">Sender -></label> <input type="email" name="sender" id=
↪"id_for_sender" required /></li>
<li><label for="id_for_cc_myself">Cc myself -></label> <input type="checkbox" name=
↪"cc_myself" id="id_for_cc_myself" /></li>
```

Note that the label suffix is added only if the last character of the label isn't a punctuation character (in English, those are ., !, ? or :).

Fields can also define their own **labelSuffix**. This will take precedence over **Form.labelSuffix**.

Binding uploaded files to a form

Dealing with forms that have *FileField* and *ImageField* fields is a little more complicated than a normal form.

Firstly, in order to upload files, you'll need to make sure that your <form> element correctly defines the enctype as "multipart/form-data":

```
<form enctype="multipart/form-data" method="post" action="/foo/">
```

Secondly, when you use the form, you need to bind the file data. File data is handled separately to normal form data, so when your form contains a *FileField* and *ImageField* , you will need to specify a second argument when you bind your form. So if we extend our ContactForm to include an *ImageField* called mugshot,we need to bind the file data containing the mugshot image:

Note: more to come soon

Testing for multipart forms

isMultipart()

If you're writing reusable views or templates, you may not know ahead of time whether your form is a multipart form or not. The **isMultipart()** method tells you whether the form requires multipart encoding for submission:

Prefixes for forms

prefix

You can put several Powerform forms inside one **<form>** tag. To give each Form its own namespace, use the prefix keyword argument:

Form fields

- *ImageField*

Core field arguments

Each Field class constructor takes at least these arguments. Some Field classes take additional, field-specific arguments, but the following should always be accepted:

required

By default, each Field class assumes the value is required, so if you pass an empty value – either **null** or the **empty** string (“”) – then **clean()** will raise a **ValidationError** exception.

label

The label argument lets you specify the “human-friendly” label for this field. This is used when the Field is displayed in a Form.

widget

The widget argument lets you specify a Widget class to use when rendering this Field. See *Widgets* for more information.

helpText

The helpText argument lets you specify descriptive text for this Field. If you provide helpText, it will be displayed next to the Field when the Field is rendered by one of the convenience Form methods (e.g., asUl()).

Like the model field’s help_text, this value isn’t HTML-escaped in automatically-generated forms.

validators

The validators argument lets you provide a list of validation functions for this field.

See the *validators* documentation for more information.

disabled

The disabled boolean argument, when set to **true**, disables a form field using the **disabled** HTML attribute so that it won’t be editable by users. Even if a user tampers with the field’s value submitted to the server, it will be ignored in favor of the value from the form’s initial data.

Built-in Field classes

Naturally, the forms library comes with a set of Field classes that represent common validation needs. This section documents each built-in field.

For each field, we describe: - The default widget used if you don't specify widget. - The value returned when you provide an empty value (see the section on required above to understand what that means).

IntegerField

- Default widget: *NumberInput*.
- Empty value: **null**
- Normalizes to: A php **integer**.
- Validates that the given value is an integer. Leading and trailing whitespace is allowed, as in php's **intval()** function.
- Error message keys: **required**, **invalid**, **maxValue**, **minValue**.

The **maxValue** and **minValue** error messages may contain **%(limit_value)s**, which will be substituted by the appropriate limit.

Takes two optional arguments for validation:

maxValue

min_value

These control the range of values permitted in the field.

BooleanField

- Default widget: *CheckboxInput*
- Empty value: **false**
- Normalizes to: A php boolean **true** or **false** value.
- Validates that the value is **true** (e.g. the check box is checked) if the field has **required=true**.
- Error message keys: **required**

Note: Since all Field subclasses have **required=true** by default, the validation condition here is important. If you want to include a boolean in your form that can be either True or False (e.g. a checked or unchecked checkbox), you must remember to pass in **required=false** when creating the **BooleanField**.

CharField

- Default widget: *TextInput*
- Empty value: '' (an empty string)
- Normalizes to: A string.
- Validates **maxLength** or **minLength**, if they are provided. Otherwise, all inputs are valid.

- Error message keys: **required**, **maxLength**, **minLength**

Has three optional arguments for validation:

maxLength

minLength

If provided, these arguments ensure that the string is at most or at least the given length.

strip

If True (default), the value will be stripped of leading and trailing whitespace.

URLField

- Default widget: *URLInput*
- Empty value: '' (an empty string)
- Normalizes to: A string.
- Validates that the given value is a valid URL.
- Error message keys: **required**, **invalid**

Takes the following optional arguments:

maxLength

minLength These are the same as *CharField.maxLength* and *CharField.minLength*.

EmailField

- Default widget: *EmailInput*
- Empty value: '' (an empty string)
- Normalizes to: A string.
- Validates that the given value is a valid email address, using a moderately complex regular expression.
- Error message keys: **required**, **invalid**

Has two optional arguments for validation, **minLength** and **minLength**. If provided, these arguments ensure that the string is at most or at least the given length.

SlugField

- Default widget: *TextInput*
- Empty value: '' (an empty string)
- Normalizes to: A string.
- Validates that the given value contains only letters, numbers, underscores, and hyphens.
- Error message keys: **required**, **invalid**

DecimalField

- Default widget: *NumberInput*
- Empty value: **null** (an empty string)
- Normalizes to: A float.
- Validates that the given value is a decimal. Leading and trailing whitespace is ignored..
- Error message keys: **required**, **invalid**, **maxValue**, **minValue**, **maxDigits**, **maxDecimalPlaces**, **maxWholeDigits**

The **maxValue** and **minValue** error messages may contain **%(limit_value)s**, which will be substituted by the appropriate limit.

Similarly, the **maxDigits**, **maxDecimalPlaces**, and **maxWholeDigits** error messages may contain **%(max)s**.

Takes four optional arguments:

maxValue

minValue These control the range of values permitted in the field, and should be given as float values.

maxDigits

The maximum number of digits (those before the decimal point plus those after the decimal point, with leading zeros stripped) permitted in the value.

maxDecimalPlaces

The maximum number of decimal places permitted.

DateField

- Default widget: *DateInput*
- Empty value: **null**
- Normalizes to: A PHP **DateTime** object.
- Validates that the given value is either a **DateTime** or string formatted in a particular date format.
- Error message keys: **required**, **invalid**.

Takes one optional argument:

input_formats

A list of formats used to attempt to convert a string to a valid **DateTime** object.

If no input_formats argument is provided, the default input formats are:

```
[
   'Y-m-d',       // '2006-10-25'
   'm/d/Y',       // '10/25/2006'
   'm/d/y'      // '10/25/06'
]
```

TimeField

- Default widget: *TextInput*
- Empty value: None
- Normalizes to: A PHP **DateTime** object.
- Validates that the given value is either a **DateTime** or string formatted in a particular time format.
- Error message keys: **required**, **invalid**.

Takes one optional argument:

input_formats

A list of formats used to attempt to convert a string to a valid **DateTime** object.

If no input_formats argument is provided, the default input formats are:

```
[
    'H:i:s',      // '14:30:59'
    'H:i:s.u',  // '14:30:59.000200'
    'H:i',        // '14:30'
];
```

ChoiceField

- Default widget: *Select*
- Empty value: **''** (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value exists in the list of choices.
- Error message keys: **required**, **invalid_choice**

The **invalid_choice** error message may contain **%(value)s**, which will be replaced with the selected choice.

Takes one extra required argument:

choices

Either an associative array to use as choices for this field, or a callable that returns such an array. This argument accepts the same formats as the **choices** argument to a model field. See the *model field reference documentation on choices* for more details. If the argument is a callable, it is evaluated each time the field's form is initialized.

MultipleChoiceField

- Default widget: *SelectMultiple*
- Empty value: **[]** (an empty list)
- Normalizes to: A list of php objects.
- Validates that every value in the given list of values exists in the list of choices.
- Error message keys: **required**, **invalid_choice**, **invalid_list**

The **invalid_choice** error message may contain **%(value)s**, which will be replaced with the selected choice.

Takes one extra required argument, **choices**, as for *ChoiceField*.

MultipleChoiceField

to be added

SplitDateTimeField

to be added

FileField

to be added

ImageField

to be added

Widgets

A widget is Powerform's representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.

The HTML generated by the built-in widgets uses HTML5 syntax, targeting <!DOCTYPE html>. For example, it uses boolean attributes such as **checked** rather than the XHTML style of **checked='checked'**.

Note: Widgets should not be confused with the form *fields*. Form fields deal with the logic of input validation and are used directly in templates. Widgets deal with rendering of HTML form input elements on the web page and extraction of raw submitted data. However, widgets do need to be *assigned* to form fields.

Specifying widgets

Whenever you specify a field on a form, Powerform will use a default widget that is appropriate to the type of data that is to be displayed. To find which widget is used on which field, see the documentation about *Built-in Field* classes.

However, if you want to use a different widget for a field, you can just use the widget argument on the field definition. For example:

```php
namespace App\Forms;


use Eddmash\PowerOrm\Form\Form;

class CommentForm extends Form
{
    /**
     * @inheritDoc
     */
    public function fields()
    {
        return [
            'name'=>Form::CharField(),
            'url'=>Form::UrlField(),
            'comment'=>Form::CharField(['widget'=>Form::TextArea()])
        ];
    }

}
```

This would specify a form with a comment that uses a larger *Textarea* widget, rather than the default *TextInput* widget.

Built-in widgets

Powerform provides a representation of all the basic HTML widgets, plus some commonly used groups of widgets in the **EddmashPowerOrmFormWidgets** module, including the *input of text*, various checkboxes and selectors, uploading files, and handling of multi-valued input.

Widgets handling input of text

These widgets make use of the HTML elements **input** and **textarea**.

TextInput

Text input: **<input type="text" ...>**

NumberInput

Text input: **<input type="number" ...>**

Beware that not all browsers support entering localized numbers in number input types.

EmailInput

Text input: **<input type="email" ...>**

URLInput

Text input: **<input type="url" ...>**

PasswordInput

Password input: **<input type='password' ...>**

HiddenInput

Hidden input: **<input type='hidden' ...>**

Note that there also is a MultipleHiddenInput widget that encapsulates a set of hidden input elements.

DateInput

Date input as a simple text box: **<input type='text' ...>**

Takes same arguments as *TextInput*, with one more optional argument:

format

The format in which this field's initial value will be displayed.

If no format argument is provided, the default format is the first format found in DATE_INPUT_FORMATS.

Textarea

Text area: <textarea>...</textarea>

Selector and checkbox widgets

CheckboxInput

Checkbox: <input type='checkbox' ...>

Takes one optional argument:

check_test

A callable that takes the value of the CheckboxInput and returns True if the checkbox should be checked for that value.

Select

Select widget: <select><option ...>...</select>

choices

This attribute is optional when the form field does not have a choices attribute. If it does, it will override anything you set here when the attribute is updated on the Field.

NullBooleanSelect

Select widget with options 'Unknown', 'Yes' and 'No'

SelectMultiple

Similar to *select*, but allows multiple selection: <select multiple='multiple'>...</select>

RadioSelect

Similar to *select*, but rendered as a list of radio buttons within <li> tags:

```
<ul>
  <li><input type='radio' name='...'></li>
  ...
</ul>
```

CheckboxSelectMultiple

Similar to *SelectMultiple*, but rendered as a list of check buttons:

```
<ul>
  <li><input type='checkbox' name='...' ></li>
  ...
</ul>
```

Creating forms from models

ModelForm

If you're building a database-driven app, chances are you'll have forms that map closely to Powerorm models. For instance, you might have a BlogComment model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model. we will be using the *Author model*.

For this reason, Powerform provides a helper class that lets you create a Form class from a Powerorm model.

For example:

```
namespace App\Forms;


use Eddmash\PowerOrm\Form\Form;
```

(continues on next page)

(continued from previous page)

```
use Eddmash\PowerOrm\Form\ModelForm;
use Eddmash\PowerOrm\Form\Validations\MaxValueValidator;
use Eddmash\PowerOrm\Form\Widgets\DateInput;
use Eddmash\PowerOrm\Form\Widgets\EmailInput;
use Eddmash\PowerOrm\Form\Widgets\NumberInput;
use Eddmash\PowerOrm\Form\Widgets\TextInput;

/**
 * Class AuthorForm
 * @package App\Forms
 * @author: Eddilbert Macharia (http://eddmash.com)<edd.cowan@gmail.com>
 */
class AuthorForm extends ModelForm
{
    protected $modelClass = 'App\Models\Author';
    protected $excludes = ['id'];
}
```

Field types

The generated Form class will have a form field for every model field specified, in the order specified in the fields attribute.

Each model field has a corresponding default form field. For example, a *CharField* on a model is represented as a *CharField* on a form. A model ManyToManyField is represented as a MultipleChoiceField.

Here is the full list of conversions:

Model Field	Form Field
AutoField	Not represented in the form
CharField	*CharField*
BooleanField	*BooleanField*
UrlField	*UrlField*
DateField	*DateField*
EmailField	*EmailField*
DecimalField	*DecimalField*
ImageField	*ImageField*
IntegerField	*IntegerField*
SlugField	*SlugField*
TextField	TextField
ForeignKey	ModelChoiceField (see below)
ManyToMany	ModelMultipleChoiceField (see below)

As you might expect, the *ForeignKey* and *ForeignKey* model field types are special cases:

ForeignKey is represented by ModelChoiceField, which is a *ChoiceField* whose choices are a model *Queryset*.

ManyToMany is represented by ModelMultipleChoiceField, which is a *MultipleChoiceField* whose choices are a model *Queryset*.

In addition, each generated form field has attributes set as follows:

- If the model field has **blank=true**, then **required** is set to **false** on the form field. Otherwise, **required=true**.
- The form field's **label** is set to the **verboseName** of the model field, with the first character capitalized.

- The form field's **helpText** is set to the **helpText** of the model field.
- If the model field has **choices** set, then the form field's **widget** will be set to **Select**, with choices coming from the model field's **choices**. The choices will normally include the blank choice which is selected by default. If the field is required, this forces the user to make a selection. The blank choice will not be included if the model field has **blank=false** and an explicit default value (the **default** value will be initially selected instead).

Finally, note that you can override the form field used for a given model field. See Overriding the default fields below.

Overriding the default fields

The default field types, as described in the Field types table above, are sensible defaults. If you have a **DateField** in your model, chances are you'd want that to be represented as a **DateField** in your form. But **ModelForm** gives you the flexibility of changing the form field for a given model.

To specify a custom widget for a field, use the **widgets()** method of **ModelForm** class. This should be a associative array mapping field names to widget classes or instances.

For example, if you want the **CharField** for the name attribute of Author to be represented by a **<textarea>** instead of its default **<input type="text">**, you can override the field's widget:

```
namespace App\Forms;

use App\Models\Author;
use Eddmash\PowerOrm\Form\Form;
use Eddmash\PowerOrm\Form\ModelForm;

class AuthorForm extends ModelForm
{
    protected $modelFields = ['name', 'email'];
    protected $modelClass = 'App\Models\Author';

    /**
     * @inheritDoc
     */
    public function widgets()
    {
        return [
            'name'=>Form::TextArea(['cols'=>80, 'rows'=>20])
        ];
    }


}
```

The widgets() method returns an associatie array with field name as key and either widget instances (e.g., **Textarea(...)**).

Similarly, you can specify the **labels**, **helpTexts** and **errorMessages** methods if you want to further customize a field.

You can also specify **fieldClasses** to customize the type of fields instantiated by the form.For example, if you wanted to use **MySlugFormField** for the slug fieldFor example, if you wanted to use **MySlugFormField** for the slug field

For example if you wanted to customize the wording of all user facing strings for the name field:

```
namespace App\Forms;

use App\Models\Author;
```

(continues on next page)

(continued from previous page)

```
use Eddmash\PowerOrm\Form\Form;
use Eddmash\PowerOrm\Form\ModelForm;

class AuthorForm extends ModelForm
{
    protected $modelFields = ['name', 'email', 'content', 'reporter', 'slug'];
    protected $modelClass = 'App\Models\Author';

    /**
     * @inheritDoc
     */
    public function widgets()
    {
        return [
            'name'=>Form::TextArea(['cols'=>80, 'rows'=>20])
        ];
    }

    /**
     * @inheritDoc
     */
    public function labels()
    {
        return [
            'name'=>"Your Name"
        ];
    }

    /**
     * @inheritDoc
     */
    public function helpTexts()
    {
        return [
            'name'=>"whats your name ?"
        ];
    }

    /**
     * @inheritDoc
     */
    public function fieldClasses()
    {
        return [
            'slug'=> MySlugField::class
        ];
    }
}
```

Finally, if you want complete control over of a field – including its type, validators, required, etc. – you can do this by declaratively specifying fields like you would in a regular **Form**.

If you want to specify a field's validators, you can do so by defining the field declaratively and setting its **validators** parameter:

```
use App\Models\Author;
use Eddmash\PowerOrm\Form\Form;
```

(continues on next page)

(continued from previous page)

```
use Eddmash\PowerOrm\Form\ModelForm;
use Eddmash\PowerOrm\Form\Validations\SlugValidator;

class AuthorForm extends ModelForm
{
    protected $modelFields = ['name', 'email', 'content', 'reporter', 'slug'];
    protected $modelClass = 'App\Models\Author';

    /**
     * @inheritDoc
     */
    public function fields()
    {
        return [
            'slug'=>Form::CharField(['validators'=>[SlugValidator::instance()]])
        ];
    }
}
```

Note: When you explicitly instantiate a form field like this, it is important to understand how **ModelForm** and regular **Form** are related.

ModelForm is a regular **Form** which can automatically generate certain fields. The fields that are automatically generated depend on the content returns by **fields()** method on which fields have already been defined declaratively. Basically, **ModelForm** will only generate fields that are **missing** from the form, or in other words, fields that weren't defined declaratively.

Fields defined declaratively are left as-is, therefore any customizations made by any of the methods shown above such as **widgets**, **labels**, **helpTexts**, or **errorMessages** are ignored; these only apply to fields that are generated automatically.

Similarly, fields defined declaratively do not draw their attributes like **maxLength** or **required** from the corresponding model. If you want to maintain the behavior specified in the model, you must set the relevant arguments explicitly when declaring the form field.

Providing initial values

As with regular forms, it's possible to specify initial data for forms by specifying an initial parameter when instantiating the form. Initial values provided this way will override both initial values from the form field and values from an attached model instance. For example:

```
$article = Article::objects()->get(['pk'=>1]);
echo $article->headline;
'My headline'
$form = new ArticleForm(['initial'=>['headline'=> 'Initial headline'], 'instance'=>
↪$article]);
echo $form['headline']->value();
'Initial headline'
```

Overriding the clean() method

You can override the **clean()** method on a model form to provide additional validation in the same way you can on a normal form.

A model form instance attached to a model object will contain an **modelInstance** attribute that gives its methods access to that specific model instance.

The save() method

Every ModelForm also has a save() method. This method creates and saves a database object from the data bound to the form. A subclass of ModelForm can accept an existing model instance as the keyword argument instance; if this is supplied, save() will update that instance. If it's not supplied, save() will create a new instance of the specified model:

```
//Create a form instance from POST data
$form = new AuthorForm(['data'=>$_POST]);
$form->save(false);

// Create a form to edit an existing Article, but use
//  POST data to populate the form.
$a = Article::objects()->get(['pk'=>1]);

$form = new AuthorForm(['data'=>$_POST, 'instance'=>$a]);
$form->save(false);
```

Note that if the form hasn't been validated, calling **save()** will do so by checking **form.errors()**. A **ValueError** will be raised if the data in the form doesn't validate – i.e., if **form.errors()** evaluates to **true**.

Form and field validation

Form validation happens when the data is cleaned. If you want to customize this process, there are various places to make changes, each one serving a different purpose. Three types of cleaning methods are run during form processing. These are normally executed when you call the **isValid()** method on a form. There are other things that can also trigger cleaning and validation (accessing the **errors()** method or calling **fullClean()** directly), but normally they won't be needed.

In general, any cleaning method can raise **ValidationError** if there is a problem with the data it is processing, passing the relevant information to the **ValidationError** constructor. *See below* for the best practice in raising **ValidationError**. If no **ValidationError** is raised, the method should return the cleaned (normalized) data as a PHP object.

Most validation can be done using *validators* - simple helpers that can be reused easily. Validators are objects that take a single argument when invoked like functions and raise ValidationError on invalid input. Validators are run after the field's **toPhp** and **validate** methods have been called.

Validation of a form is split into several steps, which can be customized or overridden:

- The **toPhp()** method on a **Field** is the first step in every validation. It coerces the value to a correct datatype and raises **ValidationError** if that is not possible. This method accepts the raw value from the widget and returns the converted value. For example, a **DateField** will turn the data into a Php DateTime or raise a **ValidationError**.
- The **validate()** method on a **Field** handles field-specific validation that is not suitable for a validator. It takes a value that has been coerced to a correct datatype and raises **ValidationError** on any error. This method does not return anything and shouldn't alter the value. You should override it to handle validation logic that you can't or don't want to put in a validator.
- The **runValidators()** method on a **Field** runs all of the field's validators and aggregates all the errors into a single **ValidationError**. You shouldn't need to override this method.
- The **clean()** method on a **Field** subclass is responsible for running **toPhp()**, **validate()**, and **runValidators()** in the correct order and propagating their errors. If, at any time, any of the methods raise **ValidationError**, the validation stops and that error is raised. This method returns the clean data, which is then inserted into the **cleanedData** associative array of the form.
- The **clean<fieldname>()** method is called on a form subclass – where <fieldname> is replaced with the name of the form field attribute. This method does any cleaning that is specific to that particular attribute, unrelated to the type of field that it is. This method is not passed any parameters. You will need to look up the value of the field in **$this->cleanedData** and remember that it will be a Php object at this point, not the original string submitted in the form (it will be in **cleanedData** because the general field **clean()** method, above, has already cleaned the data once).

  For example, if you wanted to validate that the contents of a **CharField** called **serialnumber** was unique, cleanSerialnumber() would be the right place to do this. You don't need a specific field (it's just a CharField), but you want a formfield-specific piece of validation and, possibly, cleaning/normalizing the data.

  The return value of this method replaces the existing value in **cleanedData**, so it must be the field's value from **cleanedData** (even if this method didn't change it) or a new cleaned value.
- The form subclass's **clean()** method can perform validation that requires access to multiple form fields. This is where you might put in checks such as "if field A is supplied, field B must contain a valid email address". This method can return a completely different associative array if it wishes, which will be used as the **cleanedData**.

Since the field validation methods have been run by the time **clean()** is called, you also have access to the form's **errors** attribute which contains all the errors raised by cleaning of individual fields.

Note that any errors raised by your **Form.clean()** override will not be associated with any field in particular. They go into a special "field" (called **__all__**), which you can access via the *nonFieldErrors()* method if you need to. If you want to attach errors to a specific field in the form, you need to call *addError()*.

Also note that there are special considerations when overriding the **clean()** method of a ModelForm subclass. (see the *ModelForm documentation* for more information)

These methods are run in the order given above, one field at a time. That is, for each field in the form (in the order they are declared in the form definition), the **Field.clean()** method (or its override) is run, then **clean<fieldname>()**.Finally, once those two methods are run for every field, the **Form.clean()** method, or its override, is executed whether or not the previous methods have raised errors.

Examples of each of these methods are provided below.

As mentioned, any of these methods can raise a **ValidationError**. For any field, if the **Field.clean()** method raises a **ValidationError**, any field-specific cleaning method is not called. However, the cleaning methods for all remaining fields are still executed.

Raising ValidationError

In order to make error messages flexible and easy to override, consider the following guidelines:

Provide a descriptive error code to the constructor:

```
// Good
ValidationError('Invalid value', 'invalid');

// Bad
ValidationError('Invalid value');
```

Putting it all together:

```
throw new ValidationError('Invalid value', 'invalid');
```

Following these guidelines is particularly necessary if you write reusable forms, form fields, and model fields.

While not recommended, if you are at the end of the validation chain (i.e. your form clean() method) and you know you will never need to override your error message you can still opt for the less verbose:

Raising multiple errors

If you detect multiple errors during a cleaning method and wish to signal all of them to the form submitter, it is possible to pass a list of errors to the **ValidationError** constructor.

As above, it is recommended to pass a list of **ValidationError** instances with codes and params but a list of strings will also work:

```
// Good
throw new ValidationError([
    ValidationError('Error 1', 'error1'),
    ValidationError('Error 2', 'error2'),
])

// Bad
throw new ValidationError([
    _('Error 1'),
    _('Error 2'),
])
```

Using validation in practice

The previous sections explained how validation works in general for forms. Since it can sometimes be easier to put things into place by seeing each feature in use, here are a series of small examples that use each of the previous features.

Using validators

Powerform's form (and model) fields support use of simple utility classes known as validators. A validator is merely a callable object that takes a value and simply returns nothing if the value is valid or throws a **ValidationError** if not. These can be passed to a field's constructor, via the field's validators argument, or defined on the Field class itself with the **getDefaultValidators()** method.

Simple validators can be used to validate values inside the field, let's have a look at Powerform's SlugField:

```
class SlugField extends CharField
{
    /**
     * @inheritDoc
     */
    public function getDefaultValidators()
    {
        $validators = parent::getDefaultValidators();
        $validators[] = SlugValidator::instance();
        return $validators;
    }

}
```

As you can see, **SlugField** is just a **CharField** with a customized validator that validates that submitted text obeys to some character rules. This can also be done on field definition so:

```
$slug = Form::SlugField();
```

is equivalent to:

```
$slug = Form::CharField(['validators'=>[SlugValidator::instance()]]);
```

Form field default cleaning

Let's first create a custom form field that validates its input is a string containing comma-separated email addresses. The full class looks like this:

```
namespace App\Forms;


use Eddmash\PowerOrm\Form\Fields\Field;
use Eddmash\PowerOrm\Form\Validations\EmailValidator;

class MultiEmailField extends Field
{
    public function toPhp($value)
    {
        if (empty($value)) :
            return [];
        endif;

        return explode(",", $value);
    }

    /**
     * @inheritDoc
     */
    public function validate($value)
    {
        foreach ($value as $item) :
            $validator = EmailValidator::instance();
            $validator($item);
        endforeach;
```

(continues on next page)

(continued from previous page)

```
    }

}
```

Every form that uses this field will have these methods run before anything else can be done with the field's data. This is cleaning that is specific to this type of field, regardless of how it is subsequently used.

Let's create a simple ContactForm to demonstrate how you'd use this field:

```
class ContactForm extends Form
{
    public function fields()
    {
        return [
            'subject' => Form::CharField(['maxLength'=>100]),
            'recipients' => MultiEmailField::instance(),
            'cc_myself' => Form::BooleanField(['required' => false]),
        ];
    }
}
```

Simply use **MultiEmailField** like any other form field. When the **isValid()** method is called on the form, the **MultiEmailField.clean()** method will be run as part of the cleaning process and it will, in turn, call the custom **toPhp()** and **validate()** methods.

Cleaning a specific field attribute

Continuing on from the previous example, suppose that in our **ContactForm**, we want to make sure that the recipients field always contains the address **"fred@example.com"**. This is validation that is specific to our form, so we don't want to put it into the general **MultiEmailField** class. Instead, we write a cleaning method that operates on the recipients field, like so:

```
class ContactForm extends Form
{
    public function fields()
    {
        return [
            'subject' => Form::CharField(['maxLength'=>100]),
            'recipients' => MultiEmailField::instance(),
            'cc_myself' => Form::BooleanField(['required' => false]),
        ];
    }

    public function cleanRecipients()
    {
        $data = $this->cleanedData['recipients'];
        if (!in_array("fred@example.com", $data)) :
            throw new ValidationError("You have forgotten about Fred!");
        endif;
        return $data;
    }
}
```

Cleaning and validating fields that depend on each other

Suppose we add another requirement to our contact form: if the **cc_myself** field is **true**, the subject must contain the word **"help"**. We are performing validation on more than one field at a time, so the form's **clean()** method is a good spot to do this. Notice that we are talking about the **clean()** method on the form here, whereas earlier we were writing a **clean()** method on a field. It's important to keep the field and form difference clear when working out where to validate things. Fields are single data points, forms are a collection of fields.

By the time the form's **clean()** method is called, all the individual field clean methods will have been run (the previous two sections), so **$this->cleanedData** will be populated with any data that has survived so far. So you also need to remember to allow for the fact that the fields you are wanting to validate might not have survived the initial individual field checks.

There are two ways to report any errors from this step. Probably the most common method is to display the error at the top of the form. To create such an error, you can raise a **ValidationError** from the **clean()** method. For example:

```
class ContactForm extends Form
{
    // .. everything before

    public function clean()
    {
        parent::clean();

        if (array_key_exists('cc_myself', $this->cleanedData) &&
            array_key_exists('recipients', $this->cleanedData) &&
            array_key_exists('subject', $this->cleanedData)
        ) :
            $ccMyself = $this->cleanedData['cc_myself'];
            $recipients = $this->cleanedData['recipients'];
            $subject = $this->cleanedData['subject'];
            if ($ccMyself && $recipients) :

                if (!strlen(strstr($subject, 'help'))) :
                    throw new ValidationError(
                        "Did not send for 'help' in the subject despite CC'ing␣
↪yourself."
                    );
                endif;
            endif;
        endif;
    }
}
```

In this code, if the validation error is raised, the form will display an error message at the top of the form (normally) describing the problem.

The call to **parent::clean()** in the example code ensures that any validation logic in parent classes is maintained. use **$this->cleanedData** to access cleaned field data.

The second approach for reporting validation errors might involve assigning the error message to one of the fields. In this case, let's assign an error message to both the "subject" and "cc_myself" rows in the form display. Be careful when doing this in practice, since it can lead to confusing form output. We're showing what is possible here and leaving it up to you and your designers to work out what works effectively in your particular situation. Our new code (replacing the previous sample) looks like this:

```
class ContactForm extends Form
{
```

(continues on next page)

(continued from previous page)

```
    // .. everything before

    public function clean()
    {
        parent::clean();

        if (array_key_exists('cc_myself', $this->cleanedData) &&
            array_key_exists('recipients', $this->cleanedData) &&
            array_key_exists('subject', $this->cleanedData)
        ) :
            $ccMyself = $this->cleanedData['cc_myself'];
            $recipients = $this->cleanedData['recipients'];
            $subject = $this->cleanedData['subject'];
            if ($ccMyself && $recipients) :

                if (!strlen(strstr($subject, 'help'))) :
                    $msg = "Did not send for 'help' in the subject despite CC'ing␣
↪yourself.";
                    $this->addError("cc_myself", $msg);
                    $this->addError("subject", $msg);
                endif;
            endif;
        endif;
    }
}
```

The second argument of *addError()* can be a simple string, or preferably an instance of **ValidationError**. See *Raising Validation errors* for more details. Note that *addError()* automatically removes the field from **cleanedData**.

Validators

Writing validators

A validator is a callable that takes a value and raises a **ValidationError** if it doesn't meet some criteria. Validators can be useful for re-using validation logic between different types of fields.

For example, here's a validator that only allows even numbers:

```
function validate_even($value)
{
    if ($value % 2 != 0):
        throw new ValidationError(sprintf('%s is not an even number', $value),
↪'invalid');
    endif;
}
```

You can add this to a model field via the field's validators argument:

```
namespace App\Forms;

use App\Models\Blog;
use Eddmash\PowerOrm\Exception\ValidationError;
use Eddmash\PowerOrm\Form\Form;
use Eddmash\PowerOrm\Form\ModelForm;
```

(continues on next page)

(continued from previous page)

```
class BlogForm extends ModelForm
{
    protected $modelFields = "__all__";
    /**
     * @inheritDoc
     */
    public function getModelClass()
    {
        return Blog::class;
    }

    /**
     * @inheritDoc
     */
    public function fields()
    {
        return [
          'name'=>Form::CharField(['validators'=>['App\Forms\validate_even']])
        ];
    }


}
```

Because values are converted to php before validators are run, you can even use the same validator with forms:

```
namespace App\Forms;

use Eddmash\PowerOrm\Exception\ValidationError;
use Eddmash\PowerOrm\Form\Form;

class CommentForm extends Form
{
    /**
     * @inheritDoc
     */
    public function fields()
    {
        return [
            'name' => Form::CharField(),
            'url' => Form::UrlField(),
            'even_field'=>Form::IntegerField(['validators'=>['validate_even']]),
            'moderate' => Form::BooleanField(['required' => false]),
        ];
    }



}
```

You can also use a class with a **__invoke()** method for more complex or configurable validators.

How validators are run

See the *form validation* for more information on how validators are run in forms, and Validating objects for how they're run in models. Note that validators will not be run automatically when you save a model, but if you are using a

ModelForm, it will run your validators on any fields that are included in your form. See the *ModelForm* documentation for information on how model validation interacts with forms.

Model Form Example

The Model Class

```
namespace App\Models;

use Eddmash\PowerOrm\Model\Model;

/**
* Class Author
*/
class Author extends Model
{
    public function unboundFields()
    {
        return [
            'name'=>Model::CharField(['maxLength'=>25]),
            'date'=>Model::DateField(),
        ];
    }

}
```

The form that represents the *Author Model*

The Model Form Class

```
namespace App\Forms;


use Eddmash\PowerOrm\Form\Form;
use Eddmash\PowerOrm\Form\ModelForm;
use Eddmash\PowerOrm\Form\Validations\MaxValueValidator;
use Eddmash\PowerOrm\Form\Widgets\DateInput;
use Eddmash\PowerOrm\Form\Widgets\EmailInput;
use Eddmash\PowerOrm\Form\Widgets\NumberInput;
use Eddmash\PowerOrm\Form\Widgets\TextInput;

/**
 * Class AuthorForm
 * @package App\Forms
 * @author: Eddilbert Macharia (http://eddmash.com)<edd.cowan@gmail.com>
 */
class AuthorForm extends ModelForm
{
    protected $modelClass = 'App\Models\Author';
    protected $excludes = ['id'];

    public function fields()
    {
        return [
```

(continues on next page)

(continued from previous page)

```
            'date2' => Form::DateField(
                [
                    'widget' => DateInput::instance(['class' => 'form-control']),
                    'required'=>false
                ]
            ),
            'age' => Form::IntegerField(
                [
                    'validators'=>[MaxValueValidator::instance(['max'=>10])],
                    'required'=>false,
                    'widget' => NumberInput::instance(['class' => 'form-control']),
                    'helpText'=>"What is your age"
                ]
            ),
            'email' => Form::EmailField([
                'required'=>false,
                'widget' => EmailInput::instance(['class' => 'form-control']),
            ]),
        ];
    }

    public function widgets()
    {
        return [
            'date' => DateInput::instance(['class' => 'form-control']),
            'name' => TextInput::instance(['class' => 'form-control']),
        ];
    }
}
```

The template

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
↪bootstrap.min.css"
      integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/
↪K68vbdEjh4u" crossorigin="anonymous">
<?php
/**@var $form \Eddmash\PowerOrm\Form\Form */
?>
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <form method="post" action="" enctype="multipart/form-data" novalidate>
                <?=$form;?>
                <input type="submit" value="Send" name="Send">
            </form>
        </div>
    </div>
</div>
```

The Rendering fields manually

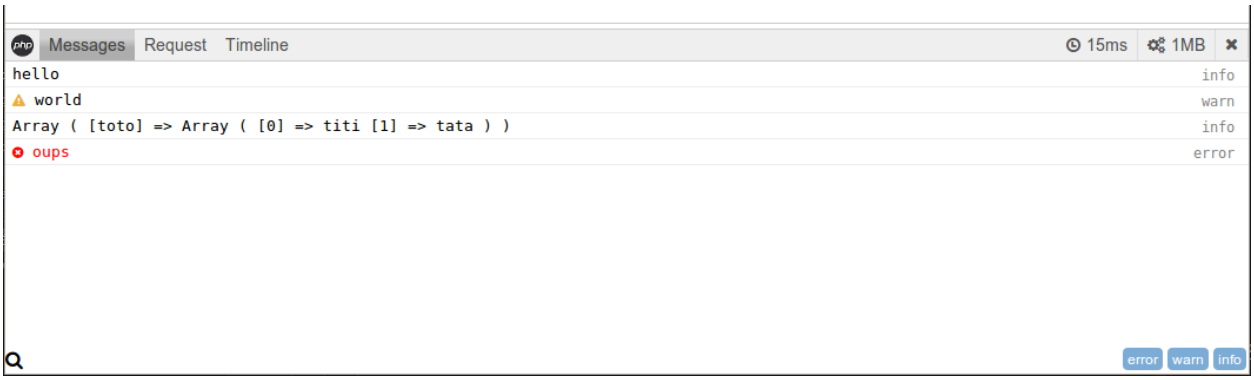
```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
↪bootstrap.min.css"
      integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/
↪K68vbdEjh4u" crossorigin="anonymous">
<?php
/**@var $form \Eddmash\PowerOrm\Form\Form */
?>
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <form method="post" action="" enctype="multipart/form-data" novalidate>

                <?php
                dump($form->nonFieldErrors());
                /**@var $field \Eddmash\PowerOrm\Form\Fields\Field */
                foreach ($form as $field):?>
                    <div class='form-group'>
                        <label for='"<?= $field->getIdForLabel(); ?>'><?= $field->
↪getLabelName(); ?></label>
                        <?= $field; ?>
                        <?= $field->getHelpText(); ?>
                    </div>
                <?php endforeach;
                ?>


                <input type="submit" value="Send" name="Send">
            </form>
        </div>
    </div>
</div>
```

6.3 Powerorm Debugbar Guide

Displays a debug bar in the browser with information from php. No more var_dump() in your code!



This is just wrapper to the PHP Debug Bar

6.3.1 Installation

Via composer **(recommended)**:

```
composer require eddmash/powerormdebug
```

Or add this to the composer.json file:

```
{
   "require": {
       "eddmash/powerormdebug": "1.1"(check latest version)
   }
}
```

6.3.2 Setup

To enable the debugbar, add it as component of the orm on the *components* setting as shown below.

```
use Eddmash\PowerOrm\BaseOrm;
use Eddmash\PowerOrmDebug\Toolbar;

$config = [
    // ..., other orm settings

    'components' => [
        Toolbar::class,
    ]
];
```

6.3.3 Usage

DebugBar is very easy to use and you can add it to any of your projects in no time.

- The debugger works by dropping all the static files it requires in the applications assets directory from where they will be served. The assets directory is set by calling the `setAssetsDirectory` method of the debugger as shown in the example below. This give you full control on how to serve this files
- You could also use `renderAssets` method to display this assets together as shown in the example below.
- Lastly to show the debugger invoke the `show` method.

Note: invoke the show() function at the end of the page so that its able to get all the sql queries performed

```
<?php

$orm = \Eddmash\PowerOrm\Loader::webRun(\App\Config\Powerorm::asArray());

/**@var $debugger \Eddmash\PowerOrmDebug\Debugger*/
$debugger =$orm->debugger;
$debugger->getDebugBar()["messages"]->addMessage("hello world!");
$debugger->setAssetsDirectory('assets');

?>
```

(continues on next page)

(continued from previous page)

```
<html>
    <head>
            <?= $debugger->renderAssets(); ?>
    </head>
    <body>
        ...
        <?php echo $debugger->show() ?>
    </body>
</html>
```

6.4 Powerorm Faker Guide

PowerOrmFaker is an extension of Faker library that generates fake data for the PowerOrm Library.

This is just wrapper to the PHP Faker

6.4.1 Installation

Via composer **(recommended)**:

```
composer require eddmash/powerormfaker
```

Or add this to the composer.json file:

```
{
   "require": {
       "eddmash/powerormfaker": "1.1"(check latest version)
   }
}
```

6.4.2 Setup

To enable the faker, just register it as a *component* with the orm on the as shown below.

```
use Eddmash\PowerOrmFaker\Generatedata;

$config = [
    // ..., other orm settings


    'components' =>[
        Faker::class,
    ],
];
```

6.4.3 Usage

```
php pmanager.php generatedata
```

Available options are :

-o, –only

The list of models to use when generating records.

```
php pmanager.php generatedata -o 'App\Models\Author' -o 'App\Models\Blog'
```

-i, –ignore

The list of models to ignore when generating records.

```
php pmanager.php generatedata -i 'App\Models\Entry'
```

-r, –records

The number of records to generate per model.

```
php pmanager.php generatedata -r 5
```

-s, –seed

To always get the same generated data. Calling generatedata twice with the same seed produces the same results

```
php pmanager.php generatedata -s 5000
```

6.4.4 Customizing the type of data generated.

This library guesses the kind of data to be set for each field on a model based on the

- name of the field, if its not able it uses,
- the type if model field is.

You have the ability to customize the kind of data generated by implementing the **FakeableInterface** in you model as shown below.

The **FakeableInterface** only has one method the *registerFormatter* method which should return an associative array of model field name as *key* and an anonymous function as its *value*.

The anonymous function should accept

- a *faker* object see *Using faker object* and
- *model instance* being populated as parameters and returns a valid php value e.g integer/string/date etc.

The *registerFormatter* method accepts one parameter, the *generator* object which you can use to register custom providers see *Adding Providers*

```
namespace App\Models;

use Eddmash\PowerOrm\Model\Model;
use Eddmash\PowerOrmFaker\FakeableInterface;

class User extends Model implements FakeableInterface
{
    public function unboundFields()
    {
        return [
            "username" => Model::CharField(['maxLength' => 50]),
```

(continues on next page)

(continued from previous page)

```
            "age" => Model::CharField(['maxLength' => 50]),
        ];

    }

    public function registerFormatter(Generator $generator)
    {
        return [
            "age" => function ($faker, $object) {
                return $faker->ipv4;
            },
        ];
    }
}
```

6.4.5 Using the faker object

```
// generate data by accessing properties

    echo $faker->randomDigit             // 7
    echo $faker->phoneNumber             // '201-886-0269 x3767'
    echo $faker->jobTitle                // 'Cashier'
    echo $faker->name;
          // 'Lucy Cechtelar';
    echo $faker->randomElements($array = array ('a','b','c'), $count = 1)    // array(
↪'c')
    echo $faker->address;
          // "426 Jordy Lodge
          // Cartwrightshire, SC 88120-6700"
    echo $faker->text;
          // Dolores sit sint laboriosam dolorem culpa et autem. Beatae nam sunt fugit
          // et sit et mollitia sed.
          // Fuga deserunt tempora facere magni omnis. Omnis quia temporibus␣
↪laudantium
          // sit minima sint.
```

See all available Localized Formatters and General Formatters on the faker object.

6.4.6 Adding Providers to Faker object

You can create you custom data providers for the faker as shown below.

The create the provider in this case we create a book provider.

```
namespace App\Provider;

use Faker\Provider\Base;

class BookProvider extends Base
{
    public function book_title($nbWords = 5)
    {
        $sentence = $this->generator->sentence($nbWords);
        return substr($sentence, 0, strlen($sentence) - 1);
```

(continues on next page)

(continued from previous page)

```
    }

    public function book_isbn()
    {
        return $this->generator->ean13();
    }
}
```

Register the custom provider with the generator on the **registerFormatter** method and now you can use the new formaters as shown below.

```
namespace App\Models;

use App\Provider\BookProvider;
use Eddmash\PowerOrm\Model\Model;
use Eddmash\PowerOrmFaker\FakeableInterface;
use Faker\Generator;

class Book extends Model implements FakeableInterface
{
    public function unboundFields()
    {
        return [
            "title" => Model::CharField(['maxLength' => 50]),
            "isbn" => Model::CharField(['maxLength' => 50]),
            "summary" => Model::CharField(['maxLength' => 50]),
        ];
    }

    public function registerFormatter(Generator $generator)
    {
        $generator->addProvider(new BookProvider($generator));

        return [
            "title" => function ($faker, $object) {
                return $faker->book_title;
            },
            "isbn" => function ($faker, $object) {
                return $faker->book_isbn;
            },
        ];
    }

}
```

For in depth details of how this work see Faker Internals: Understanding Providers

6.5 Phpgis Guide

Work in progress

6.6 The MIT License (MIT)

6.7 Change Log

Version 1.1.0-Pre-Alpha

Rewrite of the PowerOrm

Release Date: Not Released

6.7.1 Additions

- Registry

  This a registry of all the models the orm detected.

  - Manager

    This is in charge of managing all the command line related tasks for the orm, no need for users to create a migration controller to be able to work with the orm anymore

  - Doctrine Dbal

    Uses Doctrine Dbal to interact with the database.

  - Support
    - Migrations now supports all database supported by Doctrine Dbal
  - Migration Module:
    - Add Migrations commands
      - check

        - showmigration
        - migrate
        - makemigrations
        - robot
        - help
        - version
    - Migrations is able to support :
        - Alter/Remove/Add Operations relating to models.
- Forms class
    - You can deal with forms in two ways :
        - Use the default form provided by the ORM
        - this is achieved by calling the $this->orm->get_form() which helps you build your form.
        - Define a form class in the forms folder.
        - create a forms folder at the same level as the libraries folder, if it does not exist.
        - inside it, Define a class the extends PForm class
        - just like model, override the fields() method and define you forms fields
        - and just like models, you the form fields from PForm e.g. PFORM::EmailField() .
    - the benefits of defining a form class is that it makes it very easy to do custom validations i.e. that those not handled by CI_form_validation class
- From the orm orm object *$this->orm* you can be able to access the following :
    - registry = this->orm->get_registry()
    - orm version = this->orm->get_version()
    - **Orm Form = this->orm->get_form()**

        this builds a form based on the

        - orms core form class by default
        - use a custom form you have defined if you pass the form as an argument or
        - model to build a form based on a model that already exists.
- Contributor, DeConstructable interface to provide a consistent way of deconstructing objects and contributing objects to other objects

6.7.2 Improvements and Fixes

- Provides a consistent api for the models meta data, for easier access.
- A consistent approach to how checks are carried out.
- Migrations operations
- Queryset to use the mode consistent model meta
- Check system by redefining the check message levels

6.7.3 Rewrites

- The whole migration module.
  - This module saw the addition of some important class:
    - AutoDetector - rewrite, keeps track of all changes within the models and produces the migration files..
    - Executor - responsible for running the migrations applying/unapplying.
    - Graph - Keeps track of how the migrations are related to each other i.e. which migration needs to run before which.
    - Migration - this was a rewrite, this is the base class for all migrations
    - Loader - this was a rewrite, loads migrations found in the migrations folder
    - Questioner - this was a rewrite
    - Recorder - this helps in keeping track of which migrations have been applied/unapplied by storing them in the database
    - State - this was rewrite to allow to use the new registry created
  - This drops using the CI_MIGRATION module and implements a differrent approach of doing migration this was prompted by need to reduce the number of migration files the previous version was producing
- The whole console module
  - This removes the need for user to create a migration controller to be able to use the orm just copy the *pmanger.php* file located at ORM_PATH/pmanager.php to the same directory as *index.php*
  - This also provides a consistent api for adding more commands within the orm
- The whole Form Module
  - This was done to enable defining forms as classes on a separate php file.
  - This rewrite resulted in the following classes:
    - Form - this is the overall class, it keeps track of a forms fields, form errors etc
    - Field - this keeps track of information relating to a form field like which errors it has, which label to use, the value of the field etc
    - Widget - this is responsible for rendering/ creating the expected html widget eg. input, textarea, password.
    - ValidationError - this thrown if a validation fails.
  - Whilst the new Form Module has its own validation technique, it heavily relies of the Ci_form_validation class. the new validation technique is meant to be used when doing validation that is not handle by Ci_form_validation class.

  You will mostly use it in the following form methods , i.e. if you have defined a form class :
  - the forms clean() method
  - the forms clean_{field_name}() method
- *License Agreement*
- *Change Log*

Index

Symbols