CrispySystem

Release 1.1.4

Jan 24, 2019

Contents:

CrispySystem was originally built so I would have to do less during my exams. Then I made it a public small basic framework.

The framework uses a MVC approach, which is achieved using: routing, controllers, models and views. It has Dependency Injection with auto-wiring which allows you to write no-mess code that just works.

CHAPTER 1

Installation

To install the CrispySystem framework, you will need to pull it in with Composer.

Add it to the composer.json file

```
"require": {
    "stevenliebregt/crispysystem": "^1.0"
}
```

Add it to an existing project

```
"composer require stevenliebregt/crispysystem"
```

1.1 Requirements

Webserver

You will need to have a webserver with the ability to rewrite a URL.

PHP

The CrispySystem framework requires you to have at least PHP7.0.

CHAPTER 2

Getting started

To get started with CrispySystem quicly, read the following. If you want to know more about the individual components, follow the links at the bottom of this page.

2.1 Quick start

Recomended directory structure

Create the following directory structure:

```
project_root/
    app/
    config/
    public/
        index.php
    storage/
    vendor/
    composer.json
```

! Note ! The *config* and *storage* directories need to be readable and writable.

Index

To create a CrispySystem application, create an *index.php* in the *public* folder and put the following content in it:

```
<?php

use StevenLiebregt\CrispySystem\CrispySystem;
use StevenLiebregt\CrispySystem\Routing\Router;
use StevenLiebregt\CrispySystem\Routing\Route;

define('DEVELOPMENT', true);
define('ROOT', './../');

```

(continues on next page)

(continued from previous page)

```php
require ROOT . 'vendor/autoload.php';

$crispySystem = new CrispySystem();

Router::group()->routes(function() {

    Route::get('/', function() {
        return 'Hello World';
    });

});

$crispySystem->run();
```

URL rewriting

In order for the framework to function correctly, all requests will need to be rewritten to the *index.php*.

2.2 Components

2.2.1 Routing

Routing is an essential part of the framework, it allows you to map an action to a URL.

The namespace for this component is *StevenLiebregt\CrispySystem\Routing*. In this namespace are the two following classes: *Router* and *Route*.

All routes need to be added before the *CrispySystem::run* method.

Methods

Each of the following HTTP methods have a static *Route* method with the same name.

- GET (*Route::get*)
- POST (*Route::post*)
- PUT (*Route::put*)
- PATCH (*Route::patch*)
- DELETE (*Route::delete*)

Each of these methods take 2 arguments:

- The URI path to match, this needs to be a string
- The handler, this can be a closure which returns a string, or a (string) path to a *Controller* class and method.

There are also 2 special route-adding methods, these being: *Route::any*, which takes the same arguments as the other route-adding methods, but adds a route that matches any HTTP verb, and *Route::match*, which takes 3 arguments. The first argument is an array of HTTP verbs, the route should match to, and the last two arguments are the same as in the other route-adding methods.

Handler

Closure

```
<?php

Route::get('/home', function() {
    return 'Welcome home!';
});
```

Controller

```
<?php

Route::get('/home', 'HomeController.welcome');
```

This route's action will return the *welcome* method from the *HomeController*.

Naming

Routes can be given names, you can use these names to retrieve a route. For example, you can use this in a template / view to fill a *href* element with a named url. This way you can change the url without editing all templates.

The *Route::setName* method is chained off the HTTP method method.

```
<?php

Route::get('/foo', function() {
    return 'foo';
})->setName('bar');
```

You can use *Router::getRouteByName('bar')* to retrieve the above route definition.

Parameters

To add a variable part in your route, you need to add a section wrapped in curly braces. Then you need to chain the *Route::where* method off the HTTP method method. This *Route::where* method takes the following 2 parameters:

- The name of the variable part, this needs to be the same as what is between the curly braces.
- A regular expression, to which the part will need to match. If the regular expression has a capturing group, the value will be auto-wired into the handler.

```
<?php

Route::get('/products/{id}', function($id) {
    return 'I want product id: ' . $id;
})->where('id', '(\\d+)');
```

This route would match */products/193* and would return '*I want product id: 193*', but it wouldn't match */products/bar* since the parameter part does not match the regular expression.

Grouping

You can also group routes so you can add:

- path-prefixes
- handler-prefixes
- name-prefixes

These prefixes will be added to all routes within the group. To start a group, use the *Router::group* method, then chain the *Router::routes* method, which takes a closure in which you can add your routes the normal way.

```
<?php

Router::group()
    ->setPathPrefix('/api')
    ->setHandlerPrefix('Controllers\\Backend\\')
    ->setNamePrefix('api.backend.')
    ->routes(function() {

        Route::get('', function() {
            return 'Hello this is api speaking';
        });

        Route::get('/products', 'ProductsController.index')
            ->setName('products.index');

    });
```

The above example will add 2 routes, the first one will listen to */api* and the second one will answer to */api/products*. With the second one, the handler gets prefixed so it will become *Controllers\Backend\ProductsController.index*, this also goes for the route name which will become *api.backend.products.index*.

2.2.2 View

For views this framework uses the templating engine *Smarty 3* <https://www.smarty.net>.

Setting template directory

Use this method to set the template directory. From this directory templates will be sought using the *SmartyView::display* method.

The default template directory is: *ROOT/resources/templates*.

Setting the template

The method *SmartyView::template* is used to set the template to be displayed.

The method takes one parameter, this needs to be the template filename including the extension.

Displaying

! Note ! Before you use this method, make sure you have set a template.

To display the template, the value of the *SmartyView::display* method needs to be returned from the controller.

Assigning data to templates

To assign data to a variable inside a template, you can use the *SmartyView::with* method, which takes an array, with *key / value* definitions.

```
<?php

->with([
    'id' => 1235,
    'foo' => [
        'test',
        'bar',
    ],
]);
```

The above would allow you to access the variables *id* and *foo* inside the template which give you *1235* and the array with values *test* and *bar* respectively.

Example

```
<?php

$view = new SmartyView();
$view->setTemplateDir('/root/templates');
$view->template('home.tpl');
return $view->display();
```

In the above example, the class looks for the template *home.tpl* in the directory */root/templates*.

2.2.3 Controllers

Controllers will be called after matching a URL.

The framework includes a base *Controller* class which can be extended.

The base controller has a property *crispySystem* which contains a copy of the *Container* class for Dependency Injection.

Controllers must return a string.

Example

```
<?php

namespace App\Controllers;

use StevenLiebregt\CrispySystem\Controllers\Controller;
use StevenLiebregt\CrispySystem\View\SmartyView;

class ProductsController extends Controller
{
    private $view;

    public __construct(SmartyView $view)
```

(continues on next page)

(continued from previous page)

```
    {
        $this->view = $view;
    }

    public function index()
    {
        return $this->view
            ->template('index.tpl')
            ->display();
    }

    public function item($id)
    {
        return $this->view
            ->template('item.tpl')
            ->with([
                'id' => $id,
            ])
            ->display();
    }
}
```

2.2.4 Database

Connection

The *Connection* class returns a PDO object, in order for this to work you need to have a *database.php* configuration file in *ROOT/config*. This file should look something like this:

```
<?php

return [
    'driver' => 'mysql',
    'host' => '172.17.0.2',
    'port' => '3306',
    'dbname' => 'databasename',
    'user' => 'root',
    'pass' => 'secret',
];
```

To retrieve a PDO object from the class, use the *Config::getPdo* method.

Models

With this framework included is a base model (*StevenLiebregt\CrispySystem\Database\Model*) that contains some basic queries.

When extending this class, you will need to override 2 properties, those are:

- $table, the name of the table as in the database
- $fields, an array containing all the field names in the database table

The queries included in the base class are:

showTables

Runs a *SHOW TABLES* query on the database and return the result.

getAll

Runs a *SELECT* query to retrieve all records from a table, and return the result.

getOneById

Runs a *SELECT* query to retrieve one record by id from a table, and return the result.

insert

Run an *INSERT* query, the method takes one parameter which should be an array of a list of the values to insert. This array should contain the values for all fields except the *id* field, as that is expected to be an *AUTO_INCREMENT* field.

This method returns the id of the newly inserted record.

updateById

Update one or more records in the database table. Takes two arguments:

- An id, or an array of ids, these are the ids of the records to be updated
- An associative array with all the values to be updated, as where the key should be the name of the field and the value the new value of the field

This method returns the amount of rows affected by the query.

deleteById

Delete one or more records from a database table. It takes one parameter, which can be an id or an array of ids.

This method returns the amount of rows affected by the query.

2.2.5 Dependency Injection

Dependency Injection manages dependencies for you, as the name implies.

This class can be accessed by extending the base *Controller* class, as the property *crispySystem*.

! Note ! The parameters in a method or function should be type-hinted in order for this to work.

Create instance

The *createInstance* method takes the name of a class (including namespace) as argument. It resolves any dependencies for the constructor recursively, and returns the class instance.

This method always creates a new instance, in comparison to the *getInstance* method which will re-use a previously created instance, or creates a new one if it doesn't exist.

Get instance

The *getInstance* method takes the name of a class (including namespace) as argument. It resolves any dependencies for the constructor recursively, and returns the class instance.

This method will re-use a previously created instance, or create a new one if it doesn't exist.

Resolve closure

The *resolveClosure* method can be used to resolve dependencies in a closure. It takes one parameter, which should be a closure.

Resolve method

The *resolveMethod* method can be used to resolve a specific method of an instance, it takes the following arguments:

- instance, the instance of which you want to resolve a method
- method, the name of the method to resolve
- parameters **(optional)**, if given, any parameters you give in this associative array, where the key = the name of the parameter, and the value the value, will be used to fill parameter slots. If a value is not given, the default will be used or resolved.

2.2.6 Helpers

Config

All config files that reside in *ROOT/config/* will be read and cached into one array, the first-level keys in the array are the names of the config files. So if you have a file called *database.php* and *system.php* in your config directory, the configuration array will look like this:

```
[
    'database' => [
        'content from database.php'
    ],
    'system' => [
        'content from system.php'
    ],
]
```

The configuration files must return an array.

Reading the configuration

To read the configuration you can use the *Config::get* method.

If you leave the parameter empty, it will return an array containing all the configuration, if you give a key you want, you'll only receive the value of that key. Multidimensional keys are formatted like this: *database.driver*, this would map to *$config['database']['driver']*.

Functions

pr

The *pr* function is a wrapper for the PHP function *print_r*. It encapsulates the *print_r* function inside *<pre></pre>* tags so you don't have to type those manually.

vd

The *vd* function is a wrapper for the PHP function *var_dump*. It encapsulates the *var_dump* function inside *<pre></pre>* tags so you don't have to type those manually.

showPlainError

This function shows an error, with a darkred title saying *[ERROR]*, it has an option to exit after showing the error, by default this is true.

jsonify

This function sets the *Content-Type: application/json* header and returns the given data as a json_encoded string.

CHAPTER 3

Changelog

3.1 1.4.0

New features

- Added the *Route::any* method, which matches any HTTP verb
- Added the *Route::match* method, which matches a list of HTTP verbs

3.2 1.3.1

Fixes

- Fixed a routing issue when the name was not set

3.3 1.3.0

New features

- Added Smarty *config* plugin, which allows you to retrieve config settings in templates

3.4 1.2.0

New features

- Added base container
- Added complete documentation

3.5 1.1.4

New features

- Added base model for database queries

3.6 1.1.3

New features

- Added database connection class

3.7 1.1.2

New features

- Added jsonify function

3.8 1.1.1

Fixes

- Fixed some dirt that got there when refactored to PHP7.0

3.9 1.1.0

Changed features

- Refactored the project from PHP7.1 down to PHP7.0

3.10 1.0.0

This project was created