simple-mvc Documentation

Release 0.1.0

Walter Dal Mut

Sep 21, 2017

Contents

Contents:

CHAPTER 1

Getting Started

The goal is realize a web application in few steps.

See scripts into *example* for a real example. The base is create a public folder where your web server dispatch the *index.php*.

Create out from this folder the *controllers* path or whatever you want (eg. *ctrs*).:

```
- controllers
  - IndexController.php
- public
  - .htaccess
  - index.php
```

In practice you are ready. See the *.htaccess*:

```
RewriteEngine  On
RewriteCond  %{REQUEST_FILENAME}  -s  [OR]
RewriteCond  %{REQUEST_FILENAME}  -l  [OR]
RewriteCond  %{REQUEST_FILENAME}  -d
RewriteRule  ^.*$  -  [NC,L]
RewriteRule  ^.*$  index.php  [NC,L]
```

The *index.php* is the main app entry point

```
<?php
set_include_path(realpath('/path/to/src'));

require_once 'Loader.php';
Loader::register();

$app = new Application();
$app->setControllerPath(__DIR__ . '/../controllers');
$app->run();
```

The controller *IndexController.php* file should be like this

```php
<?php
class IndexController extends Controller
{
    public function indexAction()
    {
        echo "hello";
    }
}
```

See “[view](views.md)” doc for enable views supports.

Urls with dashes

If you use a dash into an URL the framework creates the camel case representation with different strategies if it is an action or a controller.:

```
/the-controller-name/the-action-name
```

Will be

```
// the-controller-name => TheControllerName
class TheControllerName extends Controller
{
    public function theActionNameAction()
    {
        //the-action-name => theActionName
    }
}
```

Bootstrap resources

You can bootstrap resources:

```php
<?php
$app = new Application();
$app->bootstrap('my-resource', function() {
    return new MyObject();
});
```

The bootstrap do not executes all hooks (lazy-loading of resources) but execute it ones only if your application needs it.

```php
<?php
// Into a controller
$resource = $this->getResource("my-resource");
$another = $this->getResource("my-resource");

// IT IS TRUE!
var_dump($resource === $another);
```

CHAPTER 2

Autoloader

simple-mvc provides two strategies for loading classes for itself and only one strategy for autoload your classes.

Classmap

The classmap loads only *simple-mvc* classes. If you have a self-designed autoloader you have to use this strategy for reduce conflicts during the autoloading process.

```
<?php
require_once '/path/to/simple/Loader.php';

// Load all simple-mvc classes
Loader::classmap();
```

PSR-0 Autoloader

If you want to use the PSR-0 autoloader you have to register the autoloader.

```
<?php
require_once '/path/to/simple/Loader.php';

set_include_path(
    implode(
        PATH_SEPARATOR,
        array(
            '/path/to/project',
            get_include_path()
        )
    )
);

```

```
14 // Load all simple-mvc classes
15 Loader::register();
```

The autoloader loads automatically namespaced classes and prefixed.

Prefix example:

```
1 <php
2
3 // Prefix -> ClassName.php
4 class Prefix_ClassName
5 {
6
7 }
```

Namespace example:

```
1 <?php
2 namespace Ns;
3
4 // Ns -> ClassName.php
5 class ClassName
6 {
7
8 }
```

CHAPTER 3

Controllers

The controller section

Init hook

Before any action dispatch the framework executes the *init()* method.

```php
<?php
class IndexController extends Controller
{
    public function init()
    {
        // The init hook
    }
}
```

Using the object inheritance could be a good choice for this hook.

```php
<?php
abstract class BaseController extends Controller
{
    public function init()
    {
        // Reusable code
    }
}
```

Next action

The *next* action goes forward to the next action appending the next view.

```php
<?php
class IndexController extends Controller
{
    public function indexAction()
    {
        $this->view->hello = "hello";


        $this->then("/index/next");
    }

    public function nextAction()
    {
        $this->view->cose = "ciao";
    }
}
```

The result is the first view (*index.phtml*) concatenated to the second view (*next.phtml*).

Redirects

You can handle redirects using the *redirect()* method

```php
<?php
class IndexController extends Controller
{
    public function indexAction()
    {
        // Send as moved temporarily
        $this->redirect("/contr/act", 302);
    }
}
```

Interact with layout and views

You can disable the layout system at any time using the *disableLayout()* method.

```php
<?php
class IndexController extends Controller
{
    public function indexAction()
    {
        // Remove layout
        $this->disableLayout();
    }
}
```

You can disable the view attached to a controller using the *setNoRender()* method

```php
<?php
class IndexController extends Controller
{
    public function indexAction()
```

```
    {
        // No this view
        $this->setNoRender();
    }
}
```

Change the layout on the fly

If you want to change your layout during an action or a plugin interaction you can use the resources manager

```
<?php
class IndexController extends Controller
{
    public function fullWithAction()
    {
        $this->getResource("layout")->setScriptName("full-width.phtml");
    }
}
```

Obviously you must use the layout manager.

Using headers

You can send different headers using *addHeader()* method

```
<?php
class IndexController extends Controller
{
    public function indexAction()
    {
        $this->addHeader("Content-Type", "text/plain");
    }
}
```

Change View Renderer

You can change the view renderer at runtime during an action execution.

```
<?php
class IndexController extends Controller
{
    public function indexAction()
    {
        $this->setRenderer("/use/me");
    }
}
```

The framework will use the *use/me.phtml*

The end.

CHAPTER 4

Views

The framework starts without view system. For add view support you have to add *view* at bootstrap.

```php
<?php

$app = new Application();
$app->bootstrap('view', function(){
    $view = new View();
    $view->addViewPath(__DIR__ . '/../views');

    return $view;
});

$app->run();
```

The framework append automatically to a controller the right view using controller and action name. Tipically you have to create a folder tree like this:

```
site
 + public
 + controllers
 - views
   - index
     - index.phtml
```

In this way the system load correctly the controller path and the view script.

Layout support

The layout is handled as a simple view that wrap the controller view.

You need to bootstrap it. The normal layout name is “layout.phtml”

```php
<?php

$app->bootstrap('layout', function(){
    $layout = new Layout();
    $layout->addViewPath(__DIR__ . '/../layouts');

    return $layout;
});
```

You can change the layout script name using the setter.

```php
<?php
$layout->setScriptName("base.phtml");
```

View Helpers

If you want to create view helpers during your view bootstrap add an helper closure.

```php
<?php
$app->bootstrap('view', function(){
    $view = new View();
    $view->addViewPath(__DIR__ . '/../views');

    $view->addHelper("now", function(){
        return date("d-m-Y");
    });

    return $view;
});
```

You can use it into you view as:

```php
<?php echo $this->now() ?>
```

You can create helpers with many variables

```php
<?php
$view->addHelper("sayHello", function($name){
    return "Hello {$name}";
});
```

View system is based using the prototype pattern all of your helpers attached at bootstrap time existing into all of your real views.

Share view helpers

View helpers are automatically shared with layout. In this way you can creates global helpers during the bootstrap and interact with those helpers at action time.

Pay attention that those helpers are copied. Use *static* scope for share variables.

```php
<?php
$app->bootstrap("layout", function(){
    $layout = new Layout();
```

```
    $layout->addViewPath(__DIR__ . '/../layouts');


    return $layout;
});

$app->bootstrap("view", function(){
    $view = new View();
    $view->addViewPath(__DIR__ . '/../views');

    $view->addHelper("title", function($part = false){
        static $parts = array();
        static $delimiter = ' :: ';

        return ($part === false) ? "<title>".implode($delimiter, $parts)."</title>" :
↪$parts[] = $part;
    });

    return $view;
});
```

From a view you can call the *title()* helper and it appends parts of you page title.

Escapes

Escape is a default view helper. You can escape variables using the *escape()* view helper.

```
<?php
$this->escape("Ciao -->"); // Ciao --&gt;
```

Partials view

Partials view are useful for render section of your view separately. In *simple-mvc* partials are view helpers.

```
<!-- ctr/act.phtml -->
<div>
    <div>
        <?php echo $this->partial("/path/to/view.phtml", array('title' => $this->
↪title));?>
    </div>
</div>
```

The partial view */path/to/view.phtml* are located at *view* path.

```
<!-- /path/to/view.phtml -->
<p><?php echo $this->title; ?></p>
```

Multiple view scripts paths

simple-mvc support multiple views scripts paths. In other words you can specify a single mount point */path/to/views* after that you can add anther views script path, this mean that the *simple-mvc* search for a view previously into the

second views path and if it is missing looks for that into the first paths. View paths are threated as a stack, the latest pushed is the first used.

During your bootstrap add more view paths

```
$app->bootstrap('view', function(){
    $view = new View();
    $view->addViewPath(__DIR__ . '/../views');
    $view->addViewPath(__DIR__ . '/../views-rewrite');

    return $view;
});
```

If you have a view named *name.phtml* into *views* folder and now you create the view named *name.phtml* into *views-rewrite* this one is used instead the original file in *views* folder.

Partials and multiple view scripts paths

Partial views follow the rewrite path strategy. If you add the partial view into a rewrite view folder, this view script is choosen instead the original partial script.

```
<?php echo $this->partial("my-helper.phtml", array('ciao' => 'hello'))?>
```

If *my-helper.phtml* is found in a rewrite point this view is used instead the original view script.

The end.

CHAPTER 5

Events

Events

- *loop.startup*
- *loop.shutdown*
- *pre.dispatch*
- *post.dispatch*

Hooks

The *loop.startup* and *loop.shutdown* is called once at the start and at the end of the simple-mvc workflow.

The *pre.dispatch* and *post.dispatch* is called for every controlled pushed onto the stack (use the *then()* method).

Hooks params

The *loop.startup* and the *loop.shutdown* have the *Application* object as first parameter.

The *pre.dispatch* hook has the *Route* object as first parameter and the *Application* object as second.

The *post.dispatch* hook has the *Controller* object as first paramter.

- The router object is useful for modify the application flow.

```
<?php
$app->getEventManager()->subscribe("pre.dispatch", function($router, $app) {
    // Use a real and better auth system
    if ($_SESSION["auth"] !== true) {
        $router->setControllerName("admin");
        $router->setActionName("login");

        $app->getBootstrap("layout")->setScriptName("admin.phtml");
```

```
    }
});
```

Create new events

```php
<?php
// Call the hook named "my.hook" and pass the app as first arg.
$app->getEventManager()->publish("my.hook", array($app));
```

You can use the self-created hook using

```php
<?php
$app->getEventManager()->subscribe("my.hook", function($app) {/*The body*/});
```

CHAPTER 6

Pull Driven Requests

Typically MVC frameworks are “push” based. In otherwords use mechanisms to “push” data to a view and not vice-versa. A “pull” framework instead request (“pull”) data from a view.

Pull strategy is useful for example during a *for* statement (not only for that [obviously]...). Look for an example:

```
<?php foreach ($this->users as $user) : ?>
<?php
    // Pull data from a controller.
    $userDetail = $this->pull("/detail/user/id/{$user->id}");
?>
<div class="element">
    <div class="name"><?php echo $userDetail->name;?> <?php echo $userDetail->surname;
↪ ?></div>
    <!-- other -->
</div>
<?php endforeach; ?>
```

simple-mvc implementation

simple-mvc has ***push*** and ***pull*** mechanisms. The *push* is quite simple and a typical operation. See an example

```
<?php
class EgController extends Controller
{
    public function actAction()
    {
        // PUSH to view a variable named <code>var</code>
        $this->view->var = "hello";
    }
}
```

The view show the pushed variable

```php
<?php echo $this->var; ?>
```

The *pull* strategy is quite similar but use the return statement of a controller to retrive all the information. Consider in advance that *simple-mvc* doesn't require a valid controller for retrive a view, that view is mapped directly. See an example

```
<!-- this view is test/miss.phtml (/test/miss GET) -->
<div>
    <h1>Missing controller and action</h1>

    <?php $data = $this->pull("/ctr/act"); ?>

    <!-- example -->
    <?php echo $data->title; ?>
</div>
```

The view require a *pull* operation from a controller named *ctr* and action *act*. See it:

```php
<?php
class CtrController extends Controller
{
    public function actAction()
    {
        $data = new stdClass();

        $data->title = "The title";

        // The return type doesn't care...
        return $data;
    }
}
```

You can use a "pull" controller as a normal controller with the attached view, but remember that when you request for a "pull" operation the view is never considered and the framework remove it without consider the output, only the *return* statement will be used.

CHAPTER 7

Examples of usage

A simple base app execution

```php
<?php
$app = new Application();

$app->run();
```

Execute with bootstrap

```php
<?php
$app = new Application();

$app->bootstrap("say-hello", function(){
    return array('example' => 'ciao');
});

$app->run();
```

Into a controller

```php
<?php
class IndexController extends Controller
{
    public function indexAction()
    {
        $element = $this->getResource('example');

        echo $element["example"];
    }
}
```

Controller Forward

You can pass to another controller using *then()*

```php
<?php
class IndexController extends Controller
{
    public function indexAction()
    {
        // Add forward action
        $this->then("/index/forward");
    }

    public function forwardAction()
    {
        // append to index or use it directly
    }
}
```

See *example* folder for a complete working example.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search