UpCloo Framework Documentation

Release 0.0.11

Walter Dal Mut

February 02, 2014

Contents

1 Introduction **3**
- 1.1 Renderers . . . 3
- 1.2 Events . . . 3
- 1.3 Services . . . 4
- 1.4 Configuration . . . 4

2 Getting Started with UpCloo Framework **5**
- 2.1 Your entry point . . . 5
- 2.2 Your base configuration . . . 6
- 2.3 Now the ActionController . . . 6
- 2.4 Test you business logic . . . 7
- 2.5 Integration testing . . . 8

3 Configuration **9**
- 3.1 Services . . . 9
- 3.2 Listeners . . . 10
- 3.3 Overload your configuration . . . 10

4 Controllers and Actions **11**
- 4.1 Interact with the event data . . . 11
- 4.2 Interact with the Request object . . . 12
- 4.3 Interact with the Response object . . . 12
- 4.4 Redirections . . . 12
- 4.5 ServiceManager . . . 13
- 4.6 EventManager . . . 13
- 4.7 Test your controllers . . . 14

5 The ServiceManager **17**

6 Listeners **19**

7 Indices and tables **21**

Contents:

CHAPTER 1

Introduction

UpCloo framework is based on ZF2 components and in particular:

- TreeRouteStack (Router)
- EventManager
- ServiceManager

1.1 Renderers

You have to define "renderers" (who render your data). The framework provides two default renderers that are:

- UpCloo\Renderer\Json
- UpCloo\Renderer\Jsonp

1.2 Events

The framework flow is event driven and the execution depends in your actions. In a valid request you reach this events list

- begin
- route
- pre.fetch
- execute
- renderer
- finish

The default flow can change on errors, redirections and exceptions, for example if a route is missing the "404" event is thrown and the flow is like this:

- begin
- route
- 404
- finish

You have to attach a listener on the "404" event in order to handle this error situation.

1.3 Services

The *ServiceManager* is responsible to provide objects to your application and is widly used into the App framework in order to select the right controller and renderer.

1.4 Configuration

The framework uses your configuration in order to bootstrap and run.

CHAPTER 2

Getting Started with UpCloo Framework

The base folder structure is: whatever you want...

We suggest something like this:

```
- configs
- src
  - Your
    - Project
      - Namespace
- tests
  - Your
    - Project
      - Namespace
- web
    - js
    - css
    - img
```

That is similar to a standard ZF2 module.

2.1 Your entry point

Into *web* direction you have to place your single entry point for your application the *index.php* file.

```
<?php
// web/index.php

$loader = require __DIR__ . "/../../vendor/autoload.php";
$loader->add("My", __DIR__ . '/../src');

$conf = include __DIR__ . "/../configs/app.php";

$config = new UpCloo\App\Config\ArrayProcessor();
$config->appendConfig($conf);

$boot = new UpCloo\App\Boot($config);
$engine = new UpCloo\App\Engine();

$app = new UpCloo\App($engine, $boot);
$app->run();
```

As you can see the first to line uses the composer autoloader in order to satisfy all your dependencies.

The configuration is loaded through the inclusion. Subsequently we create the application and after that we *run* it.

2.2 Your base configuration

We want to create a *json* response at the / address. So, we need the router and at least one controller.

```
<?php
// configs/app.php

return array(
    "router" => array(
        "routes" => array(
            "home" => array(
                "type" => "Literal",
                "options" => array(
                    "route" => "/"
                    'defaults' => array(
                        'controller' => 'My\\NM\\Index',
                        'action' => 'aMethod'
                    )
                ),
                'may_terminate' => true,
            )
        )
    ),
    "services" => array(
        "invokables" => array(
            "My\\NM\\Index" => "My\\NM\\Index",
        )
    )
)
```

2.3 Now the ActionController

The controller class is simply a *POPO* definition with just the action declared.

```
<?php
// src/My/NM/Index.php

namespace My\\NM;

class Index
{
    public function aMethod()
    {
        return array(
            "hello" => "world"
        );
    }
}
```

As you can see the method should return the value that the renderer will serialize into the response.

2.4 Test you business logic

The goal of this structure is oriented to testing. For that reason the test section is not optional!

```
// tests/My/NM/IndexTest.php

namespace My\\NM;

class IndexTest extends \PHPUnit_Framework_TestCase
{
    private $object;

    public function setUp()
    {
        $this->object = new Index();
    }

    public function testSimpleIndexMethod()
    {
        $oracleData = array(
            "hello" => "world"
        );

        $this->assertEquals($oracleData, $this->object->aMethod());
    }
}
```

Obviously this is just a simple action! Before run tests correctly we need to load classes and framework, for that use a bootstrap file.

```
<?php
// tests/bootstrap.php

$loader = require __DIR__ . '/../vendor/autoload.php'; //composer load the framework

$loader->add("My", __DIR__ . '/../src'); //Your source
$loader->add("My", __DIR__); // tests folder
```

Now run your tests:

```
phpunit --bootstrap tests/bootstrap.php tests/
```

The output should be something similar to this:

```
PHPUnit 3.7.22 by Sebastian Bergmann.

.

Time: 1 seconds, Memory: 1.25Mb

OK (1 tests, 1 assertions)
```

Now you can continue with more interesting things!

2.5 Integration testing

You can test your controller in isolation (see *Controllers and Actions*) or you can run the whole application. If you are interested in this last thing, you have to inherits from *UpClooTestWebTestCase* during testing.

```
<?php
namespace Your\NM;

use UpCloo\Test\WebTestCase;

class MyControllerTest extends WebTestCase
{
    public function setUp()
    {
        $this->appendConfig([
            "router" => [
                ... // Routes
            ],
            "services" => [
                ... // A conf
            ],
            ...
        ]);
    }

    public function testMyAction()
    {
        $response = $this->dispatch("/my-action"); //get method

        $this->assertEquals(200, $response->getStatusCode());
        //... more assert

        $content = $response->getContent();
        //...
    }
}
```

The *dispatch* method signature is:

```
public function dispatch($url, $method = "GET", array $params = array())
```

Possibile methods are:

- GET
- POST
- PUT

CHAPTER 3

Configuration

Basically only the *router* section is a must.

```
<?php
return array(
    "router" => array(
        "routes" => array(
            "home" => array(
                "type" => "literal",
                "options" => array(
                    "route" => "/"
                    "defaults" => array(
                        "controller" => "Your\\NS\\Controller",
                        "action" => "myAction"
                    )
                ),
                "may_terminate" => true
            )
        )
    )
);
```

The configuration is practically identical to ZF2 standard router configuration

3.1 Services

In addition you can configure services:

```
"services" => array(
    "invokables" => array(
        "My\\Controller\\Example" => "My\\Controller\\Example",
        "UpCloo\\Renderer\\Jsonp" => "UpCloo\\Renderer\\Jsonp",
    ),
    "factories" => array(
        "example" => function(\Zend\ServiceManager\ServiceLocatorInterface $sl) {
            return "that-service";
        }
    ),
    "aliases" => array(
        "exampleController" => "My\\Controller\\Example",
        "renderer" = "UpCloo\\Renderer\\Jsonp"
```

```
    )
),
```

The configuration is the same for ZF2 services

3.2 Listeners

When you need to hook your code on events you can specify through the *listeners* section:

```
"listeners" => array(
    "404" => array(
        array("My\\Controller\\Error", "error")
    )
)
```

Any *callable* hook is valid

```
 "listeners" => array(
    "404" => array(
        function() {
            // handle 404
        }
    )
)
```

3.3 Overload your configuration

You can pass to your *Boot* different configurations. The framework merge those together in order to obtain a single configuration.

This thing could be useful in order to obtain the right configuration for the current environment.

For example see something like this:

```
$config = new \UpCloo\App\Config\ArrayProcessor();

$config->appendConfig(include __DIR__ . "/../configs/app.php");
$config->appendConfig(include __DIR__ . "/../configs/app.{$env}.php");

$boot = new \UpCloo\App\Boot($config);

...
```

In this way the conf loaded from *app.php* is overwritten by the second configuration and so on. You can load how many conf you need.

CHAPTER 4

Controllers and Actions

Controllers are simply *POPO* object, like this:

```
class Me
{
    public function hello()
    {
        return "hello!";
    }
}
```

Controllers and actions are mapped thanks to the `TreeRouteStack` as you can see in the *Getting Started with UpCloo Framework* section.

4.1 Interact with the event data

When your action is called, the event is passed as method argument and you can interact with the *RouteMatch* in this way:

```
class Me
{
    public function hello($event)
    {
        //Play with $event
    }
}
```

The *$event* object is a *Zend\EventManager\Event* object, in few words something like this:

```
object(Zend\EventManager\Event)[31]
    protected 'name' => string 'execute' (length=7)
    protected 'target' =>
        object(Zend\Mvc\Router\Http\RouteMatch)[35]
        protected 'length' => int 7
        protected 'params' =>
            array (size=3)
            'renderer' => string 'UpCloo\Renderer\Jsonp' (length=21)
            'controller' => string 'exampleController' (length=17)
            'action' => string 'method' (length=6)
        protected 'matchedRouteName' => string 'home' (length=4)
    protected 'params' =>
        array (size=0)
```

```
        empty
    protected 'stopPropagation' => boolean false
```

The "param" contains the "RouteMatch" structure.

4.2 Interact with the Request object

Many times you need to interact with Request (Zend\Http\PhpEnvironment\Request) object. When you need to use the http request you can use "UpCloo\Controller\Request" trait.

```
<?php
namespace Your\NM;

use UpCloo\Controller\Request;

class Me
{
    use Request;

    public function hello($event)
    {
        $request = $this->getRequest();
    }
}
```

The framework hydrate your controller with the Request object only if you declare that you need it using the trait!

4.3 Interact with the Response object

The Response object (Zend\Http\PhpEnvironment\Response) follow the same of Request.

```
<?php
namespace Your\NM;

use UpCloo\Controller\Response;

class Me
{
    use Response;

    public function hello($event)
    {
        $response = $this->getResponse();
    }
}
```

4.4 Redirections

As before you have to use traits, the UpCloo\Controller\Action\Redirector to be clear

```
<?php
namespace Your\NM;

use UpCloo\Controller\Action\Redirector;

class Me
{
    use Redirector;

    public function hello($event)
    {
        $this->redirect("http://walterdalmut.com", 302);
    }
}
```

The second argument of "redirect" method is optional (302 by default) and the first argument is the redirect location.

The Redirector traits uses the "Response trait" by itself, for that reason when you use the redirector the Response traits is automatically added to your controller.

4.5 ServiceManager

You can request anything from the service locator through just using the *ServiceManager* trait.

```
<?php
namespace Your\NM;

use UpCloo\Controller\ServiceManager;

class TheHookContainer
{
    use ServiceManager;

    public function anHook()
    {
        $aService = $this->services()->get("a-service");

        ...
    }
}
```

4.6 EventManager

Inside an event you can attach and fire other events adding the EventManager trait:

```
<?php
namespace Your\NM;

use UpCloo\Controller\EventManager;

class TheHookContainer
{
    use EventManager;
```

```
    public function anHook()
    {
        // Attach something to an event
        $this->events()->attach("finish", function() {
            //Good bye cruel world!
        });

        // Trigger a custom event...
        $this->events()->trigger("my.hook.event", $this, ["name" => "a name"]);
    }
}
```

4.7 Test your controllers

You can test your controller in isolation from the entire application, you have just to prepare things that you need and inject into your controller.

See an example:

```
<?php
namespace Your\NM;

use UpCloo\Controller\EventManager;

class Controller
{
    use EventManager;

    public function myHook($event)
    {
        $this->events()->trigger("my.hook.start", $this);

        ... // do something...

        $this->events()->trigger("my.hook.finish", $this, $data);
        return $data;
    }
}
```

Your tests could be something like this:

```
<?php
namespace UpCloo\NM;

use Zend\EventManager\EventManager;
use UpCloo\Test\ControllerTestUtils;

class ControllerTest extends \PHPUnit_Framework_TestCase
{
    use ControllerTestUtils;

    private $object;

    public function setUp()
    {
        // Prepare the controller
        $this->object = new Controller();
```

```
        $this->object->setEventManager(new EventManager());
    }

    public function testWorkingAction()
    {
        $event = $this->getEventFromParams([
            "param" => "hello"
        ]);

        $data = $this->object->myHook($event);

        // asserts on data
    }
}
```

CHAPTER 5

The ServiceManager

As mentioned before, the ZF2 ServiceManger is used. You can configure your services and require for them in your controller.

In your configuration:

```
<?php
return array(
    "services" => array(
        "factories" => array(
            "example" => function($sl) {
                return new stdClass();
            }
        )
    )
)
```

In your controller you have to require the ServiceManager trait

```
<?php
namespace My\NM;

use UpCloo\Controller\ServiceManager;

class My
{
    use ServiceManager;

    public function hello($event)
    {
        $service = $this->get("example");

        return $service;
    }
}
```

CHAPTER 6

Listeners

Listeners are object that are used when an event is fired!

```
<?php
namespace My\NM;

class Error
{
    public function error()
    {

    }
}
```

Of course we have to link listeners through the configuration:

```
// configs/app.php

"services" => arrray(
    "invokables" => array(
        "My\\NM\\Error" => "My\\NM\\Error",
    )
),
"listeners" => array(
    "404" => array(
        array("My\\NM\\Error", "error")
    )
)
```

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*