rampage-php Documentation

Release 1.1.2-dev

Axel Helmert

October 17, 2014

Contents

Welcome to the reference guide to rampage-php - an addon library to ZendFramework 2.

CHAPTER 1

Quickstart

This section will show you how to set up a basic application and how to start.

We strongly recommend to use composer as dependency manager, since it will simplify you life a lot and take care of your library dependencies. In fact this tutorial will rely on it.

See getcomposer.org for download and installation instructions (It's really simple since it's just a single phar file).

1.1 Installation / Application Setup

At first we'll define the dependencies of our application. By now we only need to add `rampage-php/framework` as only dependency. This already defines dependencies to the essential zf2 components (i.e. zend-mvc):

```
php composer.phar require rampage-php/framework
```

This command will add the dependency to your composer.json. For more information on the composer.json package file see the Composer Documentation.

After doing so, we can run the install command to download all dependencies and generate the autoload files:

```
php composer.phar install
```

After executing this command, there will be a `vendor` directory containing all dependencies. To use them, all you need to do is to include `vendor/autoload.php` and all dependency classes become available to you.

1.2 Creating A Skeleton

After doing the composer magic, we only have the vendor dir. But now we want to create an application skeleton to start with. To simplify things, rampage comes with a tool to do so, a script called `rampage-app-skeleton.php`

Since rampage-php enhances zf2, it uses the zf2 layout. That means all modules are basically namespaces. In vanilla ZF2, you can only use the first namespace segment as module name which is quite limiting. Usually the first segment is the vendor name. But rampage-php enhances this behavior and adds the ability to use subnamespaces as well and map them to a dot-separated directory name.

Let's assume your application module should be namespaced *acme\myapp*. Now let's create the application layout with this modulename:

```
php vendor/bin/rampage-app-skeleton.php create acme.myapp
```

For windows this command should be:

```
.\vendor\bin\rampage-app-skeleton.php create acme.myapp
```

This will create:

- a `public` directory which will contain the webserver root.
- an `application` directory containing the application components
- the module skeleton for your application located in `application/modules/acme.myapp`

CHAPTER 2

Zend\Di Enhancements

Rampage-PHP provides a couple of enhancements to the default Di/ServiceManager implementations of zf2.

2.1 Cupling ServiceManager And Di

Rampage combines those both components more closely than zf2. The Di InstanceManager will use the ServiceManager to look up a class instance. This allows you to call `$serviceManager->get('di')->newInstance($classname);` which will respect and inject configured services at any time.

Note: Doing this call in plain zf2 would cause the Di framework to create new instances for classes that haven't been created via Di and ignoring the configured services.

2.2 Define Services For Dependency Injection

There are two options to refernce services for dependency injection which depends on your needs.

- *Option 1: ServiceManager Name/Alias*
- *Option 2: DI InstanceManager Alias*

2.2.1 Option 1: ServiceManager Name/Alias

This is the simplest way of providing your services to the di framework. Either you define the service directly with its *fqcn* or you add an alias pointing to the *fqcn* .

```
return [
    'invokables' => [
        'my\app\ClassName' => 'my\app\ClassName', // Direct naming
        'MyService' => 'my\app\AnotherClassName',
    ],
    'aliases' => [
        'my\app\SomeInterface' => 'MyService', // Alias a class/interface to a service
    ]
];
```

2.2.2 Option 2: DI InstanceManager Alias

This way is more flexible and allows you to define services and injections more precisely. To do so you should create an alias Matching your service name in your DI config:

```
// di.config.php
// Assuming "MyServiceName" and "AnotherServiceName" are defined by the ServiceManager
return [
    'instance' => [
        'aliases' => [
            // This makes the services known to the di framework.
            'MyServiceName' => 'my\app\SomeInterface', // Pointing to an interface is sufficient
            'AnotherServiceName' => 'my\app\SomeClass', // Pointing to a class is more flexible
        ],
        'preferences' => [
            // Explicit definition to
            // use the services to provide the dependencies.
            'thirdparty\SomeInterface' => 'AnotherServiceName',
            'thirtparty\AbstractClass' => 'AnotherServiceName',

            // Let Zend\Di decide which alias provides the dependency
            // (implements or inhertis the interface)
            'foo\bar\AnotherInterface' => [ 'MyServiceName', 'AnotherServiceName' ],
        ]
    ]
];
```

Note: When using an alias to provide a preference, the aliased class name must implement or inherit the the provided dependency class/interface. Zend\Di **will** do a sanity check if this is the case (i.e. to pick the matching provider)!

2.3 DIServiceFactory

Cupling ServiceManager And Di allows the library to provide a service factory to create class instances purely by dependency injection.

Example:

```
// Service config:
return [
    'factories' => [
        'MyService' => new \rampage\core\services\DIServiceFactory(ServiceImplementation::class);
    ]
];
```

2.4 DIPluginServiceFactory

As much as *DIServiceFactory* this allows defining a class to be instanciated by dependency injection. The difference to the *DIServiceFactory* is that this may be used for zf2 PluginManagers.

Example:

```
// PluginManager config:
return [
    'factories' => [
```

```
        'myhelper' => new \rampage\core\services\DIPluginServiceFactory(MyHelper::class);
    ]
];
```

2.5 Abstract Service Factory

As in zf2, there is an abstract service factory, which allows you to request a service by a classname without the need to define it explicitly. It will be automatically instanciated via the di framework then.

The difference to the zf2 implementation is, that all configured services are respected by the di framework (see *Cupling ServiceManager And Di*).

So the following code will work as expected.

di.config.php:

```
return [
    'factories' => [ 'DbAdapter' => MyDbAdapterFactory::class ],
    'aliases' => [ 'Zend\Db\Adapter\AdapterInterface' => 'DbAdapter' ]
]
```

SomeClass.php:

```
namespace my\app;

use Zend\Db\Adapter\AdapterInterface as DbAdapterInterface;

class SomeClass
{
    /**
     * @var DbAdapterInterface
     */
    protected $adapter;

    /**
     * Constructor dependency to a Zend\Db\Adapter\AdapterInterface
     */
    public function __construct(DbAdapterInterface $adapter)
    {
        $this->adapter = $adapter;
    }

    // ...
}
```

Usage:

```
namespace my\app;

// ...

// $instance will be created via Di and the "DbAdapter" service will be injected
// via constructor.
$instance = $serviceLocator->get(SomeClass::class);

// ...
```

CHAPTER 3

Module/Component Resources

New in version 1.0.

Some of your modules may need to provide public resources like images, css or js files. To allow bundeling them within your module, rampage offers a resource locator system that will automatically publish them. You don't need to copy resources manually or make your vendor directory available to the webserver.

3.1 Defining Module Resources

Defining module resources is pretty easy. You just need to add them to your zf2 module config:

```
class Module
{
    public function getConfig()
    {
        return [
            // ...
            'rampage' => [
                'resources' => [
                    // Minimal, define only the base directory:
                    'foo.bar' => __DIR__ . '/resources',

                    // More detailed, allows to define additional resource types:
                    'foo.baz' => [
                        'base' => __DIR__ . '/resources',
                        'xml' => __DIR__ . '/resources/xml',
                    ],
                ]
            ]
        ];
    }
}
```

If you're using the manifest.xml for your modules, you can define them in the resources tag:

```
<manifest xmlns="http://www.linux-rampage.org/ModuleManifest" xmlns:xsi="http://www.w3.org/2001/XMLS
        <resources>
            <paths>
                <path scope="foo.bar" path="resource" />
                <path scope="foo.baz" path="resource" />
                <path scope="foo.baz" type="xml" path="resource/xml" />
            </paths>
```

```
        </resources>
</manifest>
```

3.2 Accessing Resources In Views

To access resources in views, you can use the `resourceurl` helper. The argument passed to this helper is the relative file path prefixed with `@` and the scope like this: `@scope/file/path.css`.

Paths without the `@` prefix will be searched in the current *theme* hierarchy directly. No attempt will be made to find them in a resource location.

Example:

```
<img src="<?php echo $this->resourceUrl('@my.module/images/foo.gif') ?>" />
```

3.3 Addressing Templates

Templates will also be populated by the resource locator. You can address them the same way as *resources in view scripts* like this: `@scope/templatepath`.

Note: Paths without the `@` prefix will be searched in the *theme* directly or treated as default zf2 templates if they're not found in the theme hierarchy.

Example:

```
$viewModel = new ViewModel();
$viewModel->setTemplate('my.module/some/template');
```

3.4 Static Resource Publishing

There is also a way to publish resources to the *public* directory for static delivery. This is useful for production environments, where performance is important.

There are two ways to do this:

1. *Use The Publishing Controller*
2. *Implement/Modify The Publishing Strategy*

3.4.1 Use The Publishing Controller

New in version 1.1.1.

The easiest way to do this, is to register the resources controller for publishing.

```
// module.config.php
return [
    'console' => [
        'router' => [
            'routes' => [
                'publish-resources' => \rampage\core\controllers\ResourcesController::getConsoleRoute
```

```
            ]
        ]
    ],
];
```

You may also pass the route to *getConsoleRouteConfig()* if you don't like *publish resources* as route or create an own route yourself pointing to the *publish* action of *rampage\core\controllers\ResourcesController*.

Note: The *getConsoleRouteConfig()* method is available since 1.1.1, prior that version you have to register the route config on your own.

```
return [
    'console' => [
        'router' => [
            'routes' => [
                'publish-resources' => [
                    'options' => [
                        'route' => 'publish resources',
                        'defaults' => [
                            'controller' => 'rampage\\core\\controllers\\ResourcesController',
                            'action' => 'publish'
                        ]
                    ]
                ]
            ]
        ]
    ]
];
```

3.4.2 Implement/Modify The Publishing Strategy

The controller uses the service *rampage.ResourcePublishingStrategy* which must implement *rampage\core\resources\PublishingStrategyInterface*. By default this interface is implemented by *rampage\core\resources\StaticResourcePublishingStrategy*.

The default strategy will publish all resources to *static/* in the *public* directory.

3.5 Special Controllers/Routes

When implementing an authentication strategy which protects all of your routes from unauthorized access, you should be aware that the resource publishing strategy uses a ZF2 route/controller to publish static resources from your vendor or module directories.

The controller class is *rampage\core\controllers\ResourcesController* and it is registerd as *rampage.cli.resources* in the controller manager. The route for this controller is called *rampage.core.resources*.

If you do not allow this route/controller, public resources from your modules may not be served.

CHAPTER 4

Themes Support

Rampage-PHP allows you to register cascading themes to apply different styles to your application in a flexible way.

Themes can depend on another theme to which it will fall back when a resource is not found.

Of course you can *overwrite module resources* like templates, css, js, images, etc. In this case the resource will be loaded from your (or the parent) theme if present.

4.1 Defining Themes

Themes can be defined in your ZF2 module (or application) configuration as follows:

```
namespace my\component;

class Module
{
    public function getConfig()
    {
        return [
            'rampage' => [
                'themes' => [
                    'my.theme.name' => [
                        // Simple path definition following the default layout
                        'path' => __DIR__ . '/../theme',
                        'fallbacks' = [ 'parent.theme', rampage\core\resources\Theme::DEFAULT_THEME
                    ],

                    'my.other.theme.name' => [
                        // More detailed path definition
                        'path' => [
                            'base' => __DIR__ . '/../theme',
                            'template' => __DIR__ . '/../theme/templates',
                        ],
                        'fallbacks' = [ 'my.theme.name', 'parent.theme', rampage\core\resources\Theme
                    ]

                ]
            ]
        ];
    }
}
```

One module may define multiple themes if needed.

Note: The theme will try the fallbacks in the order in which they're defined. It will fallback flat through those themes, **not** in depth.

i.e. The fallbacks of a fallback theme are not visited.

4.2 Theme Directory Layout

Within the defined theme directory, there should be two directories.

- `public` Which contains all public assets
- `template` Which contains template files

4.3 Overwriting Module Resources

Of course it is possible to overwrite module resources, like templates or public assets, in a theme. To do so simply place the file ine a directory named like the module's resource name (See *Defining Module Resources* for details).

Example:

- **theme directory**
  - **public**
    - **module.resource.name**
      - some/public/file.css
  - **template**
    - **module.resource.name**
      - some/template.phtml
      - some-other-template.phtml

CHAPTER 5

Filesystem Abstraction

The `rampage\\filesystem` component provides an OO abstraction layer for filesystem access.

A filesystem might be a local, remote or even a virtual filesystem.

5.1 FilesystemInterface

Interface: `rampage\\filesystem\\FilesystemInterface`

This interface defines basic filesystem access via the *ArrayAccess* and *RecursiveIterator* patterns. It encapsulates the underlying filesystem and provides container like access. Much like Phar or ZipArchive

It is not intended to provide write access.

The following classes are provided as implementation for this interface:

- *LocalFilesystem*: For fs access on disk.
- GridFS: For accessing a MongoDB GridFS.

5.2 WritableFilesystemInterface

Interface: `rampage\\filesystem\\WritableFilesystemInterface`

This interface enhances the *FilesystemInterface* by defining write capabilities which are:

- adding files
- deleting files
- creating directories
- updating the access/modified timestamps (touch)

An implementation must provide these methods and respond gracefully, when they're not supported (i.e. creating directories).

The following classes are provided as implementation for this interface:

- *WritableLocalFilesystem*: for fs access on disk.
- WritableGridFS: For accessing a MongoDB GridFS.

5.3 Available Implementations

5.3.1 LocalFilesystem

The `rampage\\filesystem\\LocalFilesystem` implements the *FilesystemInterface* for a local filesystem on disk. It encapsulates `DirectoryIterator` and `SplFileInfo` to provide this functionality.

5.3.2 WritableLocalFilesystem

The `rampage\\filesystem\\WritableLocalFilesystem` implements the *WritableFilesystemInterface* for a local filesystem on disk.

It extends *LocalFilesystem* and encapsulates `DirectoryIterator` and `SplFileInfo` to provide this functionality.

CHAPTER 6

Convenience Classes

There are some convenience classes to make development of common tasks more easy

6.1 ServiceCallbackDelegator

This class allows to wrap a lazy load callback to a specific service. Lazy means this class will only instanciate the service when the callback is about to be invoked.

The service locator may be injected or changed at any time. That means you can create the callback without a service locator attached and assign it later.

The service locator must implement `Zend\\ServiceManager\\ServiceLocatorInterface`, which is the case for the zf2 ServiceManager.

Note: Invoking such a callback wrapper that has no service locator, will throw a `rampage\core\exceptions\LogicException`.

6.1.1 Usage

The constructor takes two arguments. The first one is the service name, which is required. The second, optional, argument is the method name to call. If the method name is omitted, the service object will be treated as invokable (i.e. for classes that implement `__invoke()`). **Example:**

```
// ...
$callback = new rampage\core\ServiceCallbackDelegator('MyServiceName', 'someMethod');
$callback->setServiceLocator($serviceManager); // assuming $serviceManager is a ServiceLocator

(new Zend\EventManager\EventManager())->attach('somevent', $this, $callback);
```

This example would request the service `MyServiceName` from `$serviceLocator` when `somevent` is triggered on the EventManager.

6.1.2 Injecting the service locator

The service locator can be set via `setServiceLocator()` on the callback instance at any time (See the *Usage Example* above).

6.2 GracefulArrayAccess

This class provides graceful access to arrays or objects that implement `ArrayAccess`. This means requesting a key that does not exist will not trigger a warning, but return a default instead.

6.2.1 Usage Example

```
$wrapper = new GracefulArrayAccess(['bar' => true]);

var_dump($wrapper->get('foo', 'default value'));
var_dump($wrapper->get('foo'));
var_dump($wrapper->get('bar'));
```

This example will output:

```
string(13) "default value"
NULL
bool(true)
```

CHAPTER 7

Additional Components Documentation

This section will cover additional components, that are not part of the core library.

7.1 Authentication Module (rampage.auth)

This component provides enhancments to the zf2 authentication component (Zend\Authentication).

7.1.1 Composer Installation

To obtain this component, add a dependency for `rampage-php/auth` to your composer.json.

CHAPTER 8

Glossary

ArrayAccess The array access pattern/interface defines a container object that can be accessed as array. See the ArrayAccess php documentation for details.

Traversable, Iterator The iterator or Traversable pattern allows to iterate over container objects (i.e. via foreach). See the Iterator php documentation for details.

RecursiveIterator Additionally to the *Iterator* pattern, this pattern allows a multilevel or tree iteration over a container object. See the RecursiveIterator php documentation for details.

fqcn Short for "full qualified class name". The full qualified class name is the class name including its full namespace like `foo\\bar\\baz\\ClassName` for `ClassName` in `namespace foo\\bar\\baz`.

CHAPTER 9

Overview

Rampage-PHP is a framework enhancement library for ZendFramework 2. It is supposed to offer some convenience components.

As most developers don't like to write the same code fragments over and over again. This is the point where rampage-php comes in.

Read the *Quickstart Guide* to get started.

CHAPTER 10

Key Features

- *Tight integration between Di and ServiceManager*
- XML based module configurations (XSD provided)
- *Powerful resource locators for module resource files (i.e. js and css)*
- *Advanced url locators/helpers*
- *Cascading themes support*
- Base implementation to create mergable XML config models
- *Service callback wrappers*
- *Filesystem Containers*