rampage-php Documentation

Release 1.0

Axel Helmert

October 21, 2014

Contents

Welcome to the reference guide to rampage-php.

CHAPTER 1

Overview

1.1 Rampage-PHP 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.

1.1.1 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
- Auth framework (wip)
- Cascading themes support
- Base implementation to create mergable XML config models.
- Service callback wrappers.

1.2 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).

1.2.1 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.

1.2.2 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`

1.3 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.

1.3.1 Defining Module Resources

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

```php
<?php

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/XMLSc
        <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>
```

1.3.2 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 the scope like this: *scope::file/path.css*.

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

1.3.3 Addressing templates

Templates will also be populated by the resource locator. You can address them by prepending them with the scope like this: *scope/templatepath*.

```
<?php

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

1.3.4 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.

Use the publishing controller

New in version 1.1.1.

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

```php
<?php

// module.config.php
return array(
    'console' => array(
        'router' => array(
            'routes' => array(
                '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.

```php
<?php

returnarray(
    'console' => array(
        'router' => array(
            'routes' => array(
                'publish-resources' => array(
                    'options' => array(
                        'route' => 'publish resources',
                        'defaults' => array(
                            'controller' => 'rampage\\core\\controllers\\ResourcesController',
                            'action' => 'publish'
                        ),
                    ),
                )
            )
        )
    )
);
```

Implement or 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.

1.3.5 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 2

Indices and tables

- *search*