Flint Documentation

Release 1.7.1-3-g1f64b90

Henrik Bjrnskov

May 26, 2014

Contents

Flint is a microframework built on top of Silex. It tries to bridge the gap between Silex and Symfony by bringing structure and conventions to Silex.

CHAPTER 1

Getting started

To start using Flint the easiest way is to edit your `composer.json` file to require Flint and change the Application class that is used.

```
$ php composer.phar require flint/flint:~1.0
```

```
<?php

use Flint\Application;

$application = new Application($rootDir, $debug);
```

It is recommended to subclass `Flint\Application` instead of using the application class directly.

CHAPTER 2

Controllers

Flint tries to make Silex more like Symfony. And by using closures it is hard to separate controllers in a logical way when you have more than a couple of them. To make it better it is recommended to use classes and methods for controllers. The basics are explained here but Flint takes it further and allows the application to be injected into a controller.

The first way to accomplish this is by implementing `PimpleAwareInterface` or extending `PimpleAware`. This works exactly as described in Symfony. With the only exception that the property is called `$pimple` instead of `$container`.

```php
<?php

namespace Acme\Controller;

use Flint\PimpleAware;

class HelloController extends PimpleAware
{
    public function indexAction()
    {
        return $this->pimple['twig']->render('Hello/index.html.twig');
    }
}
```

Another way is to use a base controller which have convenience methods for the most frequently used services. These methods can be seen in the source code if you look at the implementation for `Flint\Controller\Controller`.

```php
<?php

namespace Acme\Controller;

use Flint\Controller\Controller;

class HelloController extends Controller
{
    public function indexAction()
    {
        return $this->render('Hello/index.html.twig');
    }
}
```

CHAPTER 3

Routing

Because Flint replaces the url matcher used in Silex with the full router implementation a lot of new things are possible.

Caching is one of those things. It makes your application faster as it does not need to register routes on every request. Together with loading your routes from a configuration file like Symfony it will also bring more structure to your application.

To enable caching you just need to point the router to the directory you want to use and if it should cache or not. By default the `debug` parameter will be used as to determine if cache should be used or not.

```
<?php

// .. create a $app before this line
$app['routing.options'] = array(
    'cache_dir' => '/my/cache/directory/routing',
);
```

Warning: Migrating from Silex
Flint automatically enables `UrlGeneratorServiceProvider` therefor it is not needed to do this manually.

Before it is possible to use the full power of caching it is needed to use configuration files because Silex will always call add routes via its convenience methods `get|post|delete|put`. Fortunately this is baked right in.

```
<?php

// .. create $app
$app['routing.resource'] = 'config/routing.xml';

<!-- config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/routing-1

    <route id="homepage" pattern="/">
        <default key="_controller">Acme\\Controller\\DefaultController::indexAction</default>
    </route>
</routes>
```

This will make the router load that resource by default. Here xml is used as an example but `php` is also supported together with `yml` if `Symfony\Component\Yaml\Yaml` is autoloadable.

The benefit from doing it this way is of course they can be cached but also it allows you to import routing files that are included in libraries and even other Symfony bundles such as the WebProfilerBundle.

The WebProfilerBundle routes must be imported into your routing file if you intend on using it. Here's an example of what your routing file may look like in YAML format:

```
_profiler:
  resource: "/path/to/vendor/symfony/web-profiler-bundle/Symfony/Bundle/WebProfilerBundle/Resources/c
  type:     xml
  prefix:   /_profiler

_wdt:
  resource: "/path/to/vendor/symfony/web-profiler-bundle/Symfony/Bundle/WebProfilerBundle/Resources/c
  type:     xml
  prefix:   /_wdt
```

Also it will make it easier to generate routes from inside your views.

```
<a href="{{ app.router.generate('homepage') }}">Homepage</a>
```

This is also possible with Silex but with a more verbose syntax. The syntax can be even more precise by using the twig functions that is available in the Twig bridge for Symfony. To enable those add the twig bridge to your composer file.

```
{
    "require" : {
        "symfony/twig-bridge" : "~2.0"
    }
}
```

Now it is possible to use the functions inside your Twig templates.

```
<a href="{{ path('homepage') }}">Homepage</a>
<a href="{{ url('homepage') }}">Homepage</a>
```

CHAPTER 4

Default Parameters

The two constructor arguments `$rootDir` and `$debug` are also registered on the application as parameters. This makes it easier for services to add paths for caching, logs or other directories.

```
<?php

$app = new Flint\Application(__DIR__, true);
$app['debug'] === true;
$app['root_dir'] === __DIR__;
```

CHAPTER 5

Custom Error Pages

When finished a project or application it is the small things that matter the most. Such as having a custom error page instead of the one Silex provides by default. Also it can help a lost user navigate back. Flint makes this possible by using the exception handler from Symfony and a dedicated controller. Both the views and the controller can be overridden.

This will only work when debug is turned off.

To override the error pages the same logic is used as inside Symfony. The logic is very well described in their documentation.

Only difference from Symfony is the templates must be created inside `views/Exception/` directory. Inside the templates there is access to `app` which in turns gives you access to all of the services defined.

To override the controller used by the exception handler change the `exception_controller` parameter. This parameter will by default be set to `Flint\\Controller\\ExceptionController::showAction`.

```
<?php

// .. create $app
$app->inject(array(
    'exception_controller' => 'Acme\\Controller\\ExceptionController::showAction',
));
```

To see what parameter the controller action takes look at the one provided by default. Normally it should not be overwritten as it already gives a lot of flexibility with the template lookup.

CHAPTER 6

Pimple Console

6.1 Helper

Flint have a helper that provides access to a pimple instance or in the case of Flint access to you application object.

```
<?php

class SomeCommand extends Command
{
    public function execute(InputInterface $input, OutputInterface $output)
    {
        $pimple = $this->getHelperSet()->get('pimple');
    }
}
```

To register the helper do this.

```
<?php

$app = new Symfony\Component\Console\Application;
$app->getHelperSet()->set(new Flint\Console\PimpleHelper($pimple));
```

6.2 Application

Warning: This is deprecated and it is adviced to use `Flint\Console\PimpleHelper` instead.

`Flint\Console\Application` is an extension of the base console application shipped with Symfony. It gives access to Pimple in commands.

```
<?php

namespace Application\Command;

use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class MyCommand extends \Symfony\Component\Console\Command\Command
{
    protected function execute(InputInterface $input, OutputInterface $output)
```

```
    {
        $pimple = $this->getApplication()->getPimple();
    }
}
```

CHAPTER 7

Configuration

Every application need to have some parameters configured based on environment or other parameters. Flint comes with a `Configurator` which reads `json` files and sets them as parameters on your application.

```
<?php

use Flint\Application;

$app = new Application($rootDir, $debug);
$app->configure('config.json');

// Or use the service directly
$app['configurator']->configure($app, 'app/config/prod.json');
```

Note: For more information about how configuration loading works read the Tacker documentation.

Warning: When using Silex version 1.0.0 or earlier it is not possible to load configurations in the boot method. This is because when adding a listener to the *dispatcher* service it will get the routes and a bunch of other services which means it is too late.