The Silex Sunset

Alexander M. Turek

about:me

Freelance software developer.
Builds web applications with php.
Likes Symfony.
~~Frankfurt~~ ~~Munich~~ ~~Berlin~~ Düsseldorf.
Father of ~~three~~ four.
Likes digging into legacy applications.

Silex

PHP microframework
based on Symfony components

Hello World

A working application in one file!

```php
<?php

// web/index.php
require_once __DIR__.'/../vendor/autoload.php';

$app = new Silex\Application();

$app->get(
    '/hello/{name}',
    function ($name) use ($app) {
        return 'Hello '.$app->escape($name);
    }
);

$app->run();
```

Silex under the Hood

- Symfony HTTP Kernel
- Symfony Routing
- Symfony Event Dispatcher
- Pimple (DI Container)

Goodbye World

The Symfony core team stopped maintaining Silex.

The end of Silex

January 12, 2018  Fabien Potencier

What about Silex in a Symfony 4 world? During the last few months, and as an exercise when working on Flex, I have migrated several applications from Silex to Symfony 4. And the conclusion is that Symfony 4 feels like using Silex.

For all these reasons, I would say that Silex is not needed anymore. So, we've decided to not support Symfony 4 in Silex, or at least not add the new features added in 3.4. The current stable version of Silex is still maintained for bugs and security issues. But its end of life is set to **June 2018**.

Don’t panic (yet).

- Symfony components will be around after Silex' EOL.
- Pimple can be considered feature-complete.
- You should get rid of Silex eventually, but no need to rush.

Silex' real EOL

Symfony 4.4 will be a **long term support** version published in November 2019.

Roadmap

Nov 2019
Release

Nov 2022
End of support for bug fixes

Nov 2023
End of support for security fixes

TIP Get this information in JSON format: https://symfony.com/roadmap/4.4.json

The Silex Sunset

Any Silex application can be refactored to a state in which there are equal configurations for Silex and Symfony for that application.

OpenCFP

Open source application
for conferences
to manage a call for papers.

Built on top of ~~Silex~~ Symfony.

opencfp / opencfp

Watch 29 | Unstar 428 | Fork 178

Code | Issues 26 | Pull requests 2 | Projects 0 | Wiki | Insights

Switch to Symfony #895

Edit

Merged chartjes merged 1 commit into `opencfp:master` from `derrabus:symfony` on 18 Dec 2017

Conversation 40 | Commits 1 | Checks 0 | Files changed 44 | +1,870 −1,622

derrabus commented on 7 Dec 2017 • edited — Contributor

This PR switches the application to Symfony.

- [x] Upgrade to Symfony 3.4.
- [x] Eliminate container-awareness.
- [x] Convert middlewares to event subscribers.
- [x] Configure a Symfony kernel for OpenCFP.
- [x] Switch front controller to the Symfony kernel.
- [x] Switch console to the Symfony kernel.
- [x] Switch integration tests to the Symfony kernel.
- [x] Remove the `Application` and all providers.
- [x] Remove `silex/silex`.
- ~~Raise php requirement to 7.1.~~
- ~~Upgrade to Symfony 4.0.~~

This PR switches OpenCFP to a new Symfony kernel and removes all dependencies to the Silex package.

Fixes #618.

Reviewers

chartjes ✓

localheinz

mdwheele ●

BackEndTea ●

Assignees

chartjes

mdwheele

BackEndTea

Labels

enhancement

Projects

None yet

Pimple

A very simple implementation of a dependency injection container.

```php
$container = new \Pimple\Container();

$container['my_service'] = function () {
    return new MyService();
};

$container['my_other_service'] = function ($c) {
    return new MyOtherService(
        $c['my_service']
    );
};

$container['my_other_service']->doStuff();
```

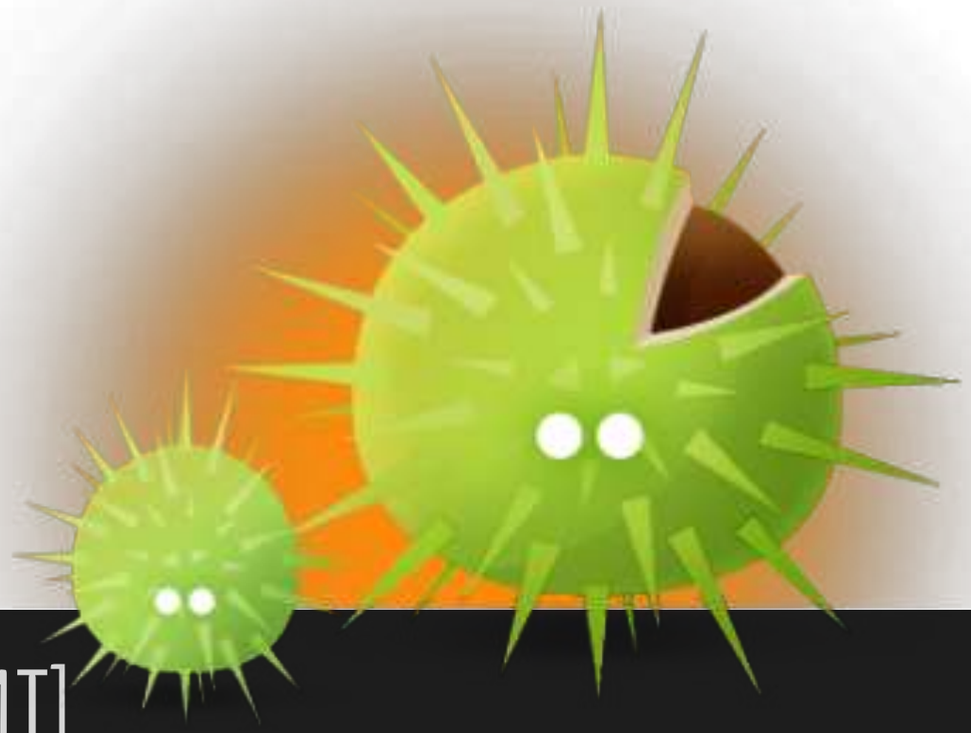
Pimple Vs. Symfony DI

```php
$app['my_service'] = function () {
    return new MyService();
};

$app['my_other_service'] = function ($app) {
    return new MyOtherService(
        $app['my_service']
    );
};
```

```yaml
my_service:
    class: Acme\MyApp\MyService

my_other_service:
    class: Acme\MyApp\MyOtherService
    arguments:
        - '@my_service'
```



Symfony

Pimple Vs. Symfony DI

Pimple	Symfony Dependency Injection
Services are defined by factory functions.	Services are defined by a semantic configuration.
No auto-discovery.	Can discover services via PSR-4.
No auto-wiring.	Can guess dependencies via reflection.
Ready to use after configuration.	Ready to use after compilation.
Frozen after first service lookup.	Frozen after compilation.
Array-like access, provides a wrapper for PSR-11.	PSR-11 compatible interface.

Integration of 3rd-Party Libraries

- Libraries can be integrated into a Silex application using service providers.
- Symfony uses bundles for this purpose.
- Challenge: Find bundles that replace the service providers your application uses.

The Application Class

- Acts as your kernel.
- Acts as your DI container.
- Acts as a configurator for routing, DI and events.
- A façade to framework features (abort, stream, escape, json,…)
- No counterpart in Symfony.

What's the S in SOLID again?

- Silex violates the single responsibility principle on purpose.
- Developers are tempted to add even more utilities to the Application class.
- Result: `$app` is accessed directly throughout the codebase.

Move Helpers into Services

```php
class Application extends \Silex\Application
{
    public function helpMe($param)
    {
        // Insert helper logic here.
    }
}
```

```php
class Application extends \Silex\Application
{
    public function __construct(array $values = array())
    {
        parent::__construct($values);

        $this['my_helper'] = function () {
            return new MyHelper();
        };
    }

    public function helpMe($param)
    {
        return $this['my_helper']->helpMe($param);
    }
}
```

```php
$app->helpMe('with this');
```

```php
$app['my_helper']->helpMe('with this');
```

Problem: Service Lookup

- Silex application is used for service lookups.
- Symfony DI supports service location, but the interface is incompatible.

```php
namespace Acme\MyApp\Controller;

use Silex\Application;

class MyController
{
    public function myAction(Application $app)
    {
        $parameters = [];

        // Insert controller logic here.

        return $app['twig']
            ->render(
                'my_template.html.twig',
                $parameters
            );
    }
}
```

Solution A: Dependency Injection

- Controllers as services.
- The controller is not aware of the container anymore.
- Very time-consuming refactoring, if you have a lot of controllers.

```php
namespace Acme\MyApp\Controller;

use Twig\Environment;

class MyController
{
    private $twig;

    public function __construct(Environment $twig)
    {
        $this->twig = $twig;
    }

    public function myAction()
    {
        $parameters = [];

        // Insert controller logic here.

        return $this->twig
            ->render(
                'my_template.html.twig',
                $parameters
            );
    }
}
```

Solution B: PSR-11

- No need to register controllers as services.
- The controller is not aware of the container implementation anymore.
- Easier refactoring step.

```php
namespace Acme\MyApp\Controller;

use Psr\Container\ContainerInterface;

class MyController
{
    public function myAction(
        ContainerInterface $container
    ) {
        $parameters = [];

        // Insert controller logic here.

        return $container->get('twig')
            ->render(
                'my_template.html.twig',
                $parameters
            );
    }
}
```

I've built a package for that.

Psr11ServiceProvider

This service provider registers a PSR-11 compatible container as a service inside a Silex application.

PSR-11 enables developers to write code that is aware of the service container without coupling it to a specific container implementation, thus allowing to switch to another service container more easily.

build passing

Installation

Use Composer to install the package.

```
composer require derrabus/silex-psr11-provider
```

Usage

Once you have registered the service provider, you can either access the container as service `service_container` or as a controller argument by using `Psr\Container\ContainerInterface` as type hint.

```php
$app->get(
    '/hello/{name}',
    function ($name) use ($app) {
        return 'Hello '.$app->escape($name);
    }
);
```

```php
$app->before(function ($request, $app) {
    // Do something.
});
```

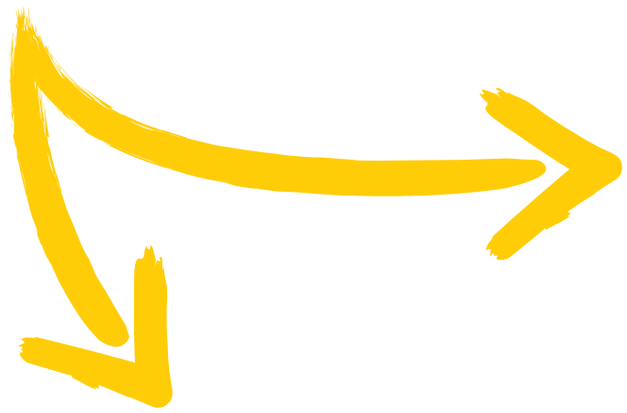
```php
$app->on(
    KernelEvents::REQUEST,
    function ($event) {
        // Do something.
    }
);
```

Anonymous Functions

Anonymous functions cannot be compiled into the container or the router.

Event Subscribers

```php
$app->on(
    KernelEvents::REQUEST,
    function ($event) {
        // Do something.
    }
);
```



```php
namespace Acme\MyApp\Event;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;
use Symfony\Component\HttpKernel\KernelEvents;

class DoSomethingListener implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return [KernelEvents::REQUEST => 'onKernelRequest'];
    }

    public function onKernelRequest($event)
    {
        // Do something.
    }
}
```

```php
$app->extend('dispatcher', function ($dispatcher) {
    $dispatcher->addSubscriber(new DoSomethingListener());

    return $dispatcher;
});
```

Middlewares

- Execute logic before the controller is called.
- A fancy way to define event listeners.
- Can be migrated to event subscribers as well.

```php
$app->before(function ($request, $app) {
    // Do something.
});
```

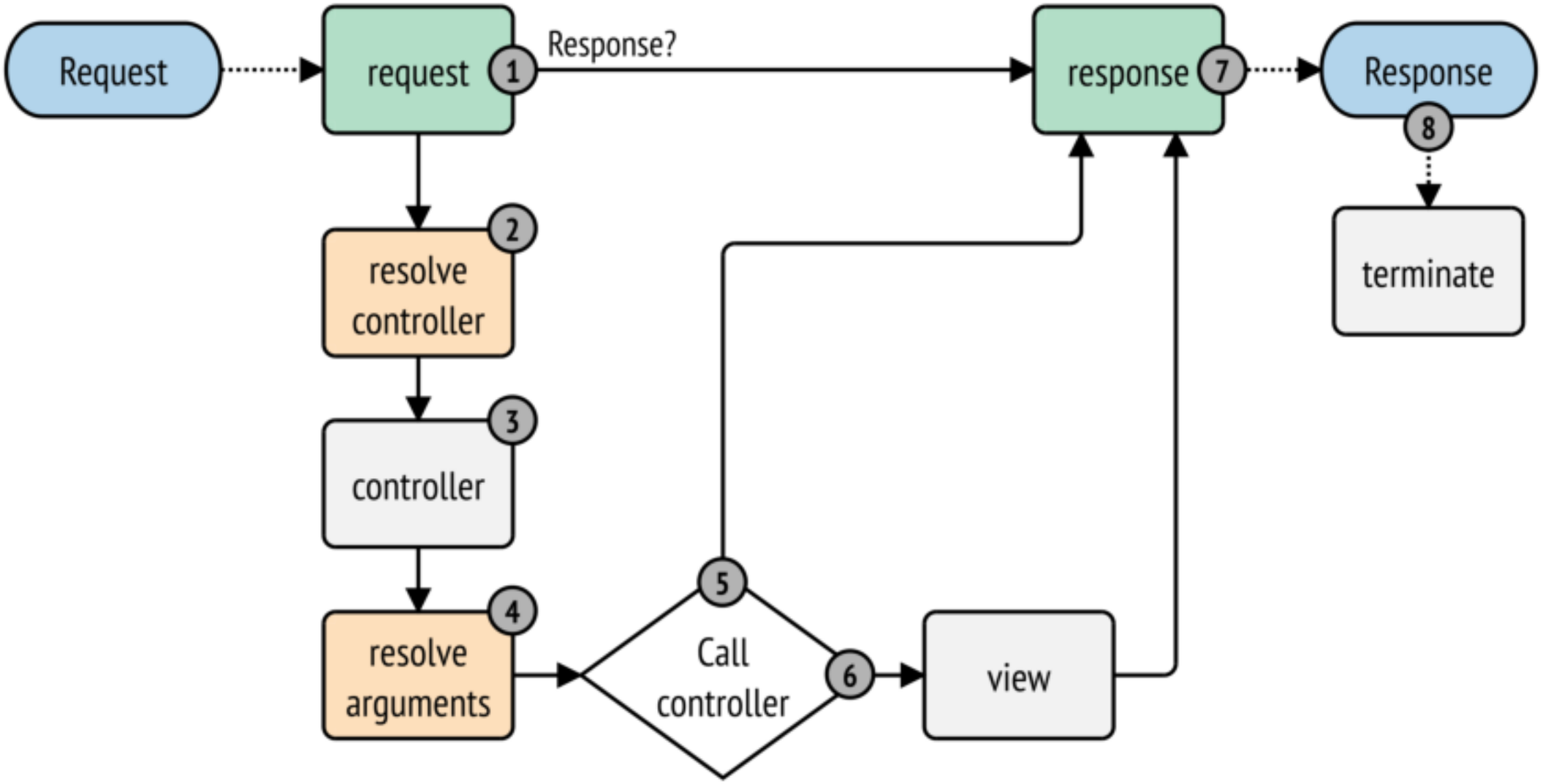
```php
public function before($callback, $priority = 0)
{
    $this->on(
        KernelEvents::REQUEST,
        function ($event) use ($callback) {
            if (!$event->isMasterRequest()) {
                return;
            }

            $ret = $callback($event->getRequest(), $this);

            if ($ret instanceof Response) {
                $event->setResponse($ret);
            }
        },
        $priority
    );
}
```

Middlewares

Middleware	Symfony Event	Default Priority
`$app->before()`	`kernel.request`	0
`$app->after()`	`kernel.response`	0
`$app->finish()`	`kernel.terminate`	0
`$app->error()`	`kernel.exception`	-8
`$app->view()`	`kernel.view`	0



Symfony Event Cycle

Route Middlewares

- Middlewares can be attached to routes!
- Effect: Event listener is only called if the route matches.

```
$csrfChecker = function ($request, $app) {
    $checker = $app['csrf_validator'];

    if (!$checker->isValid($request)) {
        return new RedirectResponse('/dashboard');
    }
};

$app->post(
        '/talk/create',
        'talk_controller:processCreateAction'
    )
    ->bind('talk_create')
    ->before($csrfChecker);
```

Generalize route middleware so it can be run for all routes.

Solution A: Marker Attribute

- Replace route middleware with an attribute.
- Register a global middleware that checks for the attribute.
- Refactor middleware to event subscriber.

```php
$app->before(function ($request, $app) {
    if (!$request->attributes->get('_check_csrf')) {
        return;
    }

    $checker = $app['csrf_validator'];

    if (!$checker->isValid($request)) {
        return new RedirectResponse('/dashboard');
    }
});

$app->post(
        '/talk/create',
        'talk_controller:processCreateAction'
    )
    ->bind('talk_create')
    ->value('_check_csrf', true);
```

https://github.com/opencfp/opencfp/pull/888

Solution B: RegEx on URI

- Replace route middleware with a global middleware that runs a regex against the request URI.
- Useful for firewall-like middlewares.

```php
public function onKernelRequest($event)
{
    $uri = $event->getRequest()->getRequestUri();

    if (\preg_match('/^\/(talk|profile)/', $uri)) {
        // check for speaker permissions
    }

    if (\preg_match('/^\/admin/', $uri)) {
        // check for admin permissions
    }

    if (\preg_match('/^\/reviewer/', $uri)) {
        // check for reviewer permissions
    }
}
```

Routing

Routing

```php
$app
    ->get(
        '/',
        'my_controller:homeAction'
    )->bind('home');

$app
    ->get(
        '/imprint',
        'my_controller:imprint_action'
    )->bind('imprint');

$app
    ->get(
        '/login',
        'security_controller:loginFormAction'
    )->bind('login');

$app
    ->post(
        '/login',
        'security_controller:loginCheckAction'
    )->bind('login_check')
    ->value('_check_csrf', true);
```

```yaml
home:
    path: /
    controller: my_controller:homeAction
    methods: [GET]

imprint:
    path: /imprint
    controller: my_controller:imprintAction
    methods: [GET]

login:
    path: /login
    controller: security_controller:loginFormAction
    methods: [GET]

login_check:
    path: /login
    controller: security_controller:loginCheckAction
    methods: [POST]
    defaults:
        _check_csrf: true
```

Dump Router Configuration to YAML

- Our routing configuration does not contain any executable code anymore.
- Boot the application and dump the configured routes to YAML.
- Yes, this file is going to be huge. Refactor it after switching to Symfony.

https://gist.github.com/derrabus/4d7b7b3a6ffc0c1ccc1037ce2a66b13c

Ready to Switch?

- Our Controllers are Symfony-compatible.
- $app is used for service/routing configuration only.
- All event listeners bypass Silex' façade.
- Direct service lookups (if any) via PSR-11 only.
- Router configuration can be imported by Symfony.

Let's do it!

```
$ composer require symfony/flex
```

Optional!

Helps you with bootstrapping your Symfony application, but messes with your file structure.

```
$ composer require symfony/http-kernel
$ composer require symfony/framework-bundle
```

```php
namespace Acme\MyApp;

use Symfony\Bundle\FrameworkBundle\FrameworkBundle;
use Symfony\Component\Config\Loader\LoaderInterface;

final class Kernel extends \Symfony\Component\HttpKernel\Kernel
{
    public function registerBundles()
    {
        return [
            new FrameworkBundle(),
        ];
    }

    public function getCacheDir()
    {
        return $this->getProjectDir().'/var/cache/'.$this->getEnvironment();
    }

    public function getLogDir()
    {
        return $this->getProjectDir().'/var/logs';
    }

    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(
            $this->getRootDir().'/config/config_'.$this->getEnvironment().'.yml'
        );
    }
}
```

Temporary Framework Switch

- Switch to your new Symfony kernel.
- Test the application.
- Switch back to compare with the old behavior.
- Remove the switch if you're confident that the Symfony application works.

```php
use Acme\MyApp\Kernel;
use Silex\Application;
use Symfony\Component\HttpFoundation\Request;

// Insert your own logic here.
$debug = (bool) getenv('MYAPP_DEBUG');

if (getenv('MYAPP_USE_SYMFONY')) {
    $kernel = new Kernel(
        $debug ? 'dev': 'prod',
        $debug
    );
} else {
    // Add your old bootstrap logic here.
    $kernel = new Application([
        'debug' => $debug
    ]);
}

$request  = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

Swapping Out Libraries

- Writing decoupled code helps you in the long run.
- Find commonalities and make use of them.
- Find differences and use wrappers.
- Draft a migration path with small steps.

References

- The Silex Sunset
  https://medium.com/@derrabus/the-silex-sunset-d0fb5f1129df
- OpenCFP: Switch to Symfony (PR)
  https://github.com/opencfp/opencfp/pull/895
- OpenCFP: Migrate to Symfony 4 (Ticket)
  https://github.com/opencfp/opencfp/issues/618

Thank You

https://twitter.com/derrabus
https://github.com/derrabus
http://facebook.com/derrabus