Symfony2 & Performance

Alexandre Salomé

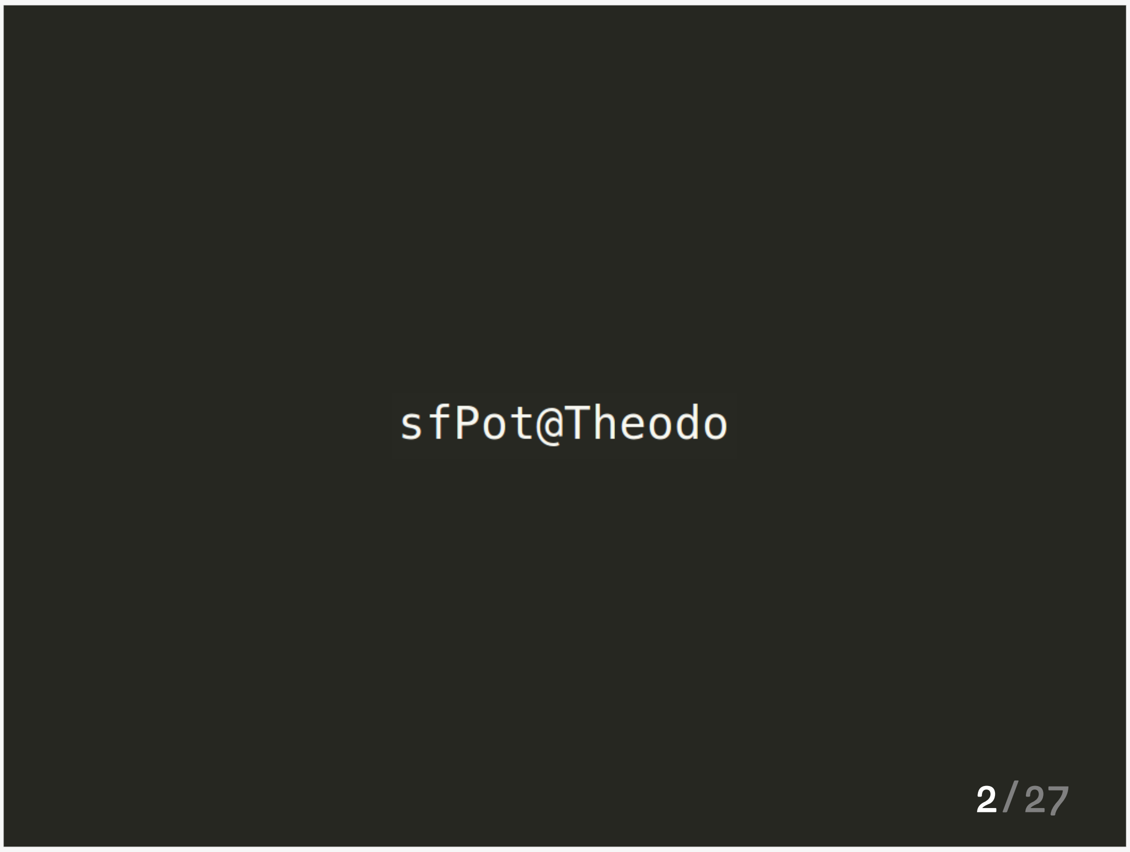
Alexandre Salomé – October 8th, 2012

I work for **Sensio Labs** since now more than 3 years.

Since 2 years, I'm a **consultant** for the company, and I take part of many important projects, going from a small team to a big group.

In this rich context, I use **Symfony2** as framework for most of my developments.

Today I try to show you how to take the best part from Symfony2 and reach a higher performance with your application.

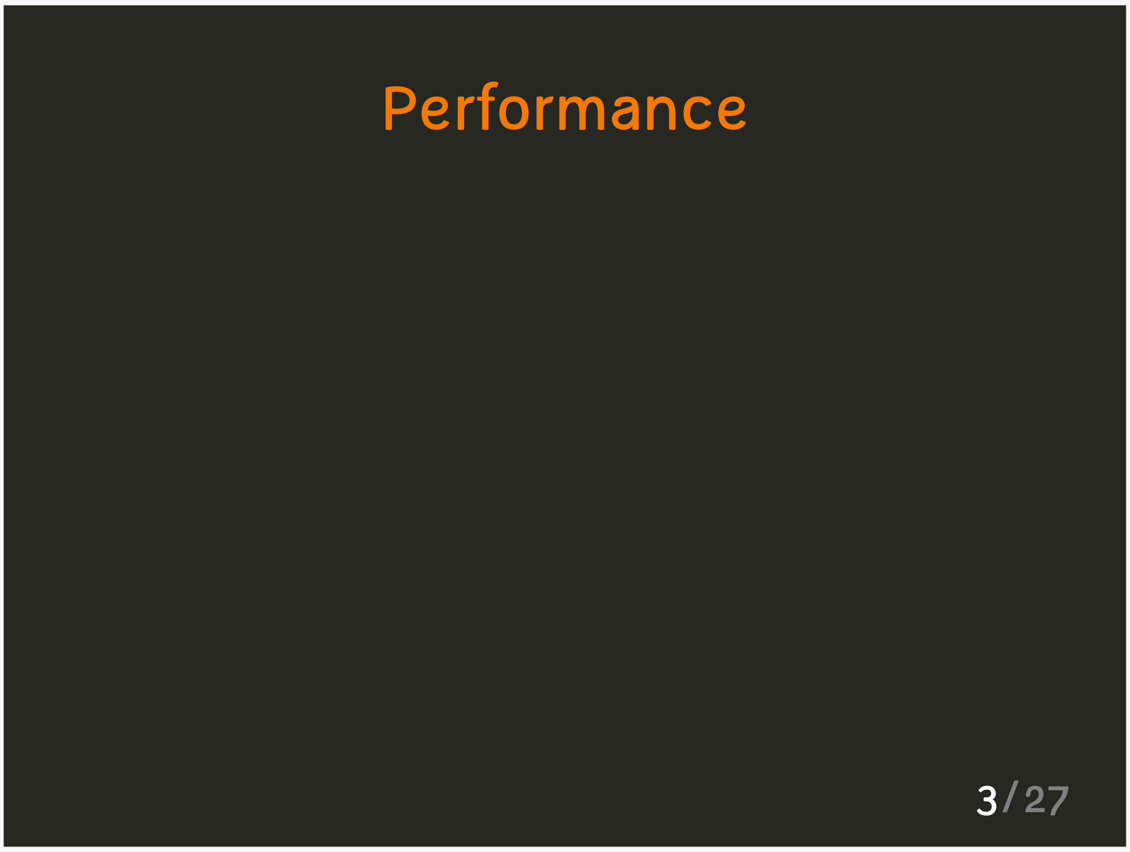


sfPot@Theodo

Thanks to **Theodo** for hosting the event.

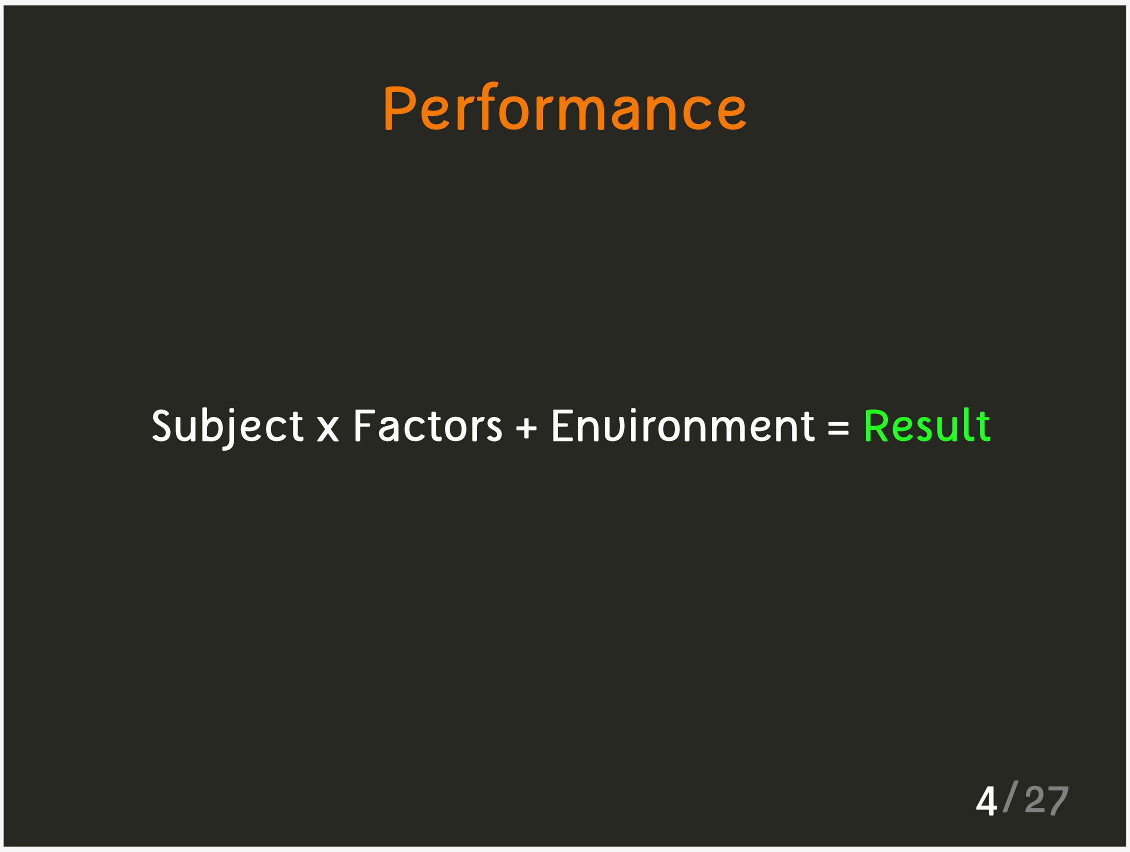
Different companies host this free event, the sfPot. There is a presentation, then a drink, a nice formula!

To organize the events and manage it, one association : Afsy. The association take care of organization of the event, the newsgroup and the website : afsy.fr



Performance

What is performance?



Performance

Subject x Factors + Environment = Result

Definition:
Optimal result a **subject** can reach.

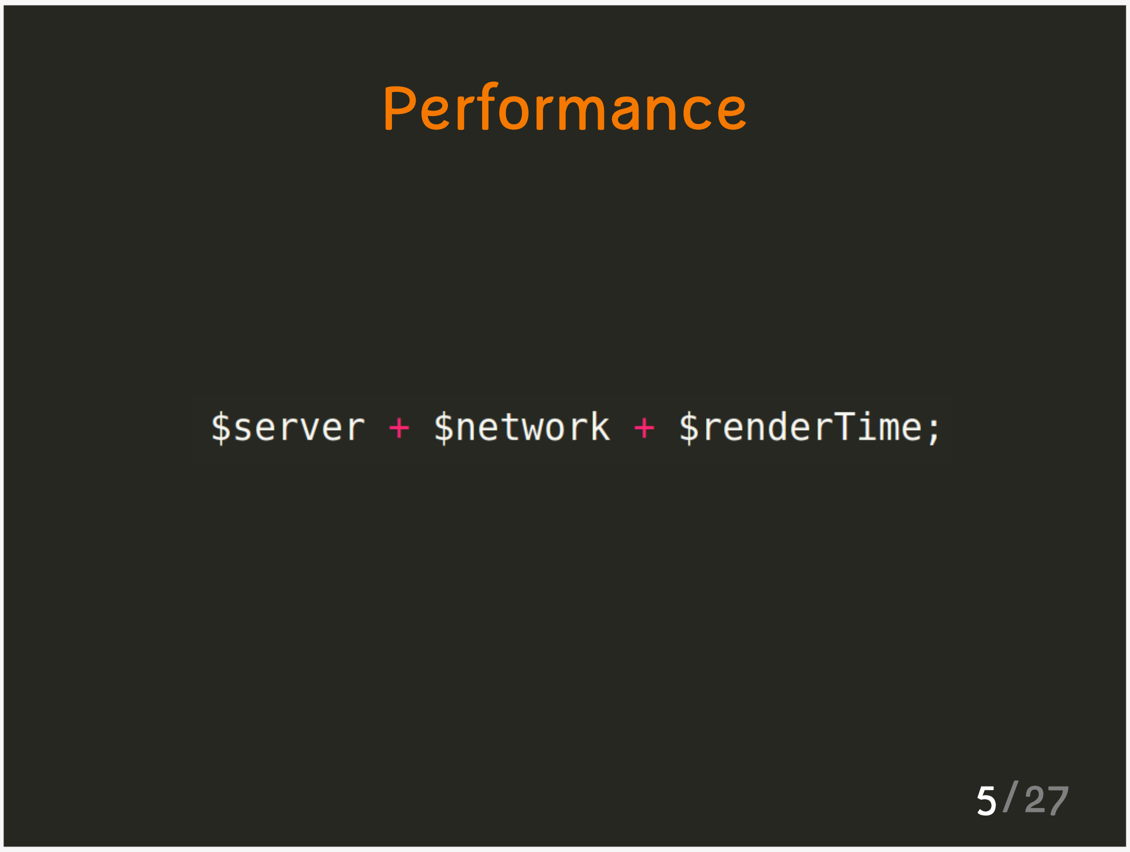
Examples: an athlete, a chicken, a trading-order or an artist.

We will firsti interest to **factors** we can attenuate.

Examples: what our athelete eats, chicken conditions.

Other conditions don't depend of us: this is called the **environment**.

Examples: gravity for our athele, news for a traing-order



Performance

```
$server + $network + $renderTime;
```

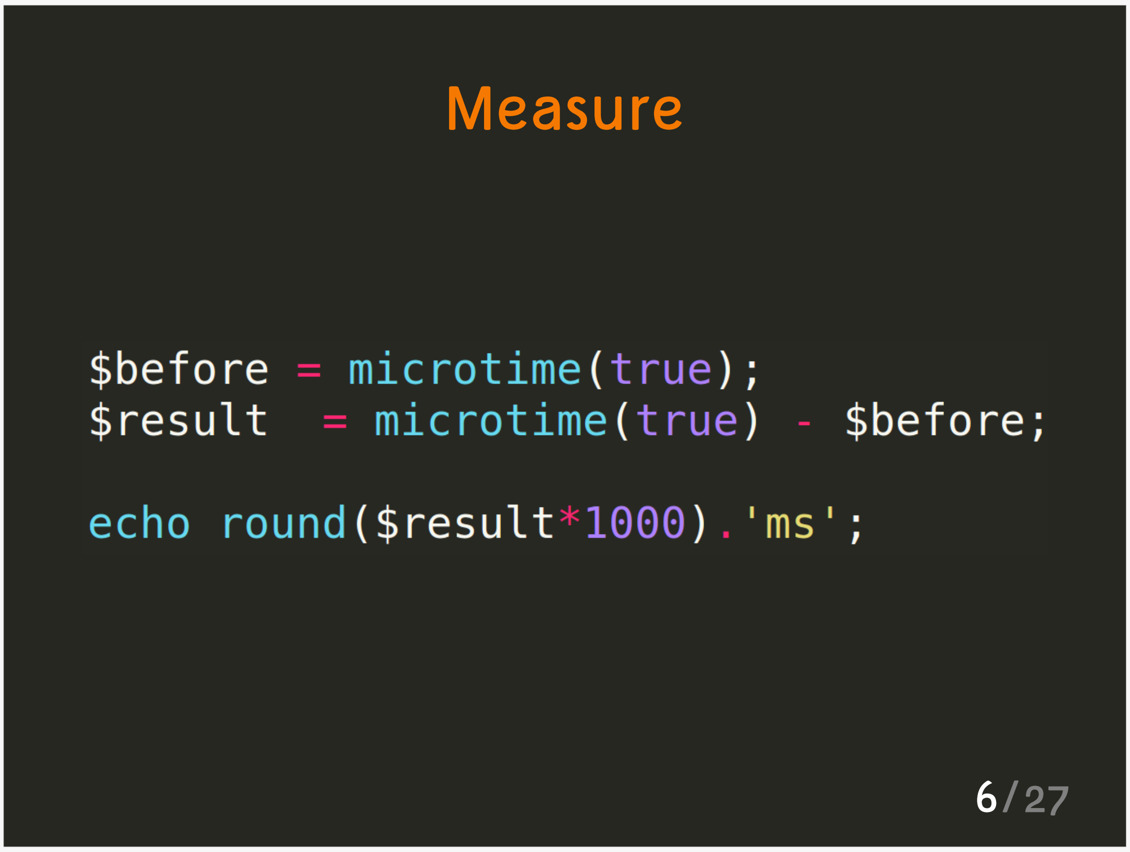
Peformance is the **sum of an all** :
- Server resources (disk, memory, network)
- Platform setup (APC, PHP)
- Application
- Network latency
- Download and client-side processing

Global performance = **SYS + APP + FRONTEND**

These are jobs of **system administrator** and **frontend engineer**.

This presentation won't talk of those two subjects, even if they're very important.

You must be **aware of it**, even if you don't plan to make it your job.



Measure

```
$before = microtime(true);
$result  = microtime(true) - $before;

echo round($result*1000).'ms';
```

You must have a good scientific method and be precise : 50 users/ minture doesn't mean anything.

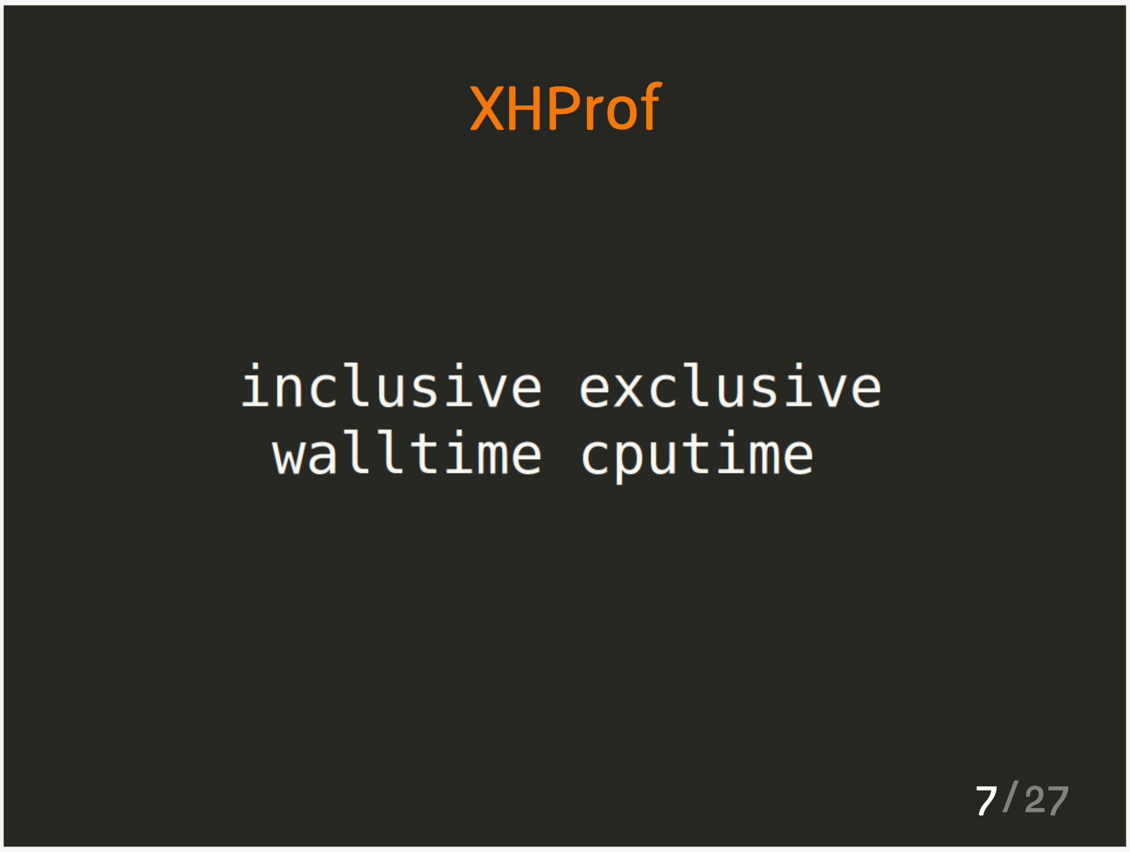
Proceed with **repeated shots, aggregate your results in a meaningful way**.

See book in references.

Be aware of your **environment** and study it. You must be able to confirm hypothesis you make based on it.

Do simple tests: die('ok');

Get out of Symfony2 to test your environment, even your application. Be aware ofthe overall context of your project.



XHProf

inclusive exclusive
walltime cputime

7/27

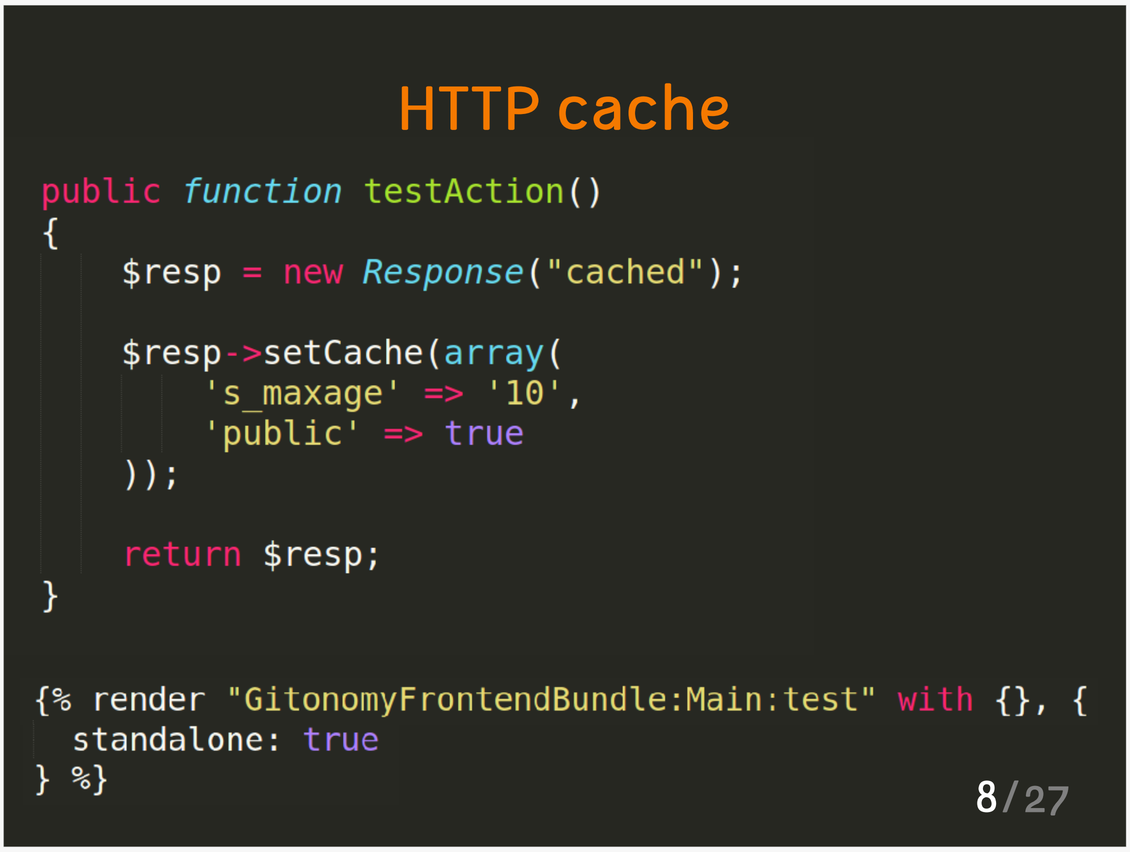
To analyze **results of XHProf**, you must know different principles of the reports: **wall time** and **processor time**.

The **processor time** is the time spent by processor making computations, without interruptions.

The **wall time** contains interruptions related to blocking states of a drive, a network or a busy processor.

The **inclusive time** is the time including the sub-tasks.

Exclusive time is the time spent without sub-tasks.



HTTP cache

```
public function testAction()
{
    $resp = new Response("cached");

    $resp->setCache(array(
        's_maxage' => '10',
        'public' => true
    ));

    return $resp;
}
```

```
{% render "GitonomyFrontendBundle:Main:test" with {}, {
  standalone: true
} %}
```

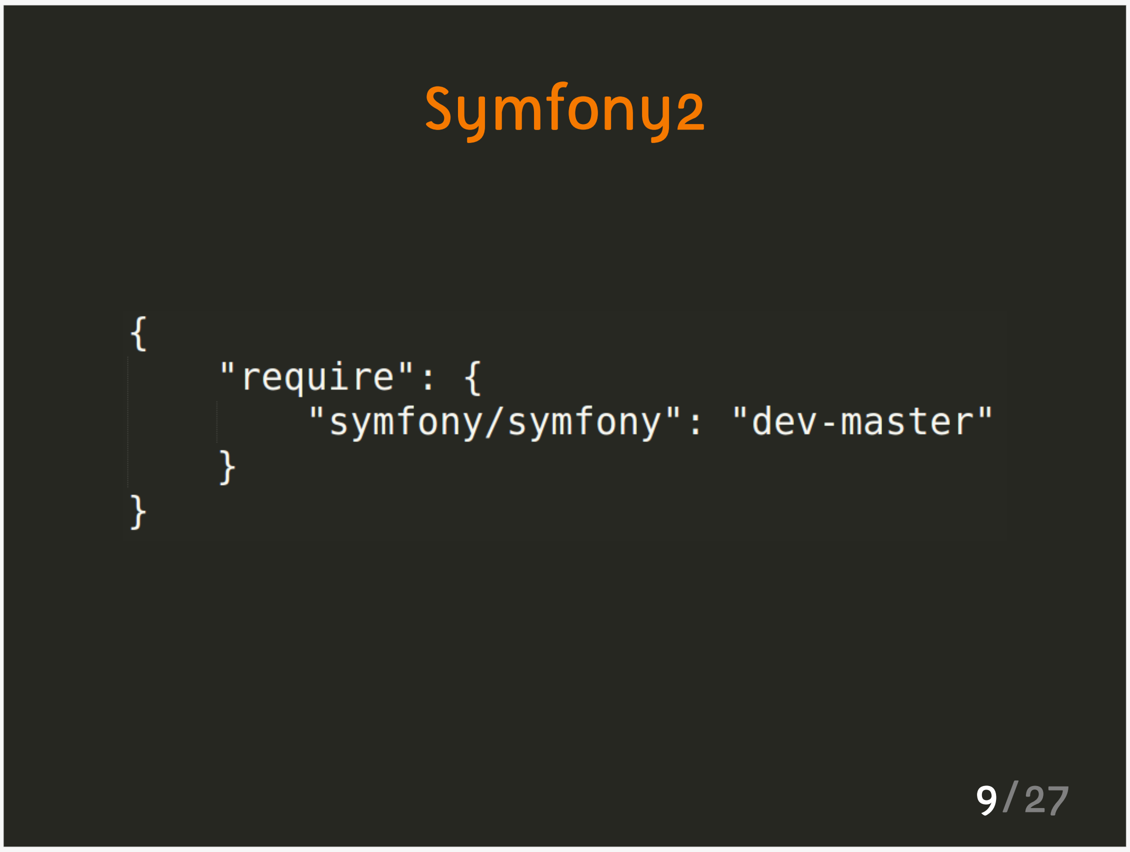
Symfony2 gives a major recommendation : The usage of HTTP protocol and his cache.

Symfony2 also uses **ESI**. This technology allows to cache small part of pages easily, quickly (*as shown on the slide*).

Usage of Varnish of even the *AppCache* allow to increase performances of the application, if it was correctly constructed in this way.

Expiration is quick and useful, whereas **validation** requires to start the application each time.

It's a **major recommendation**, you must be familiar with those methods to use them at a correct moment. **HTTP is the basis of Web, you must know it.**



Symfony2

```
{
    "require": {
        "symfony/symfony": "dev-master"
    }
}
```

9/27

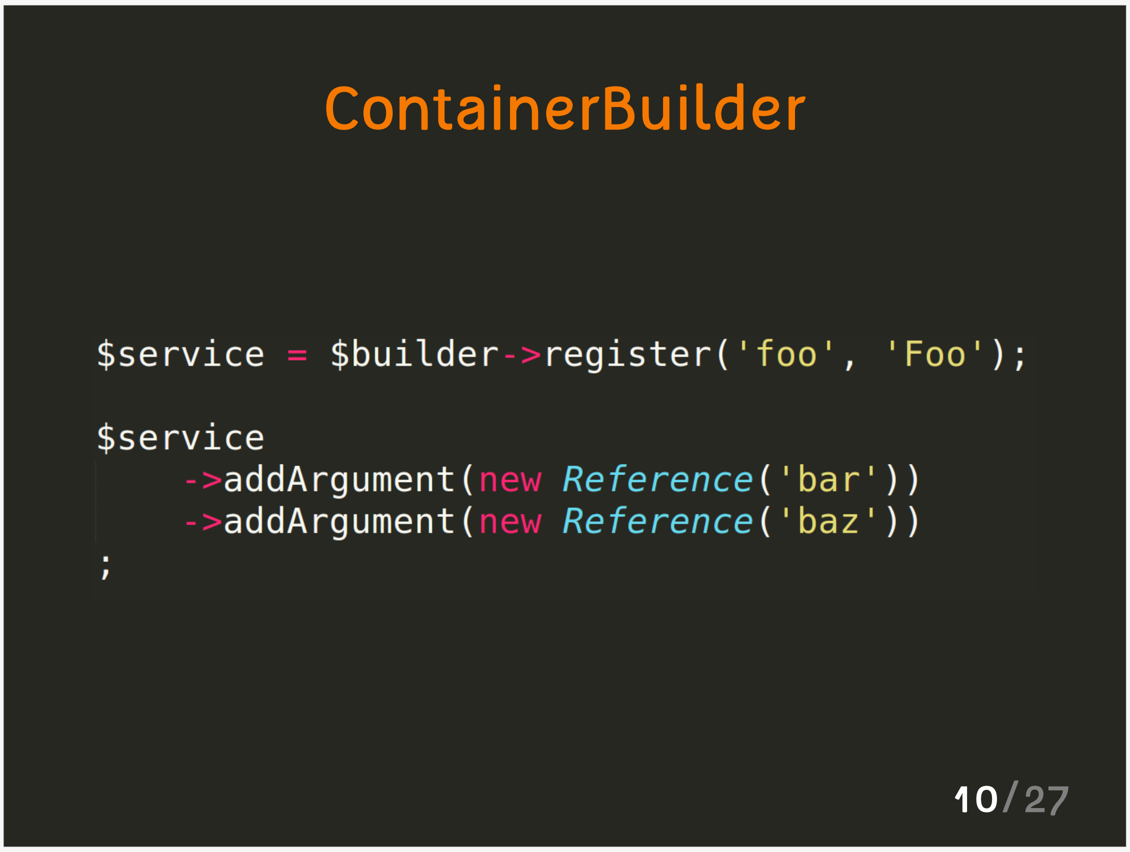
We reviews **the system** (APC, network, hard drive) and **the frontend**.

We also understood that **HTTP cache** must be known properly to be able to use it efficiently.

Let's focus now specifically on Symfony2.

2 points:
- Symfony2;
- Vendors;

Symfony2 is not most of your codebase. There are also many libraries and extensions available, as we will see.



ContainerBuilder

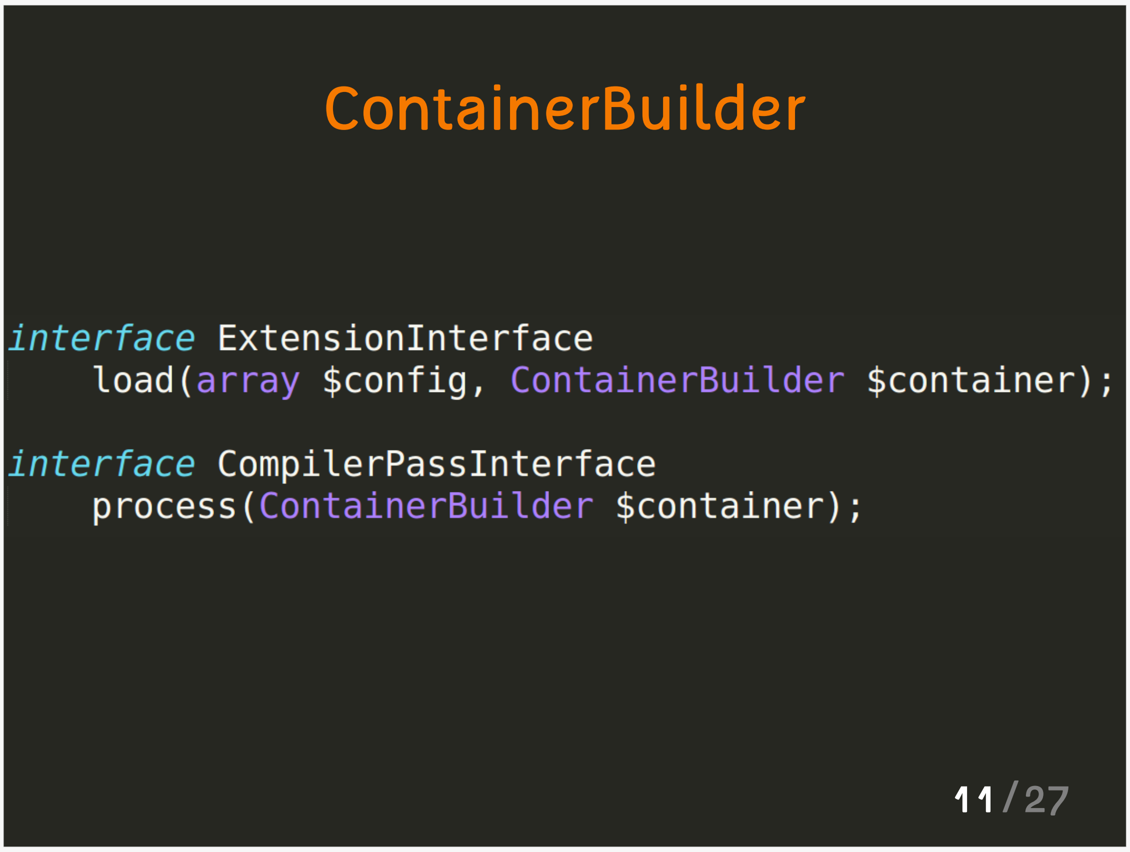
```php
$service = $builder->register('foo', 'Foo');

$service
    ->addArgument(new Reference('bar'))
    ->addArgument(new Reference('baz'))
;
```

Within Symfony2, we work around a service container. This container is constructed from configuration files.

The **construction of the container** allows up to optimize processing by easing as much as possible construction of our services.

Usage of inline-services is a good way of optimizing service container. It allows to:
- reduct number of calls to container
- reduct number of services in container
- delete unused services (and not public)

You must optimize your work on this part to improve **construction of performant services**.



ContainerBuilder

```
interface ExtensionInterface
    load(array $config, ContainerBuilder $container);

interface CompilerPassInterface
    process(ContainerBuilder $container);
```

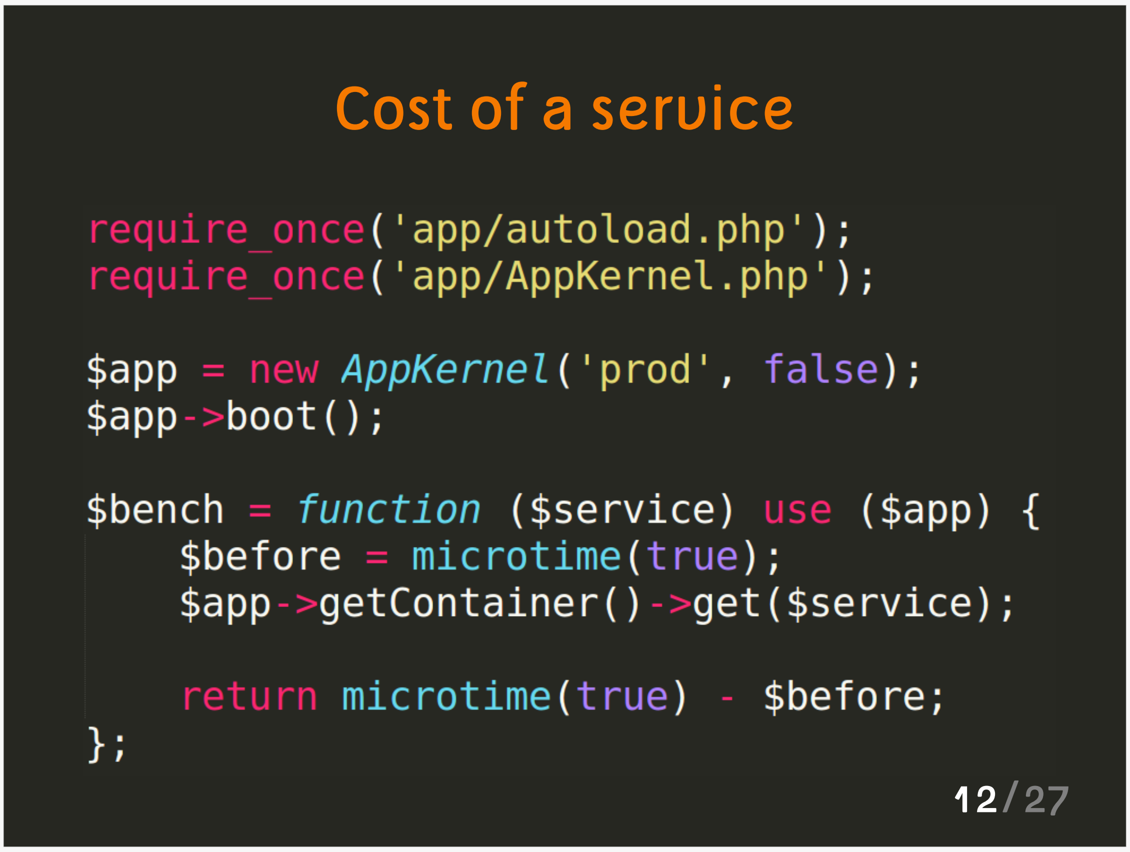
The extension above is a **service container extension**. It method load is only called when the container must be constructed (usually once in production).

You extension may be **configurable**, allowing to change the behavior of the application, depending on the environment.

All the processing you are making in those methods is never done again.

CompilerPasses will be executed after all extensions. They will allow you to aggregate by tags, and optimize service container.

ContainerBuilder interface should be familiar to you if you want to manipulate the service container.



Cost of a service

```
require_once('app/autoload.php');
require_once('app/AppKernel.php');

$app = new AppKernel('prod', false);
$app->boot();

$bench = function ($service) use ($app) {
    $before = microtime(true);
    $app->getContainer()->get($service);

    return microtime(true) - $before;
};
```

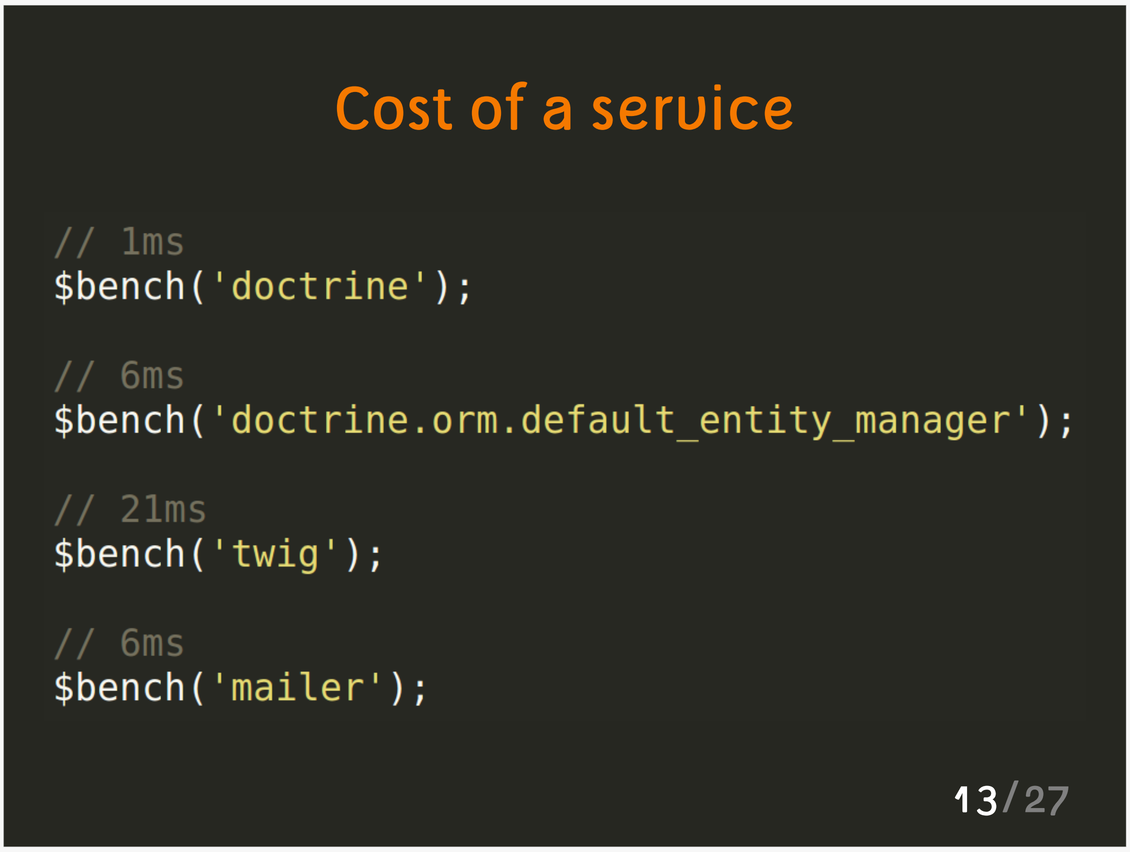
Let's make a simple exercice: measure the time a service takes to be constructed and returned.

We start container to access a given service. We will interest to **time a service takes to be constructed**.

This time is contained in:

- every call to *get('service')* in our controller
- every dependency injection we make of it

We must know also that a service construction time includes the construction time of dependencies.



Cost of a service

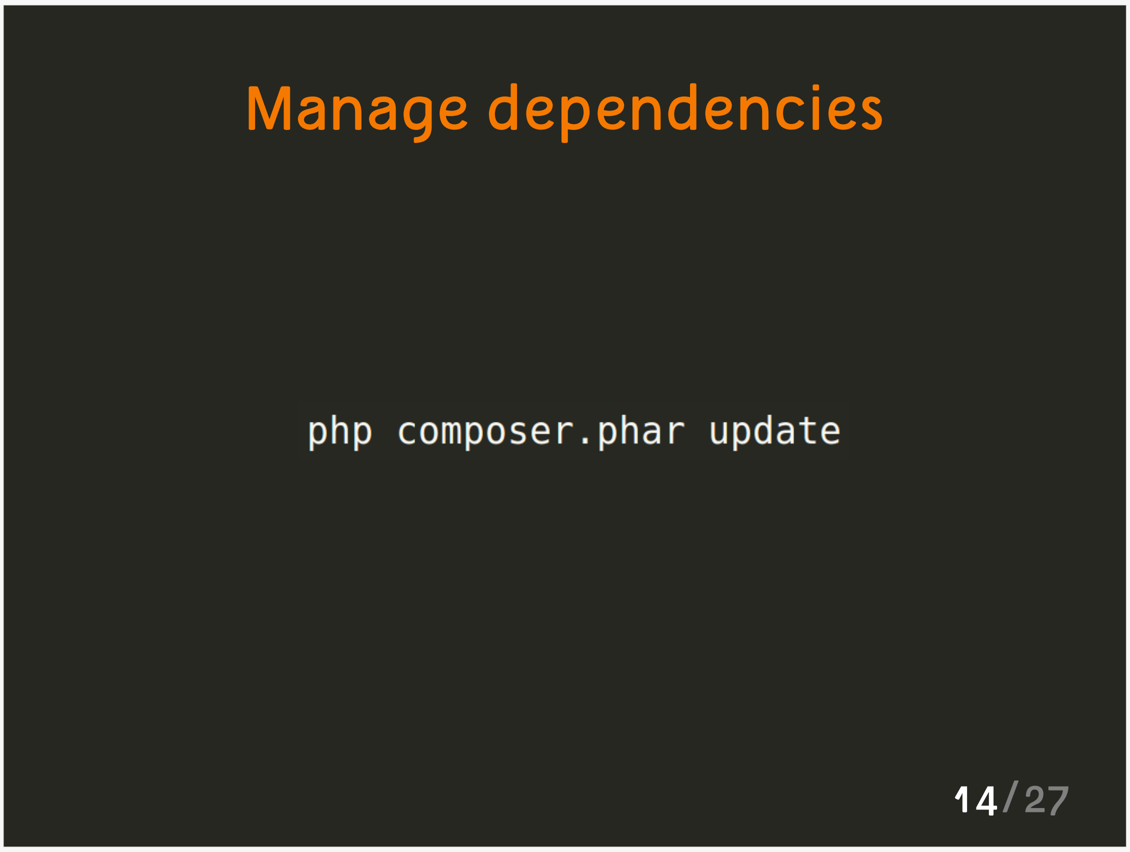
```
// 1ms
$bench('doctrine');

// 6ms
$bench('doctrine.orm.default_entity_manager');

// 21ms
$bench('twig');

// 6ms
$bench('mailer');
```

Those tests were made on a modern server, provided by Dedibox.

Here, we only instanciate a service, we didn't make usage of it yet.

The **construction time** is different from the **usage time**.

Remarque : loading time of Twig is explained by the time of dependency constructions. Twig, as a standalone component, is very performant.



Manage dependencies

```
php composer.phar update
```

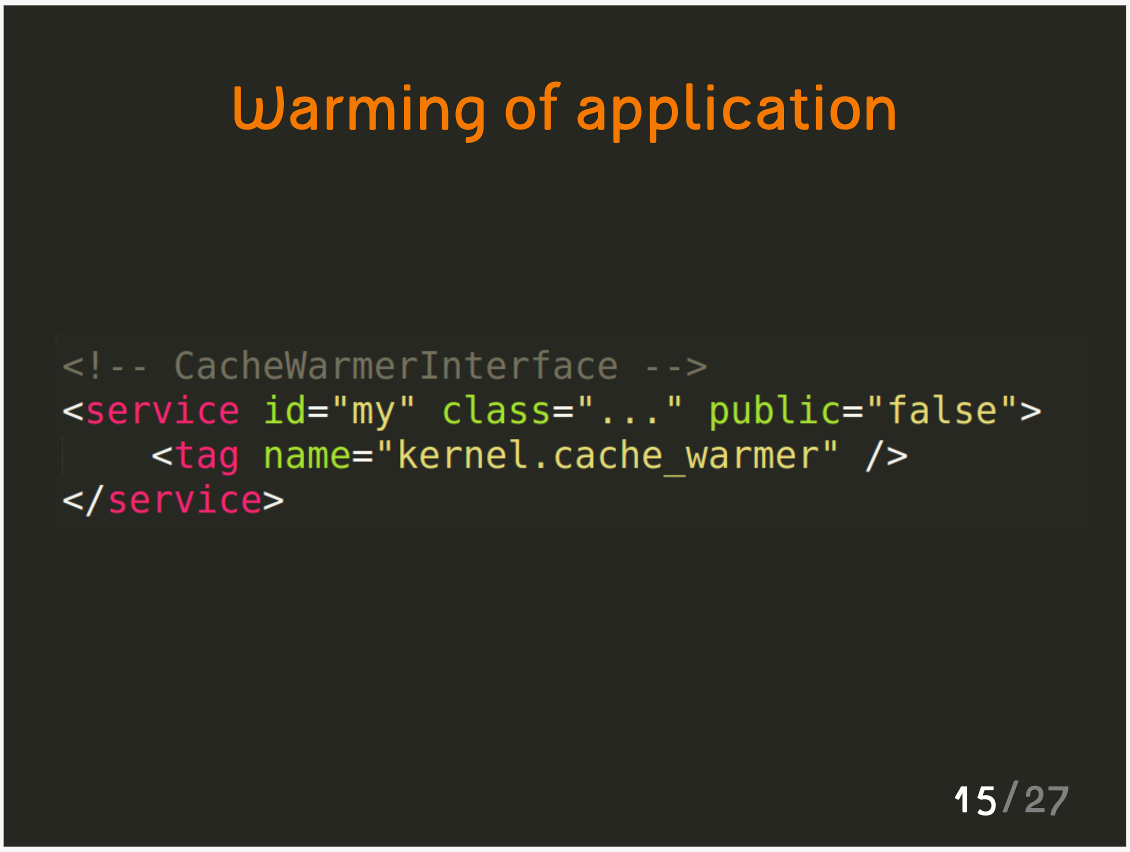
With Symfony2, **dependency** has all it meaning:
- Dependency injection with a container
- Dependency management with Composer

We have : Doctrine, Propel, Twig, Assetic, Monolog, Buzz, Bundles...

First way of improving an application performance is to **update dependencies**. Updating dependencies is **natural** for an application to stay performant.

Example : upgrading PHP from 5.3 to 5.4 makes you at least a 20% difference in performance.

If **you can't** update your dependencies, your application has a **big problem**.



Warming of application

```
<!-- CacheWarmerInterface -->
<service id="my" class="..." public="false">
    <tag name="kernel.cache_warmer" />
</service>
```

15/27

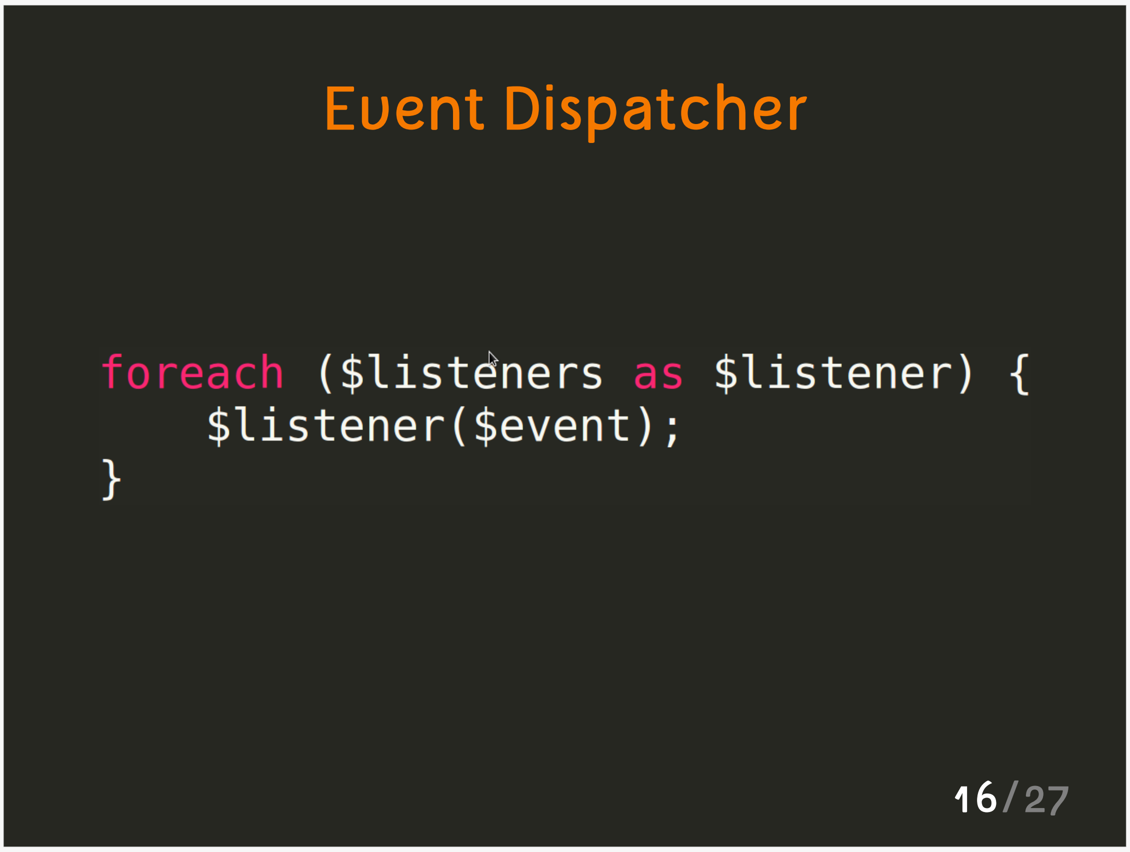
Warming of the application is used to pre-fill the cache before any request.

It's used to cache Doctrine metadatas, Twig templates, routing, and so on.

To use it, you only have to **tag a service** and honor a **CacheWarmerInterface**.

Usage
- Preloading of metadatas
- Precompilation of given elements

Limitation
- Same cache of every request



Event Dispatcher

```
foreach ($listeners as $listener) {
    $listener($event);
}
```

16/27

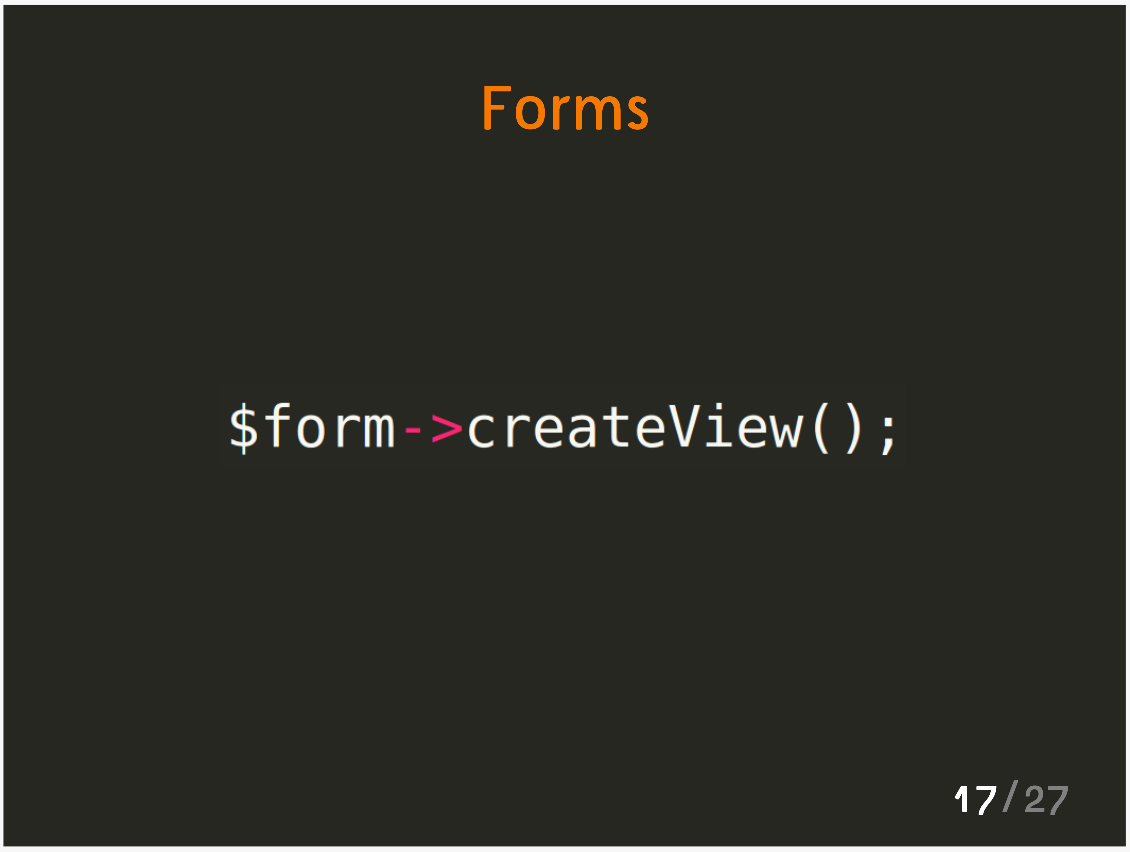
Very **simple** and performant. Overall code is small

In the full-stack framework, this component is used as a POPO.

Be careful about number of listeners you put on kernel.request, kernel.response and kernel.controller.

In a listener, return as much as possible to avoid computations. Be aware of the **criticity of process** when you write the code.

Dependencies of this listener will natureally be constructed with object and raise the cost of those listeners.



Forms

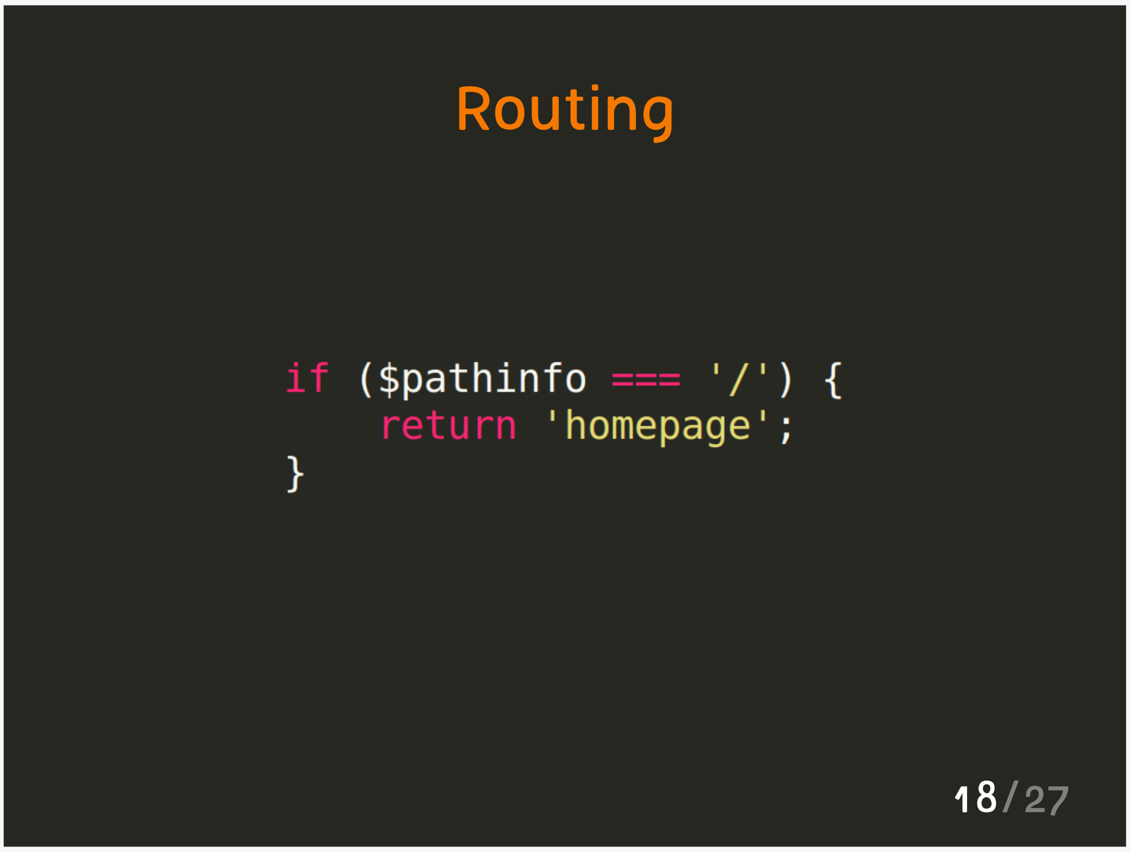
```
$form->createView();
```

17/27

Performance of form component in **2.1 is much more higher than 2.0** : http://symfony.com/blog/form-goodness-in-symfony-2-1

The first method of performance optimization in your application is to **declare your form types as services in your container**. This allow to component to reuse your *Type* objects accross forms.

I recommend you supports from Symfony Live San Francisco 2012 de Bernhard Schussek on forms, they are perfect.



Routing

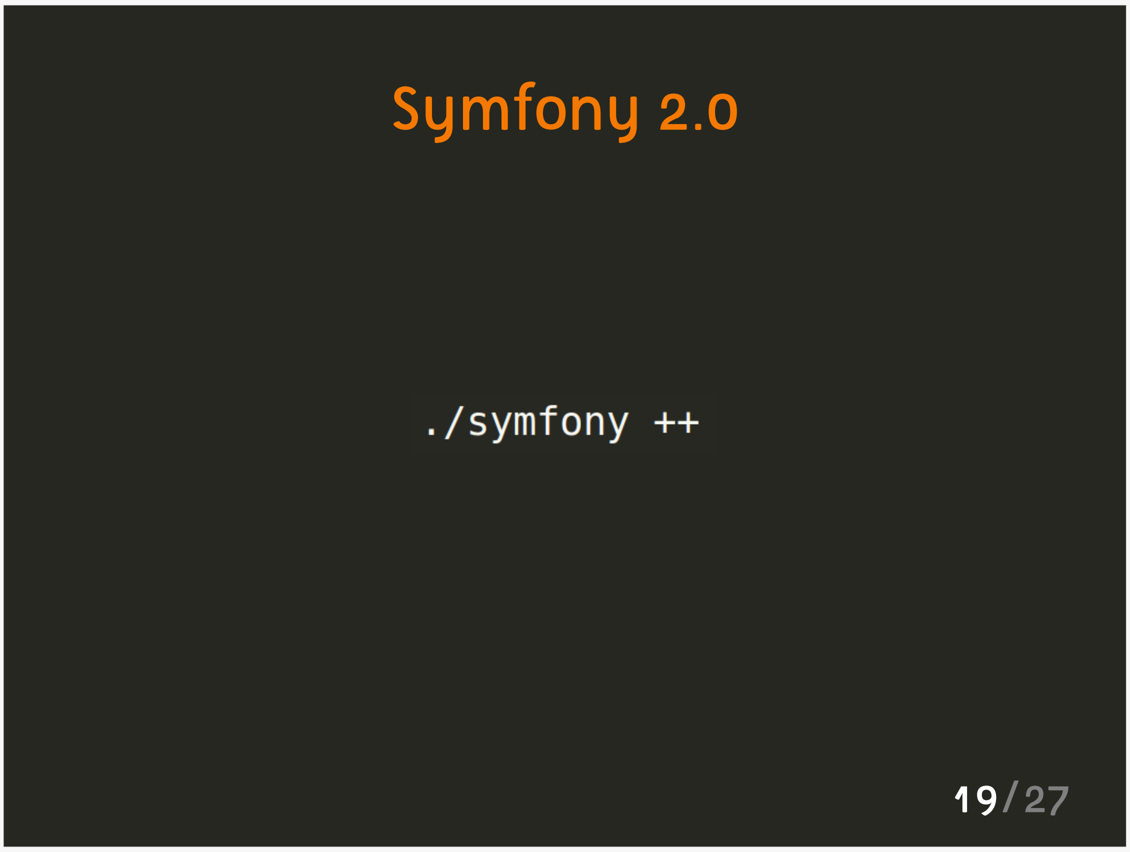
```
if ($pathinfo === '/') {
    return 'homepage';
}
```

The **first matched route** will be used. So if your home page is the first route of your routing, she well be the first tested. **Order of definition** is important for the performance of it.

This definition only has one starting point: *app/config/routing(_dev)?.yml*

Cache is composed of two elements: *UrlMatcher* and *UrlGenerator*. One can be loaded without the other → we don't load informations to generate a URL when we are doing routing.

It's possible to export this cache to Apache, to optimize even more routing time.



Symfony 2.0

```
./symfony ++
```

19/27

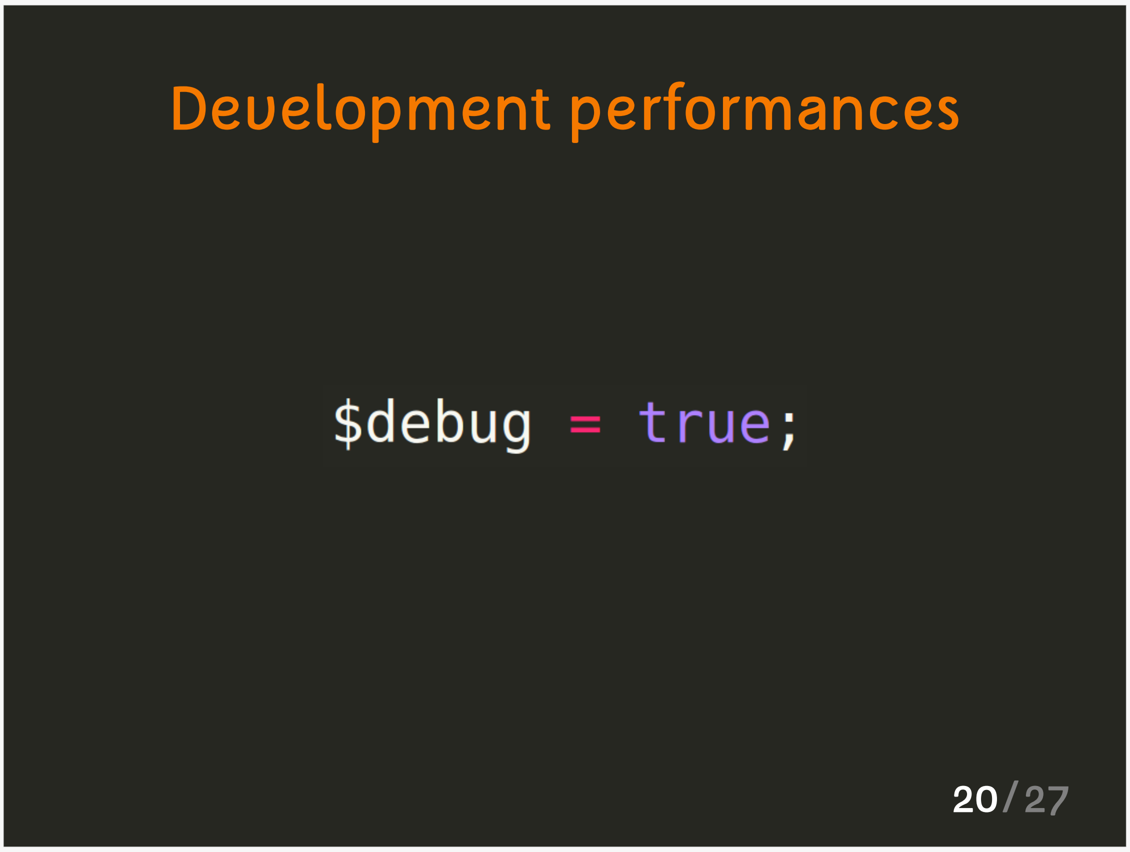
Some verifications are usual with Symfony 2.0:

Autoloading cache is now transparent, thanks to composer. With 2.0, it's another story. Check your files!

- APC cache for **Doctrine**
- Update dependencies (deps, deps.lock)

It's stronly recommeded to switch to Symfony 2.1 to take benefit from last progress on project.

Migration from 2.0 to 2.1 is not complicated, it is mechanic. The file UGRADE-2.1.md gives you a list of all changes to operate.



Development performances

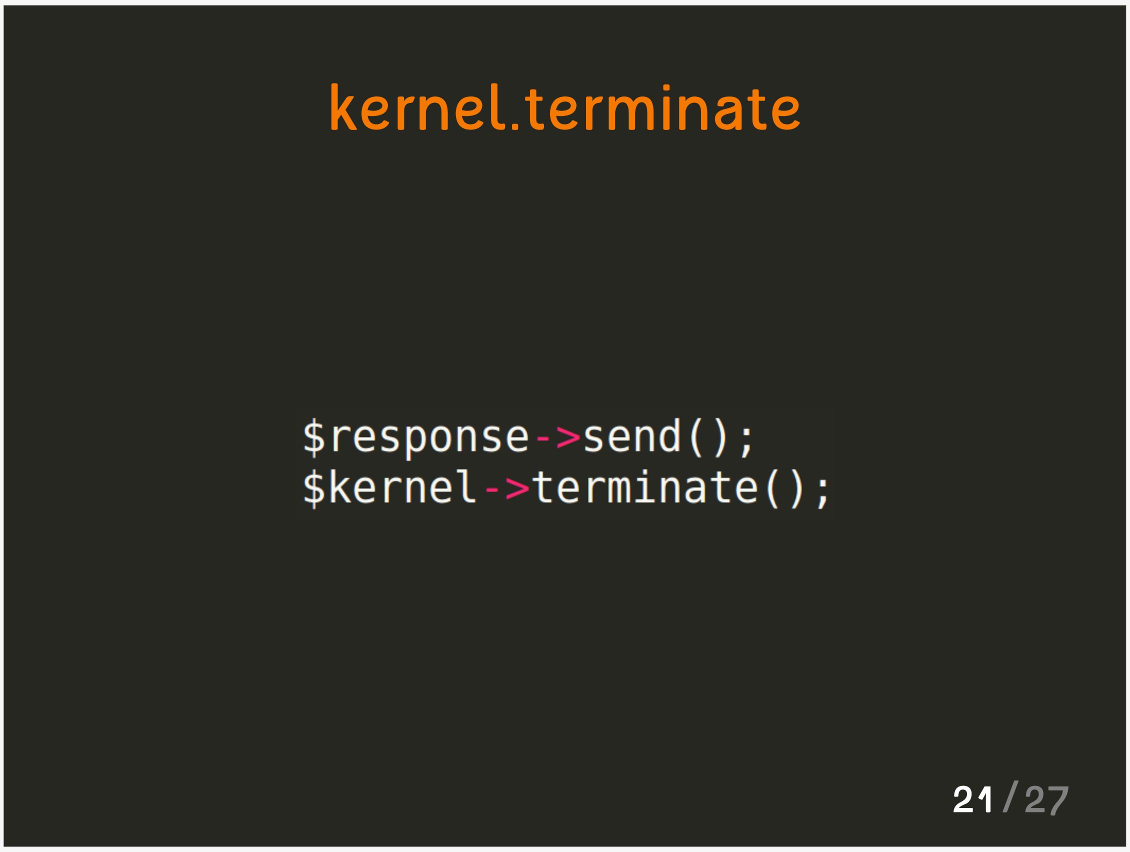
```
$debug = true;
```

Performance in development is important, as much as performance in production. Be sure to have an **efficient** development environment.

Counter-performant : Samba mouting, APC missing, remote servers. All time you gain is development needs to be taken.

Be careful to some heavy bundles, making development process harder. DiExtraBundle is a sample of bundle you should not use : those bundles store an enormous amount of data in your profiler. Extraction and storage of it on each request is costful while developing.

Never say ”page takes 3 seconds to load, it's OK we are in development environment”



kernel.terminate

```
$response->send();
$kernel->terminate();
```

The event **kernel.terminate** is launched in event dispatcher after the response is sent to the client.

This event may be used, for example, to send a mail.

This allows to quickly send response to the client and process some other things after that.

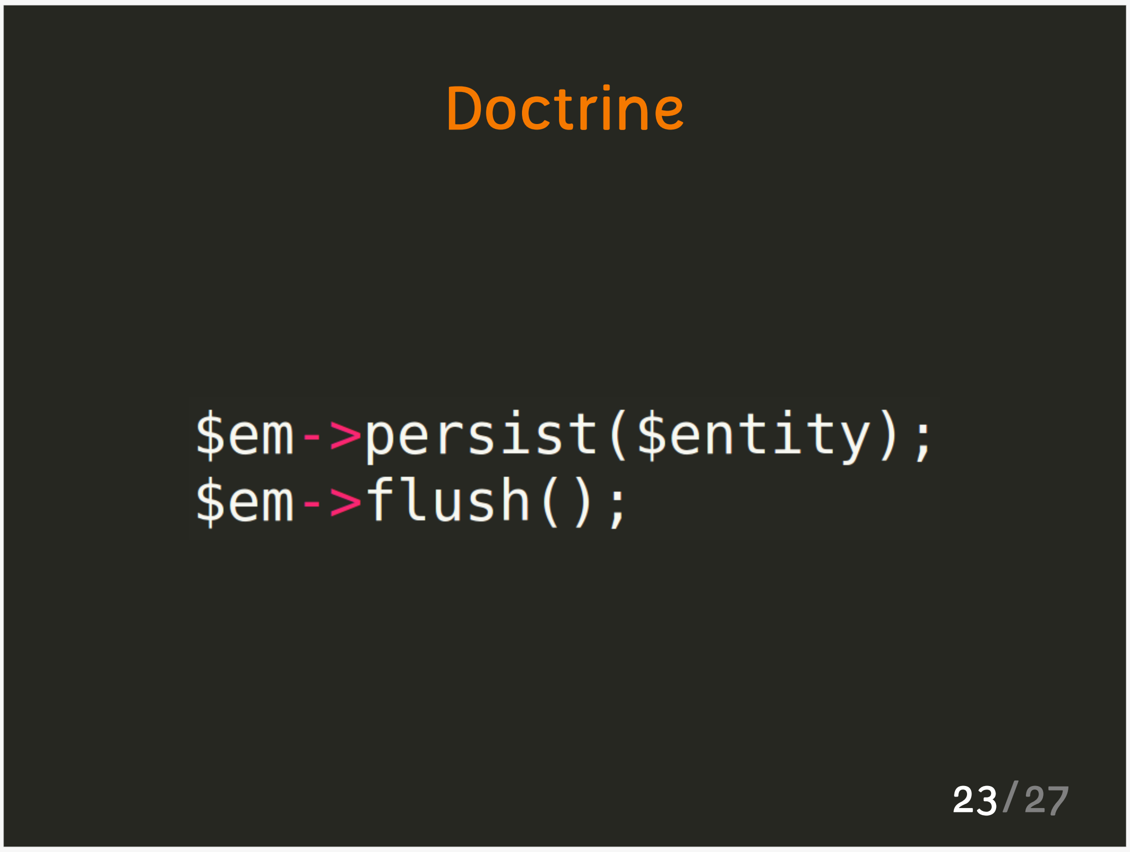
It works with **PHP-FPM.**

Limitations :
- impossible to modify the response
- needs an appropriate PHP installation

Otherwise, kernel.termine is executed before sending the response

Vendor

Now, let's see how to enhance performance of our **dependencies**.



Doctrine

```php
$em->persist($entity);
$em->flush();
```

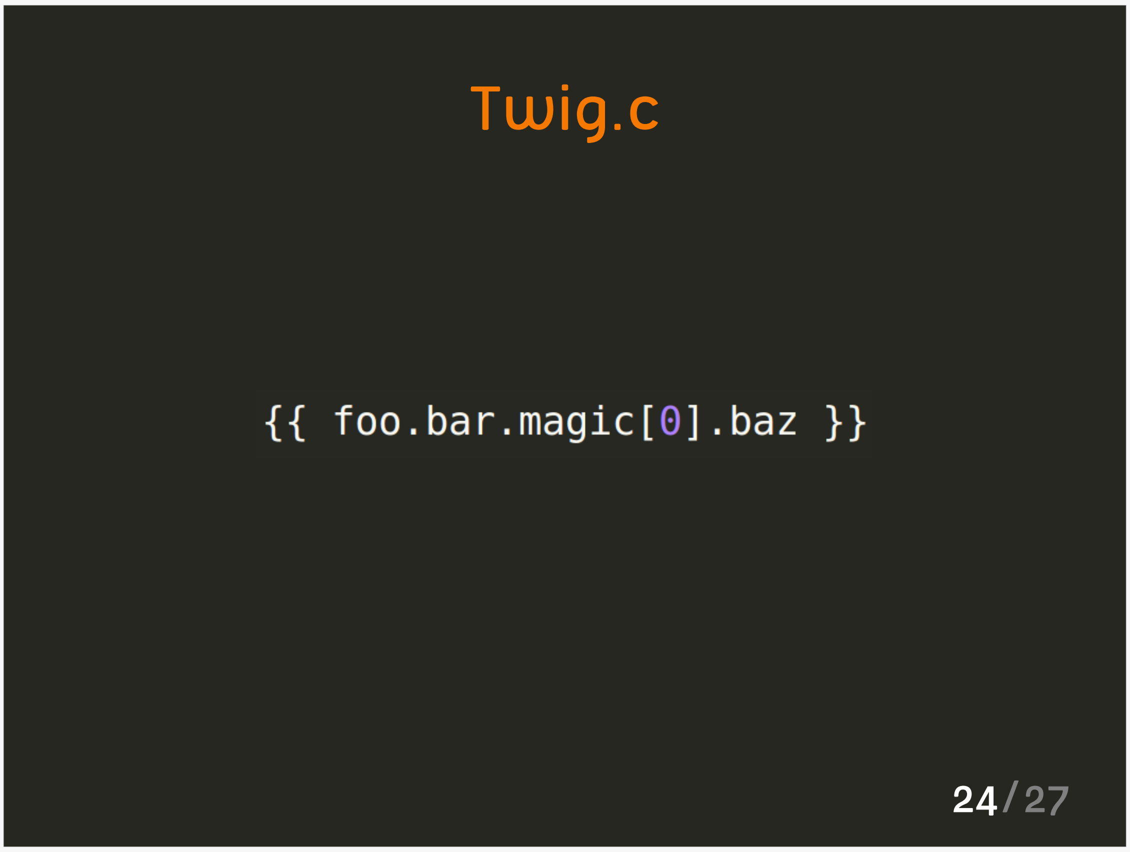
The most spread dependency with Symfony2 is surely Doctrine.

Be careful about **persist/flush** ratio. Call to a flush is more expensive than a call to persist.

Watch your **transactional calls** and know about them.

Each transaction of your application is stored in a **UnitOfWork**. This UnitOfWork stores different SQL orders to execute and will execute them in a transaction.

You must finally be careful about your metadata cache. If your application needs it, request cache and/or results cache.

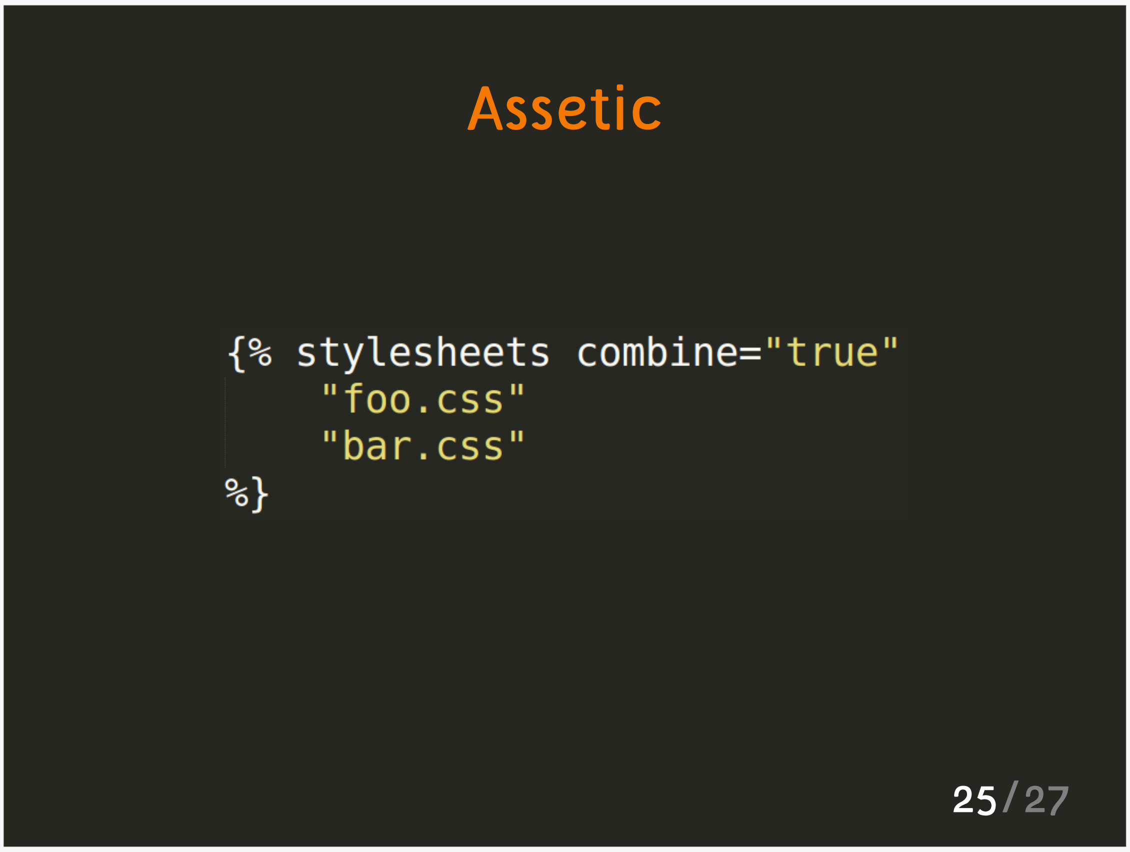


Twig.c

```
{{ foo.bar.magic[0].baz }}
```

Twig has a C-module, allowing to optimize the rendering time of Twig templates.

Resolution of attributes in an expression is costful in Twig. A C extension was so created for this part.



Assetic

```
{% stylesheets combine="true"
    "foo.css"
    "bar.css"
%}
```

Assetic is a real step toward proper stylesheets and assets management. His model is so that the result is the best: we serve static pre-compiled files.

Unfortunately, misconfigured, Assetic becomes counter-performant in development environment.

A technic for it is to use combine=”true” in the definition.

If loading time is really too long, use the **watch mode** instead of controllers method.

References

- Livres
  - Release It
  - How to measure anything
- Articles
  - Pingdom

Release It! : Design and Deploy Production-Ready Software
- Michael T. Nygard

How to Measure Anything: Finding the Value of Intangibles in Business
- Douglas W. Hubbard

Pingdom: http://pingdom.com

The end