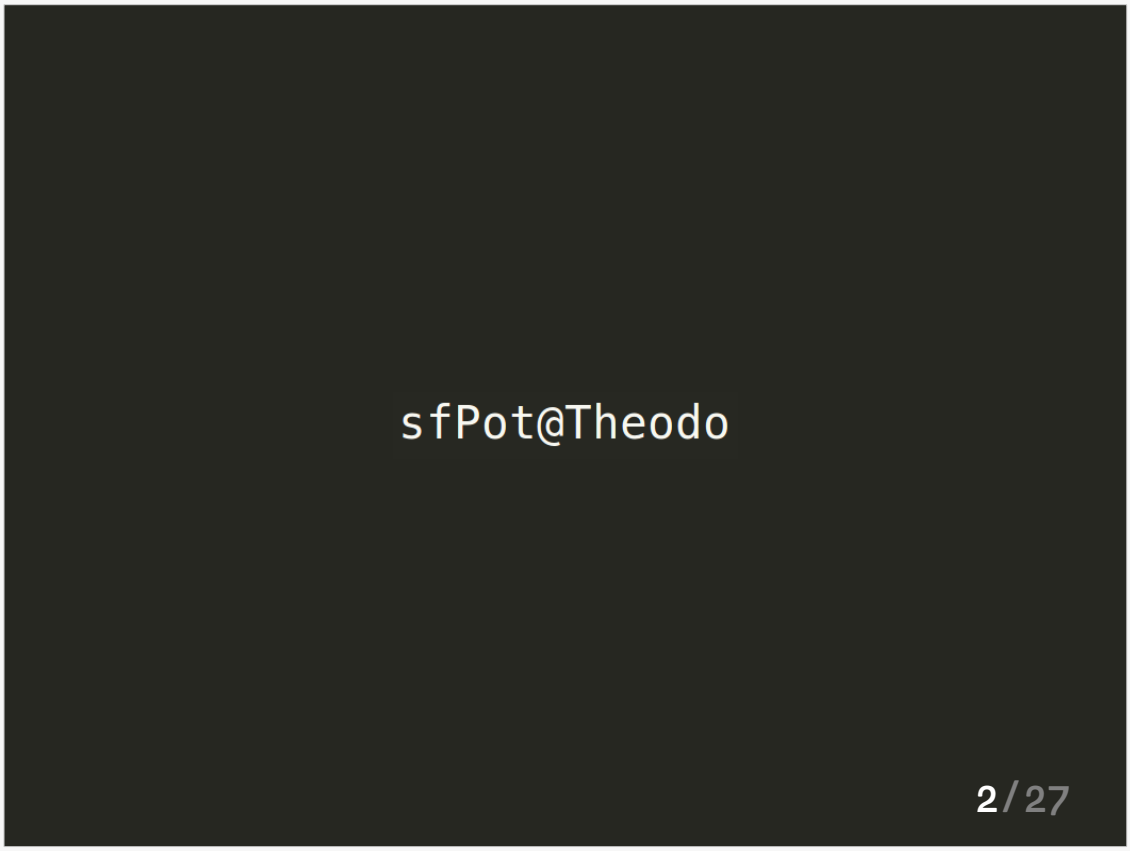
Alexandre Salomé - Le 8 octobre 2012

Je travaille pour **Sensio Labs** depuis maintenant plus de 3 ans.

Depuis 2 ans, j'occupe le poste de c**onsultant**, poste qui me permet de prendre part à un nombre importants de projets, allant de la petite équipe à la multi-nationale.

C'est donc dans un contexte riche que j'utilise **Symfony2** comme framework pour la plupart de mes développements.

Fort de ces expériences, j'essaie aujourd'hui de vous montrer comment au mieux exploiter Symfony2 pour atteindre une meilleure performance.



Je remercie **Theodo** pour l'hébergement.

Différentes sociétés hébergent à tour de rôle ces événements libres, les sfPot. Une présentation et un verre après, c'est une formule sympa !

Pour gérer tout ce mouvement, une association : l'**Afsy**. L'association s'assure de l'organisation de l'événement, du site afsy.fr.

Qu'est-ce que la performance ?

Définition :
Résultat optimal que peut obtenir un **sujet**.

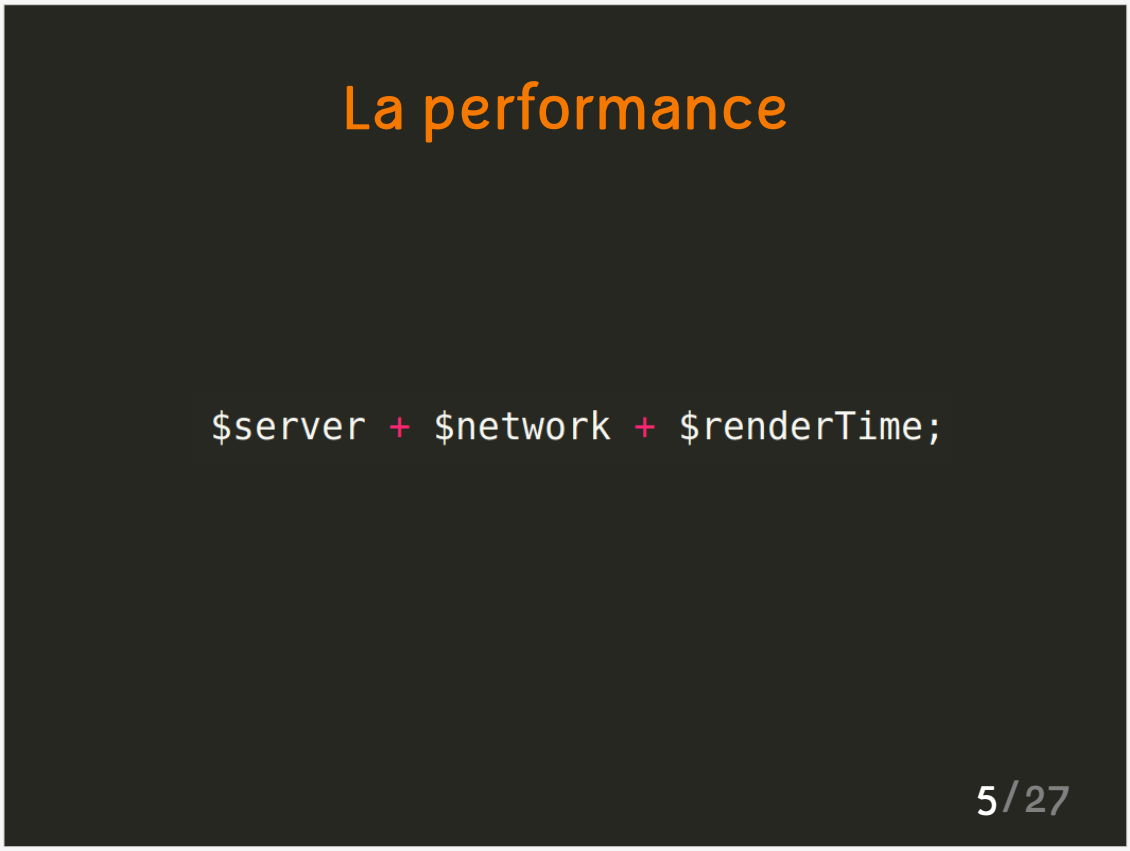
Exemples : la performance d'un athlète, la performance d'une poule, la performance d'un placement boursier ou encore la performance d'un artiste.

On s'intéresse aux **facteurs** sur lesquels on peut influencer.

Exemples : le régime de l'athlète, les conditions d'élevage de la poule.

D'autres par contre ne dépendent pas de nous et sont les contraintes de l'**environnement**.

Exemples : la gravité pour l'athlète, l'actualité pour un placement boursier.



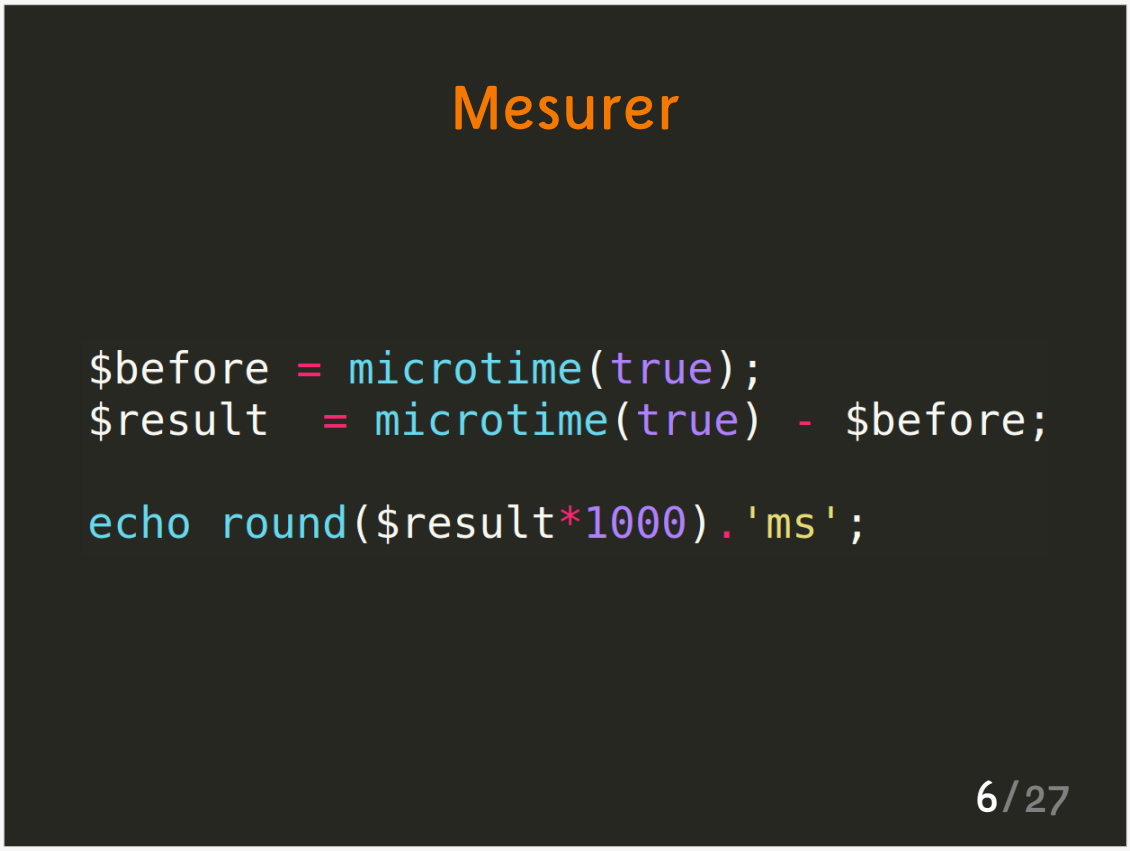
La performance est **la somme d'un tout** :
- ressources serveur (disque, mémoire, réseau);
- installation de la plateforme (APC, version PHP);
- application;
- latence réseau;
- téléchargement et rendu côté client;

Performance globale = **SYS + APP + FRONTEND**

Ce sont les métiers d'**administrateur système** et le métier d'**ingénieur frontend.**

Cette présentation n'abordera pas ces deux sujets très importants.

Vous devez être **sensible à ces sujets**, même si vous n'en faites pas votre métier.



Ayez une bonne démarche scientifique et soyez précis : 50 utilisateurs / minute ne veut rien dire.

Faites des **tirs répétés**, **aggrégez les résultats de manière pertinente** (distribution/moyenne/écart type/variance).

Voir bouquin en référence

Prenez en compte **l'environnement**, étudiez-le et confirmez les hypothèses ou au contraire, infirmez les.

Faites des tests simples: die('ok');

Sortez de Symfony2 pour tester votre environnement, votre application, soyez conscient du contexte global de votre projet.

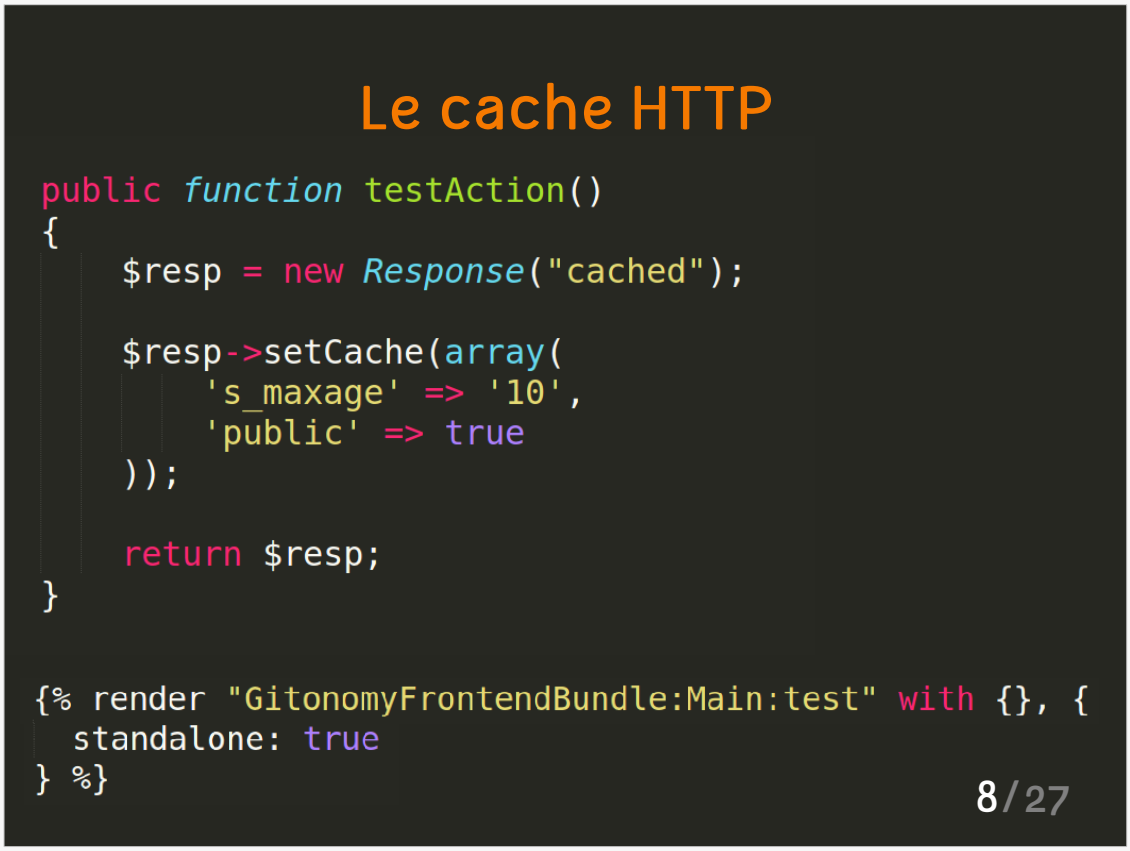
L'analyse des **résultats de XHProf** suppose une connaissance de ses différents principes : **temps ressenti** et **temps processeur**.

Le **temps processeur** est le temps passé par le processeur à faire des calculs, sans interruption.

Le **temps ressenti** *(wall-time)* comprend les interruptions liées à des états bloquants : lecture disque, lecture réseau, processeur occupé.

Le **temps inclusif** correspond au temps incluant le temps des sous-tâches.

Le **temps exclusif** correspond au temps passé sans les sous-tâches.



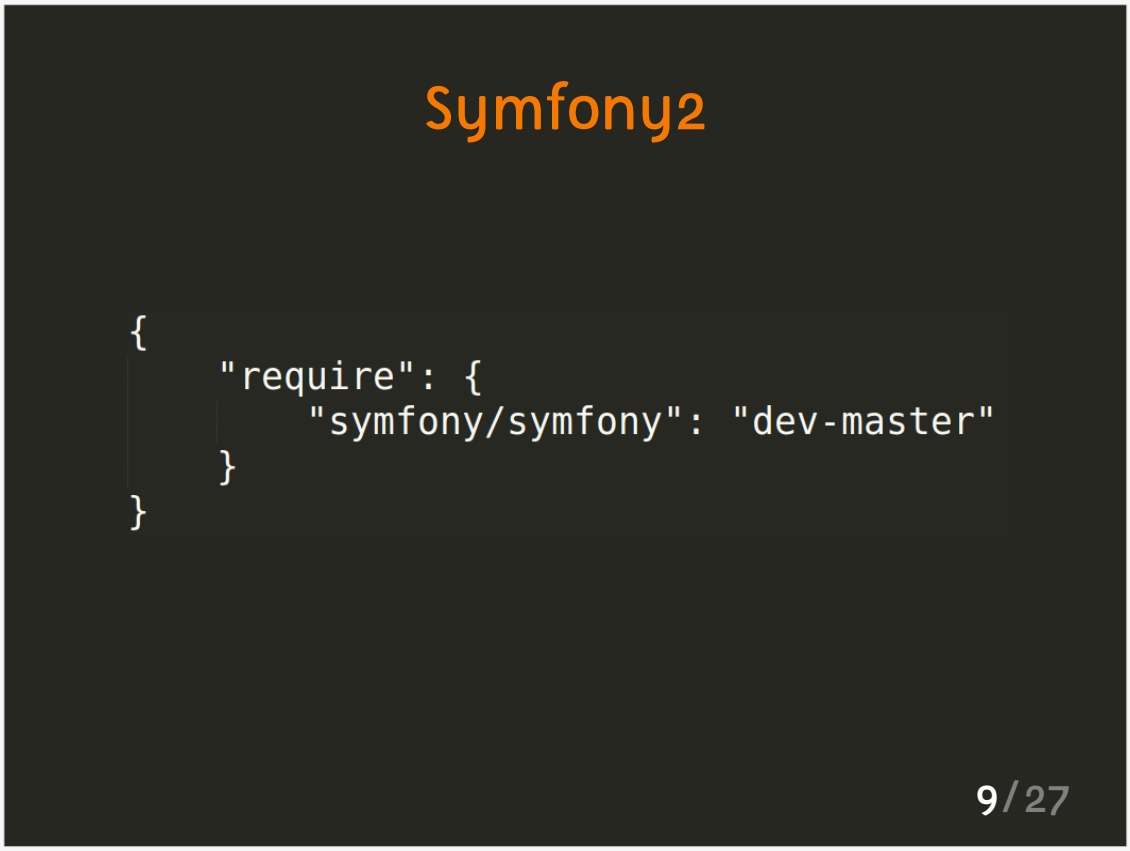
Symfony2 donne une recommandation majeure : L'utilisation du protocole HTTP et de son cache.

Symfony2 utilise également **l'ESI**. Cette technologie permet de mettre en cache des fragments de page facilement, rapidement (*cf exemple sur le slide*).

L'utilisation d'un Varnish ou même du *AppCache* permet d'augmenter les performances d'une application, si elle a été conçue avec un cache HTTP.

L'**expiration** est rapide et pratique, la **validation** suppose que chaque requête démarre l'application.

C'est une **recommandation majeure**, vous devez connaître cette méthode pour la mettre en oeuvre au moment opportun. Le HTTP est la base du Web, **vous devez le maîtriser**.



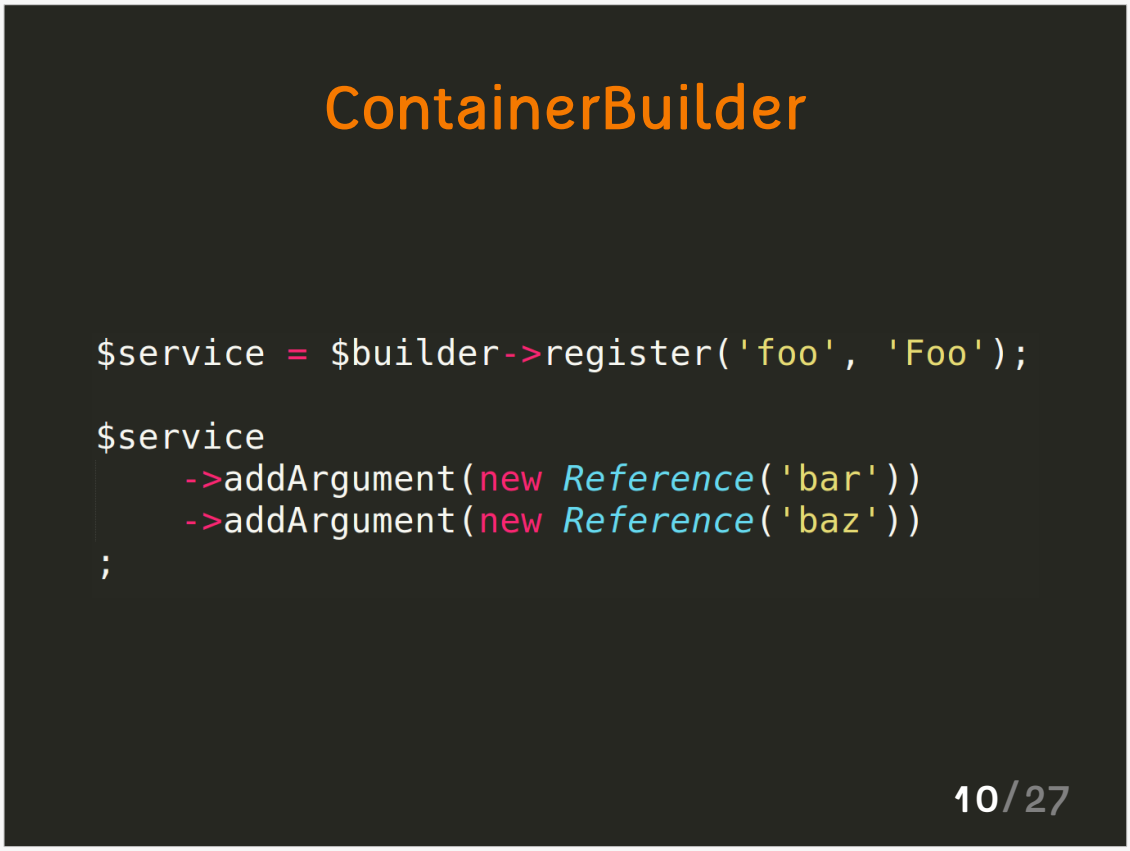
Nous avons écarté le domaine du **système** (APC, réseau, disque), le domaine du **front-end** (Javascript, CSS).

Nous avons également vu que le **cache HTTP** doit donc être maîtrisé pour pouvoir être utilisé efficacement.

Concentrons nous maintenant sur Symfony2.

Nous traiterons deux points :
- Symfony2;
- Vendors;

Symfony2 n'occupe pas la majorité de notre base de code, elle s'appuie également beaucoup sur des services tiers.



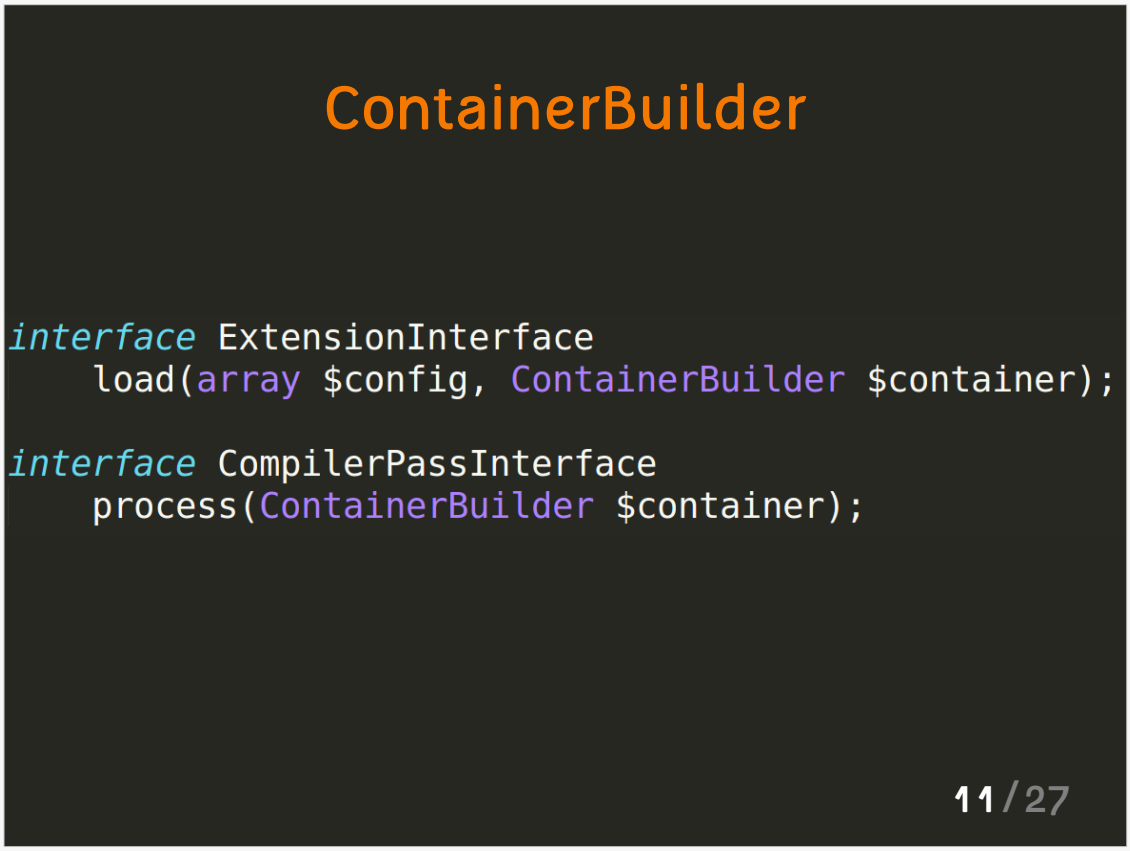
Dans Symfony2, on travaille autour d'un conteneur de services. Ce conteneur est construit à partir d'un fichier de configuration.

La phase de **construction du conteneur** va nous permettre d'optimiser les traitements en simplifiant le plus possible la construction de nos services.

L'utilisation de services *inline* (en-ligne) est une bonne méthode pour optimiser le conteneur de service. Elle permet de !

- réduire le nombre d'appels au conteneur;
- réduire le nombre de services du conteneur;
- supprimer les services non-utilisés (via public=”false”)

Vous devez optimiser cette phase pour avoir une **construction de service performante**.



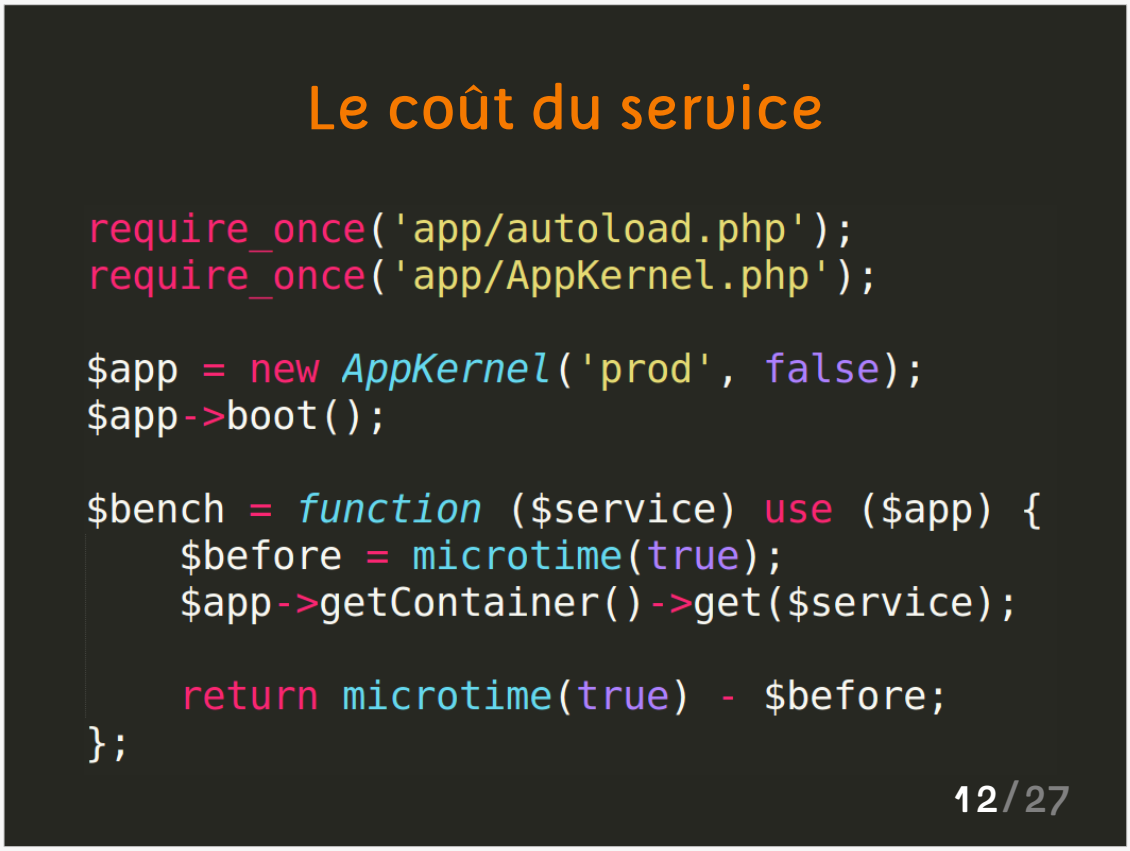
L'extension est une **extension de conteneur de services**. Sa méthode load n'est appelée qu'au moment où le conteneur doit être reconstruit.

Votre extension peut être **configurable** permettant notamment de configurer différemment l'extension, selon l'environnement, par exemple..

Tout le traitement fait dans ces méthode n'est plus à faire.

Les **CompilerPass,** elles, vont être éxécutées après toutes les extensions. Elles vont permettre d'aggréger par tag, et d'optimiser le conteneur de services.

L'interface de **ContainerBuilder** doit vous être familière si vous voulez manipuler le conteneur.



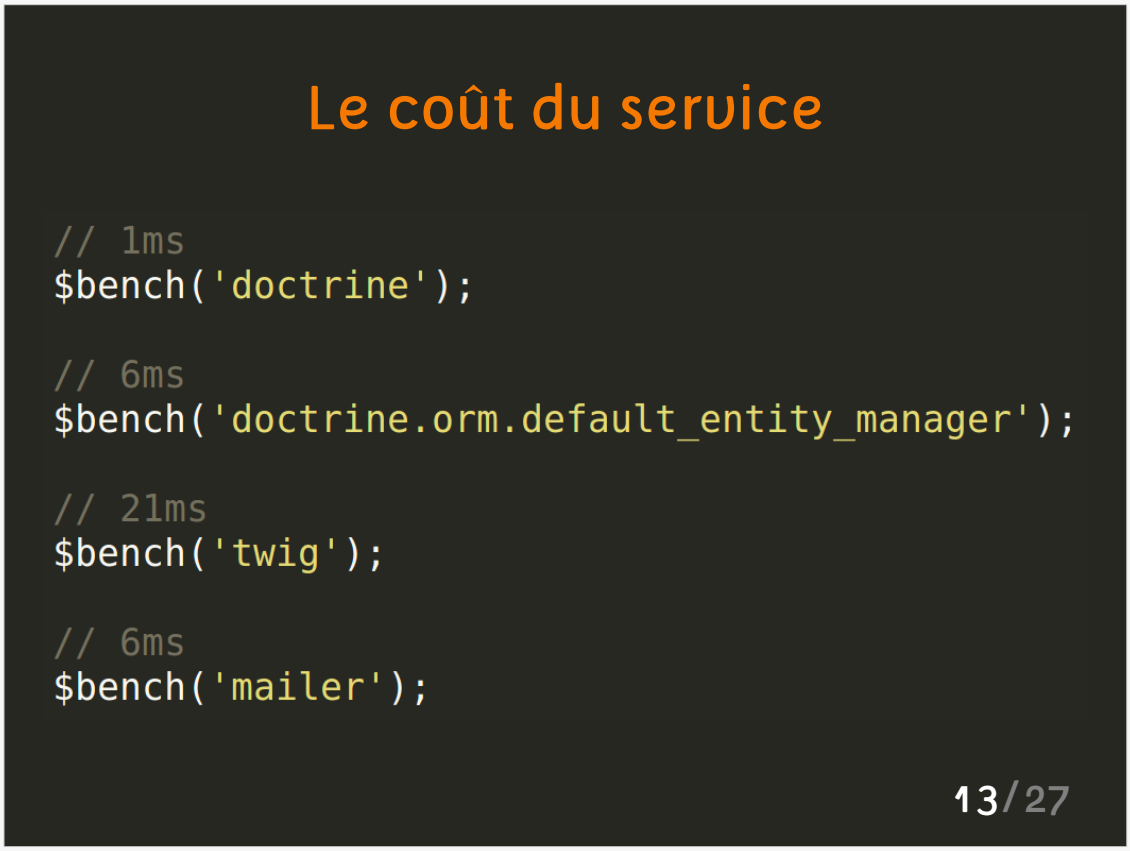
Faisons un exercice simple : mesurons le temps que met un service donné à se construire.

On ne fait que démarrer le conteneur de services pour accéder à un service donné. On s'intéresse au **temps que met ce service à être construit**.

Précisons avant cela où ce coût sera ressenti :

- Lorsqu'on fera un appel à *get('service')* dans notre contrôleur
- Lorsqu'on injectera cette dépendance dans un service.

Il faut bien comprendre également que le temps de construction d'un service inclus également le temps de construction de ses dépendances.



Ces tests ont été fait sur une Dédibox. Ils mettent en avant une chose : la construction des services peut être coûteuse.

De plus on n'a fait qu'instancier le service, on ne l'a pas encore utilisé.

Le **temps de construction** est différent du **temps de traitement**, d'utilisation.

Remarque : le temps de chargement de Twig est dû à ses dépendances. Seul, Twig est beaucoup plus rapide.

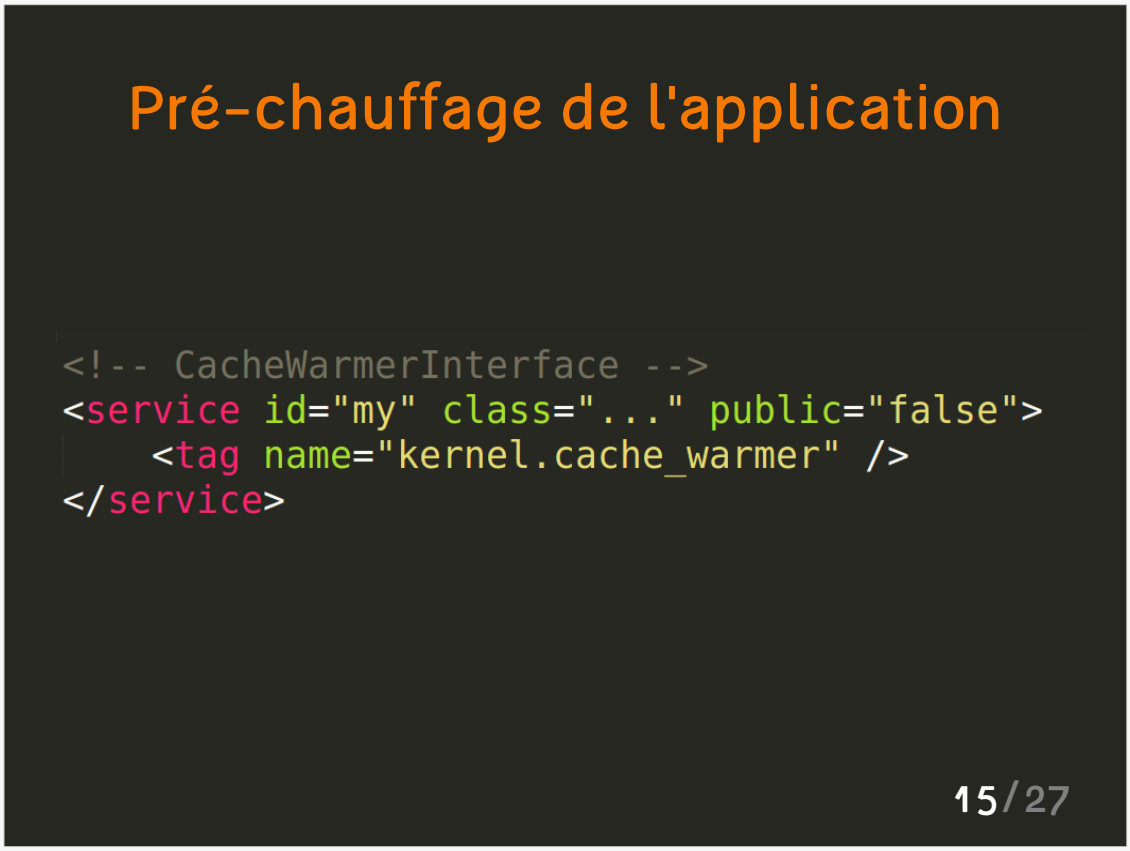
Avec Symfony2, le terme **dépendance** prend tout son sens :
- Injection de dépendance
- Gestionnaire de dépendances

On retrouve en vrac : Doctrine, Propel, Twig, Assetic, Monolog, Buzz, Bundles (Sensio, Knp, JMS)

La première manière d'améliorer la performance d'une application est de **mettre à jour ses dépendances**. Mettre à jour est **naturel** pour permettre à une application d'être ou de rester performante.

Un exemple : la mise à jour de PHP 5.3 en PHP 5.4 vous fait gagner au moins 20% de performance !

Si **vous ne pouvez pas** mettre vos dépendances à jour, votre application est **en danger**.



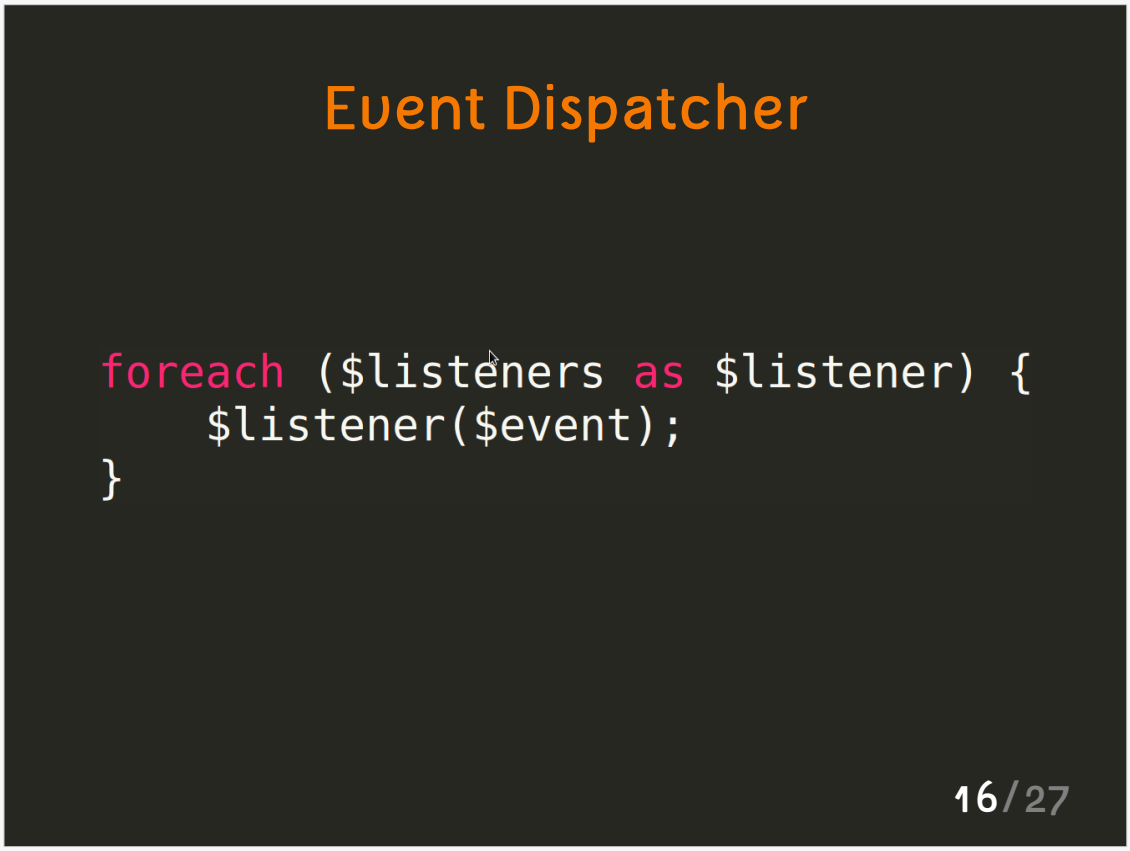
Le **pré-chauffage** est utilisé pour pré-remplir le cache avant la première requête.

Il est utilisé notamment pour mettre en cache les méta-données Doctrine, les templates Twig, le routing, etc.

Pour cela, il n'y a qu'un **service à tagguer**.

Utilisation
- Chargement des méta-données
- Précompilation de certains éléments

Limitation
- Même cache pour toutes les requêtes



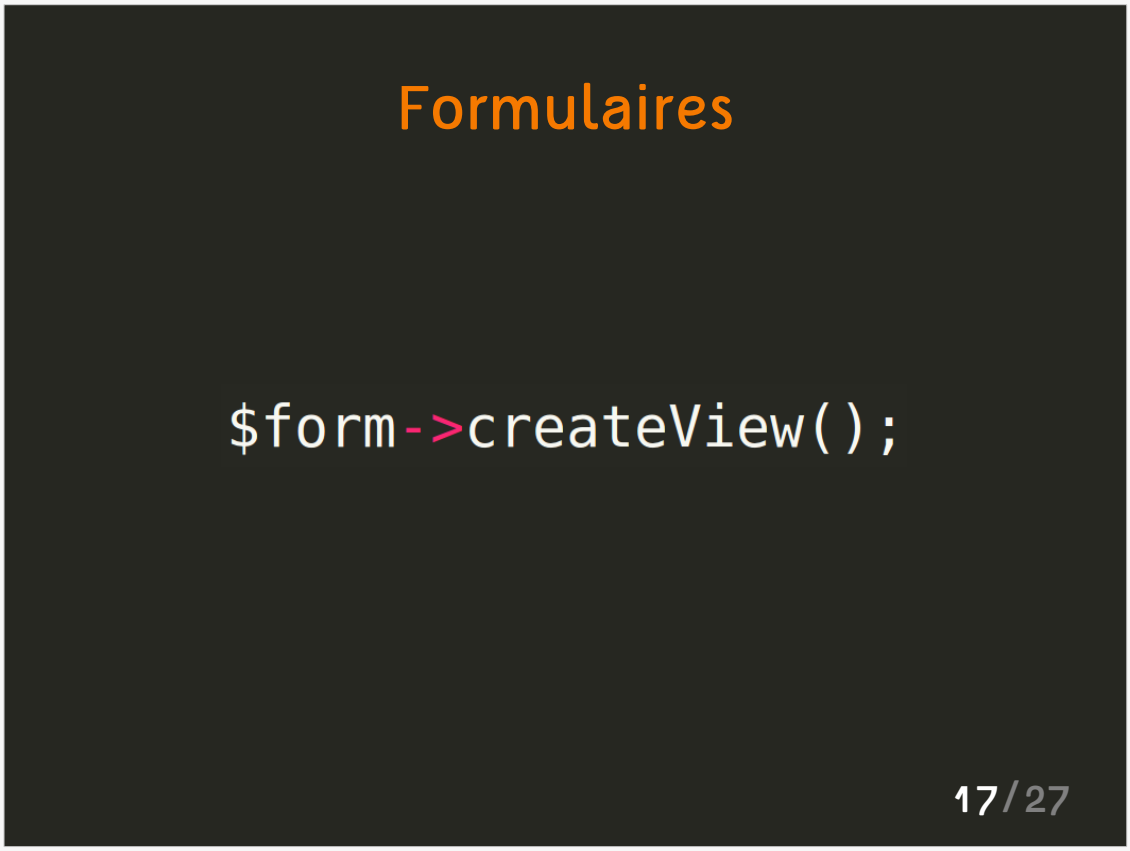
Suffisamment **simple** pour être performant

Dans le framework full-stack, ce composant est utilisé comme un POPO.

Attention à la multiplication des listeners sur kernel.request, kernel.response, kernel.controller.

Dans un listener, écartez en premier les cas particuliers et soyez conscient de la **criticité du traitement** que vous écrivez.

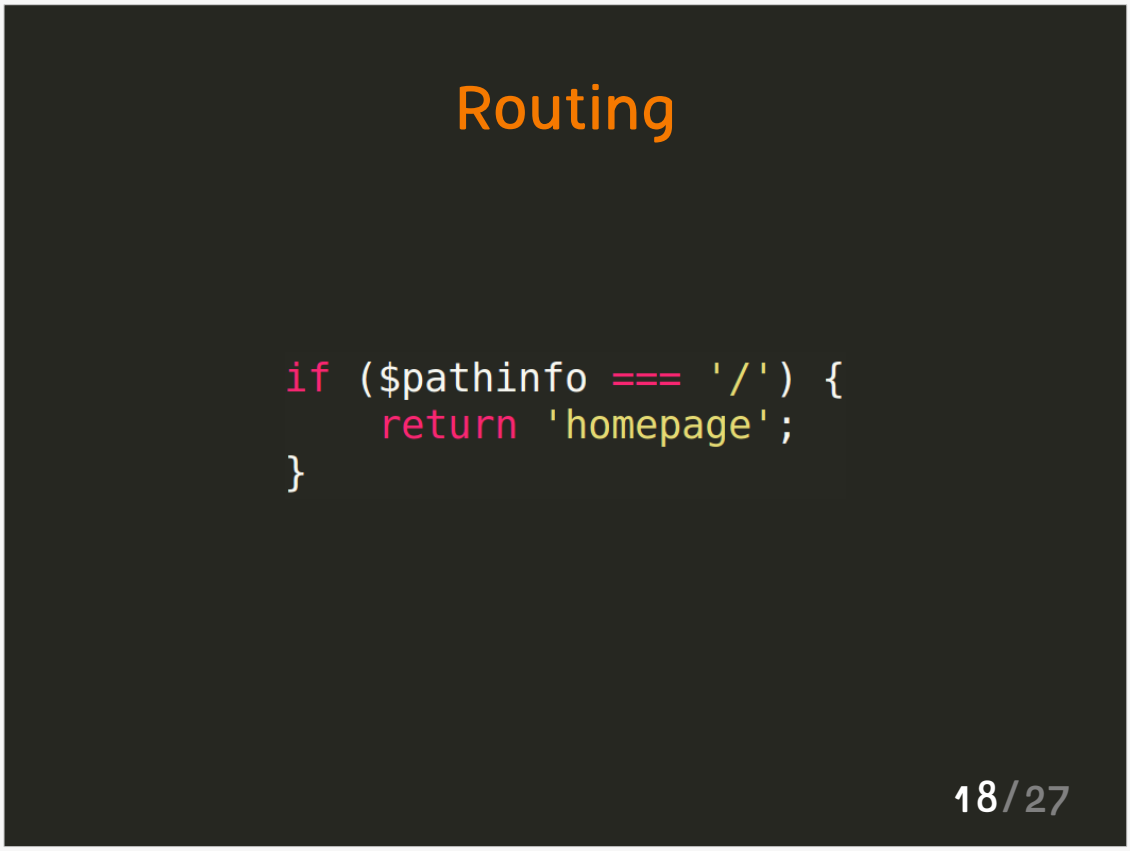
Les **dépendances** de ce listener seront naturellement construites avec l'objet, et augmentent donc le coût de ces listeners..



La performance du composant de formulaires en **2.1est nettement supérieure à celle en 2.0 :** http://symfony.com/blog/form-goodness-in-symfony-2-1

Une première façon d'optimiser la performance de ses formulaires est de **déclarer ses formulaires dans le conteneur de services**. Cela permet au composant de réutiliser vos objets *Type* pour la construction de différents formulaires.

Finalement, je vous conseille les supports du Symfony Live San Francisco 2012 de Bernhard Schussek sur les formulaires, ils sont parfaits.

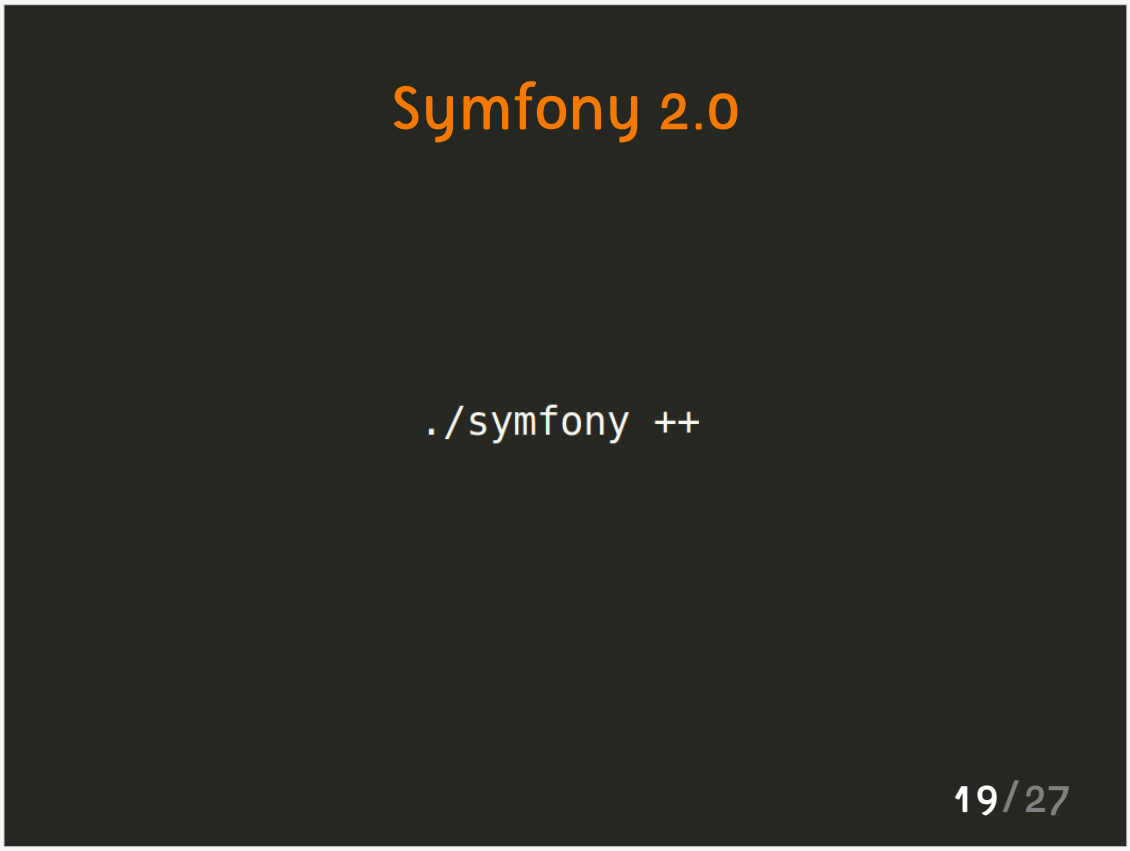


La **première route trouvée** sera utilisée. Ainsi si votre page d'accueil est chargée en première, elle sera la première à être testée. L'**ordre de définition** des routes est donc très important pour la performance de votre routing.

Cet définition n'a qu'un point de départ : *app/config/routing(_dev)?.yml*

Le **cache** sera composé de deux éléments : *UrlMatcher* et *UrlGenerator*. L'un peut être chargé sans l'autre → on ne charge pas les infos pour générer une URL au moment du routing.

Il est possible d'exporter ce cache dans Apache pour plus optimiser le temps de routing.



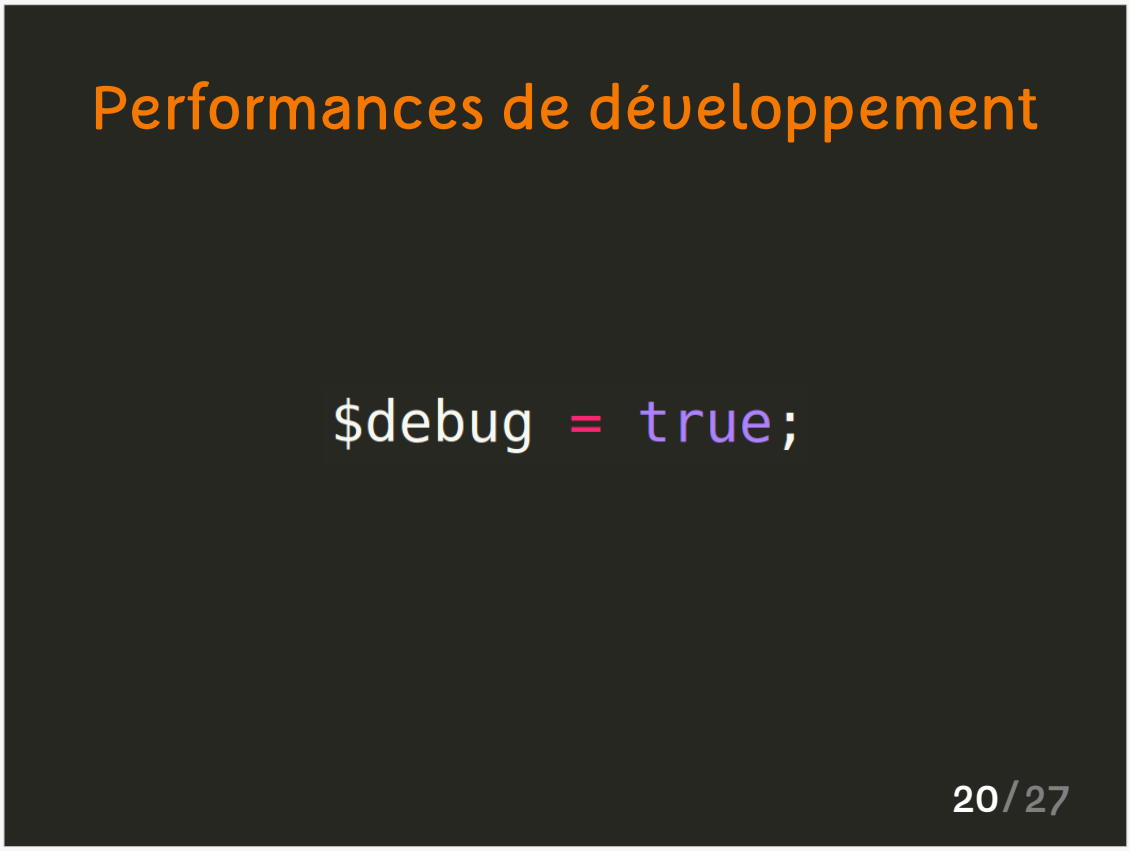
Certaines vérifications sont d'usage en Symfony 2.0 :

Le cache d'autoloading est maintenant transparent grâce à composer. En 2.0, ça ne l'est pas. Vérifiez donc vos fichiers !

- Cache APC pour **Doctrine**
- Mettre à jour ses dépendances (deps, deps.lock)

Il est fortement recommandé de mettre à jour Symfony en 2.1 afin de profiter des dernières avancées.

La **migration de 2.0 vers 2.1** n'est pas compliquée, c'est surtout des mises à jour mécaniques. Le fichier UGRADE-2.1.md liste tous les changements à opérer.

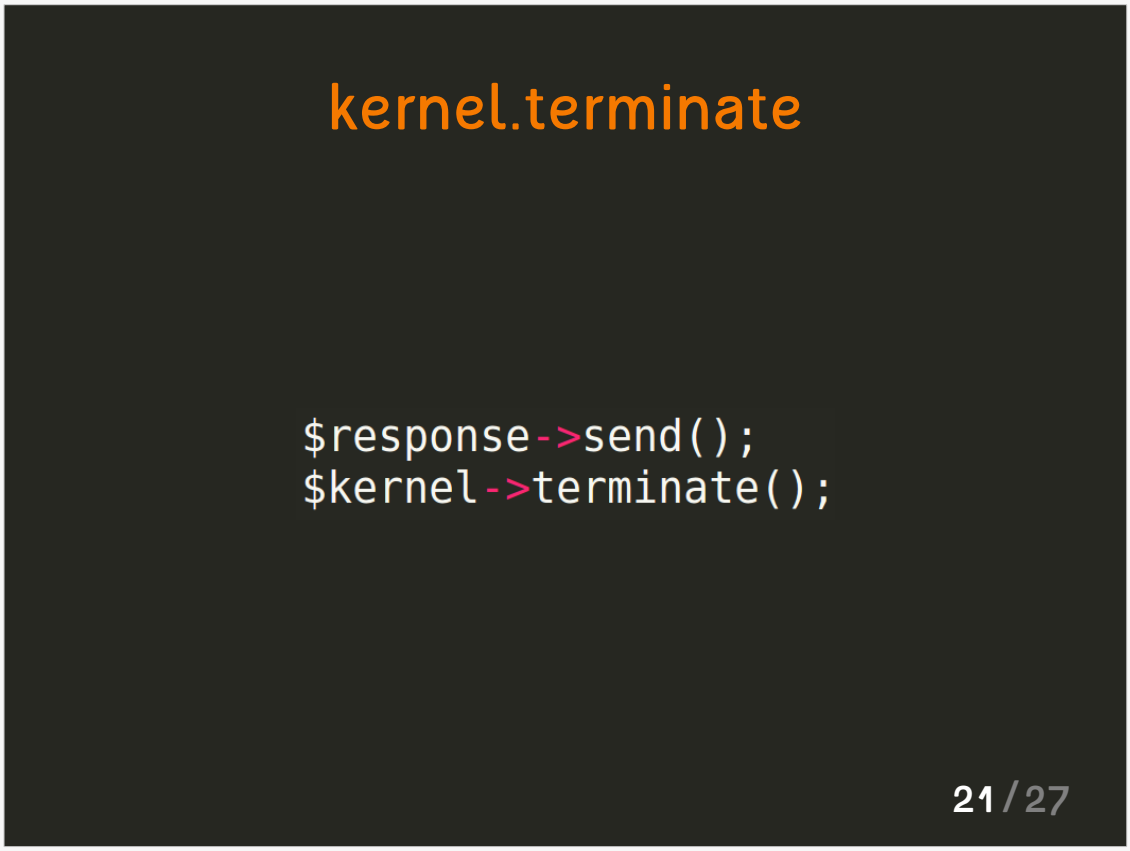


La performance en développement est très importante, tout aussi que la performance en production. Veillez à avoir un environnement de développement **efficace**.

Contre-performant : montages Samba, absence d'APC, serveurs distants. Tout le temps qu'on peut gagner en développement est à prendre.

Attention également aux bundles qui alourdissent les process. Le DiExtraBundle en un exemple de bundle à ne pas utiliser : ces bundles stockent un nombre important de données. L'extraction et le stockage de ces informations à chaque requête peut être coûteuse en développement.

Ne pas dire ”La page met 3 secondes, c'est normal on est en développement”



L'événement **kernel.terminate** est lancé dans l'event-dispatcher après que la réponse ait été envoyée au client.

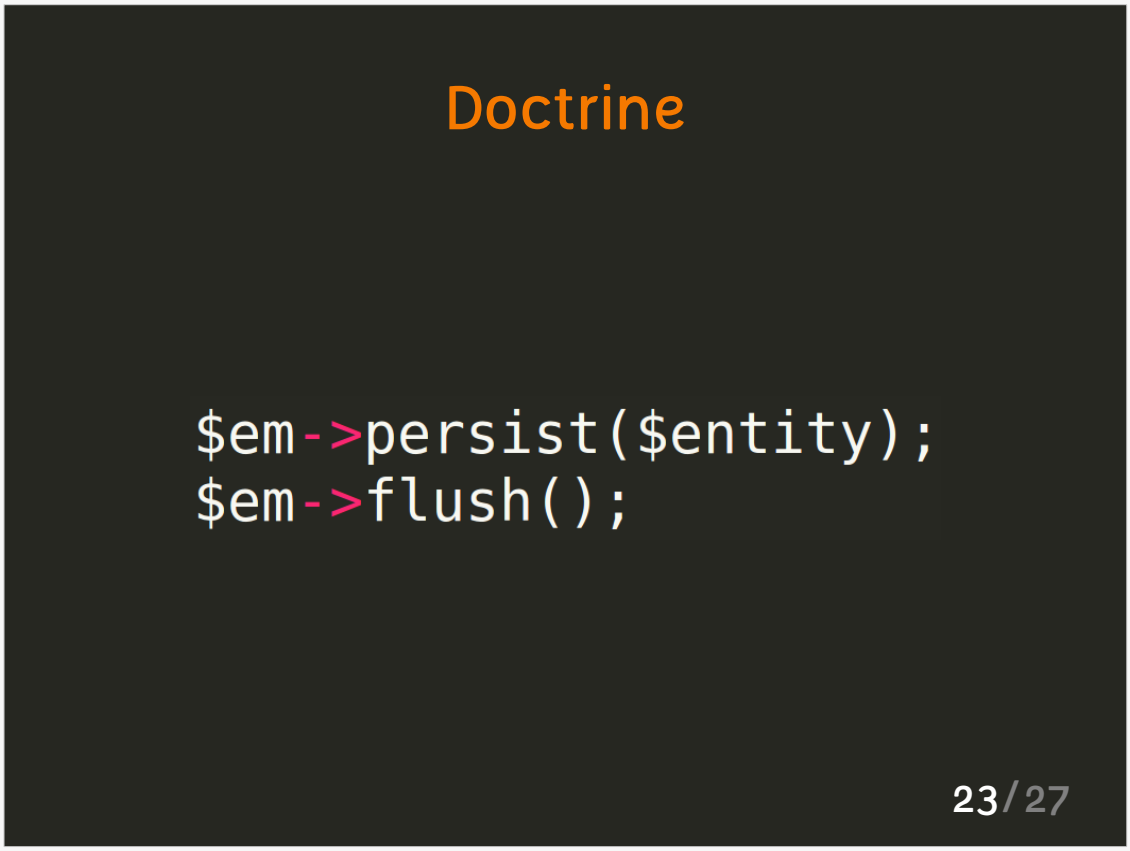
Cet événement peut-être utilisé pour par exemple envoyer un mail.

Cela permet de renvoyer rapidement la réponse au client pour ensuite faire ces traitements.

Fonctionne avec **PHP-FPM**, par exemple.

Limitations :
- Impossible de modifier la réponse envoyée
- Nécessite une install PHP compatible
- Sinon kernel.terminate est éxécuté avant de mettre fin à la réponse au client

Maintenant, nous allons voir comment améliorer **la performance de nos dépendances**.



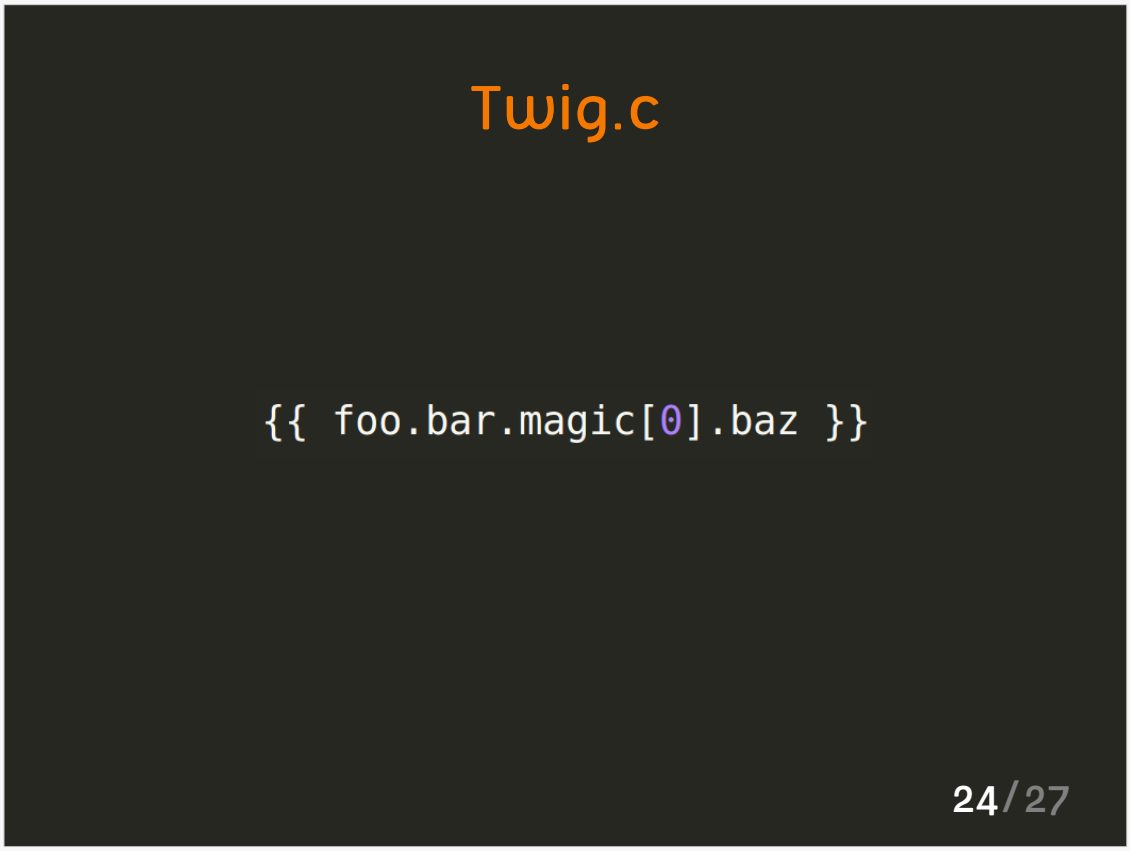
La dépendance la plus répandue avec Symfony2 est sûrement Doctrine.

Attention au ratio **persist/flush**. L'appel à un flush est significativement plus coûteux qu'un appel à persist.

Bien veiller **aux appels transactionnels** votre application.

Chaque **transaction** de votre application est stockée dans une **UnitOfWork**. Cette UnitOfWork stocke les différents ordres à exécuter et exécutera les ordres SQL en une transaction.

Il faut finalement bien veiller à l'activation du cache de méta-données. Si votre application le nécessite, le cache de requêtes et/ou le cache de résultats.



Twig dispose d'un module en C, permettant d'optimiser le rendu des templates Twig

La résolution des attributs dans une expression est très coûteuse. C'est la partie la plus coûteuse; on crée donc une extension C pour cette partie

Assetic est une véritable avancée pour la gestion des feuilles de style et les fichiers Javascript. Son fonctionnement est telle que sa performance en production est infaillible : on ne sert que des fichiers statiques.

Malheureusement, mal configuré, Assetic devient vite contre-performant en développement.

Une technique pour accélérer le temps consiste à utiliser **combine=”true”** dans la définition.

Si le temps de chargement devient vraiment trop long, utilisez le **mode watch** plutôt que les contrôleurs.

Références

- 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

Say fini