Audit de performance

Table des matières

1. État des lieux

1.1. Environnement de travail du projet initial

PHP	5.5.9
Symfony	3.1
Doctrine/ORM	2.5

1.2. Analyse de la performance

Nous utilisons Blackfire pour évaluer la performance de l'application.

L'application sera testée en mode de production pour ne pas être biaisé par la perte de performance induite par l'utilisation du Web Profiler.

Le fichier « .blackfire.yml » à la racine du projet contient les tests que Blackfire effectuera à chaque analyse afin de vérifier que notre application est bien en mode production.

- **Profil de la page de login :**

https://blackfire.io/profiles/458adfa5-99d2-40a4-afc1-7eac33d2a128/graph

Temps de génération de la page		**Mémoire utilisée**
1,56 sec		9,72MB
Temps accès processeur	**Temps accès système de fichiers**	
1 sec	551 ms	

- **Profil de la page d'accueil :**

https://blackfire.io/profiles/8bfe706c-e6e0-4f59-b990-e9ea81c67623/graph

Temps de génération de la page		**Mémoire utilisée**	**DB Connection**	**Requêtes SQL**
1,08 sec		14.9 MB	21,6 ms	1 (1,65 ms)
Temps accès processeur	**Temps accès système de fichiers**			
645 ms	433 ms			

Remarques :

- Méthode « **loadClass** » de la classe « **Symfony\Component\Debug\ DebugClassLoader** » :

Cette méthode représente environ 57 % du temps exclusif dans les deux profils. La classe DebugClassLoader est dépréciée depuis Symfony 4.4 donc la méthode en question ne sera plus appelée suite à la mise à jour du projet vers Symfony 5.3.

- Méthodes de la classe « **Composer\Autoload\ClassLoader** » :

  La méthode « **findFile** » est appelée 304 fois et représente 27% du temps inclusif contre 0,2 % du temps exclusif. Cela signifie que le coût de performance vient d'une ou plusieurs fonctions appelées dans « **findFile** ».

  La méthode « **findFile** » appelle uniquement la méthode « **findFileWithExtension** ». Cette dernière méthode est appelée lorsque l'autoloader de composer ne trouve pas une classe dans sa classmap. Elle représente 26,6 % du temps inclusif contre 1,4 % du temps exclusif.

  La méthode « **findFileWithExtension** » appelle les fonctions « **file_exists** » (94,5 %) et « **strtr** » (0,10 %).

  C'est donc de la fonction « **file_exists** » que provient le coût principal de performance. Elle représente en effet 25,4 % du temps exclusif.

  Nous optimiserons l'autoloader de Composer afin qu'il cherche les classes à importer dans une « classmap ». S'il y trouve la classe, la fonction « file_exists » ne sera pas appelée et notre application sera plus rapide.

2. Amélioration des performances de l'application

2.1. Optimisations pour Symfony

2.1.1. Mise à jour vers Symfony 5.3

La version 3 de Symfony n'étant plus maintenue, nous avons migré l'application sur la version 5.3 de Symfony qui est la version actuellement stable du framework.

2.1.2. Compilation du conteneur de service dans un seul fichier

Symfony compile le conteneur de service en plusieurs petits fichiers par défaut.

Définir le paramètre « **container.dumper.inline_factorie** » sur true permet de compiler l'intégralité du conteneur dans un seul fichier et peut améliorer les performances lors de l'utilisation du préchargement de classes depuis le cache « OPcache » dans PHP 7.4 ou les versions plus récentes.

```
# config/services.yaml
```

```
parameters:
    # ...
    container.dumper.inline_factories: true
```

2.1.3. Gestion des variables d'environnement en production

À chaque requête, notre application traite les fichiers « .env.* ».

Il est possible de générer un fichier « .env.local.php » optimisé qui remplace tous les autres fichiers de configuration.

Il suffit pour cela de lancer la commande : *composer dump-env prod*

2.1.4. Configurer le cache Realpath

Lorsqu'un chemin relatif est transformé en chemin réel et absolu, PHP met en cache le résultat pour améliorer les performances.

Les applications qui ouvrent de nombreux fichiers PHP, comme les projets Symfony, doivent utiliser au moins ces valeurs :

```
; php.ini
; maximum memory allocated to store the results
realpath_cache_size=4096K

; save the results for 10 minutes (600 seconds)
realpath_cache_ttl=600
```

Remarque : ce cache est désactivé lorsque l'option de configuration open_basedir est activée.

2.2. Optimisations pour PHP

2.2.1. Activation de l'OPCache et préchargement des scripts dans l'OPcache au démarrage du serveur

Pour faire fonctionner les scripts PHP, le moteur Zend transforme le code afin de l'exécuter plus facilement. Le résultat cette compilation du code est appelé « OPCode ».

Sans cache, chaque script PHP est analysé puis compilé en OPCode avant d'être exécuté.

En mettant l'OPCode en cache, on évite tout ce travail d'analyse du script PHP. Depuis PHP 5.5, l'extension « OPcache » permet de faire cela.

L'OPcache s'active dans le fichier php.ini :

```
zend_extension=opcache
```

```
…
[opcache]
; Determines if Zend OPCache is enabled
opcache.enable=1
```

Par ailleurs, Symfony fournit un script de préchargement prêt à l'emploi qui sera compilé et exécuté lors du démarrage du serveur, et dont l'intérêt sera de précharger d'autres fichiers dans le cache OPcache. Toutes les fonctions et classes définies dans ces fichiers seront disponibles aux requêtes, jusqu'à ce que le serveur soit éteint.

Cela se règle également dans le fichier php.ini :

```
opcache.preload=/path/to/project/config/preload.php
```

Remarque : Le préchargement n'est pas supporté sur Windows.

La configuration par défaut d'OPcache n'est pas adaptée aux applications Symfony, il est donc recommandé de modifier ces paramètres comme suit :

```
; php.ini
pcache.memory_consumption=256
opcache.max_accelerated_files=20000
```

2.2.2. Ne vérifiez pas les « timestamps » des fichiers php

Dans les serveurs de production, les fichiers PHP ne doivent jamais changer, à moins qu'une nouvelle version de l'application ne soit déployée. Cependant, par défaut, OPcache vérifie si les fichiers mis en cache ont changé leur contenu depuis leur mise en cache. Cette vérification introduit une surcharge qui peut être évitée comme suit :

```
; php.ini
opcache.validate_timestamps=0
```

Si nous faisons cela, à chaque déploiement, nous devrons vider et régénérer le cache d'OPcache manuellement ou en redémarrant le serveur Web afin que les modifications apportées prennent effet.

2.3. Optimisation de l'autoloader de Composer

Dans les serveurs de production, les fichiers PHP ne doivent jamais changer, à moins qu'une nouvelle version de l'application ne soit déployée. C'est pourquoi on peut optimiser l'autoloader de Composer en lui demandant d'analyser l'ensemble de l'application afin de créer une « classmap » qui contiendra toutes les classes dont l'application aura besoin en production.

Cette « classmap » est stockée dans vendor/composer/autoload_classmap.php.

Il faudra pour cela exécuter la commande suivante :

```
composer dump-autoload --no-dev --classmap-authoritative
```

Remarques :

--no-dev : exclut les classes qui ne sont nécessaires que dans l'environnement de développement (c'est-à-dire les dépendances de require-dev et les règles autoload-dev)

--classmap-authoritative : convertit l'autoloading PSR-0/4 en classmap pour obtenir un autoloader plus rapide (revient à faire --optimize). Cette option indique également que si quelque chose n'est pas trouvé dans la classmap, l'autoloader ne doit pas tenter de regarder le système de fichiers selon les règles PSR-4.

3. Gains de performances liées aux optimisations

- **Profil de la page de login après optimisations :**

https://blackfire.io/profiles/4f2797df-8eca-4b0f-89d0-83125181f887/graph

Temps de génération de la page		Mémoire utilisée
22.6 ms		2.72 MB
Temps accès processeur	**Temps accès système de fichiers**	
15.7 ms	6.89 ms	

Comparatif : https://blackfire.io/profiles/compare/83b17174-102d-4e63-a312-7f67ff76d8fb/graph

Temps de génération de la page		Mémoire utilisée
-98%		-72%
Temps accès processeur	**Temps accès système de fichiers**	
-98%	-99%	

- **Profil de la page d'accueil après optimisations :**

https://blackfire.io/profiles/7403654c-0a46-4e9e-9119-1c7929107f0d/graph

Temps de génération de la page		Mémoire utilisée	DB Connection	Requêtes SQL
28.2 ms		3.62 MB	1.46 ms	1 (663 µs)
Temps accès processeur	**Temps accès système de fichiers**			
17.7 ms	10.6 ms			

Comparatif : https://blackfire.io/profiles/compare/11ede12c-8367-4e47-93ab-d19a20ed103d/graph

Temps de génération de la page		Mémoire utilisée	DB Connection	Requêtes SQL
-97%		-76%	-20,14 ms	1 (-982 µs)
Temps accès processeur	**Temps accès système de fichiers**			
-97%	-98%			