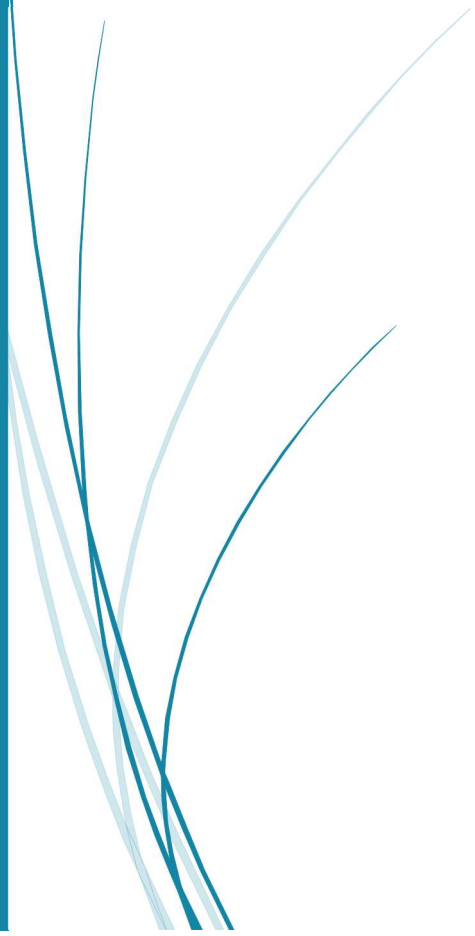
03/06/2022

Améliorez une application existante de ToDo & Co

Projet P8 - OpenClassRooms



Serge Pillay

Table des matières

Introduction

Contexte

L'entreprise **ToDo & Co** est une startup dont le corps de métier est une application permettant de gérer ses tâches quotidiennes.

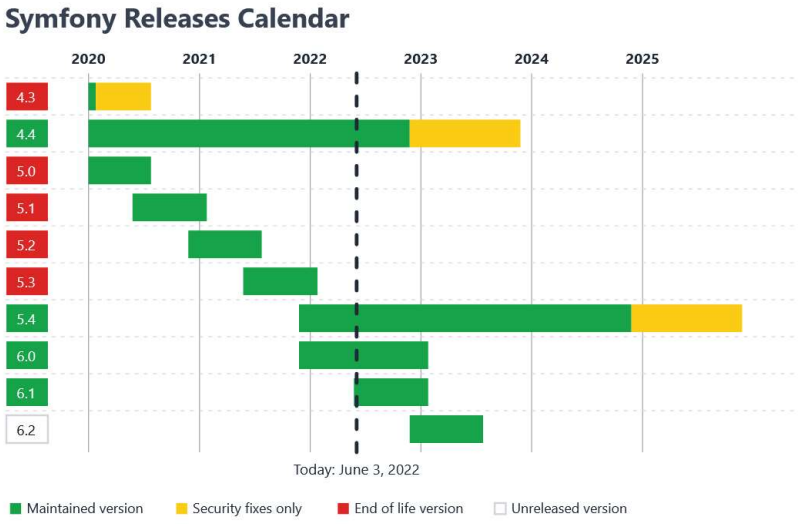
L'application vient a dû être développée à toute vitesse pour permettre de montrer à de potentiels investisseurs que le concept est viable (on parle de Minimum Viable Product MVP).

Le choix du développeur précédent a été d'utiliser le Framework PHP Symfony.

L'entreprise ToDo & Co ayant réussi à lever des fonds pour permettre le développement de l'entreprise et surtout de l'application

Urgence de version

L'application a été mise à jour vers une version plus ressente par souci de maintenance, mais aussi de sécurité. Il faut savoir que la version 3.1 de Symfony n'est plus maintenue, il était donc normal de faire une mise à jour. La version 5.4 a été choisie pour une plus longue période de maintenance.



	Avant mise à jour	Après mise à jour
php	>=5.5.9	>=7.2.5
symfony/symfony	3.1.*	5.4.*
doctrine/orm	^2.5	^2.12
bootstrap	3.3.7	5.1.3

Qualité du code

Pour faire l'analyse 2 sites Internet ont été utilisés :

- **Codacy**, pour les erreurs de code et les bonnes pratiques.
- **Code Climate**, pour la partie maintenabilité mais aussi la qualité du code.

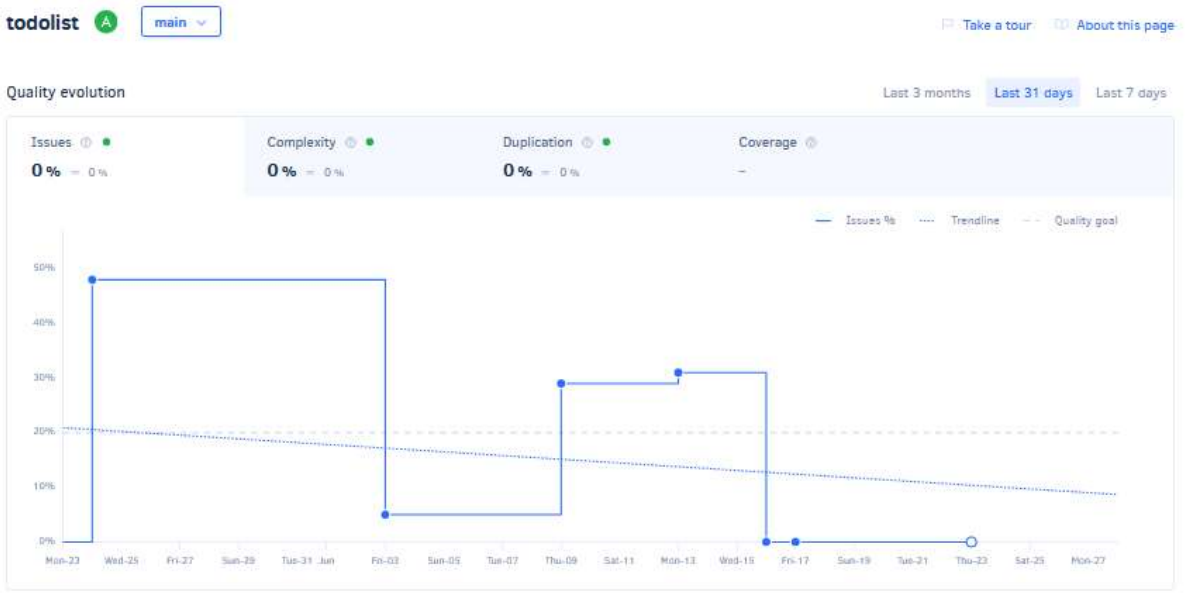
Les deux sites sont très similaires au premier abord, mais une deuxième vérification du code est toujours bénéfique, et les deux analyses sont grandement complémentaires.

L'analyse avec Codacy

Selon Codacy, les statistiques de complexité et de duplication de code sont à 0 %, le nombre d'issues est de 0 %.

La note globale du site est « A », ce qui est la meilleure note.

Ce qui, dans la globalité, présente une très bonne qualité de code. À savoir : le grade est calculé sur le nombre d'issues pour 1k lignes de code.



Les différentes issues sont répertoriées par catégories, mais aussi par niveau, elles représentent les problèmes de code venant du site.

Info : Le type de problème le moins critique apparaîtra en bleu ; par exemple, les problèmes de style de code.

Avertissement : ce type de problème apparaîtra en jaune. Vous devez être prudent avec ceux-ci, ils sont basés sur les normes et les conventions du code.

Erreur : les types de problèmes les plus dangereux s'affichent en rouge. Prenez le temps de les corriger, bien que le code puisse s'exécuter, ces problèmes montrent que le code est très susceptible de poser des problèmes. Ces problèmes sont sujets aux bogues et / ou peuvent avoir de graves problèmes de sécurité et de compatibilité.

Une configuration des exclusions de fichiers a été faite pour ne pas prendre en comptes les lignes de code généré par Symfony et les différentes librairies utilisées.

Lien du rapport Codacy :

https://app.codacy.com/gh/Urza45/todolist/dashboard?branch=main

L'analyse avec Code CLimate

Selon Code Climate, la maintenabilité de l'application est optimale, car il n'y a pas beaucoup de mauvaises pratiques (Code Smells), duplications et autres problèmes. Ce qui représente les meilleures notes possibles à avoir sur son site.

Une configuration des exclusions de fichiers n'a pas été faite comme pour Codacy, pour ne pas prendre en comptes les lignes de code généré par Symfony et les différentes librairies utilisées, car nous utilisons la version gratuite de Code Climate.

Le schéma ci-dessus représente l'évolution de la dette technique du projet.

Il faut savoir que le projet a été suivi depuis le début, même avant la mise à jour du site. On peut aisément dire que la mise à jour de Symfony a réglé beaucoup de soucis.

Performances

Analyse Blackfire

Pour effectuer une analyse Blackfire, ne pas oublier de passer en mode prod (env=prod), sachant que le debugger de Symfony prend une grande part des ressources lorsqu'il est activé.

L'analyse de Blackfire donne :

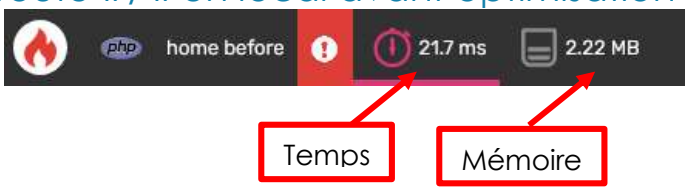
- Le temps de chargement du site.
- La mémoire utilisée pour son chargement,
- Les fonctions utilisées et leur nombre d'appels. Chaque fonction est détaillée avec son temps d'exécution et la mémoire utilisée.

Un schéma est aussi consultable pour suivre le plan d'exécution du site, avec une couleur spécifique pour les fonctions les plus gourmande.

Les tests ici présents, sont faits avec la licence gratuite de Blackfire, ce qui limite l'analyse du site.

À savoir : avec la version payante des recommandations d'optimisation sont proposés

Analyse de la route « / » en local avant optimisation



Les fonctions appelées (triées par nombres d'appels décroissants) :

Lors de cette première analyse nous pouvons constater un nombre très important d'appel des fonctions telle spl_autoload_call, loadclass, Composer\Autoload\IncudeFile, Composer\Autoload\ClassLoader ::findFile ...

Ce qui amène naturellement vers l'optimisation de l'autoloader de Composer.

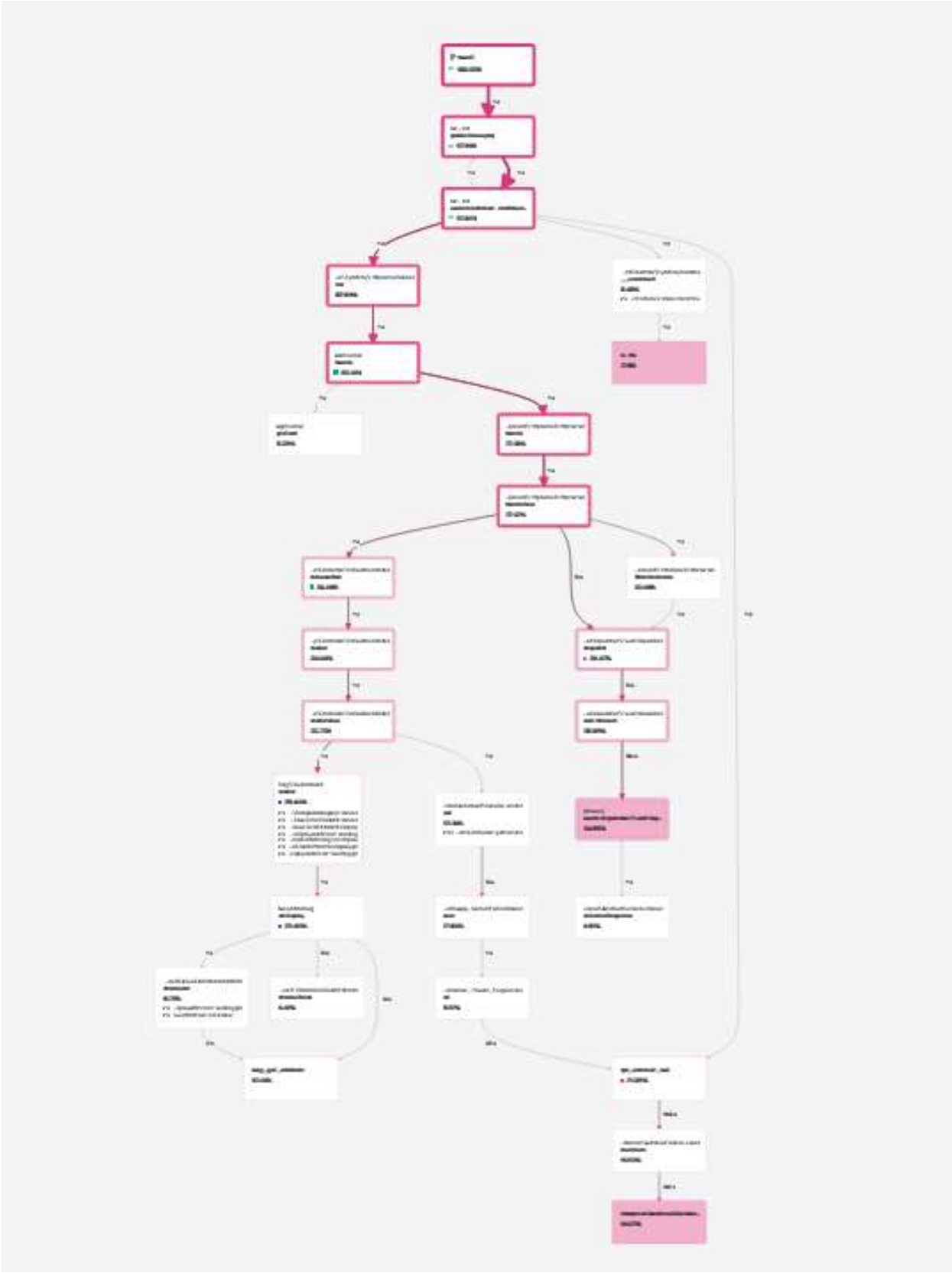


Figure 1 : route "/" avant optimisation

https://blackfire.io/profiles/65d8ef08-6f79-4845-9f81-167180d2d91a/graph?settings%5Bdimension%5D=wt&settings%5Bdisplay%5D=landscape&settings%5BtabPane%5D=nodes&selected=&callname=main()&constraintDoc=

L'autoloader utilisé lors du développement de l'application est optimisé pour trouver des classes nouvelles et modifiées.

Dans les serveurs de production, les fichiers PHP ne doivent jamais changer, sauf si une nouvelle version de l'application est déployée. C'est pourquoi vous pouvez optimiser l'autoloader de Composer pour analyser l'intégralité de l'application une fois et créer une "carte de classe" optimisée, qui est un grand tableau des emplacements de toutes les classes et qui est stockée dans **vendor/composer/autoload_classmap.php**.

Exécutez cette commande pour générer le nouveau mappage de classe (et l'intégrer également à votre processus de déploiement) :

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

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

--classmap-authoritative crée un mappage de classe pour les classes compatibles PSR-0 et PSR-4 utilisées dans l'application, et empêche Composer d'analyser les classes qui ne se trouvent pas dans le mappage de classe.

```
PS D:\WAMP\www\todolist> composer dump-autoload --no-dev --classmap-authoritative
Generating optimized autoload files (authoritative)
Generated optimized autoload files (authoritative) containing 4638 classes
PS D:\WAMP\www\todolist>
```

Analyse de la route « / » en local après optimisation

Au premier abord nous constatons une amélioration du temps de chargement 19,6 ms au lieu de 2,21 ms

Les fonctions appelées (triées par nombres d'appels décroissants) :

Function calls	% Excl.	% Incl.	Calls
Composer\Autoload\ClassLoader::findFile			369
dirname			228
spl_autoload_call			164
Composer\Autoload\ClassLoader::loadClass			163
Composer\Autoload\includeFile			161
str_starts_with			129
Composer\Autoload\includeFile@1			106
Composer\Autoload\ClassLoader::loadClass@1			106
spl_autoload_call@1			106
class_exists			101
…nent\EventDispatcher\EventDispatcher::addListener			63
Composer\Autoload\includeFile@2			55
Composer\Autoload\ClassLoader::loadClass@2			55
spl_autoload_call@2			55
ure}::event-dispatcher/EventDispatcher.php/265-271			30

Les nombres d'appels quand a eu non pas évolués, cependant ces appels se font de manière un peu différente comme le montre le schéma ci-dessous.

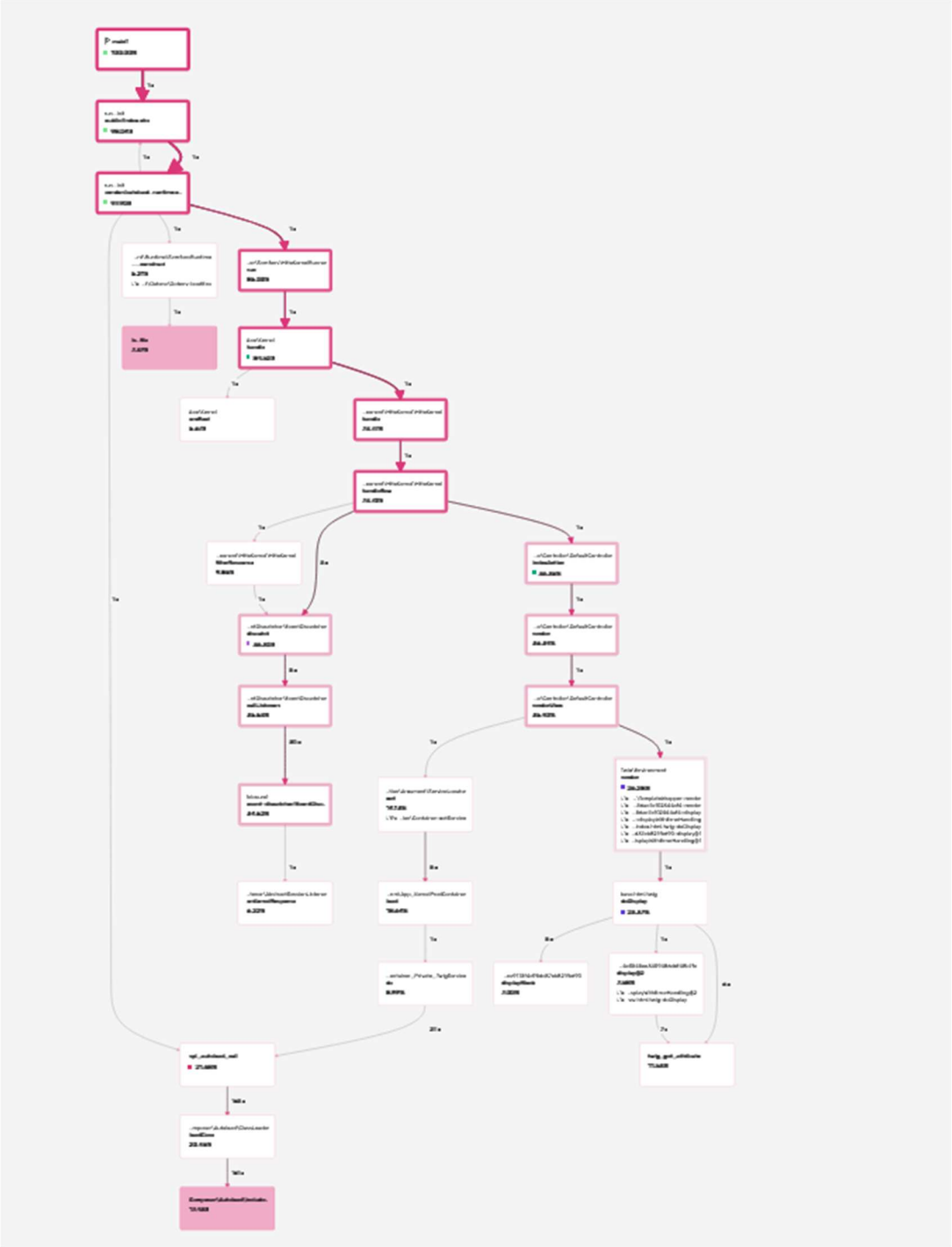


Figure 2 : : route "/" après optimisation

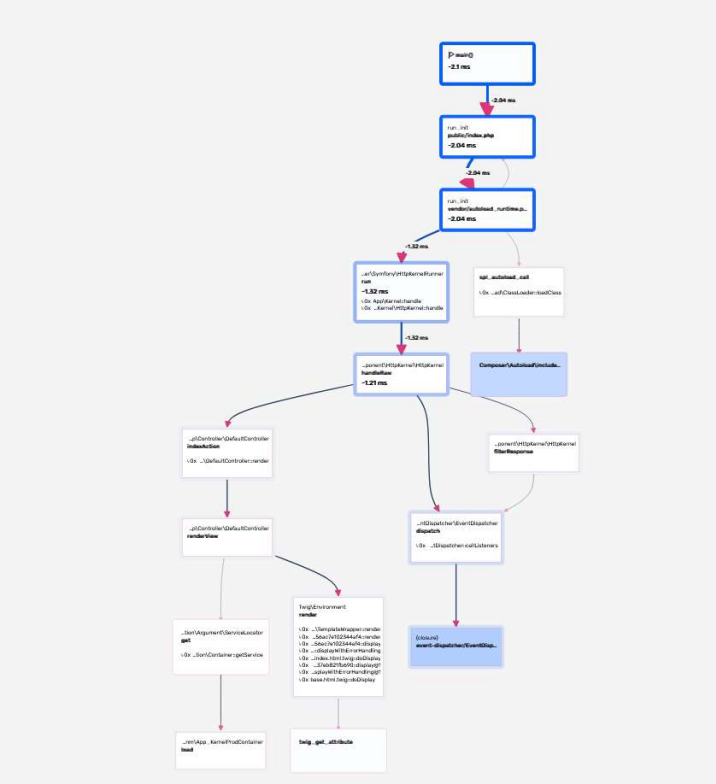
https://blackfire.io/profiles/7246feaf-965b-49ba-8609-979a6b305742/graph?settings%5Bdimension%5D=wt&settings%5Bdisplay%5D=landscape&settings%5BtabPane%5D=nodes&selected=&callname=main()&constraintDoc=

Blackfire nous permet de comparer les deux analyses :

Nous voyons que nous avons gagné presque 10% de temps de chargement et 0,2% de gain de mémoire.

Nous avons également économisé quelques appels de fonctions :

Function calls	% Incl.	Calls
composerRequire15b8f52ceec9a1ab5919888472a09cb6		-17
Symfony\Bundle\FrameworkBundle\Routing\Router::getGenerator		-2
class_exists		-1
main()		+0
run_init::public/index.php		+0
Symfony\Component\HttpFoundation\Session\Session::save		+0
…ent\HttpFoundation\Session\Storage\NativeSessionStorage::save		+0
session_write_close		+0
…App_KernelProdContainer::getAnnotations_CachedReaderService		+0
spl_autoload_call		+0



En bleu dans le graphe ci-contre, les classes pour lesquels un gain a été obtenu.

Lien du graphe