AUDIT DE L'APPLICATION – TODO LIST de ToDo & Co

L'audit de qualité du code et de performance a été réalisé après la correction d'anomalies et l'ajout de fonctionnalités sur le Minimum Viable Product. Après un état des lieux, un plan pour améliorer le code est proposé. Quant à la performance, le chargement des classes par composer a été optimisée réduisant ainsi considérablement le temps d'exécution de l'application PHP.

1. Qualité du code

• Dette technique

Le code PHP respecte les normes PSR-2, c'est-à-dire des règles strictes en terme de nommage, d'indentation et d'espacement. Il est commenté avec PHPDoc et le répertoire GitHub où a été déposé le projet, après avoir subi une revue de code automatisé par Codacy, a obtenu le badge A qui certifie la bonne qualité du code.

• Améliorations possibles

La version actuelle de l'application symfony est 3.2. Elle n'est plus maintenue par SensioLabs et une migration au minimum vers la version 3.4 est requise afin de bénéficier d'un support longue durée avec des corrections contre les failles de sécurité ou des bugs signalés.

Le formulaire de connexion, contrairement aux autres, n'a pas recourt au composant Form de symfony et présente donc un risque de faille CSRF en l'absence d'un token. Il faudra y remédier.

2. Performance

- ### Blackfire.io

Blackfire.io est un service disponible sur navigateur et en ligne de commande pour mesurer la performance des applications PHP. Ce profileur présente de nombreuses avantages. Il ne requiert aucune modification du code et permet de trouver les causes des problèmes de performance à la différence du web profiler fourni par le framework Symfony qui est bien plus limité. De plus, la plateforme stocke et permet d'analyser et de comparer des profils de mesure.

Cet outil a été utilisé pour réaliser l'audit de performance de l'application de ToDo & Co en environnement de production afin de désactiver le web profiler de symfony qui aurait altérer les résultats.

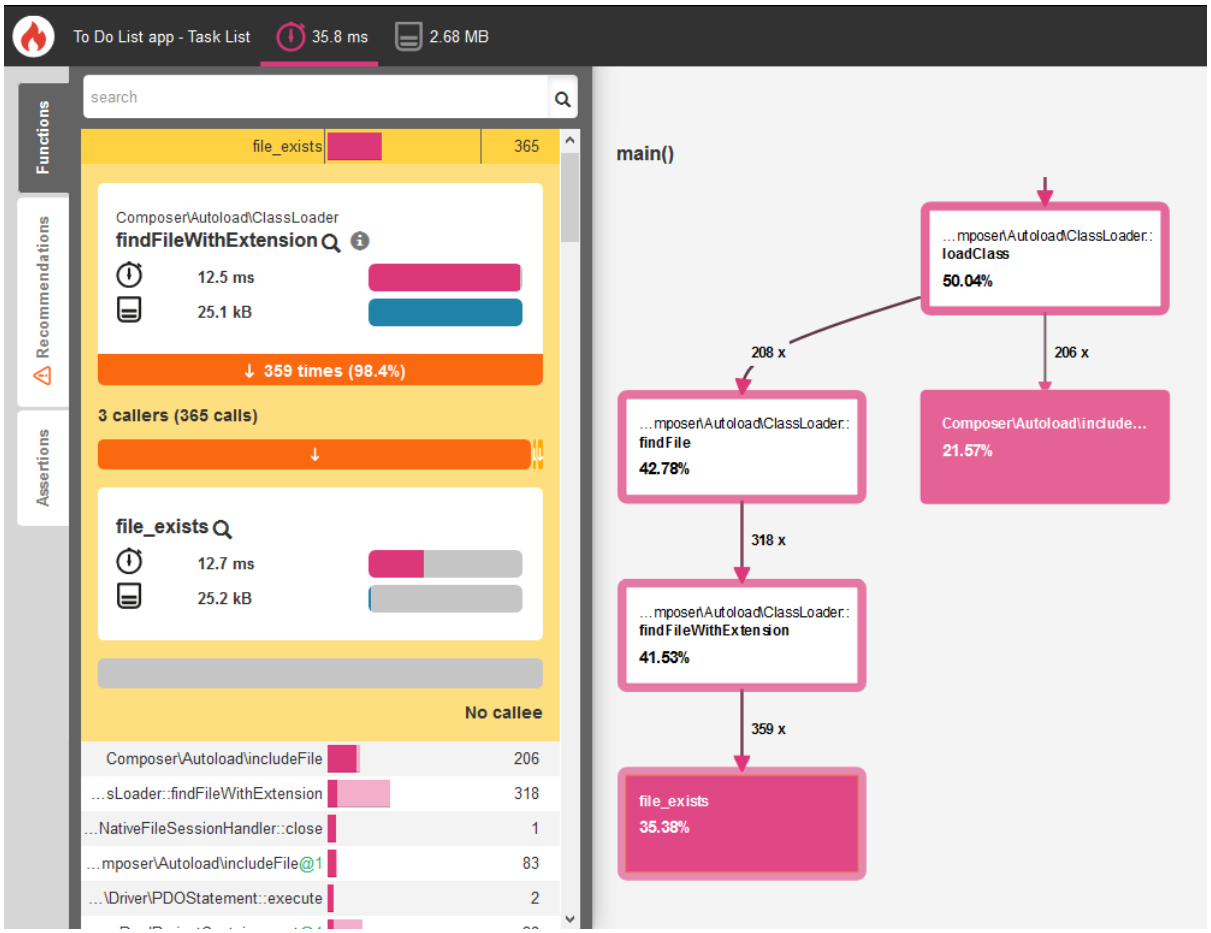
- ### Dette technique

Blackfire.io fournit des métriques, notamment sur le temps de génération en millisecondes d'une page en PHP ou bien sur la mémoire utilisée en mégabytes. Voici le tableau qui récapitule ces informations sur l'ensemble des routes et méthodes existantes de l'application :

Route	Méthode HTTP	Temps d'exécution - ms	Mémoire - mb
/	GET	32.9	2.55
/login	GET	21.7	1.67
/login_check	POST	34.3	2.19
/logout	GET	15.2	1.22
/tasks	GET	35.8	2.68
/tasks/create	GET	46.8	3.35
/tasks/create	POST	48.4	3.75
/tasks/{id}/edit	GET	54.8	3.39
/tasks/{id}/edit	POST	42.6	3.18
/tasks/{id}/toggle	GET	71.8	2.54
/tasks/{id}/delete	GET	41.6	2.66
/users	GET	33.9	2.58
/users/create	GET	54.6	3.53
/users/create	POST	57.9	4.02
/users/{id}/edit	GET	66.2	3.55
/users/{id}/edit	POST	75.5	3.35

Il faut savoir que les données collectées sont tributaires de la mémoire vive, du processeur et du disque dur de l'ordinateur qui ont générés les scripts, et qui eux-mêmes fonctionnent localement avec wampserver. Mais sur l'ensemble des pages, on peut affirmer que les résultats obtenus sont homogènes, et ne semblent pas dépasser outre mesure ce qu'on attend d'une application performante. Ainsi, le temps d'exécution moyen d'une route est de 46,11 millisecondes, soit largement en dessous de la seconde.

Voici dans le détail le profil de la page de la liste des tâches :



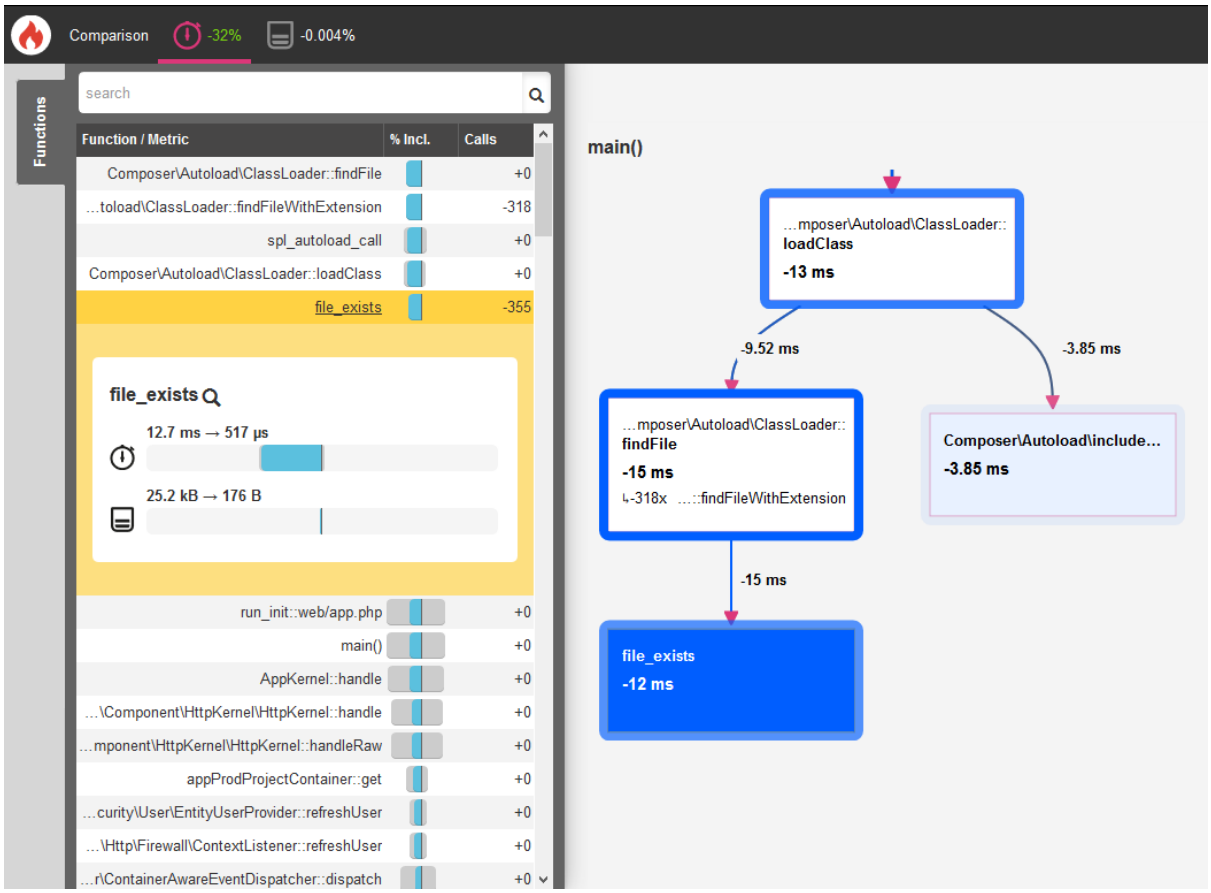
Le bloc de gauche donne la liste de toutes les fonctions exécutées par le script PHP. Les plus lentes en fonction du temps d'exécution sont en haut de la liste. Quant au chemin en surbrillance, à droite, il montre les fonctions les moins performantes et aide à choisir lesquelles doivent être optimiser. Comme pour l'ensemble des pages, on note que les deux fonctions les plus coûteuses en ressources sont file_exists() et incudeFile(). Elles sont appelées depuis composer et sont donc natives au framework Symfony.

Si on se focalise sur file_exists(), on constate que le temps d'exécution exclusif, c'est-à-dire sans tenir compte de l'appel à d'autres fonctions, est de 12.7 millisecondes et représente 35.38% du temps total de l'exécution du script. Si elle n'appelle pas de fonctions, elle est appelée à 365 reprises, et presque systématiquement par findFileWithExtension(). Ce nombre excessif d'appels s'explique par le fonctionnement de composer, autrement dit l'outil en charge du chargement des classes.

- Optimisation

Heureusement, il est possible d'optimiser ce chargement avec la commande composer dump-autoload –o qui évite la vérification systématique de fichiers dans l'autoloader de composer. Après avoir exécutée cette commande à la racine du projet, un deuxième profil a été créé pour la page de la liste de tâches. Il a été comparé avec le premier profil.

Voici le graphe de comparaison :



Dans un graphe de comparaison Blackfire, une fonction dont la performance a été amélioré est de couleur bleue. On note qu'il n'existe plus d'appels vers file_exists() est que le temps d'exécution inclusif, c'est-à-dire en tenant compte de l'appel à d'autres fonctions, s'est réduit de 12 millisecondes. Quant à incudeFile(), même si le nombre d'appels vers cette fonction est resté inchangé, elle a tout de même gagné 3.85 millisecondes. Et surtout le script a gagné 32% de temps de chargement. C'est le même impact positif que l'on constate sur l'ensemble des pages grâce à cette optimisation. Pour ce qui est de la mémoire vive, il n'y a pas de changement notable.