Audit de Qualité & Performance

Ce document a pour but de faire un état des lieux de la qualité et de la performance de l'application mais également de présenter des pistes d'amélioration.

Table des matières

Introduction

L'application a été développée sous Symfony 3.1, afin de respecter le délai de mise en production, nous avons fait le choix de migrer vers une version de Symfony plus récente (3.4).

Cela nous a donc permis d'améliorer légèrement la performance de l'application et d'assurer la stabilité du code. Pour autant, il reste des points qui peuvent être améliorés que nous allons voir ici.

Nous ne pourrons pas étudier toutes les pistes d'amélioration sur l'ensemble des pages, donc nous nous concentrerons sur la page de login.

Les principaux outils utilisés pour faire cette analyse seront :

- Pour la qualité :
  - L'outil incorporé à Symfony : « Profiler ».
  - Le site de Codacy qui va nous donner une note qualitative du code
- Pour la performance :
  - L'outil incorporé à Symfony : « Profiler ».
  - L'outil BlackFire

I. L'outil incorporé à Symfony « Profiler »

Introduction

Profiler est donc un outil développé par Symfony et 100% adapté pour toutes applications. Cet outil est disponible uniquement lorsque l'application est en statut de développement, il est facilement accessible, puisqu'il s'agit de la « barre de menu » Symfony qui est en bas lorsque l'on navigue en local.

Pour information cette barre est assez lourde et peut ralentir l'exécution de vos requêtes et donc détériorer la performance.

1. Analyse des éléments de la barre de l'outil Profiler

Sans ouvrir l'outil, on nous donne déjà un certain nombre d'informations. De la gauche vers la droite, on va retrouver :

- **Request :** Le Status Code de la page (200)
- **Response :** Le nom de la route sur lequel nous sommes (login)
- **Performance :** Le temps de chargement pour contrôler et afficher la page (152ms)
- **Performance :** La mémoire utilisé (2.0MB)
- **Logs :** Les erreurs, alertes & Dépréciations (11)
- **Routing / Cache :** Le nombre d'appel effectué vers le cache (4) et le temps d'exécution (0,86ms)

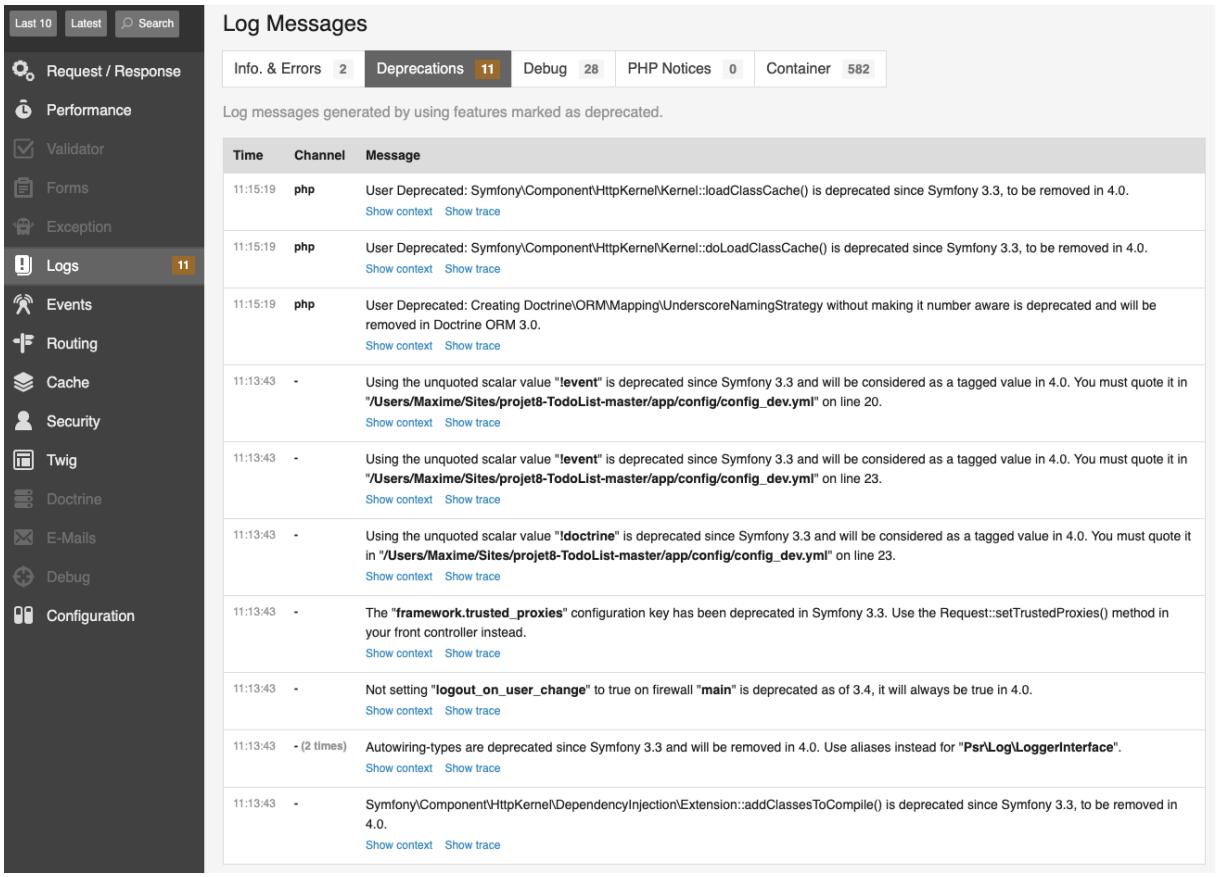
- **Security :** Les informations liés à l'utilisateur avec lequel nous sommes authentifié (anonyme)
- **Twig :** Le temps de chargement uniquement de l'affichage de la vue (13ms)
- **Configuration :** La version de Symfony utilisée (3.4.37)

En passant, la souris sur chacun de ces points, on peut voir plus d'informations détaillées.

Bon déjà avec tous ces indicateurs, on peut s'imaginer qu'il est facile d'avoir une première idée globale sur ce qu'il est possible de faire.

2. Analyse de la Qualité via Profiler

Lorsque l'on parle de qualité du code via le Profiler, on ne va pas parler de l'écriture du code en elle-même mais plutôt du niveau de sécurité ou de mise à jour des différents modules. Comme une image parle plus que des mots, regardons cela à travers le logs du Profiler :



A travers le Logs, on peut voir un certain nombre d'alertes. Dans la partie ci-dessus, il s'agit plus précisément des « Depreciations » qui sont aux nombres de 11 et si on les lit, on va vite s'apercevoir qu'il s'agit de notifications qui sont liées à notre version.

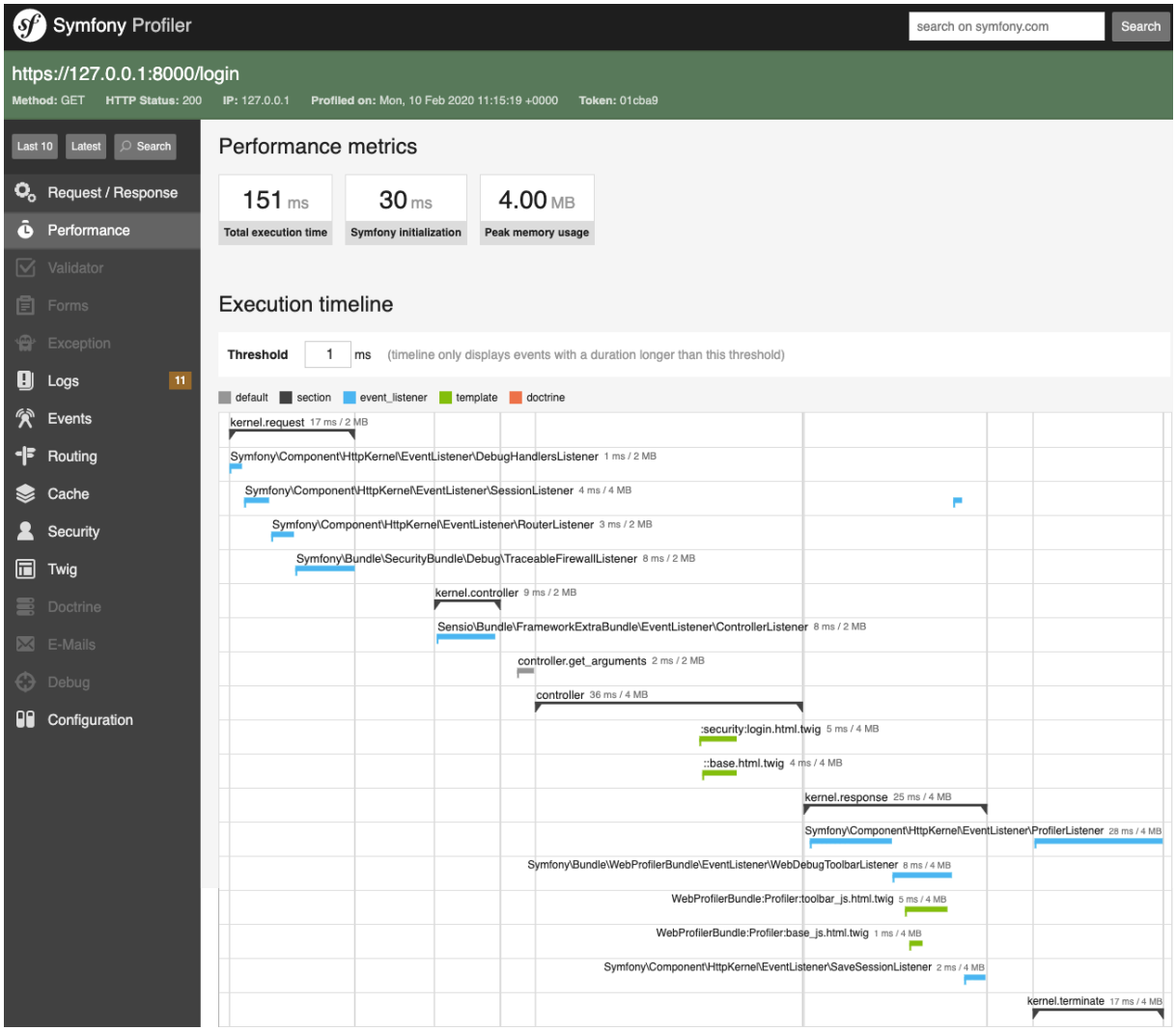
Un moyen facile d'y remédier serait de passer sur la version 4.4 de Symfony, pour facilement enlever toutes ces notifications ou au moins en résoudre 80%.

Pour ce qui est de qualité du code qui sera écrit et qui doit respecter un maximum de normalisation, on se tournera vers Codacy dans le prochain chapitre.

3. Analyse de la performance via Profiler

a. Présentation

En cliquant, sur l'indicateur du temps de chargement, on nous amène directement vers l'outil d'analyse de performance de Profiler.



- En haut, on retrouve les détails de la requête sur laquelle nous sommes.
- Dans le bandeau de gauche, on retrouve l'ensemble des différentes options que nous avons détaillées plus haut.
- Sur la page principale, on retrouve alors :
    - 3 métriques de performance
    - La ligne détaillée d'exécution de la requête

b. Analyse de la performance

Tout d'abord, on regarde les 3 métriques de performances qui nous sont présentés :

- Total execution time : 151ms
- Symfony initialization : 30ms
- Peak memory performance : 4MB

C'est 3 indicateurs nous permettre d'avoir une idée globale sur le poids de l'exécution de la requête pour accéder à la page /login. Dans l'ensemble, ces indicateurs sont corrects pour cette page.

Rentrons dans le détail.

Afin d'analyser facilement les différents éléments exécutés lors de la requête, Symfony Profiler nous mâche un peu le travail en catégorisant les différentes ressources pour nous permettre une meilleure lisibilité.

Grâce à ses indicateurs, on peut voir que :

- 5 sections sont exécutées
- L'event listener est principalement appelé pour gérer l'authentification
- Le module de template est appelé plusieurs fois afin de constituer la page de vue
- Doctrine n'est pas appelé (en effet, nous n'avons pas fait appel à la base de données).

Bon tout cela est bien sympa mais ne nous dit pas quel élément est le plus critique et qui mérite des améliorations de performance. Pour cela, on fera appel à l'outil BlackFire qui est capable de mesurer plus en détail les performances.

II. Analyse de la Qualité du code avec Codacy

Introduction

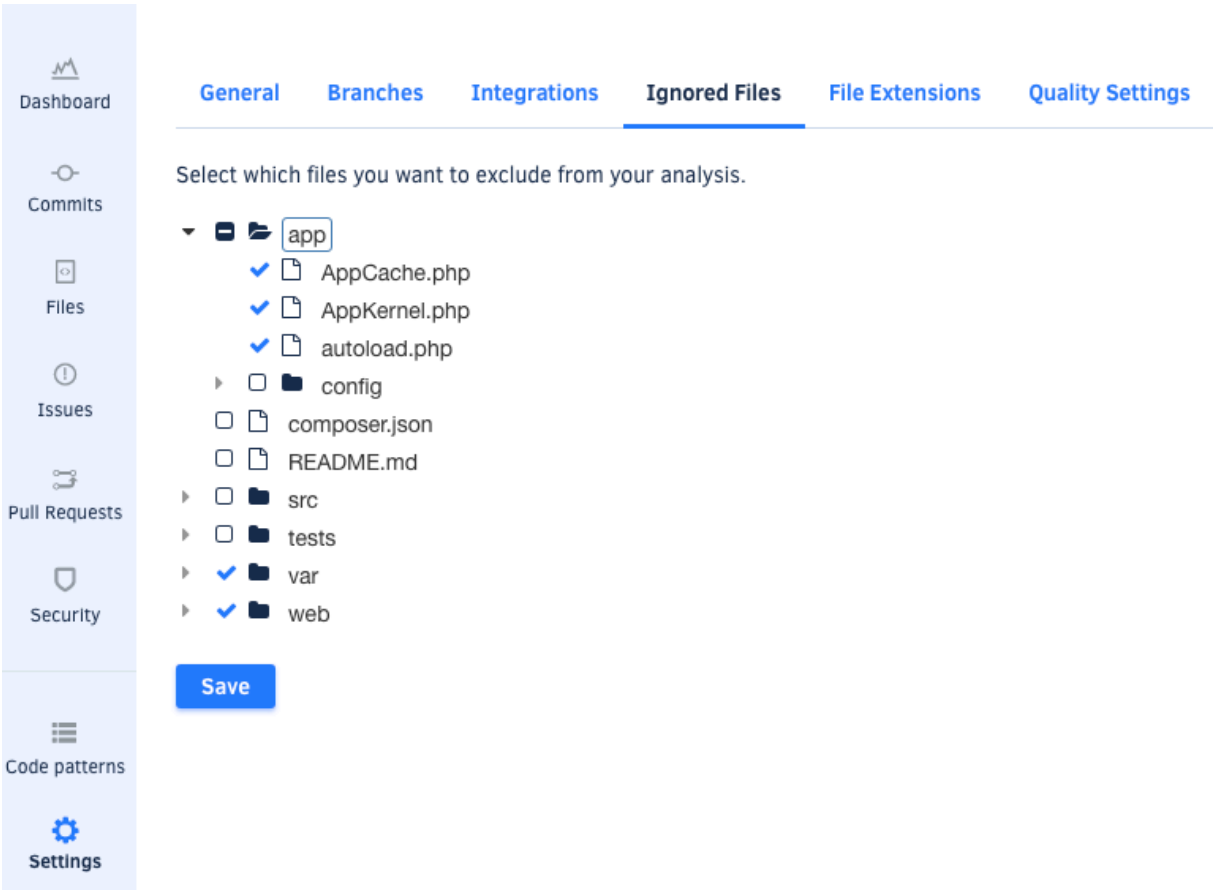
Codacy est un site qui va analyser le code des différentes branches choisies dans un repository Git.

Pour ajouter un repository à Codacy, ce n'est pas compliqué, il vous suffit de créer un compte avec vos comptes GitHub et ensuite d'indiquer à Codacy quel repository vous souhaitez analyser.

1. Bien paramétrer son repository pour analyse

Avant l'analyse, il est important de bien paramétrer ce que Codacy doit analyser. Il y a forcément des points du code qu'il n'est pas nécessaire d'analyser, comme des éléments auto-généré par Symfony ou des rapports de tests.

Pour cela, il faudra se rendre dans :



Les dossiers sélectionnés, sont ceux qui seront ignorés lors de l'analyse du code. Ici nous avons choisi de mettre de côté :

- Une partie du dossier App
- Le dossier var qui contient la majorité des fichiers de Symfony
- Le dossier web qui contient des ressources mais aussi le rapport de couverture des test

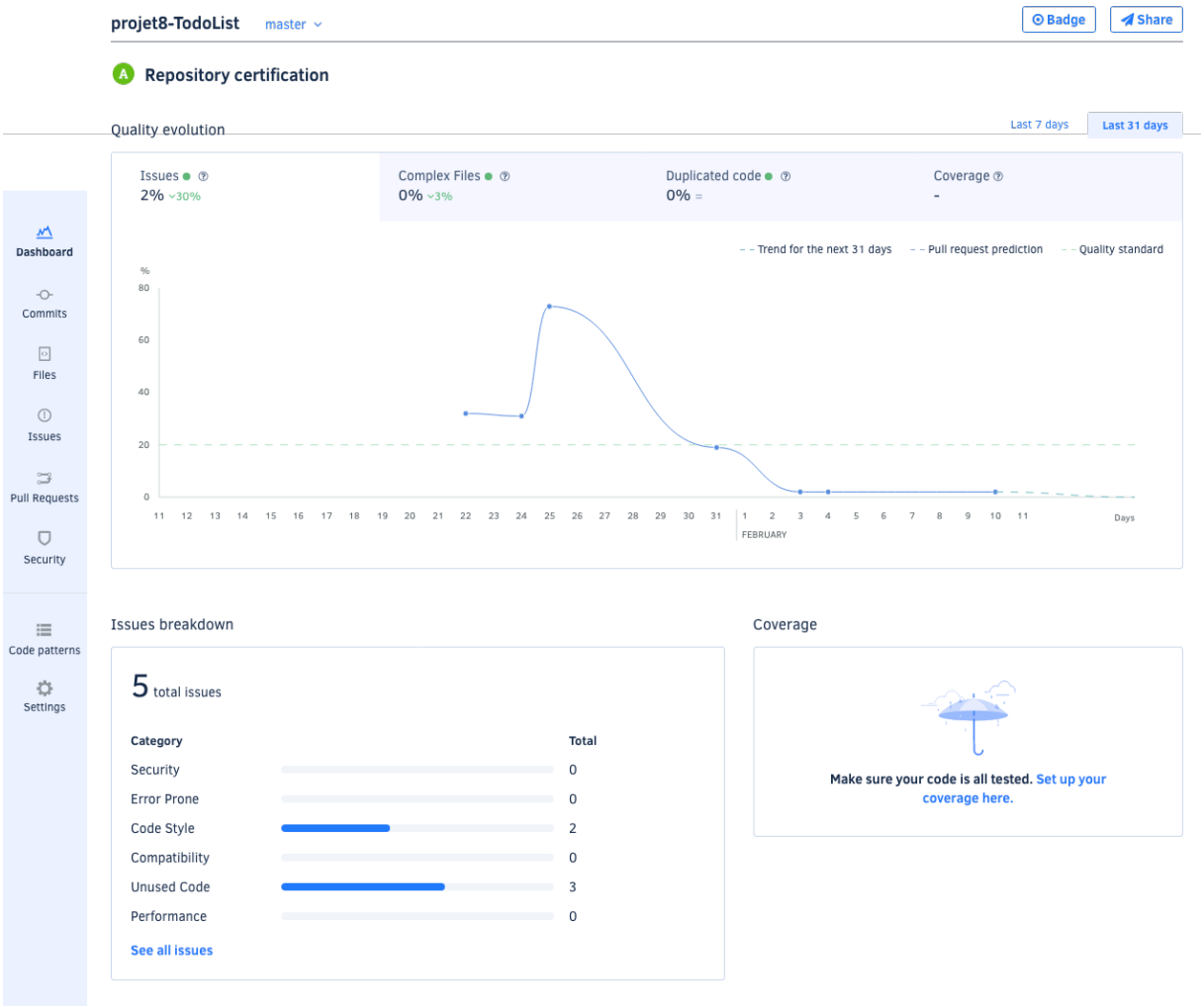
Si besoin, il est donc assez facile de modifier les différents éléments que l'on souhaite analyser.

2. Analyse du dashboard

Une fois le repository choisi, Codacy va vous présenter un dashboard avec différents indicateurs. On trouvera notamment :

- La certification du repository qui est ici la plus haute de « A »
- Un score d'issues avec une courbe de progression en fonction du temps.
- Un breakdown des issues par catégories
- Un score de complexité des fichiers
- Un score de code fait en doublon

Le dashboard se présente comme ci-dessous :



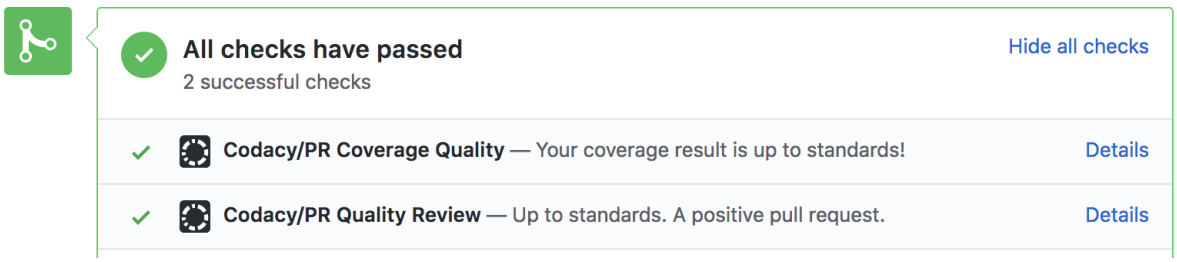
Ce que l'on peut voir sur ce dashboard c'est que le projet présentait au départ de gros problème de qualité avec un taux d'issue supérieur à 60%.

La majorité de ces issues ont été corrigé en quelques pull request, malgré tout on constat qu'il reste encore 5 issues qui pourrait être corrigé, qui sont liés au style du code (2) ou à du code non utilisé (3).

3. Les petits plus de Codacy

L'analyse automatisée de vos pull request !!

Et oui, fini les de pusher des pull request (PR) avec du code pénalisant. A chaque PR, sur GitHub, un contrôle de qualité est effectué comme la forme ci-dessous :



Cela nous permet donc en un coup d'œil de savoir si l'on peut pousser de manière confiante sans altérer la qualité du code.

Pour autant, Codacy ne pourra vous dire si votre code est bien fonctionnel. Il fonctionne uniquement comme un correcteur d'orthographe ou de grammaire.

III. Audit de Performance avec BlackFire

L'application sera amenée à évoluer rapidement, ce qui implique des contraintes d'augmentation du volume de données et du trafic.
Afin de pouvoir au mieux anticiper ces contraintes, les performances de l'application doivent être suivies tout au long du processus de développement afin de pouvoir au mieux les optimiser.

L'application web BlackFire permet d'optimiser les performances de nos projets web grâce à une lecture simple et rapide des requêtes que l'on effectue.

1. Installation

L'utilisation de BlackFire est gratuite seulement 15 jours, et ensuite demande un accès payant.

Pour installer BlackFire, il faudra simplement suivre les étapes indiquées ici : https://blackfire.io/docs/up-and-running/installation

Il faudra donc installer :

- L'Agent BlackFire et le configurer
- PHP Probe
- Profiling Client (il s'agit de l'extension Chrome ou Mozilla)

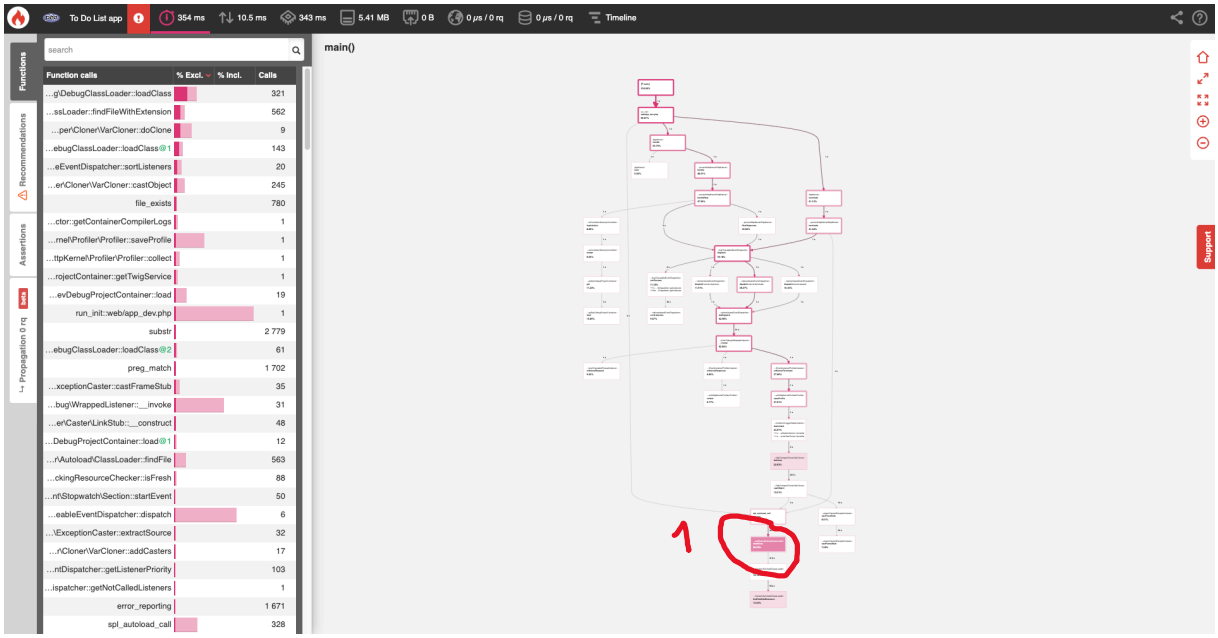
Une fois installée, il suffira simplement de cliquer sur l'extension pour pouvoir générer un rapport initial sur la requête ou page que l'on souhaite.

2. Rapport Graphique BlackFire

Le rapport se divise en 3 parties dont 2 majeures :

- En haut, la bande avec des KPI assez équivalente à celle de Symfony Profiler
- A gauche, la liste de toutes les fonctions appelés lors de l'exécution de la requête
- A droite, le graphique représentant l'exécution de la requête avec en surbrillance le chemin le plus critique

Ci-dessous un exemple de rapport produit pas BlackFire sur la requête d'accès à la page de login :

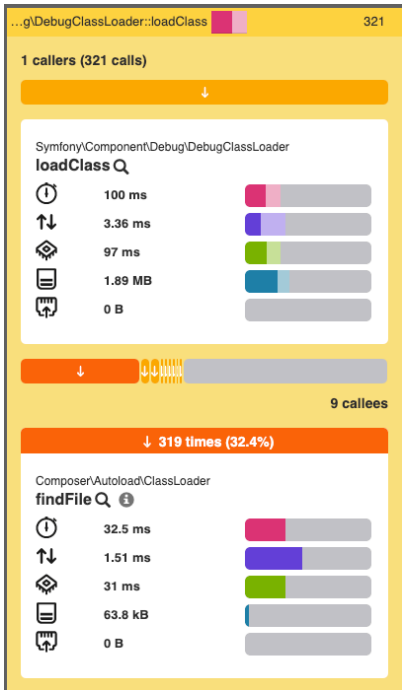


3. Analyse du rapport BlackFire

L'analyse du rapport est assez simple, car dans la liste à gauche, les fonctions appelées sont classées à partir de la fonction la plus chronophage.

Dans notre rapport, il se trouve que la fonction la plus critique est « loadClass » que l'on voit en rose foncée (point 1) sur le graphique.

Regardons donc plus en détail cette fonction :



En cliquant sur cette fonction, on peut voir qu'elle est appelée 321 fois et qu'elle fait appel à 9 fonctions différentes dont une qui prend un volume particulier.

Il s'agit de la fonction findFile dans Autoloader de Composer qui est appelé 319 fois et représente à elle seule une masse de 32,4% du délai de chargement de la fonction loadClass.

Il va donc falloir regarder comment faire pour réduire ces appels et améliorer la performance. Pour cela, rendons-nous directement sur la documentation de composer : https://getcomposer.org/doc/articles/autoloader-optimization.md

4. Optimisation de la performance & analyse des actions effectuées

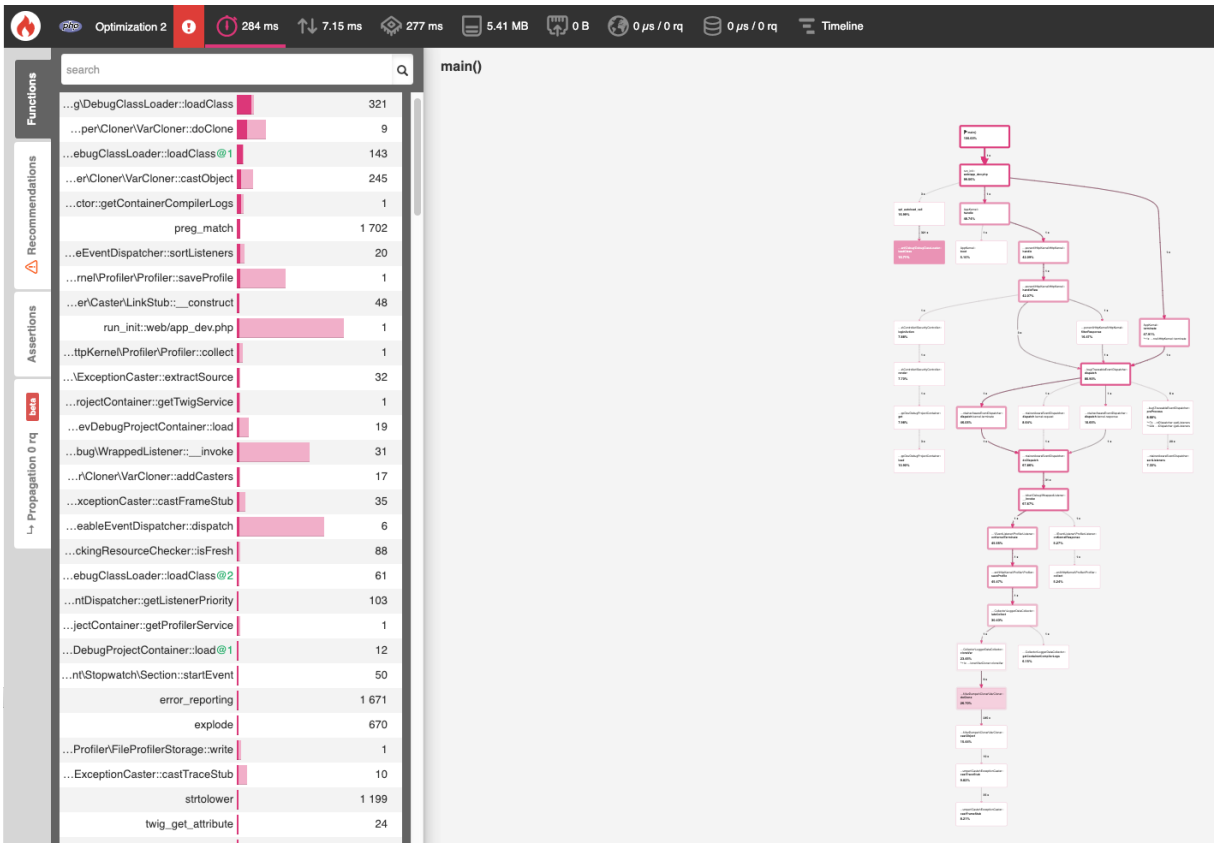
On voit qu'il existe 3 étapes pour améliorer la performance de composer. Nous allons effectuer la première étape d'optimisation et analyser s'il y a eu une amélioration.

Pour se faire, lançons la commande du premier niveau d'optimisation : « composer dump-autoload -o ». Cette commande a pour but de créer un fichier avec toutes les routes des class qui facilitera l'accès aux fichiers.

Regardons maintenant l'impact de cette commande sur la performance via :

- Nouveau graphique d'optimisation
- Détail de la fonction la plus lourde appelée
- Comparaison du rapport initial et du nouveau rapport d'optimisation

Tous ces rapports se génèrent grâce à BlackFire. Ci-dessous le nouveau graphique d'optimisation :



Dans un premier temps on peut voir plusieurs points :

- Le temps d'exécution global a diminué en passant de 354ms à 284ms
- L'architecture d'exécution est différente comme le montre le graphique
- La fonction la plus lourde reste inchangée (loadClass de la classe Debug)

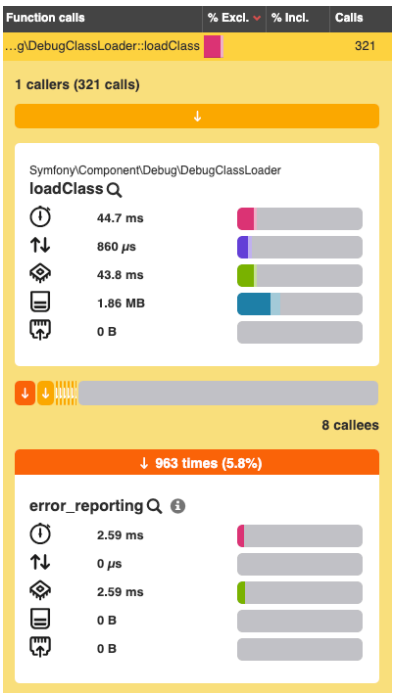
Regardons dans un premier temps les détails de loadClass pour voir si certains points ont changé.

En cliquant sur cette fonction, on peut voir qu'elle est toujours appelée 321 fois mais qu'elle fait appel à seulement 8 fonctions au lieu de 9.

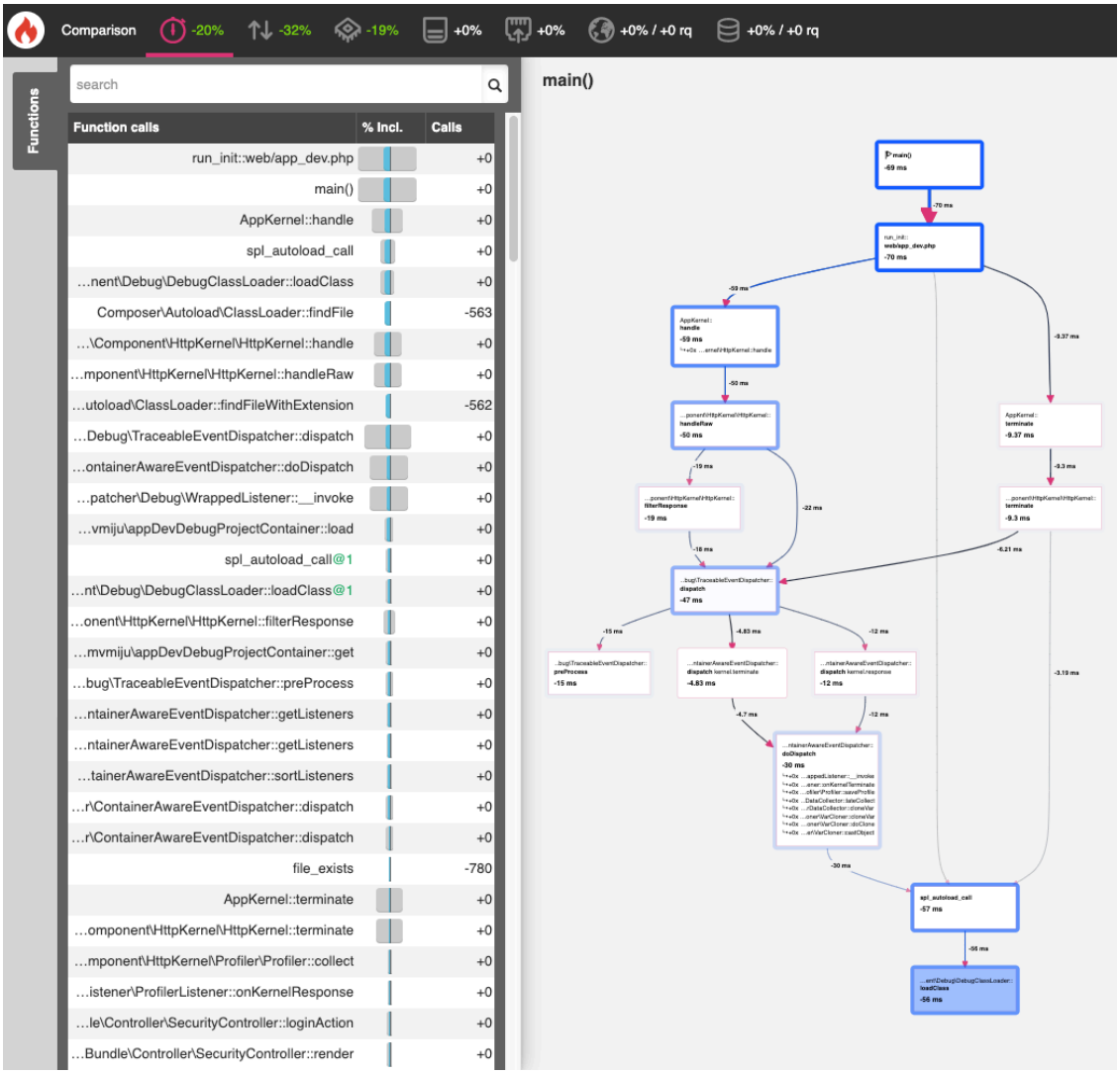
On note que l'exécution de cette fonction ne prend plus que 44,7ms au lieu de 100ms précédemment, ce qui est une amélioration importante.

Le temps d'exécution du CPU sur cette fonction a lui aussi été divisée par 2.

Enfin on constate que la fonction findFile de Composer n'est plus appelée ce qui explique assez bien ce gain de temps.



Maintenant jetons un coup d'œil à l'outil de comparaison de BlackFire qui va nous permettre de confronter les 2 rapports avant/après :



Tout de suite, on voit que l'ensemble des indicateurs dans la barre BlackFire en haut sont verts :

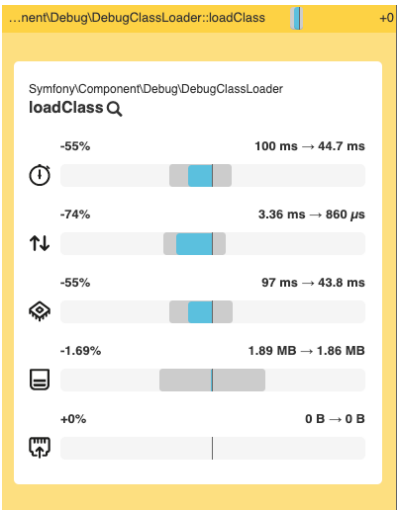
- -20% sur le temps globale d'exécution
- -32% sur les aller / retour des requêtes concernant la BDD et aussi des fichiers qui concernent la vue
- -19% de temps d'activité pour le CPU.

Tout cela est donc très positif.

Maintenant si on regarde la partie graphique, on voit qu'une des fonctions appelés est affichée avec un fond bleu. Ce code couleur correspond à la fonction qui a bénéficié de la plus grosse réduction. En la sélectionnant, on peut voir qu'il s'agit de la fonction loadClass de la classe Debug :

On voit qu'effectivement la réduction du temps d'exécution que l'on a souhaité optimiser a bien été impacté en priorité.

Dans la liste des fonctions qui sont présente dans le rapport de comparaison, il y a également la fonction findFile de Composer. On constate que celle-ci n'est plus appelée ce qui explique notre succès sur cette optimisation.



5. Pousser l'amélioration

Ayant une contrainte de temps, toutes les améliorations n'ont pas pu être effectuer, mais on peut déjà donner plusieurs points d'amélioration de performances possible :

- En production, le composant Profiler de Symfony ne sera pas activée, celui-ci est assez chronophage. Donc par défaut, cela signifie que l'application sera plus rapide en production
- Exécuter les étapes 2 & 3 d'amélioration recommandées par composer

Conclusion

Aujourd'hui l'application a une certaine stabilité mais convient d'être améliorée sur le plan sécurité et performance.

Concernant la sécurité, une migration vers la version la plus stable de Symfony est recommandée (Symfony 4.4) mais devrait nécessiter quelques adaptation du code.
Concernant la performance, un suivi du projet avec BlackFire serait recommandé dans un premier temps sur des pages qui font appel à de lourdes ressources. De plus des tests de performances peuvent être mis en place : https://blackfire.io/docs/cookbooks/tests.
Les tests conseillés doivent être fait sur le nombre d'appel et non sur le temps d'exécution.