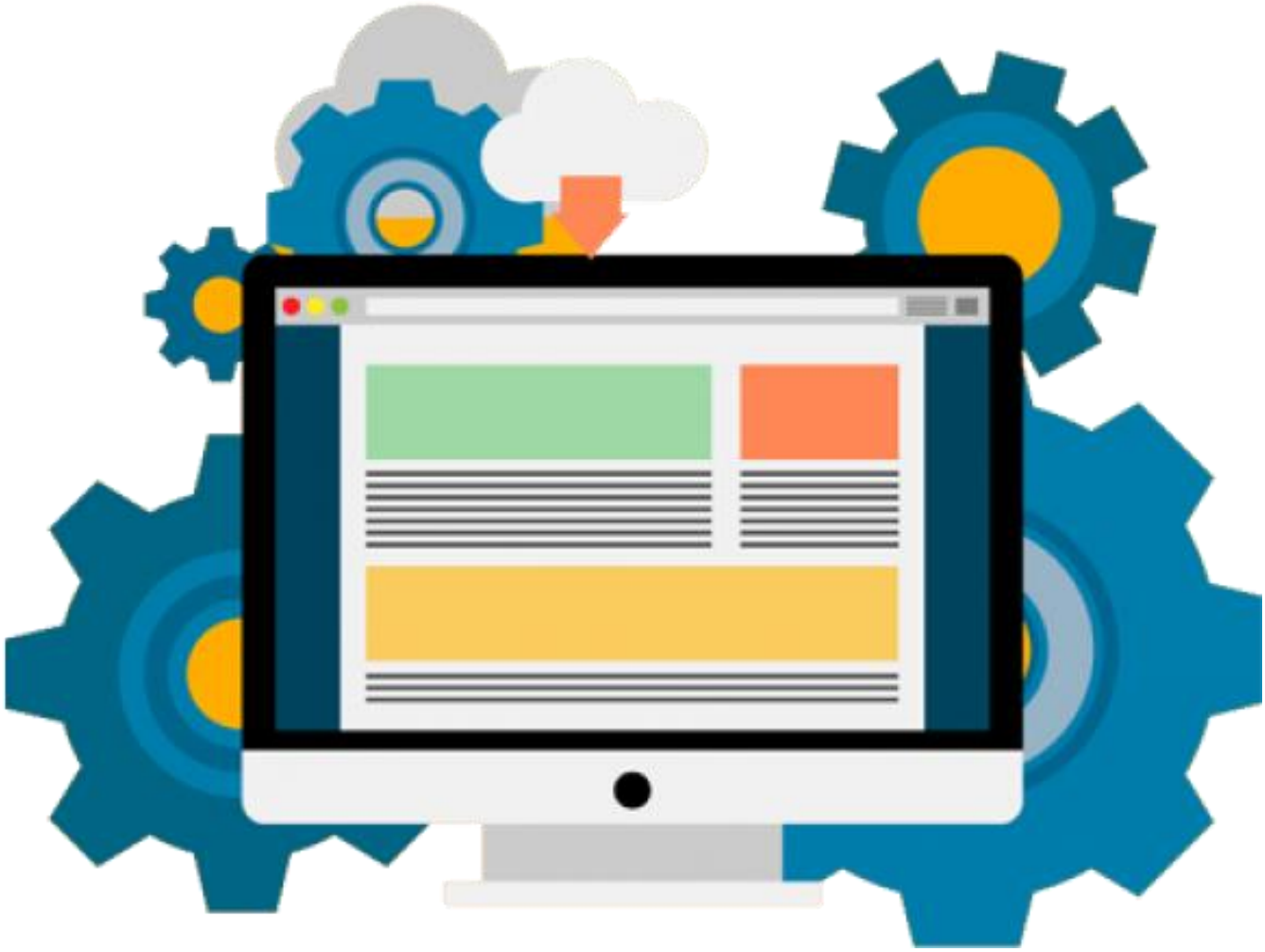
TODOLIST

Audit performance & qualité



Thomas LEFEBVRE

20/11/2021

Table des matières

.1 Avant-propos :

Le but de cet audit est de déterminer précisément, à l'aide des données obtenues via le profiler d'application Blackfire.io, la dette technique et les manques à gagner en termes de performance de l'application Todolist.

Ce rapport détaillera également toutes les améliorations apportées à l'application sur une branche git à part, qui sera mergeable sur la branche main dès qu'on le souhaitera, sans conflit et avec tous les tests unitaires passés.

Nous passerons en revue chaque point critique remonté par le profiler et proposerons, pour chacun, des solutions techniques adaptées et réalisables facilement pour combler au mieux ces failles de performance.

Une fois toutes les failles de performance auditées et solutionnées théoriquement avec un temps estimé de développement, cet audit proposera également un ensemble de mesure visant à améliorer la maintenabilité et réduire la dette technique sur le long terme de l'application, afin de garantir que son évolution sera pérenne pour un coût de développement réduit et des performances augmentées pour un ratio coût/efficacité très positif.

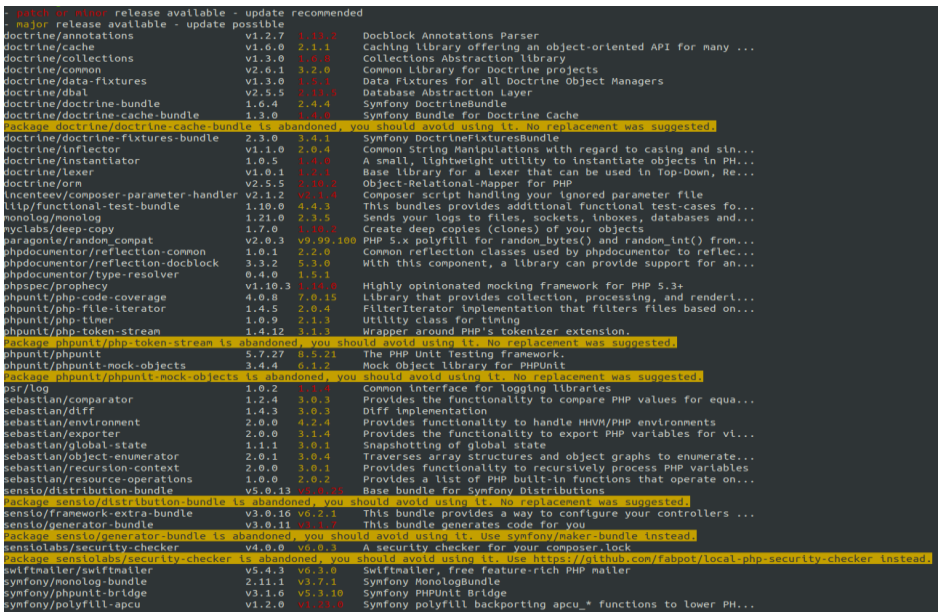
Il est a noté que les temps de chargement des écrans sont à minimiser dans un cadre de production du fait que la probe Blackfire.io met un certain temps à récolter les informations.

.2 État des lieux de l'application

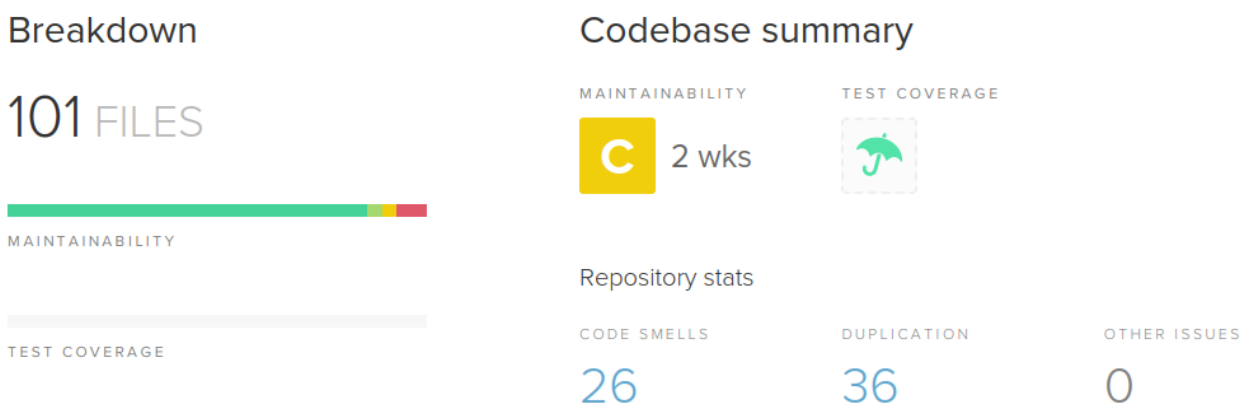
L'état de l'application à sa réception était le suivant :

- Version de php : 5.5.9
- Version de symfony : 3.1
- Dépréciation des packages :

On note que que plus de 30 packages sont largement déprécié et nécessitent des updates majeures. Certains ont même été abandonnés pour cette version de Symfony et il faudra leur trouver des remplaçants dans les versions ultérieures du framework.



- Analyse code climate :



L'analyse code climate montre une importante dette technique qui est principalement due à l'ancienneté de la version symfony et l'intégration de bootstrap, points qui seront à revoir par la suite.

.3 Améliorations de la qualité du code

.3.1 Configuration docker

La première amélioration apportée au projet lors de sa reprise a été de mettre en place un work flow moderne de développement en passant par une configuration Docker. Cela nous permet de changer de version de PHP, d'extensions et de réglages serveurs facilement, et de pouvoir faire profiter toute l'équipe de ces modifications avec un simple pull du projet.

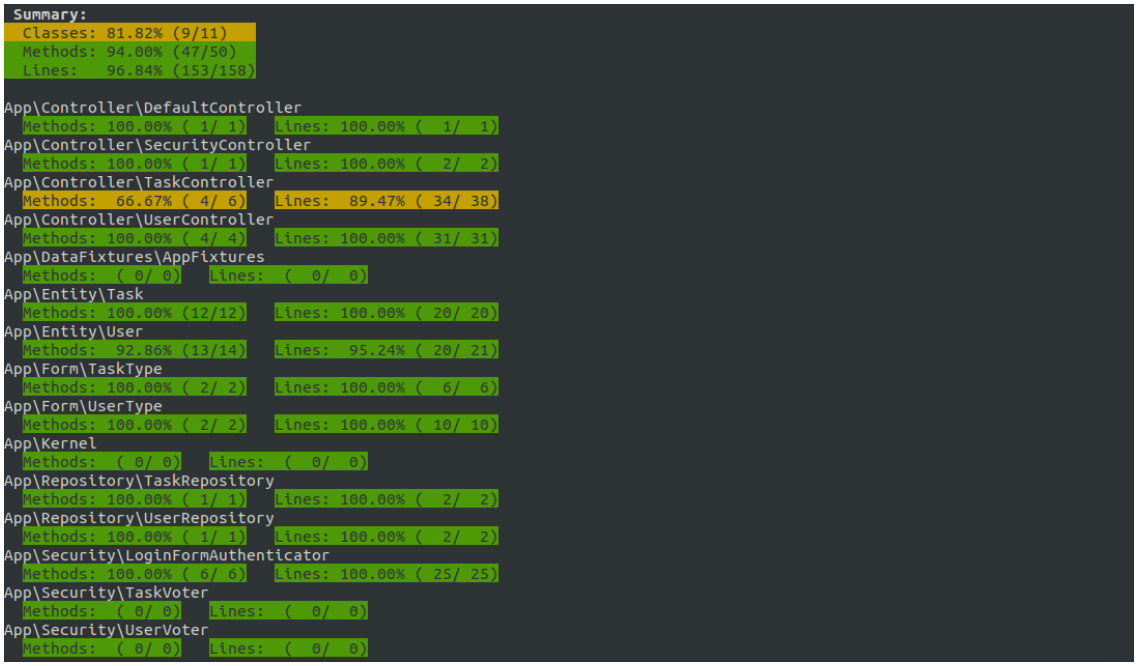
Dans l'optique de l'intégration de développement junior, c'est une pratique idéale pour retirer de la complexité lors de l'arrivée sur le projet, et un gain de temps considérable de mise en place de l'application pour tous les développeurs.

A la racine du projet on trouvera un docker-compose.yml qui décrit chaque image utilisée, un Dockerfile qui regroupe toute la configuration serveur, et un dossier php qui contient le vhost pour le

serveur apache. Il serait utile de prévoir une réorganisation des fichiers du projet afin de ne pas mélanger les dossiers de l'application de ceux de la configuration docker.

.3.2 Tests unitaires, fonctionnels et Fixtures

Ensuite, la mise en place de tests unitaires et fonctionnels a permis de maîtriser chaque aspect de l'application avec un code coverage de 80 à 96% avant tout nouveau développement.



Ce niveau de test ne devra pas descendre en dessous de 70 %. Seule la logique des controller est testée, ainsi que ce qui diffère du comportement natif de Symfony dans les entités. Il ne s'agit pas de tout tester, mais de couvrir les parties critiques de l'applicatif.

Dans l'optique de faire des tests efficaces en conditions réelles, un système de fixtures (données test) a été mis en place. Les données test sont reset à chaque test lancé, ainsi ils sont tous indépendant les uns des autres, et la mise en place de l'application s'en trouve facilitée.

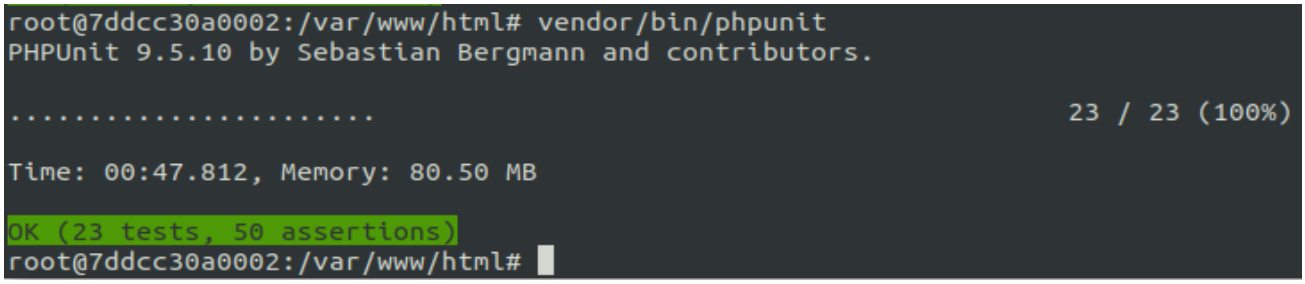
Deux types de tests ont été utilisés :

- Les tests unitaires :

Peu nombreux sur cette version de l'application du fait de sa simplicité coté entité.

- Les tests fonctionnels :

Ces tests reprennent tous les return et les throw des controllers pour tester chaque renvoie de chaque requête selon ce qu'on en attend.



Au total, ce sont 23 tests et 50 assertions qui ont été ajoutés et qui couvrent actuellement 100 % du code critique de l'application.

Du coté des fixtures, il ne s'agit pas d'un stress test donc le choix a simplement été fait de créer quelques entités de base User et Task, afin de pouvoir travailler sur les controllers pour les tests fonctionnels. Nous avons 5 Users possédant chacun 10 Tasks, dont la moitié sont marquées comme « faites ». 5 tasks sans User ont également été créés pour tester le rôle admin, qui seul peut gérer les Tasks sans User (user_id = null)

Se reporter au readme d'installation du projet à la racine du dépôt git pour les commandes lançant ces fixtures.

.3.3 Update symfony & bundles, version de php

Une fois les tests et les fixtures mises en place et après un passage obligatoire sur symfony 3.3 pour installer le bundle de fixture, les améliorations majeures à apporter se sont trouvées être du coté des versions de php, symfony et ses bundles.

Le choix a été fait, dans un premier temps, de passer sur la version LTS (long term support) de symfony, actuellement la 4.4. Cette version permet déjà un boost majeur coté stabilité, sécurité et performance de l'application, et nous rend possible une montée de php et des bundles suffisantes pour un premier état statisfaisant de Todolist.

Voici l'état dans lequel l'app se trouve sur la branche dédiée à l'audit et l'application des conseils de performance :

Version de php : 7.4

Version Symfony : 5.4 LTS

Bundles : Voir le composer.json, les versions sont à jour avec la 5.4

.3.4 Best practices et ajouts de code

.3.4.1 Templates des pages erreurs

Des templates propres à l'application ont été ajouté pour les erreurs 403, 404 et 500 afin de gérer ces cas proprement sans retour direct d'apache ou de symfony. Cela permet d'en montrer le moins possible sur la configuration ou les données sensibles aux utilisateurs.

.3.4.2 Voters

Dans l'écosystème Symfony, une pratique recommandée en ce qui concerne les droits d'accès aux ressources est de passer par un système de Voter. Ce système, rapide à mettre en place, permet de centraliser tous les droits d'accès à toutes les ressources selon les rôles, plutot que de les gérer par route, ce qui disperse les modifications à apporter en cas d'évolution. En l'occurence, dans src/Security, on trouvera un UserVoter qui gère les accès par rôle, et un TaskVoter qui gère l'appartenance des tâches et les droits de modification.

.3.4.3 EntityManager

En Symfony 4+, les interfaces permettent de nous faciliter la vie quand à l'utiliser des classes principales du framework. Pour gérer les entités, l'EntityManagerInterface est toute indiquée, et elle est désormais autowired dans tous les controllers qui en ont besoin depuis le consctruct. De ce fait, $this→em permet d'accèder à toutes les méthodes de gestion d'entité (comme persist ou flush) sans repasser par le chargement d'un container doctrine à la volée.

.3.4.4 Injection de dépendance

Idem que précédemment, toutes les dépendances sont désormais autowired dans la méthode ou le construct si réutilisées dans plusieurs méthodes, afin de ne pas surcharger le code intutilement avec de multiples appels aux containers.

.3.4.5 Symfony Flex

Flex est une amélioration majeure de la version 3,4, et il est fortement recommandé de passer sur ce mode de fonctionnement. Flex permet une automatisation de l'installation des bundles et de leur activation, une gestion des dépendances depuis config/bundles.php qui facilite la vision des bundles autowired, une structure moderne en MVC, et nous fait tout simplement passer dans le nouvel écosystème Symfony.

.3.4.6 Variables d'environnement

Flex permet également la reconnaissance des fichiers en .env, .env.local, qui permettent de versionner ou non des variables d'environnement plus ou moins sensibles ou propre à chaque environnement. On parle de la configuration base de donnée, d'un mailer, ou simplement de variables changeantes selon certains cas, des tokens JWT si l'app en contient, etc. Il est fortement conseiller de passer par un .env.local pour tout développement, et de ne pas versionner ce fichier.

.3.4.7 Validateurs sur Entités et formulaires

Symfony 3.4+ permet de rapidement valider ses formulaires et les entités qu'ils modifient à l'aide du composant Validator. Ce dernier rajoute des vérifications sur, par exemple, le type de champ, le nombre max de caractère, mais aussi l'unicité de tel ou tel champ en base de donnée, le type de fichier uploadé, etc.

.3.4.8 Stratégie de cache : opcache, apc doctrine, etc

Afin de tester les performances de l'application en condition plus proche d'une production, il a été mis en place un système de cache à plusieurs niveaux :

- Un cache http via le composant Symfony. Ce composant n'est pas fait pour de la production, mais il aide à se représenter et à mesurer les performances en conditions plus réelles, et à tester le cache HTTP de l'app.
- Un cache doctrine apc via le composant Symfony pour accélérer la récupération de data.
- Un cache php (opcache) pour optimiser les performances du langage.

Les résultats se trouvent dans la partie comparatif de cet audit, page 10.

.3.4.9 Naming des methodes

Les noms des méthodes ont été updatés pour y retirer le terme « action » qui est déprécié si l'on veut strictement respecter les normes actuelles.

.3.4.10 Authenticator Custom

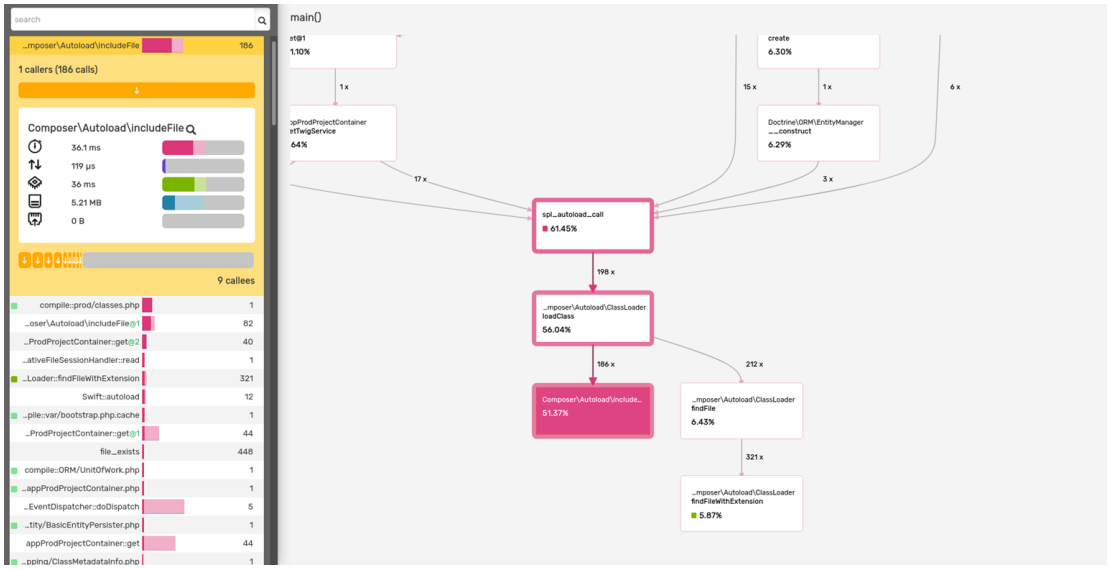
Dans une optique d'amélioration de la sécurité de l'app et de réduction du futur temps de développement en cas de personnalisation de ce composant (ce qui arrive souvent), une authentification custom a été mise en place selon les best practices Symfony 5. On notera également le rajout d'un token csrf au login.

.4 Les principaux nœuds de performance

Todolist est une application pour le moment relativement simple et il est d'autant plus important que ses fondations soient les plus solides possibles pour supporter des évolutions régulières sans perdre en performance. Garder des point de passage dans l'application qui posent problème en terme de temps de traitement, soit à cause d'une fonction, soit de ce qu'elle appelle, installe la suite du développement dans un goulot d'étranglement se resserrant de plus en plus vers un état critique qui ne sera plus acceptable pour un utilisateur.

Voici les différentes points, écran par écran, remontés par Blackfire.io :

.4.1 Autoload composer



Problème :

Le problème principal de cet écran est une classe de Composer, **includeFile,** ici en couleur rose, qui sert à appeler les vendors via l'autoloading. Ce que montre le graph c'est que, en l'état, le principal goulot d'étranglement de performance est cet autoload, qui prend un walltime de 36ms (et autant de temps coté CPU) sur l'ensemble de la requête pour afficher l'écran. Il requiert également une mémoire non négligeable de 5Mo par requête.

Ce n'est pas à optimiser dans la partie du code développé par l'équipe, puisque Composer est un gestionnaire de dépendance externe à Symfony et qui n'est pas remanié par le développement, mais ça reste très facile à faire pour un gain de performance significatif.

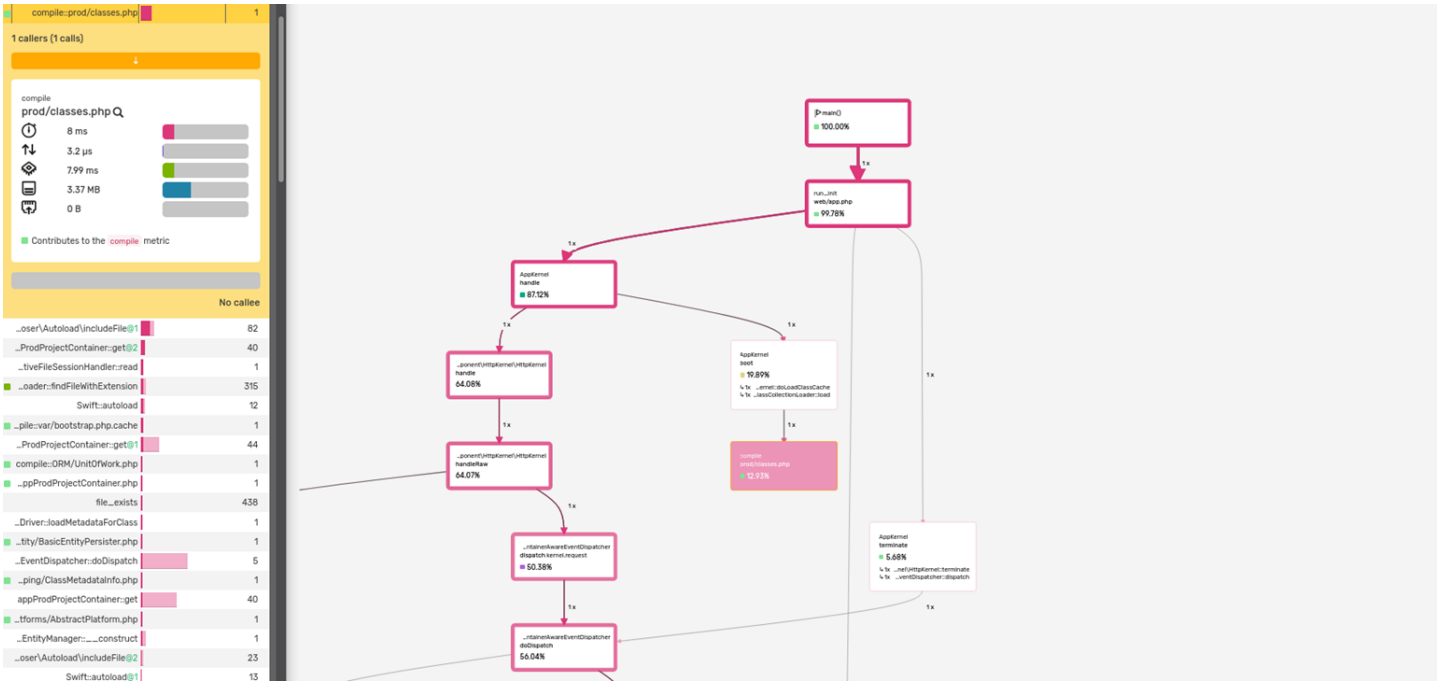
Solution :

La solution est très simple à appliquer et commune à tous les environnements php gérés via composer. Dans un contexte de production et pendant le déploiement d'une application, afin d'optimiser le chargement de toutes ces ressources via l'autoload, composer dispose d'une commande à lancer pour optimiser automatiquement ses classes :

composer dump-autoload –optimize

Cette commande permet de dire à composer de passer en cadre de production et de passer par un mode d'autoloading plus éfficace. Cela aura pour effet de réduire le walltime d'environ 10 %

.4.2 prod/classes.php



Problème :

Le deuxième point névralgique problématique de cet écran home est le chargement d'un fichier php **classes.php** appellé par le Kernel. On remarque que, depuis le kernel, deux fichiers sont appellés :

- Component\HttpKernel\Kernel::doLoadClassCache

- Component\ClassLoader\ClassCollectionLoader::load

Ce qui veut dire que si ce que le Kernel cherche à charger ne se trouve pas en cache, il va recharger toutes les ressources nécessaires (token de sécurité, variable d'environnement, paramètres router, etc).

Solution :

Il faudra bien s'assurer que le système de cache en production est activé et optimisé afin que ces classes se trouvent toujours en cache. Lors de tous les déploiements, on peut lancer les commandes suivantes :

bin/console cache:clear -e prod

bin/console cache:warmup -e prod

On pourra également mettre en place un cache HTTP avancé de type Varnish. Après la mise en cache de ces ressources, Blackfire.io ne reportera plus ce fichier. On ajoutera que dans les recommandations du profiler, une stratégie de cache des annotations doctrine est proposée :

```
# app/config/config_prod.yml
doctrine:
  orm:
    metadata_cache_driver: apc
```

Elle permet de réduire le temps des requêtes SQL en donnant accès au mapping des entités à Doctrine via un cache plutôt que la database à chaque requête. Il ne faut pas s'attendre à des résultats trop importants, mais ça participera d'une amélioration globale de l'application.

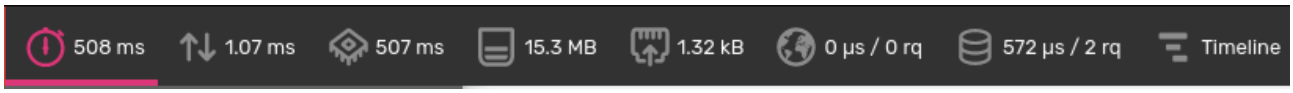
.5 Comparatif de performance avant/après

Après avoir identifié les problèmes majeurs, il a été facile de reprendre les principales routes pour les vérifier et ainsi obtenir un comparatif avant/après des solutions proposées. Ce faisant, nous avons des données précises qui montrent le gain de performance après notre intervention :

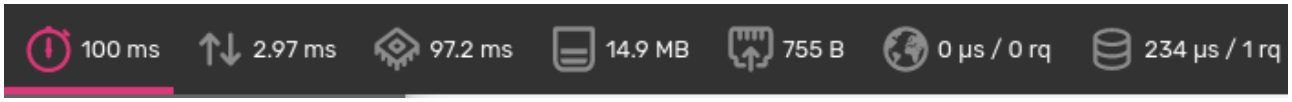
.5.1 État de l'application route par route avant intervention :

.5.1.1/login :

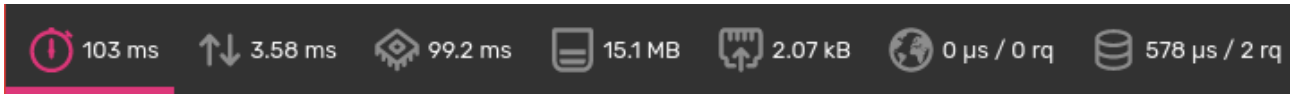
.5.1.2/login_check



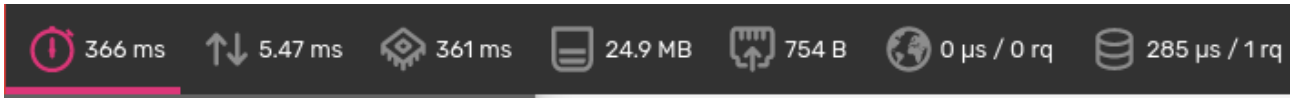
.5.1.3/app.php



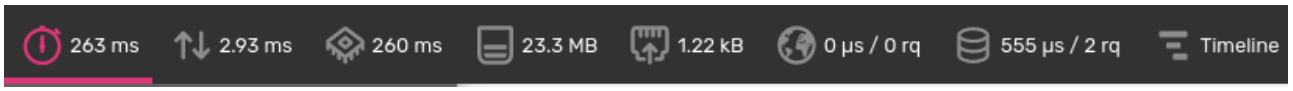
.5.1.4/tasks



.5.1.5task/create



.5.1.6user/create



.5.2
5.2 État de l'application après solutions proposées :

.5.2.1/login :

Soit une diminution du temps de chargement de 87%

.5.2.2/ :

Soit une diminution du temps de chargement de 90%

.5.2.3/tasks :

Soit une diminution du temps de chargement de 92%

.5.2.4tasks/create

Soit une diminution du temps de chargement de 95%

.5.2.5/users/create

Soit une diminution du temps de chargement de 97%

On notera que grâce au cache ACPU, les requêtes Doctrine ne repartent pas à chaque appel de la route, ce qui fait gagner un temps considérable lors du chargement, et limite la surcharge du serveur en cas de forte affluence, rendant l'app scalable sur un très grand nombre d'utilisateurs. Le gain de RAM est également massif, et le stockage trois fois moins encombré en moyenne.

.6 Corrections d'anomalies selon le CDC

Deux correctifs principaux ont été demandés par le PO afin de stabiliser l'état de l'application du coté utilisateur :

.6.1 Rôles User/Admin et Accès

Dans un premier temps, la notion de rôles était critique pour faire fonctionner Todolist correctement. Il a été demandé d'implémenter deux rôles : user et admin. Ces rôles déterminent les droits d'accès à certaines pages. En l'occurrence :

- */user/** : admin only

.6.2 Relation User/Tasks

Ensuite, il a fallu que chaque utilisateur n'interagisse qu'avec les Tasks qu'il possède uniquement. Pour ce faire, nous sommes passés par un lien oneToMany entre User et Tasks, afin de n'afficher via la query doctrine que les taches de l'utilisateur de la session. De plus, un Voter Symfony vérifie que le user a les droits suffisant sur l'entité affichée.

.6.3 Ajout d'une liste d'utilisateurs

Dans l'optique de mieux gérer les utilisateurs, en tant que rôle Admin, une liste des utilisateurs avec une option d'édition a été ajoutée sur la route */users*

.7 Solutions supplémentaires

Pour continuer d'améliorer l'application et d'anticiper sur sa future dette technique ou de performance, il serait dès à présent utile de mettre en place :

- Un cache varnish et/ou redis en cas de forte augmentation des tasks par user
- Passer par Webpack encore et sass pour la partie front, ainsi que moderniser le rendu css pour l'acceptabilité coté utilisateur
- Ajouter un système dragNdrop pour la gestion des tâches par catégorie
- Se séparer de bootstrap ou monter en version 5 sans jquery pour gagner en performance coté client
- Mettre en place un système de pagination de ressources
- Continuer la migration vers php 8

- Mettre en place des tests automatisés via github CI, travis, coveralls
- Installer un système devops de livraison continue via Capistrano
- Penser à une web progressive app pour la partie mobile qui représentera la majorité du trafic de l’app
- Installer phpstan level 7 pour faciliter l’intégration de junior et le suivi des best practices php