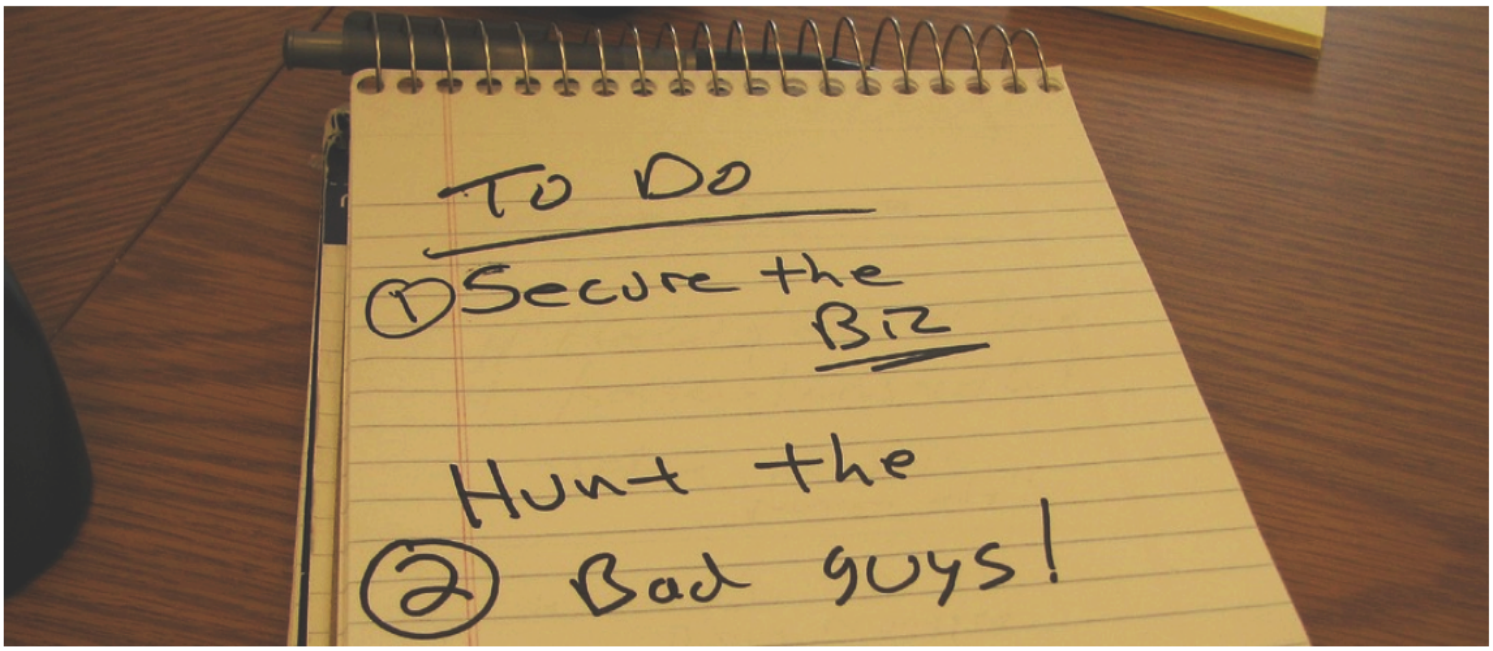
To Do List app

Créer un utilisateur

Se déconnecter

Bienvenue sur Todo List, l'application vous permettant de gérer l'ensemble de vos tâches sans effort !



Créer une nouvelle tâche

Consulter la liste des tâches à faire

Consulter la liste des tâches terminées

TODO-LIST

Audit de qualité et de performance

Section 1 : Audit de qualité

1.1 Approche

L'approche globale de cet audit de qualité du code de l'application Todolist se déroulera de la manière suivante:

- Analyse de l'existant
    - Qualité du code et des bonnes pratique (utilisation d'un outil de revue de code +revue de code manuelle)
    - Revue de la sécurité
    - Revue sur l'expérience utilisateur
- Classification des changements à effectuer
    - Changement court terme
    - Un point sur la version de l'application

1.2 Analyse préliminaire

Comme indiqué dans la section "*1.Approche*", cette analyse préliminaire se déroule en 2 parties. Analyse automatique en utilisant "Codacy" pour faire un tour d'horizon de l'application et en ressortir les éléments à améliorer.
Ensuite en se basant sur les bonnes pratiques du framework Symfony, de l'UI etc… nous allons faire un premier bilan

1.2.1 Analyse Codacy

Après une première revue, voici les éléments qui en sont ressortie:

	Style du code	**Securité**	**Code non utilisé**
Item repertoriés	27	8	6
En Warning	4	4	6

À titre d'information	23	0	0

Note: les retour sur le code Symfony sont marqués comme => "Framework"

Style du code:

Severité	Pattern	Nombre d'items	Classes concernés
Information	Nom de variable trop court	12	3 => **FeatureContext** 2 => **TaskAllocationContext** 3 => **UserContext** 1 => **TaskController** 1 => **UserController** 1 => **Framework**
Information	Éviter les accès direct à des méthodes statiques	8	1 => **FeatureContext** (test end to end) 7 => **Framework**
Information	Eviter les "Else" statement dans le code	2	**TaskAllocationContext**
Information	Non respect de la convention de nommage pour les méthode boolean	1	**Task**
Warning	Complexité cyclomatique (nombre de point de décision i.e. trop de if)	2	**Framework**
Warning	Longueur de méthode excessive	1	**Framework**
Warning	Complexité Npath (nbre possible d'exécution d'une fonction i..e comporte trop de chemin possible)	1	**Framework**

Sécurité:

Severité	Pattern	Nombre d'items	Classes concernés
Warning	Échapper certaines valeurs	4	**Framework**

Code non utilisé:

Severité	Pattern	Nombre d'items	Classes concernés
Warning	Argument de fonction non utilisé	3	1 => **SecurityController** 1 => **TaskType** 1 => **UserType**
Warning	Variable non utilisé	2	1 => **FeatureContext** 1 => **TaskAllocationContext**
Warning	Attribut non utilisé	1	1 =>**TaskControllerFunctionalTest**

1.2.2 Analyse manuelle

En ce qui concerne la revue manuelle (revue de code, fonctionnelle etc…), pour ce qui est la revue du code, les points suivants sont remontés:

Revue de code:

Le code devrait de manière générale être conçu autour des services, Symfony le permet très facilement à l'aide de la configuration dans le dossier service.yml ou par injection de dépendance. J'ai effectué un petit changement au niveau du contrôleur de façon à externaliser le control des utilisateurs connectés (la tâche peut être supprimé que par le user qui l'a créé etc…). voici quelques avantages à utiliser les services:

Utilisation des services

- Permet de pouvoir réutiliser l'objet ailleurs plus facilement

- Le maintien du code est plus facile
- Permet une reprise du code par un autre développeur plus facilement
- Augmenter les performances de l'application
- Un code plus clean et plus agréable à maintenir

Revue fonctionnelle:

Expérience utilisateur

- Il manque une indication/ un bouton pour revenir à l'accueil
- Pas de bouton pour la liste des utilisateurs enregistrés sur l'application
- Pas de bouton pour accéder à la page où gérer les utilisateurs de l'application
- Dans la page user pas de bouton/ fonctionnalité pour supprimer un utilisateur
- Lors de la modification d'un utilisateur, le mot de passe doit obligatoirement être changé (hors quand on modifie un utilisateur on ne veut pas forcément changer son mot de passe)

Sécurité:

Control d'accès

- Les utilisateurs anonymes peuvent créer des tâches ou même les modifiés => amélioration apporté, seul les utilisateurs de l'application peuvent effectuer des actions sur les tâches.

Paramètre "Secret":

- Dans le fichier "parameters.yml" , le paramètre "secret" est laissé avec une valeur par défaut. Hors celui-ci devrait être changé (par une série , symbole et/ou chiffres d'au moins 32 caractères), pour entre autre encrypter les cookies pour la fonctionnalité "remember me", ou pour l'ESI (Edge Side Include) etc...

1.3 Point d'amélioration rapide

En amélioration rapide, peuvent être fixé les point suivants:

Scope	Items	Estimation
Correction Codacy	Tous les points relevés (hors Framework)	2 jours (correction + adaptation des test automatiques etc…)
Revue du code	Re-factoré le controller Ajouter des services Créer des méthodes dans les repository	5 jours (correction + adaptation des test automatiques etc…)
Revue Fonctionnelle	Tous les points	2 jours (correction + adaptation des test automatiques etc…)
Securité	Tous les points	1 (correction + adaptation des test automatiques etc…)

1.4 Migration du framework

L'application étant sous une version de Symfony 3.1 (plus supporté), l'upgrade vers une version LTS (Long Time Support) était de priorité majeur (car la version utilisée jusqu'à présent n'était plus maintenue depuis Juillet 2017).
Rester en version non maintenue signifie

- Pas de bug fix
- Pas de fixe de sécurité.
- On ne profite pas des améliorations sur le framework

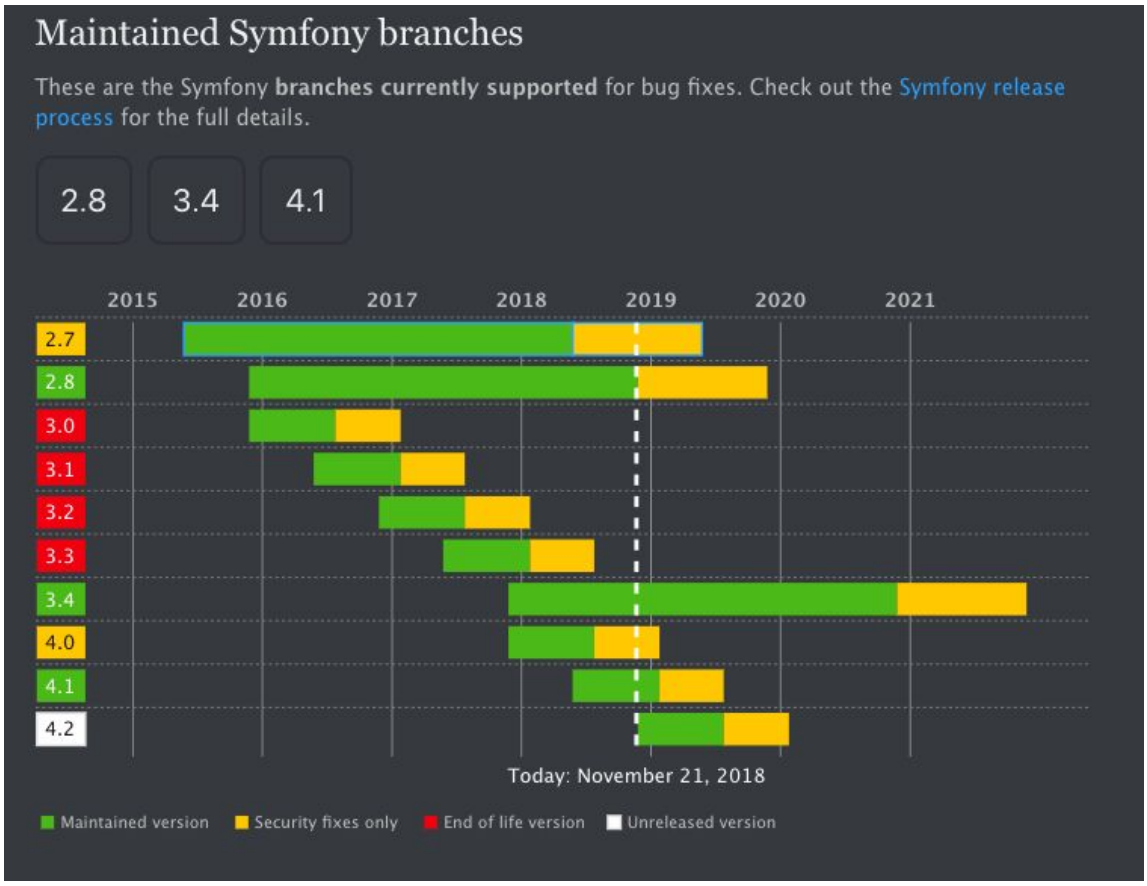
Cependant, de façon à mitiger le risque de changer de version du framework (rétro-compatibilité, obsolescence de certaines méthodes utilisées etc…) et donc de ne pas mettre l'application en péril dans le temps imparti, un upgrade vers une version mineur était plus sage. La version LTS de la série 3 est la **3.4.**

- Maintenue jusqu'à Novembre 2021
- Pas de changement drastique dans la structure du code
- Peu d'obsolescence à corriger (lien sur l'approche de Sensiolab en ce qui concerne la retro comptabilité https://symfony.com/doc/3.4/contributing/code/bc.html)
- Avantage d'avoir migrer vers une version stable

Nous voyons effectivement qu'avec la feuille de route de l'équipe Symfony, le support de la version sera effectuée jusqu'à 2021.

Note: *Cependant quelque partie mineur de l'application devenait un peu obsolète, tout à été corrigé (voir détails section 4. Annexes =>"Deprecated corrigé aprés migration V3.4")*

ROADMAP:



1.5 Nouveau processus d'implémentation

Comme indiqué dans le document de collaboration, une série de contrôles sont mises en place de façon à réduire la dette technique de l'application, en résumé ce processus consiste à (pour plus de détails, lire la documentation de collaboration) :

- S'assurer de la non régression
    - Lancer les test automatiques existant avant déploiement

- Implémenter de nouveaux test automatique (sur les nouvelles fonctionnalités (approche intéressante comme le Test Driven Development ou Behavior Driven Development)

- Établir une culture de faire du code de qualité
    - part l'utilisation de best practice,
    - l'utilisation constante d'un outil de revue du code dans le processus de livraison etc...

- Intégrer tout ce processus dans le cycle de vie de livraison, en les intégrants aux estimations données au utilisateurs

Section 2 : Audit de performance

2.1. Organisation de l'audit de performance

2.1.1 Introduction general

Afin de conduire cet audit sur la performance actuelle de l'application, nous allons utiliser l'outils "Blackfire" qui nous permettra de profiler toutes les pages de "Todolist". Celui-ci nous fournira des données afin d'identifier les points d'améliorations.

Dans les grandes lignes, voici la liste des mesures rapportées par Blackfire, ainsi que les fonctionnalités que nous utiliserons:

Metrics:

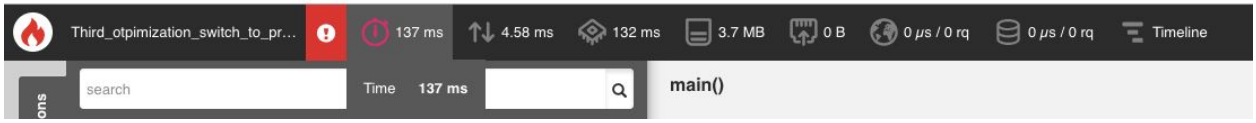
- **Wall Time** => le temps global que PHP prend pour exécuter le code (les instruction exécutée processée par le CPU, le volume de donnée en mémoire, le temps de réponse des services appelés sur le réseau etc…)

- **CPU time** => Le temps de réponse du processeur (temps que va mettre le processus I/O et le réseau)

- **I/O time** => Ce sera le temps de réponse du réseau (base de donnée, requête HTTP pour web service etc…) et/ou la lecture sur le disque

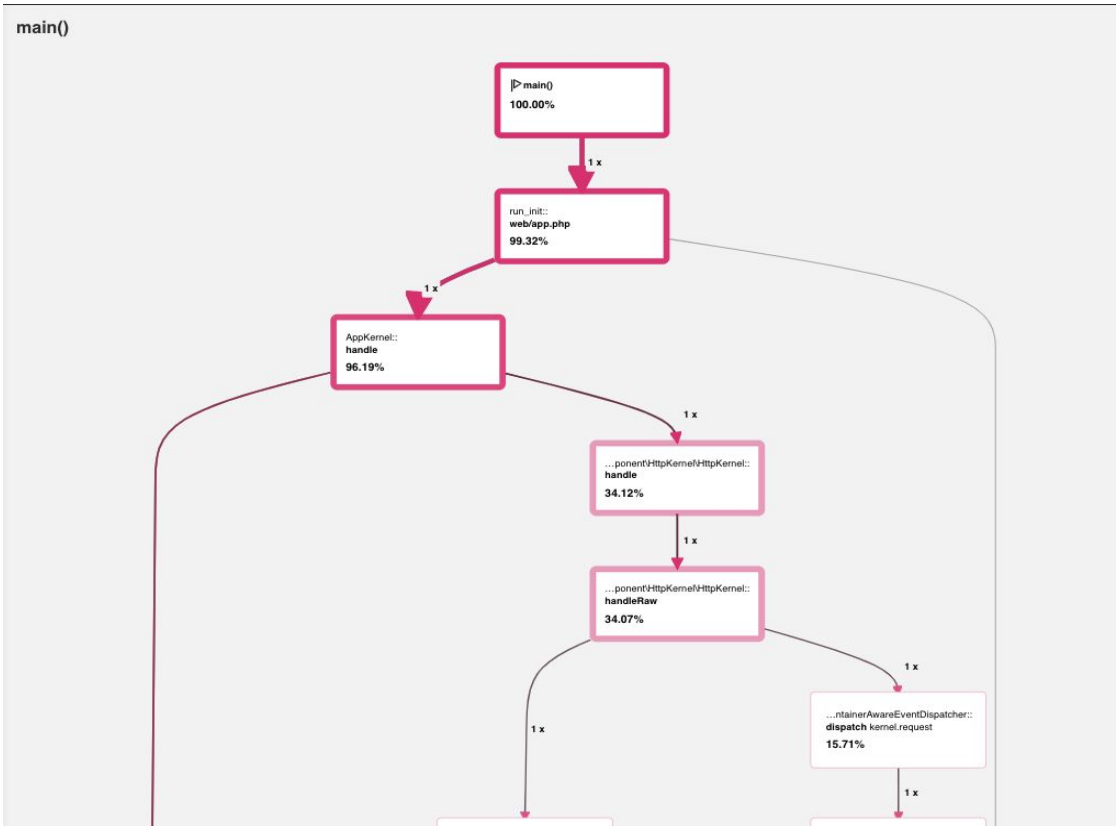
- **MB** => la mémoire utilisé par PHP pour générer la page

- **Network** => temps de transfert de data sur le réseau



Le graphique:

- Ceci est la représentation de l'exécution du script avec le nombre d'appels entre les fonctions (les appelés et les appelants etc…)



Note: Une timeline est aussi disponible mais pour ma part elle est moins lisible que le graphe

L'option compare:

- Cette option est importante pour mesurer l'évolution de la performance (positive ou négative)

Note: Le temps (dans le contexte de la performance de l'application) est finalement composé de plusieurs paramètres. Donc pour ce qui est de notre audit, toutes les metrics liés à la machine comme le CPU, l'I/O, le réseau (si local ou en remote avec transfert d'un gros payload de plusieurs mb) etc.. viennent en complément puisqu'on ne peut pas se baser exclusivement sur ces données la, vu que selon la capacité de la machine celles-ci peuvent être sensiblement différentes. Cependant, nous devons nous pencher sur les aspects suivants ,

- Le nombre d'appels d'une fonction (si trop élevé il y a peut-être moyen d'améliorer le code)
- Quel est le temps inclusive (temps d'exécution d'une fonction utilisant d'autres fonctions) et exclusif (temps d'exécution seul de la fonction)
- Le nombre de requêtes sql etc…

Important: la partie généralement très coûteuse en temps est le front end, chargement des images, du js etc… Plus on améliore les performance d'une page qui load par exemple, plus on a de chance d'avoir un impact sur le visiteur (il ya une corrélation directe entre le gain en temps de performance et le chiffre d'affaire d'une société ayant pour but la vente en ligne). Pour une vue globale (même si le but de cette exercice est plutôt un profil backend avec blackfire, cela servira de base de travail au développeur front end), un profile de chaque page a été effectué avec l'outil de "Lighthouse" qui est un plugin google chrome(qui se base sur les performances, l'accessibilité, les best practices etc…), les résultats sont plutôt moyen est beaucoup de choses peuvent être améliorés, la totalité des audits peuvent être consultés, tous les fichiers pdf sont dans le zip sauvé dans le même répertoire que ce document.

2.1.2 déroulement de l'audit

L'audit de performance se déroulera en 3 étapes:

- Analyse de la situation actuelle => ceci nécessite donc de prendre un snapshot de la performance actuelle de l'application, qui nous servira de base de départ pour l'optimisation

- Optimisation de la performance

- Comparaison après toutes les améliorations possible

2.2 Audit de performance

Pour chaque page de l'application, un profil sera établis.

2.2.1 Page d'acceuil:

Before:

Wall time	I/O	CPU	Memory	SQL Queries
335 ms	5.36 ms	330 ms	3.51 MB	

Graphe disponible à l'url suivante "**AS_IS_HomePage**":

https://blackfire.io/profiles/5f8fe35c-46dd-41d9-a625-acd166bd961f/graph

After:

Wall time	I/O	CPU	Memory	SQL Queries
53.6 ms	1.47 ms	52.1 ms	1.74 MB	

Graphe disponible à l'url suivante "**AfterOpt_HomePage**"":

https://blackfire.io/profiles/ed4656a8-822d-4821-b574-458732ef906b/graph

2.2.2 Page listant les taches:

Before:

Wall time	I/O	CPU	Memory	SQL Queries
454 ms	11.9 ms	442 ms	4.33 MB	2.48 ms/1rq

Graphe disponible à l'url suivante "**AS_IS_ListTasks**":

https://blackfire.io/profiles/b87e6460-8f41-4fb2-a876-b9fa69fe1ebc/graph

After:

Wall time	I/O	CPU	Memory	SQL Queries
94.2 ms	5.97 ms	88.3 ms	2.71 MB	4.72 ms/1rq

Graphe disponible à l'url suivante "**After_Opt_ListTask**":

https://blackfire.io/profiles/a8d44f1e-cf65-48a7-ae65-67c68bd2d8e5/graph

2.2.3 Page pour créer une tâche:

Before:

Wall time	I/O	CPU	Memory	SQL Queries
509 ms	6.7 ms	502 ms	5.01 MB	

Graphe disponible à l'url suivante "**AS_IS_CreateTask**"::

https://blackfire.io/profiles/90515942-aaa6-4131-a92d-1dcc973b7368/graph

After:

Wall time	I/O	CPU	Memory	SQL Queries
109 ms	7.42 ms	102 ms	3.32 MB	

Graphe disponible à l'url suivante "**After_Opt_CreateTask**":

https://blackfire.io/profiles/28e031a9-2fdc-4bcd-a49e-d28a7c342447/graph

2.2.4 Page pour editer un tâche:

Before

Wall time	I/O	CPU	Memory	SQL Queries

551 ms	14.6 ms	536 ms	5.53 MB	1.31 ms / 2rq

Graphe disponible à l'url suivante "**AS_IS_EditTask**":

https://blackfire.io/profiles/662ddb0a-e5ce-46a5-8a06-8e0bdd4a9b31/graph

After

Wall time	I/O	CPU	Memory	SQL Queries
130 ms	5.44 ms	125 ms	3.62 MB	1.1 ms / 2rq

Graphe disponible à l'url suivante "**After_Opt_TaskEdit**":

https://blackfire.io/profiles/45614788-50d6-406c-979c-432c86224cda/graph

2.2.5 Page de login

Before:

Wall time	I/O	CPU	Memory	SQL Queries
369 ms	14.2 ms	355 ms	3.53 MB	

Graphe disponible à l'url suivante "**AS_IS_LoginPage**":

https://blackfire.io/profiles/e0790ec3-b3de-4e62-8ee2-befcce2e8c1d/graph

After:

Wall time	I/O	CPU	Memory	SQL Queries
56.8 ms	1.51 ms	55.3 ms	1.76 MB	

Graphe disponible à l'url suivante "**After_Opt_Login**":

https://blackfire.io/profiles/00255caa-aa52-4380-85e9-b2c1e85b797e/graph

2.2.6 Page pour créer un utilisateur:

Before

Wall time	I/O	CPU	Memory	SQL Queries
563 ms	12.1 ms	551 ms	5.41 MB	4.41 ms/1rq

Graphe disponible à l'url suivante "**AS_IS_CreateUser**":

https://blackfire.io/profiles/bfb57103-d5c7-4386-9e8d-594fe5a1ab18/graph

After

Wall time	I/O	CPU	Memory	SQL Queries
129 ms	5.24 ms	124 ms	3.68 MB	489 micro s/1rq

Graphe disponible à l'url suivante "**After_Opt_UserCreate**":

https://blackfire.io/profiles/299e9947-3d3f-463b-b7d8-3bbd1c55b035/graph

2.2.7 Page pour lister les utilisateurs:

Before

Wall time	I/O	CPU	Memory	SQL Queries
427 ms	9.42 ms	418 ms	4.26 MB	1.51 ms/ 2rq

Graphe disponible à l'url suivante "**AS_IS_ListOfUsers**":

https://blackfire.io/profiles/3b243e80-60ca-48d5-8b66-2e58914e9c5c/graph

After

Wall time	I/O	CPU	Memory	SQL Queries
88.3 ms	6 ms	82.3 ms	2.79 MB	985 ns/ 2rq

Graphe disponible à l'url suivante "**After_Opt_ListOfUsers**":

https://blackfire.io/profiles/767a3d7b-45f6-4663-91cc-ecd75af356db/graph

2.2.8 Page pour modifier un utilisateur:

Before

Wall time	I/O	CPU	Memory	SQL Queries
629 ms	12.8 ms	616 ms	5.47 MB	335 ns

Graphe disponible à l'url suivante "**AS_IS_EditUser**":

https://blackfire.io/profiles/cc6dfd17-a584-4ca7-bf3f-dfb2a5249852/graph

After

Wall time	I/O	CPU	Memory	SQL Queries
144 ms	7.4 ms	137 ms	3.7 MB	624 ns

Graphe disponible à l'url suivante "**After_Opt_UserEdit**":

https://blackfire.io/profiles/a993d923-9444-449a-850d-7b434b1b67f4/graph

2.3 Amélioration apportée

En vue de la comparaison entre le profil de l'application avant et après changement, on voit clairement la différence (tableau ci-dessous récapitulatif du gain de performance).

Composer:

Dans la première évaluation de l'application, pratiquement la moitié de la consommation des ressources étaient attribué au chargement des classes avec l'autoloader de composer.

La façon la plus simple d'optimiser ce chargement est de déclarer les classes (non répertoriées) dans la **class_map** utilisée par composer, en utilisant la commande suivante "*composer dumpautoload -o*".

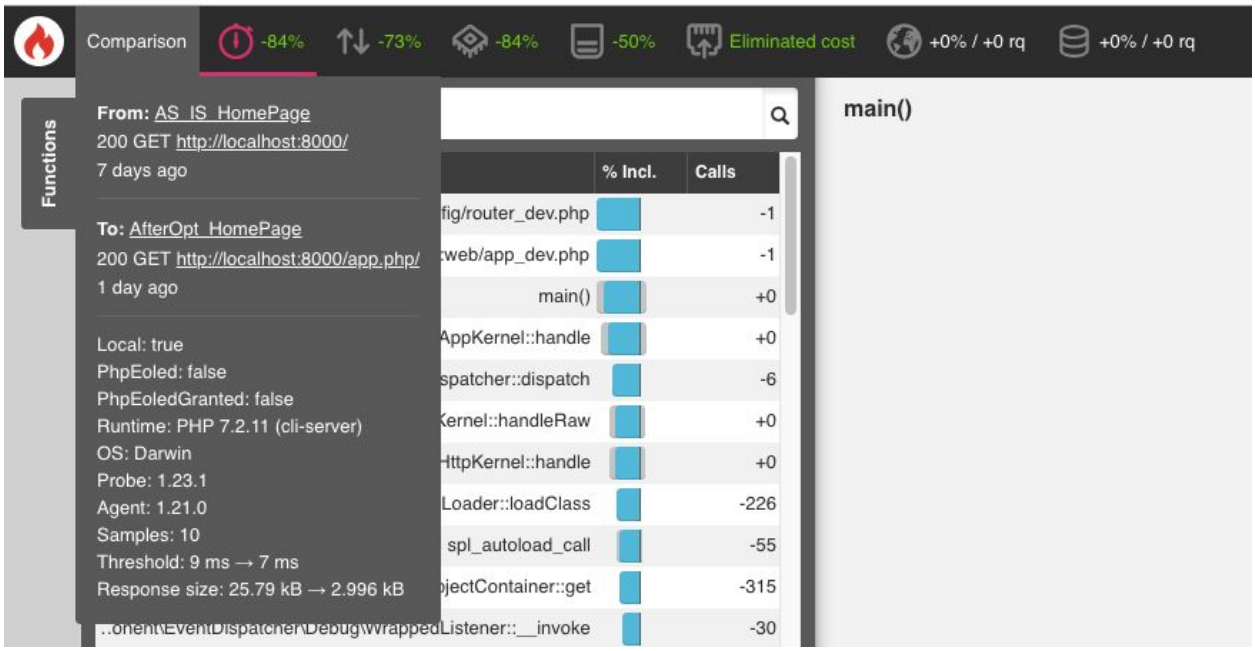
<u>Passage en production:</u>

L'environnement du framework que l'on utilisait jusqu'à présent était celui de développement, qui comparé à la production est bien plus lent (comme indiqués ci-dessous). L'environnement de production de Symfony est nativement optimisée pour être performantes, ceci dit comparé à l'environnement de développement, on peut voir entre autres:

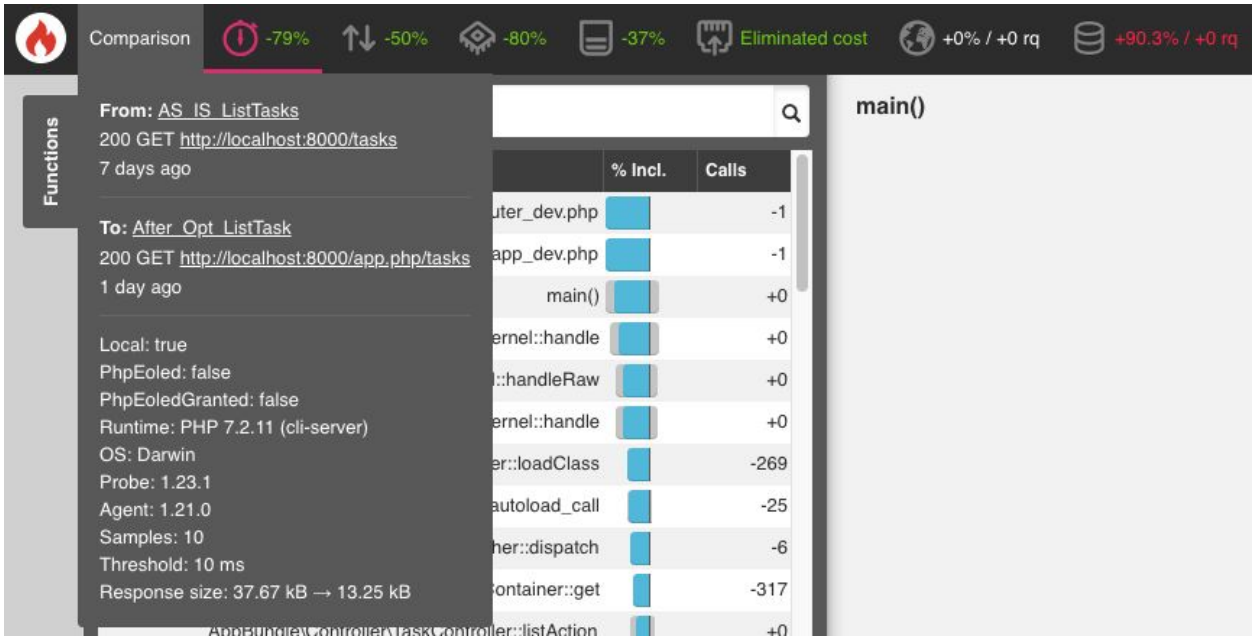
- Pas de web profiler à loader
- Moins de choses reporté dans logs (config_dev.yml => monolog / level => debug, alors que pour l'environnement de production, seul les erreurs sont loguées) etc…

En terme de comparaison, voici les profils établis comparant l'avant à l'après changement, et le gain de performance:

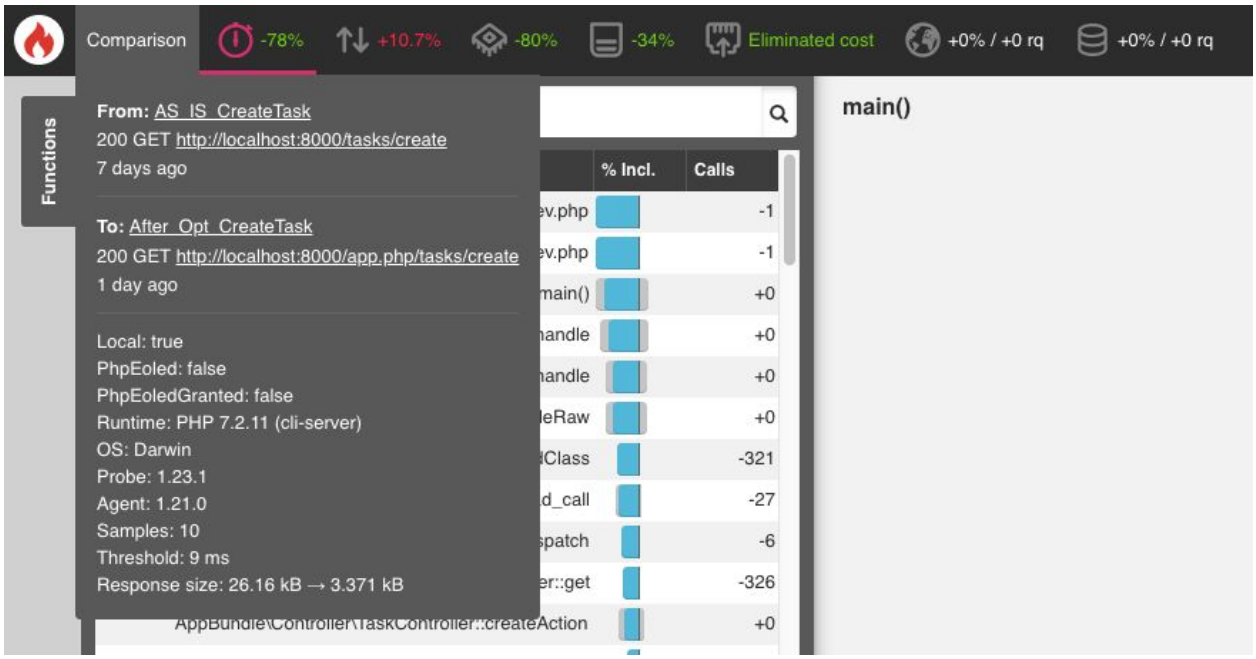
Page d'accueil:



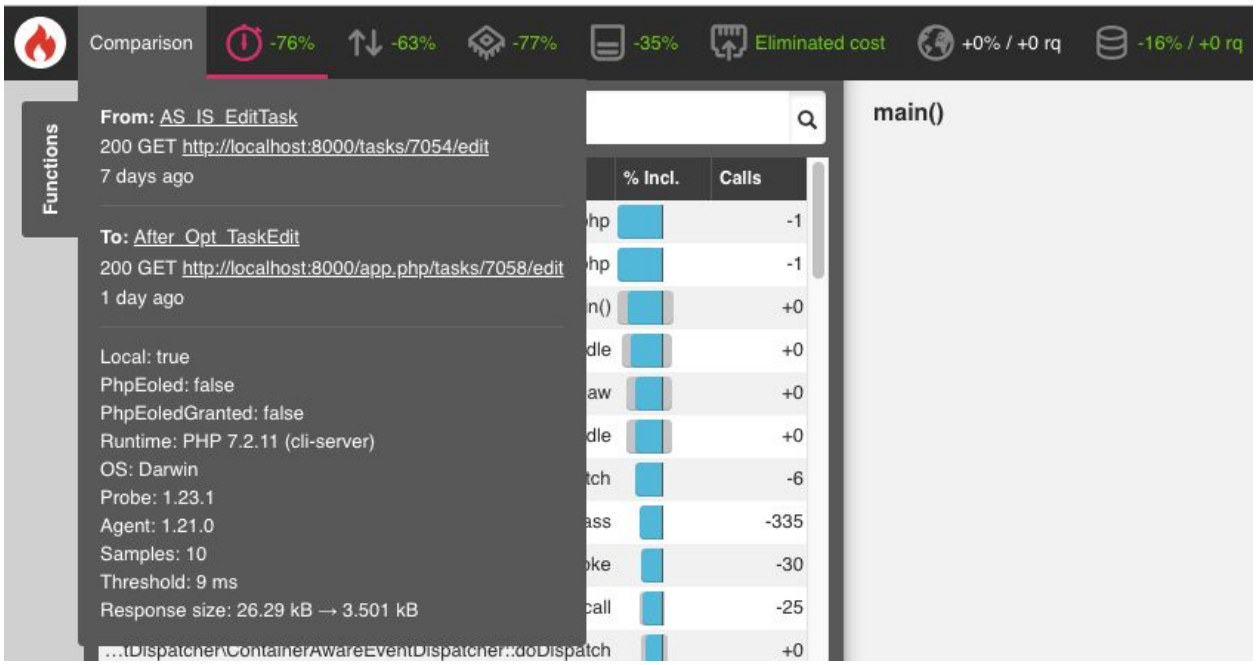
Page listant les taches:



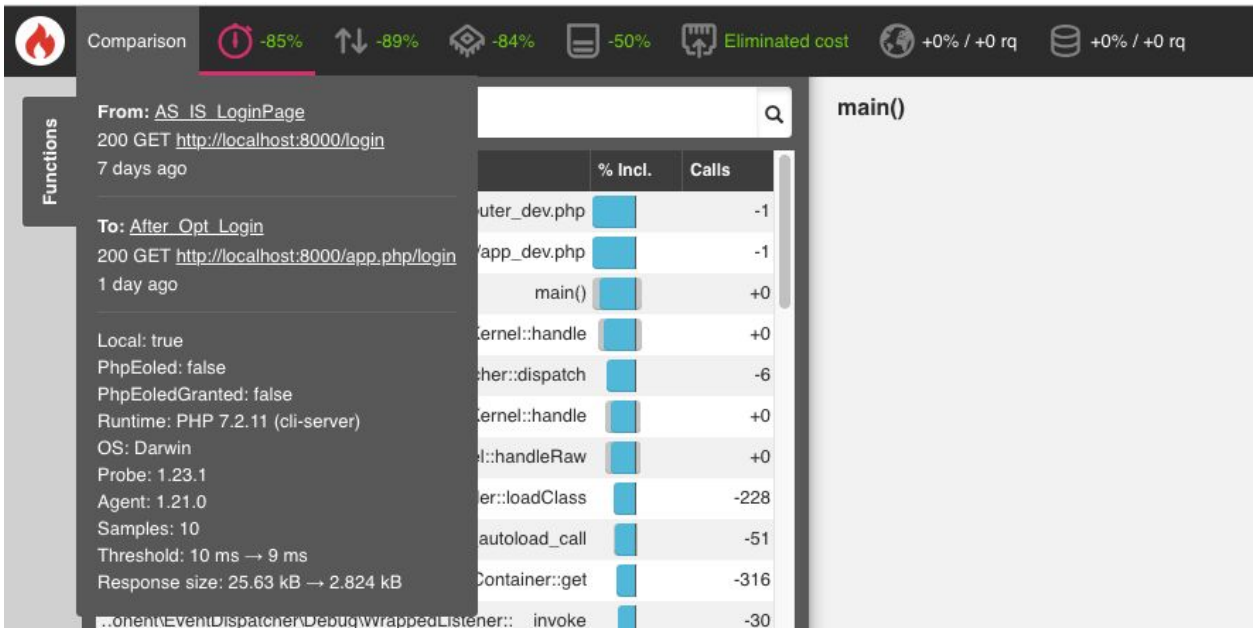
Page pour créer une tâche:



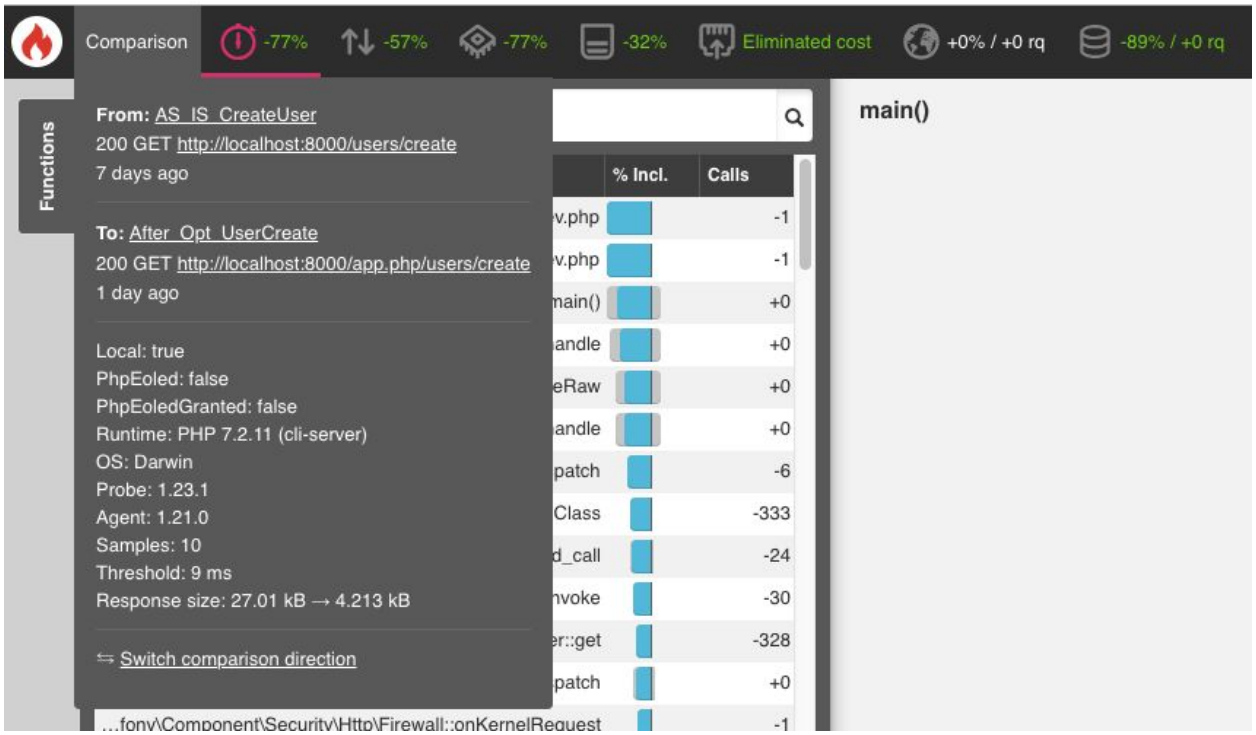
Page pour éditer un tâche:



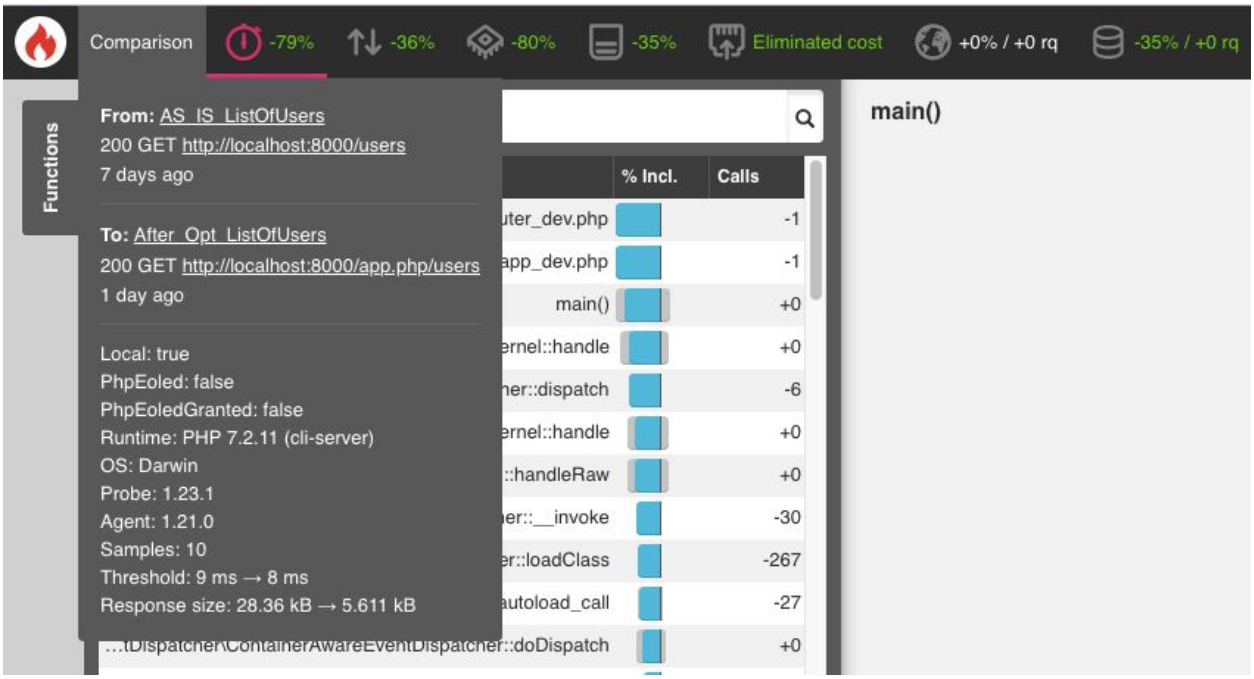
Page de login:



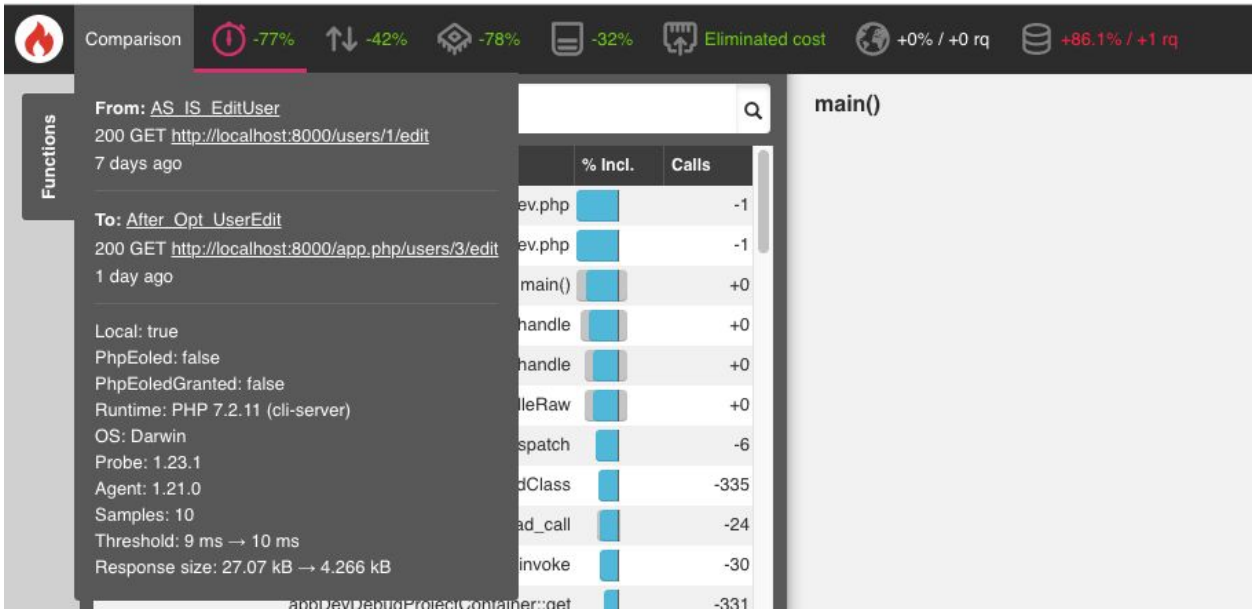
Page pour créer un utilisateur:



Page pour lister les utilisateurs:



Page pour modifier un utilisateur:



2.4 Next Step

D'autres actions peuvent être mise en place rapidement au niveau du serveur web de façon à accélérer l'application, en voici certaine (proposé dans la documentation de Symfony voir *section 3.Annexe* "*Performance Symfony*") à mettre en place:

En résumé, dans le php.ini section => *"**Zend OPcache**"*,

on peut augmenter la mémoire utilisé par OPcache, afin de stocker plus de fichiers compilés, et le nombre maximum de fichier stocké dans le cache

opcache.memory_consumption=256
opcache.max_accelerated_files=20000

En production, les fichiers ne changent pas (hors nouveau déploiement) cependant pour ces fichiers, mise en cache, aucune vérification n'est nécessaire, donc le paramétrage du control du timestamp par OPcache peut être mis à "0"
opcache.validate_timestamps=0

Note: après chaque changement en dev et chaque déploiement, il faut vider les cache (voir documentation, *section 3. Annexes "Performance Symfony"*).

Alloue plus de mémoire pour les chemins transformés (de relatif à absolu)
realpath_cache_size=4096K (par défaut sur la version actuellement utilisé *7.2.11*)
.
Sauve le résultat pendant 10 minutes:
realpath_cache_ttl=600

D'autres optimisation peu coûteuses sont tout à fait possible tel que l'implémentation du cache de "Doctrine ORM" (mise en cache de la transformation du DQL en SQL, du résultat retourné etc…), d'un reverse proxy comme celui de "Symfony", ou encore la mise en place de "Varnish" (reverse proxy) pour encore plus d'optimisation.

3. Conclusion

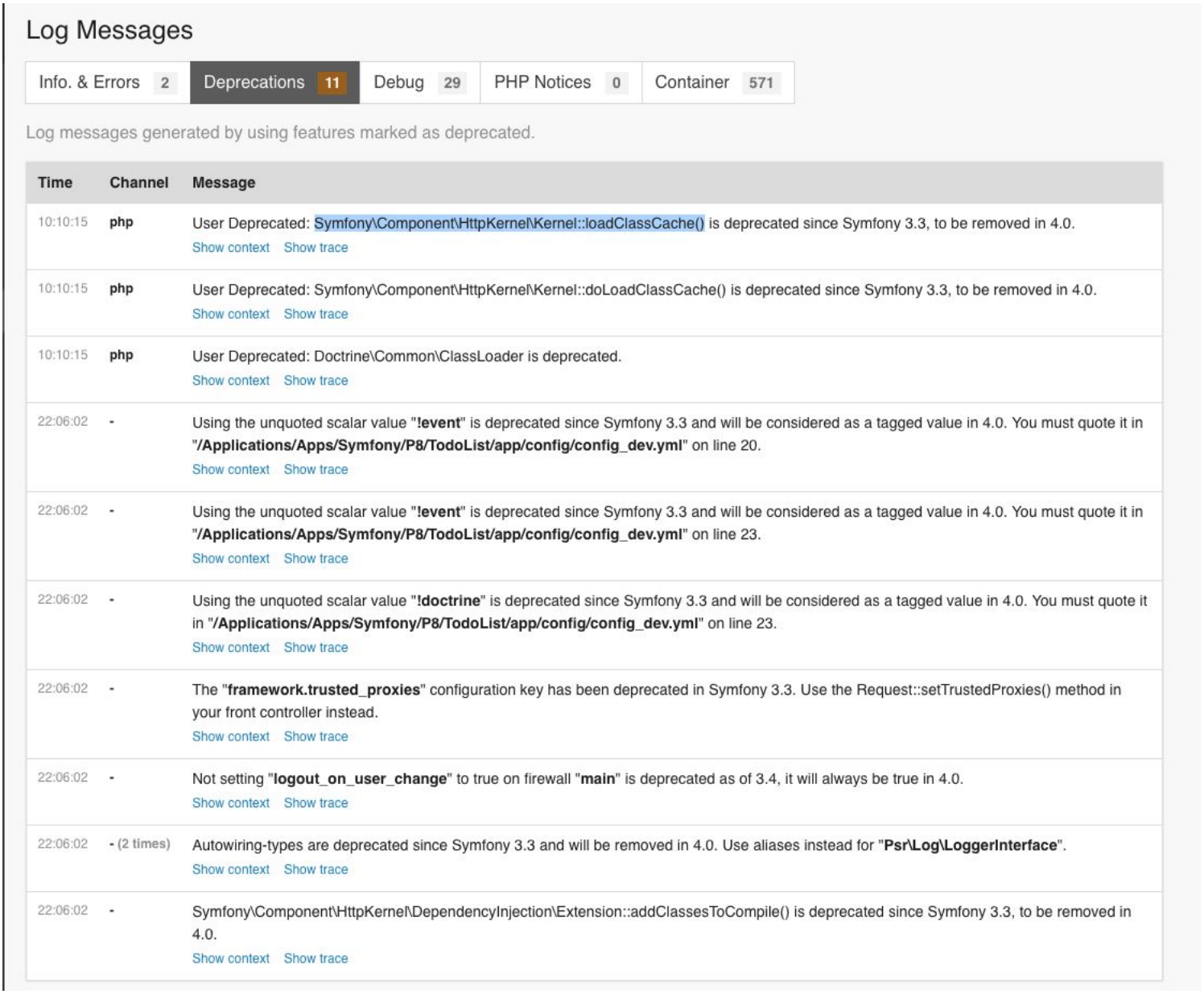
En vue de l'analyse effectué et reporté dans ce document, je recommanderai de finaliser tous les développement (qui sont assez facile à réaliser), de corriger les petites imperfections, de mettre en place un processus de déploiement (avec tous les contrôles de qualités recommandés) avant de monter celle-ci sur un serveur de production.

4. Annexes

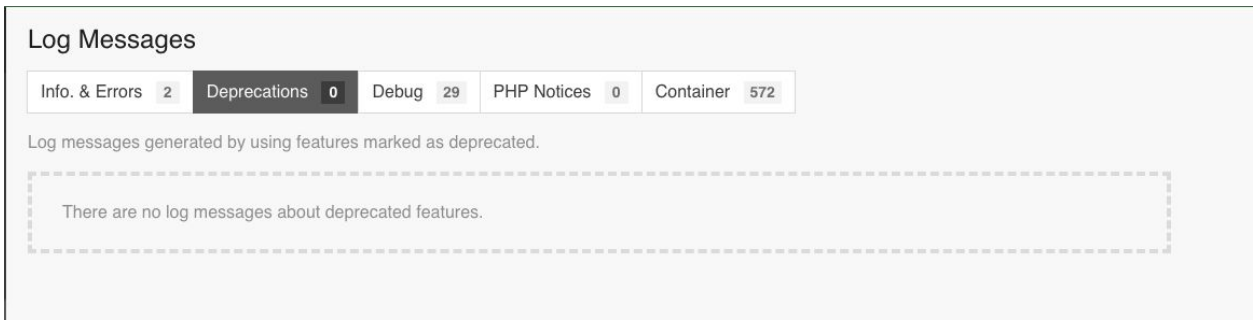
Codacy:

https://github.com/codacy/php-codacy-coverage

Deprecated corrigé aprés migration V3.4 :



Tout à été mis à jour:



CookBook Blackfire:

https://blackfire.io/docs/book/index

Composer (section dump autoload):

https://getcomposer.org/doc/03-cli.md#dump-autoload-dumpautoload-

Performance Symfony:

https://symfony.com/doc/3.4/performance.html

HttpCache Symfony:

https://symfony.com/doc/3.4/http_cache.html#symfony-gateway-cache

Varnish:

https://symfony.com/doc/3.4/http_cache/varnish.html
(sur le site de Symfony)

https://varnish-cache.org/
(documentation officielle de Varnish)

Doctrine:

https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/caching.html