

RAPPORT D'AUDIT

Pour l'application

ToDo & Co



Contexte

ToDo & Co est une startup dont le cœur de métier est une application permettant de gérer ses tâches quotidiennes. L'entreprise a réussi à lever des fonds pour permettre son développement et celui de l'application.

L'application, développée avec le framework PHP Symfony, a dû être écrite à la hâte pour convaincre les futurs investisseurs que le concept est viable.

Notre mission est de contribuer à l'amélioration de la qualité de l'application et ce à tout niveau ;

- Qualité de travail et de collaboration sur le projet.
- Qualité utilisateur.
- Qualité de code.
- Qualité de performance.

Camile Ghastine (Mars 2021)

Sommaire

I) Qualité de travail et de collaboration sur le projet	3
1) Règle de collaboration	
2) Mise à jour de Symfony	
3) Implémentation des tests	
II) Qualité utilisateur	5
1) Amélioration	
2) Pistes d'amélioration	
III) Qualité de code	7
1) Standard de code	
2) Bonne pratique mise en place	
3) Analyse avec Code Climat	
IV) Qualité de performance	9
1) Optimisation des requêtes	
2) Paramétrages pour optimiser l'application	
3) Pistes d'amélioration	
Conclusion	13

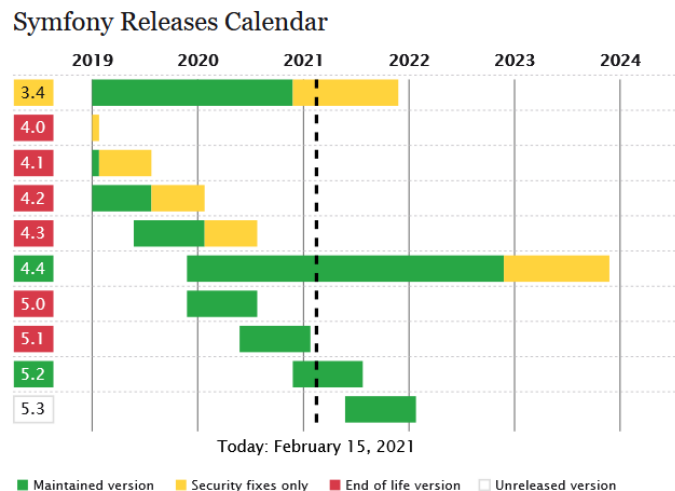
I) Qualité de travail et de collaboration sur le projet

1) Règles de collaboration

Pour cette section, reportez-vous au document interne nommé [Doc-collaboration.pdf](#). Vous le trouverez aussi en anglais à la racine du projet sous le nom [contribution.md](#).

2) Mise à jour de Symfony

Le projet initial est codé avec le Framework **Symfony version 3.1**.



Cette version n'étant plus maintenue, il nous a paru essentiel en tout premier lieu de faire évoluer le projet vers une version Symfony plus récente.

Dans un premier temps, nous avons migré le projet vers la **version 4.4** qui est la version stable de long terme.

Pour cela, nous avons :

- Migré en Symfony 3.4 (en modifiant le fichier *composer.json*).
- Régulé les dépréciations entraînées par cette migration.
- Migré en Symfony 4.0 (en modifiant le fichier *composer.json*).
- Mis à jour les bibliothèques dont les versions ne sont pas compatibles avec Symfony 4.
- Installé Symfony flex
- Réorganiser les fichiers (contrôleurs, entités, vues et de configuration) en suivant l'architecture imposée par Symfony flex
- Migré en Symfony 4.4 (en modifiant le fichier *composer.json*).
- Régulé les dépréciations entraînées par cette migration.

Nous avons migré le projet en Symfony 4.4 un peu à la hâte (sans réflexion préalable), et ce n'est que tardivement que nous nous sommes posé les questions :

- Devons-nous rester avec la LTS ("Latest Long-Term Support Release"), c'est-à-dire la **version 4.4** ?
- Ou devons-nous faire évoluer le projet vers la "Latest Stable Release" (**version 5.2** Aujourd'hui) ?

La première option a l'avantage de la **stabilité** pendant encore plus d'un an et demi à l'heure où nous écrivons. C'est sans hésitation l'option qui aurait été retenue si nous devions livrer cette application à un client.

ToDo&Co étant propriétaire de son application et possédant sa propre équipe de développeur, nous avons fait le choix de la deuxième option afin de profiter de toute la puissance de Symfony 5 :

- Nouvelles fonctionnalités
- Amélioration de la sécurité
- Amélioration de la performance

Le seul inconvénient de ce choix est une petite perte de temps pour des mises à jour vers les versions mineures plus récentes. Ce désagrément nous a paru mineur au regard des bénéfices potentiels. D'ailleurs, au moment de la migration vers Symfony 5, la dernière version stable était la version 5.1. Puis, celle-ci n'étant plus maintenue, nous avons évolué très facilement vers la version 5.2 qui est aujourd'hui la dernière version stable.

Notons enfin que le passage à **Symfony 5.1**, c'est fait différemment du passage à Symfony 4.4.

Pour cela, nous avons :

- Créé simplement un nouveau projet Symfony 5.1.
- Configuré ce projet
- Copié-collé les fichiers de notre projet ancien projet vers notre nouveau projet (celui disponible sur Github).

3) Implémentation des tests

























Une partie de notre travail sur l'application doit **OBLIGATOIREMENT** être allouée au test de celle-ci. En effet, nous devons mettre en place des tests unitaires et fonctionnels dont les objectifs sont multiples :

- Faciliter la maintenance de l'application.
- Faciliter l'implémentation de nouvelles fonctionnalités.
- Faciliter le travail en équipe.
- Créer par le biais des tests une documentation de l'application.

Notre politique de tests mise en place est la suivante :

- Atteindre les 100% de couverture de code par les tests dans tous les contrôleurs à l'aide de tests fonctionnels.
- Compléter en testant les autres classes à l'aide de tests unitaires pour :
 - Couvrir les points importants non-couverts.
 - Tester le maximum de cas limites.

La mise en place de cette politique nous a permis d'obtenir une couverture de code de 100% :

	Code Coverage							
	Lines			Functions and Methods			Classes and Traits	
Total		100.00%	264 / 264		100.00%	69 / 69		100.00% 18 / 18
■ Controller		100.00%	64 / 64		100.00%	11 / 11		100.00% 4 / 4
■ DataFixtures		100.00%	42 / 42		100.00%	5 / 5		100.00% 2 / 2
■ Entity		100.00%	50 / 50		100.00%	30 / 30		100.00% 2 / 2
■ Form		100.00%	18 / 18		100.00%	2 / 2		100.00% 2 / 2
■ Repository		100.00%	10 / 10		100.00%	3 / 3		100.00% 2 / 2
■ Security		100.00%	45 / 45		100.00%	11 / 11		100.00% 3 / 3
■ Service		100.00%	35 / 35		100.00%	7 / 7		100.00% 3 / 3

A l'avenir, des tests devront être mis en place :

- Dès qu'une nouvelle fonctionnalité sera implémentée.
- Dès qu'un bug sera découvert.

Nous n'avons pas pour exigence d'atteindre les 100% de couverture de code. Néanmoins, nous devons y tendre en gardant à l'esprit que le plus important est de tester les points clefs de notre application et le maximum de cas limites.

II) Qualité utilisateur

1) Améliorations

Afin d'améliorer l'expérience utilisateur, nous avons procédé à la correction de plusieurs anomalies et implémenté quelques nouvelles fonctionnalités.

Tâches attachées à un utilisateur

Dans le projet initial, les tâches n'étaient rattachées à aucun utilisateur. Pour remédier à cela, nous avons :

- Créé une relation **OneToMany** entre les entités *User* et *Task*.
- Ajouté un **constructeur** à la classe *Task* attendant une instance de *User* en paramètre.
- Passé *\$this->getUser()* en paramètre lors de la création d'une nouvelle instance de *Task* dans le *TaskController*.
- Affiché dans les vues Twig le nom du créateur des tâches affichées *{{task.user.username}}*

Gestion des rôles et des autorisations

Lorsque nous avons repris le projet, seul le `ROLE_USER` était disponible et aucune autorisation particulière n'était mise en place. Pour remédier à cela, nous avons :

- Adapté le *UserType* pour que, lors de la modification d'un *User*, la possibilité nous soit offerte de choisir le `ROLE_ADMIN`.
- Mis en place des **Voters** et utilisé *\$this->is_granted()* afin de :
 - **Restreindre l'accès** aux pages relatives à la gestion des tâches au `ROLE_USER` et celles relatives à la gestion des utilisateurs au `ROLE_ADMIN`.
 - **Restreindre la suppression** des tâches au créateur de celles-ci (ou au `ROLE_ADMIN` pour les tâches "anonymes").
- Modifié les vues Twig en conséquence (barre de navigation, boutons "supprimé", etc.) avec *is_granted()*.

Contraintes sur les entités

Les contraintes sur les entités étaient initialement très sommaires. Pour remédier à cela, nous avons :

- Créé des **contraintes d'unicité**, *@UniqueEntity*, sur *username* et *email*
- Mis en place des contraintes plus fines en utilisant les annotations des attributs des entités *User* et *Task* :
 - Utilisation de *@Assert\Regex* pour *username* et *password* de l'entité *User*.
 - Utilisation de *@Assert\Length* pour *title* de l'entité *Task*.

Navigation sur le site

Sur la version précédente de l'application, la navigation entre les pages était entièrement à revoir avec des liens morts ou inexistant. Pour remédier à cela, nous avons :

- Créé une **barre de navigation** adaptée au rôle de l'utilisateur connecté.
- **Réactivé les liens morts** comme le lien d'accès aux "tâches réalisées" avec la création d'une requête spécifique.

En plus des points cités ci-dessus, de nombreuses anomalies ou améliorations mineures que nous n'énumérerons pas ici, ont été corrigées.

2) Piste d'amélioration

Le travail d'amélioration réalisé jusqu'à maintenant n'est pas suffisant. Avant de mettre l'application en production, de nombreux points devront être retravaillés tant sur le frontend que sur le backend.

Amélioration de l'interface utilisateur (front end)

Il y a un gros travail à réaliser sur la partie frontend de l'application avec :

- La mise en place d'une charte graphique cohérente.
- Une mise en page, une esthétique et des design travaillés et plaisants.
- L'amélioration de la partie responsive.

- La mise en place d'un code couleur en fonction de l'urgence ou de la complexité de la tâche, des tâches faites ou à faire.

Nouvelles fonctionnalités (back end)

L'application, bien que prometteuse, est pour le moment un peu basique. Elle mériterait d'être enrichie de nombreuses fonctionnalités et options permettant d'améliorer le confort utilisateur. Voici quelques pistes qui pourraient être envisagées :

- Afficher le chapô d'une tâche dans la liste des tâches.
- Créer et afficher une date de modification et le *username* de l'utilisateur ayant modifié la tâche.
- Ajouter une date limite de réalisation d'une tâche.
- Paginer l'affichage des tâches.
- Mettre en place un système de tri des tâches.
- Permettre la suppression d'un utilisateur à un utilisateur ayant le `ROLE_ADMIN`.
- Gérer certaines requêtes en AJAX (tri, pagination, task isDone, etc.)

III) Qualité de code

1) Standard de code

Afin de faciliter le travail collaboratif et respecter certains standards, nous avons modifié le code pour qu'il suive quelques règles simples :

- Les classes, les méthodes et certaines parties du code ont été commentées.
- La présentation et l'indentation du code ont été revues.
- Le code a été modifié de manière à respecter les standards de code de Symfony basés sur les standards PSR-1, PSR-2, PSR-4 et PSR-12.

2) Bonne pratique mise en place

Architecture MVC

Dans le projet de départ, c'est l'architecture Modèle-Vue-Contrôleur (MVC) qui a été choisie. Nous approuvons ce choix et avons conservé cette architecture pour les avantages qu'elle apporte :

- Clarté et efficacité en conception avec la séparation des données de la vue et du contrôleur.
- Gain de temps de maintenance et d'évolution du site.
- Plus grande souplesse pour organiser le développement collaboratif du site (indépendance des données, de l'affichage et des actions)

Principe de responsabilité unique

Afin de décharger les contrôleurs de la logique métier, nous avons mis en place :

- Différents **services** :
 - Pour gérer la soumission des formulaires : *TaskFormHandler* et *UserFormHandler*
 - Pour gérer la vérification des tokens : *ToogleTokenHandler*
- Des **voters** : *TaskVoter* et *UserVoter*

Sécurité

Plusieurs formulaires étaient dépourvus de token et n'étaient donc pas protégés contre les failles CSRF. Pour remédier à cela, nous avons :

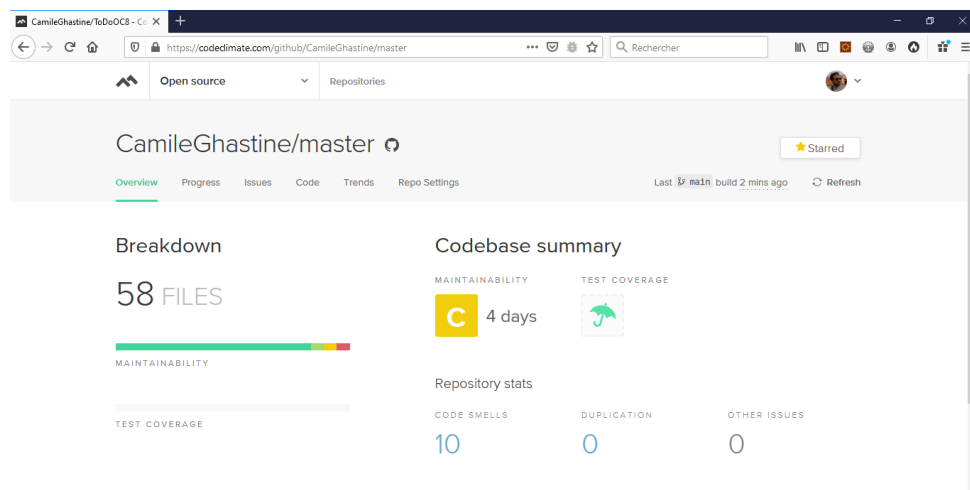
- Ajouté un token dans les vues et vérifié ce token pour :
 - Le formulaire de suppression de task
 - Le formulaire pour marquer une tâche comme faite
- Mis en place un système d'authentification personnalisé avec Guard pour se décharger de la génération et de la vérification du token lors de la connexion d'un utilisateur.

3) Analyse avec Code Climat

Afin de suivre la qualité du code de l'application, nous avons choisi d'utiliser **CodeClimate**.

<https://codeclimate.com/github/CamileGhastine/ToDoOC8>

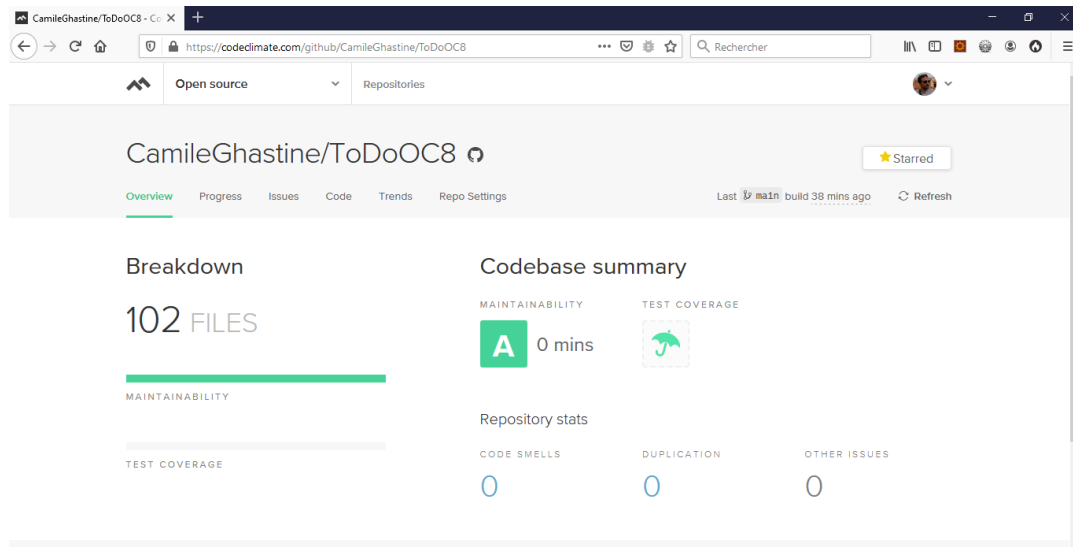
Note Code climat du projet initial (fichier PHP uniquement)



Lorsqu'on a récupéré le projet, en ne considérant que les fichiers PHP, il obtenait la **note de C** pour la qualité du code.

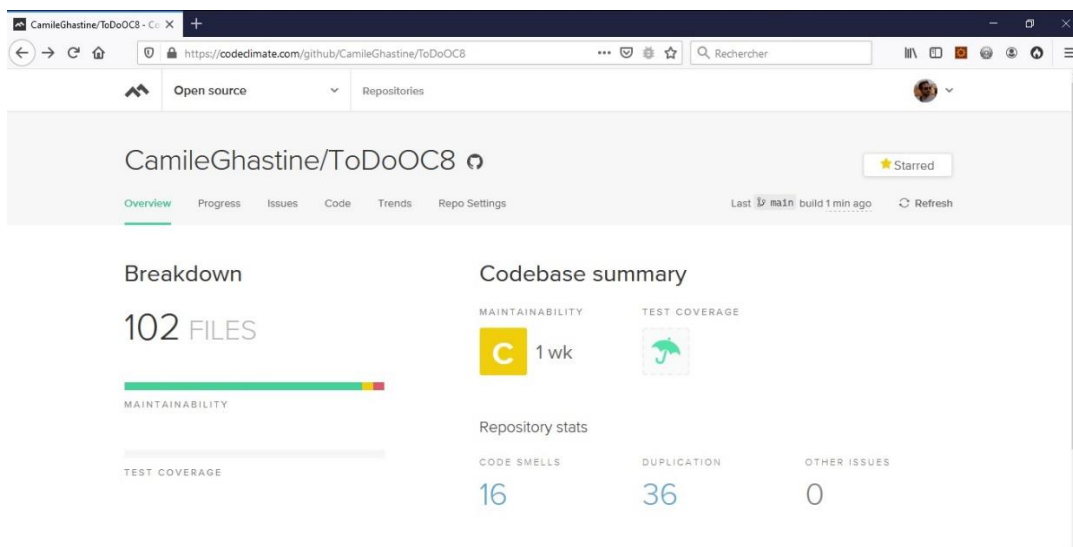
Cette note est trop éloignée des standards de qualité que l'on souhaite pour notre application.

Note Code climat à ce jour (fichier PHP uniquement)



Comme on peut le voir sur la seconde capture d'écran ci-dessus, les différentes stratégies mises en place nous ont permis d'améliorer la qualité du code initial et de conserver cette qualité lors du développement des nouvelles fonctionnalités.

Code climat : Fichier javascript



L'analyse de la qualité du code par Code Climat nous montre qu'il y a 52 modifications à faire (16+36). Ces 52 modifications concernent uniquement des fichiers javascript. Il est donc important que les développeur frontend travaillent sur la qualité du code javascript afin d'obtenir la note de A.

IV) Qualité de performance

Comme on l'a vu précédemment, le passage en Symfony 5.2 permet déjà un gain de performance pour notre application.

Une analyse des métriques offertes par **Blackfire** et par la **web debug tool bar** de Symfony, nous permet de noter que notre application ne mérite pas qu'on s'attarde trop longuement à l'amélioration de ses performances. En effet, les différents temps d'accès ou la mémoire consommée ont des valeurs tout à fait acceptables. Ceci est principalement dû à la petite taille de notre projet.

Néanmoins, il serait dommage de ne pas modifier quelques points (essentiellement des paramètres), afin de rendre notre application plus performante.

1) Optimisation des requêtes

Lorsqu'on demande l'affichage de la liste des tâches, la barre de debug de Symfony ci-dessous, nous indique que la base de données a été interrogée 6 fois.



- 1 requête est due à la recherche dans la base de données des informations relatives à l'utilisateur connecté.
- 1 requête est due au `findAll()` de la méthode `listAction()` du contrôleur `TaskController` pour récupérer dans la base de données toutes les tâches.
- Les 4 autres requêtes sont dues à l'affichage du nom du créateur d'une tâche dans la vue Twig avec l'appel `{{task.user.username}}`.

Ici, on voit poindre le problème d'une application avec des centaines de tâches créées par des centaines d'utilisateurs. Il faudrait alors des centaines de requêtes pour afficher notre liste de tâches.

Pour régler ce problème, la solution des requêtes jointes s'impose. Ceci devrait permettre de réduire à 2 le nombre de requêtes.

```
public function findAllWithUser(): array
{
    return $this->createQueryBuilder('t')
        ->addSelect('u')
        ->leftJoin('t.user', 'u')
        ->getQuery()
        ->getResult()
    ;
}
```

Comme attendu, la barre de debug de symfony nous indique bien que le nombre de requête est maintenant passé à 2 (ce, quel que soit le nombre de tâches et de créateurs de tâches).



On a réalisé la même optimisation pour l'affichage des tâches faites.

2) Paramétrages pour optimiser l'application

Généralités

Le temps d'affichage d'une page est une donnée très importante dans le cadre d'une application web. En effet, aujourd'hui 40% des utilisateurs abandonnent une application si elle ne s'affiche pas en moins de 3s. Il y a deux ans, ce temps était de 5s. Un autre exemple criant est celui d'Amazon qui déclare que 100ms gagné sur l'affichage d'une page, c'est 1% de chiffre d'affaire en plus.

Le temps d'affichage d'une page sur la machine d'un client dépend de nombreux facteurs. Il y a plusieurs de ces facteurs que nous ne pouvons pas maîtriser : la puissance, la configuration, le navigateur ou l'accès au réseau du client par exemple.

Néanmoins, coté serveur, il y a des facteurs sur lesquels nous pouvons influencer. Blackfire, en mode premium, nous donne par défaut quelques métriques qui nous permettent d'analyser la performance de notre application :

- Temps d'accès à la page :
- Mémoire consommée :
- Temps d'accès au système de fichiers :
- Temps d'accès au processeur :
- Quantité de données passées par le réseau :

En fonction des objectifs de notre application et des ressources dont on dispose, il est important de :

- Concentrer notre travail sur l'optimisation de certaines métriques par rapport à d'autres.
- Réaliser des arbitrages judicieux lorsqu'une optimisation entrainera un gain pour une métrique et une perte pour une autre.

Passage en environnement de production

En passant en environnement de production, la barre de développement et le web profiler de Symfony sont désactivés. Ces derniers ayant un fort impact sur la performance de l'application, un simple changement d'environnement va considérablement améliorer les performances.

Bien sûr, ceci ne constitue pas une amélioration. En effet, lorsque notre application sera prête à passer en production, c'est bien sûr dans cet environnement qu'elle sera utilisée.

Environnement	Métriques Blackfire pour la page affichant les tâches
Développement	
Production	

Dans le tableau comparatif ci-dessus, on constate que le simple fait de passer en environnement de production divise par plus de 2 les différents temps d'accès et fait gagner plus de 50% de mémoire consommée.

En environnement de production, on constate que la somme des temps d'accès est bien inférieure à 1 seconde. Ceci représente un temps tout à fait acceptable pour notre application.

Néanmoins, il peut être intéressant de mettre en place des paramétrages simples qui vont nous permettre d'améliorer facilement les performances de l'application.

Optimisation des performances

La documentation de Symfony nous propose plusieurs paramétrages qui permettent d'améliorer les performances de notre application : <https://symfony.com/doc/4.4/performance.html#performance-service-container-single-file>

Nous avons appliqué les différentes solutions et après analyse des métriques Blackfire, nous avons retenu les paramétrages ci-dessous :

- Utilisation du cache de code OPcache en l'activant dans le fichier *php.ini* :
`zend_extension = c:\wamp64\bin\php\php7.2.10\ext\php_opcache.dll`
- Maximisation des performances en paramétrant OPcache dans le fichier *php.ini* :
`opcache.memory_consumption=256`
`opcache.max_accelerated_files=20000`
- Annulation de la vérification de l'horodatage des fichiers PHP dans le fichier *php.ini* :
`opcache.validate_timestamps=0`
- Configuration de realpath cache PHP dans le fichier *php.ini* :
`realpath_cache_size=4096K`
`realpath_cache_ttl=600`

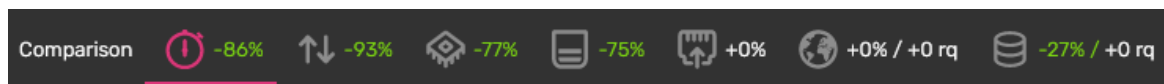
- Optimisation de l'auto-chargement des classes de composer en tapant la commande :
`composer dump-autoload --no-dev --classmap-authoritative`

Optimisation	Métriques Blackfire pour la page affichant les taches
Avant	
Après	

D'après les métriques relevées dans le tableau ci-dessus, on constate que nos optimisations ont apporté un réel gain de performance à notre application.

Blackfire nous permet de comparer ces deux profils (avant et après optimisation), afin de pouvoir quantifier le gain de performance pour notre application.

La lecture du comparatif ci-dessus montre sans équivoque l'incroyable gain de performance entraîné par les différents paramétrages opérés.



Remarque : Ici, nous nous sommes contentés de limiter notre étude à la page d'affichages des tâches. Les gains de performance de ces paramétrages sont du même ordre de grandeur pour les autres pages.

Pour les rapports de performance détaillés, reportez-vous à l'annexe nommée : [Annexe-profils_Blackfire.pdf](#)

3) Piste d'amélioration

Ces quelques paramétrages nous ont permis un gain de performance très important. Néanmoins, sans entrer dans des améliorations de performance chronophages dont le gain de performance ne justifie pas le temps passé, il y a quelques autres améliorations faciles à mettre en place.

Passage à la dernière version de PHP

Lorsqu'on regarde le journal des modifications (<https://www.php.net/releases/>) du langage PHP, on constate que chaque version amène, en plus de nouvelles fonctionnalités, une amélioration des performances.

Pour notre projet, on pourrait passer à **PHP 8** ou au moins à **PHP 7.4**.

Paramétrage possible grâce à PHP 7.4

En PHP 7.4, la documentation de Symfony, nous propose plusieurs paramétrages améliorant les performances :

- Décharger le conteneur de services dans un seul fichier pour améliorer les performances lors du pré-chargement des classes en paramétrant le fichier `config/services.yaml` de notre projet :
parameters:
`container.dumper.inline_factories: true`
- Utiliser le pré-chargement des classes de OPcache dans le fichier `php.ini` :
`opcache.preload=/path/to/project/config/preload.php`

Bonne pratique de Doctrine ORM

Il existe de nombreuses recommandations concernant Doctrine ORM de Symfony qui améliorent les performances comme celle que nous avons déjà appliquée en utilisant les requêtes jointes.

Lorsque le nombre de données gérées par l'application deviendra très important, il pourra être intéressant de s'intéresser à :

- L'étude des requêtes provoquant des goulots d'étranglement
- La mise en cache de ces données pour garantir un accès plus rapide à ces dernières.

Mise en cache HTTP

En plus de la mise en cache des résultats de la base de données, une mise en cache HTTP permettrait d'améliorer le confort utilisateur avec un affichage plus rapide des pages sur la machine du client.

Conclusion

Notre travail sur **ToDo & Co** aura permis en amont de corriger de nombreuses imperfections et de mettre en place de nombreuses améliorations avec :

- La correction de certaines anomalies
- La mise en place de nouvelles fonctionnalités
- La mise en place de règles de collaboration
- La mise à jour de Symfony
- L'implémentation de tests unitaires et fonctionnels
- L'amélioration de la performance de l'application.

Le but de ce travail étant de mettre en place des bases solides et saines afin de d'assurer que l'application soit qualitative tant au niveau du code, de la performance, de l'expérience de travail et de collaboration et aussi bien sûr que de l'expérience utilisateur.

Ces fondations mises en place ne sont pas un aboutissement, mais le début d'un travail qui devra se poursuivre. Tout au long de ce rapport, de nombreuses pistes sont proposées afin d'améliorer encore l'application à tous les niveaux.

La créativité de notre équipe, la maintenance de l'application et les retours utilisateurs, nous permettront de trouver encore de nouvelles pistes d'amélioration et de rendre **ToDo & Co** toujours meilleur pour tous.