

Audit de qualité du code et de performance

1. Contexte
2. Audit de performance
3. Audit de qualité et couverture de tests

1. Contexte

1.1. Présentation de l'application

L'application auditée et améliorée est une appli web de gestion de tâches à faire créée par Todo & Co, elle a été développée rapidement afin d'avoir une Minimum Viable Product à montrer à de potentiels investisseurs.

1.2. Axes d'améliorations

Afin d'améliorer ce produit plusieurs tâches ont dues être accomplies :

- L'implémentation de nouvelles fonctionnalités
- La correction des anomalies
- L'implémentation de tests automatisés

De plus, il était préférable de réduire la dette technique du projet, pour cela plusieurs axes ont été suivis :

- La mise à jour de la version des différents packages utilisés (dont Symfony)
 - o Symfony a été passé en version 4.4, qui est la version LTS pour le moment
- La mise à jour de la version de PHP utilisée (PHP 7.4)
- La correction des points remontés par les analyses (qualité du code et normes PSR)
- La mise au propre du code selon de bonnes pratiques (moins de code métier dans les Controller)

2. Audit de performance

2.1. Présentation de l'action et de l'outil

Un audit de performance a été réalisé avec Blackfire, qui permet de profiler des requêtes et de récupérer différentes métriques, dont le temps de chargement ou bien encore la mémoire utilisée ou les requêtes SQL. C'est un outil très utile pour faire le benchmark d'un site et pour comparer les effets de changements de code sur la performance de ce dernier. Ici un profiling a été réalisé sur chaque page avant et après optimisation.

2.2. Axes d'optimisation

OPcache étant déjà activé, c'était un élément d'optimisation qu'il n'était pas nécessaire de prendre en compte, cependant la première optimisation a été de passer le projet d'un mode d'environnement de développement à celui de production.

De plus la commande suivante a été lancée sur le projet :

composer dump-autoload --optimize --classmap-authoritative

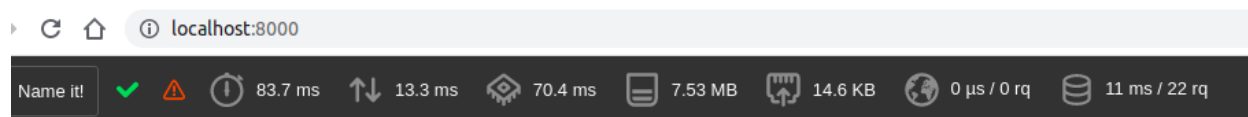
Celle-ci permet d'optimiser l'autoloader en mettant en cache les classes utilisées, cependant cette commande doit être relancée à chaque ajout de nouvelles classe, il est donc conseillé de la réaliser seulement en environnement de production.

2.3. Comparaisons

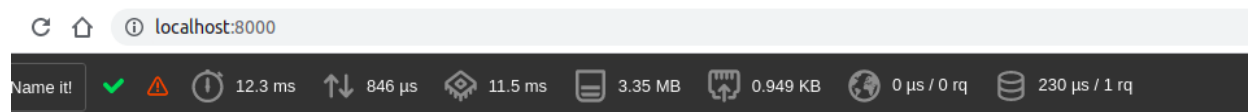
Voici une comparaison des différentes requêtes avant et après optimisation :

2.3.1. Page d'accueil

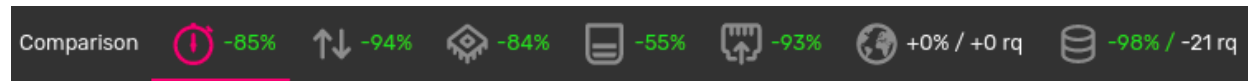
Avant :



Après :

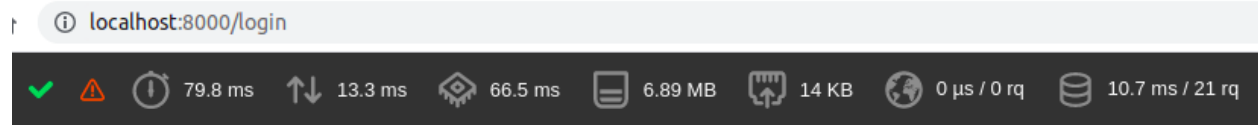


Comparaison :

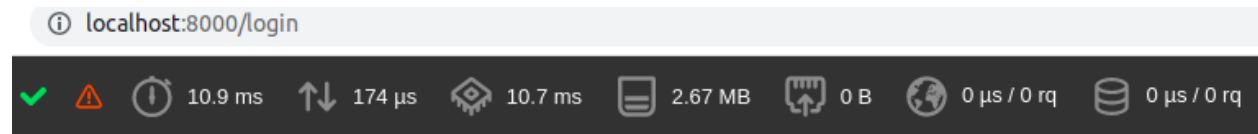


2.3.2. Page de login

Avant :



Après :

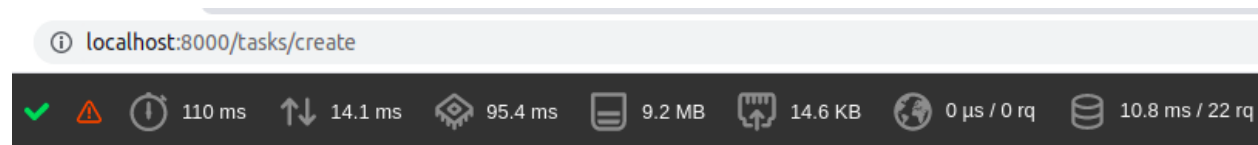


Comparaison :

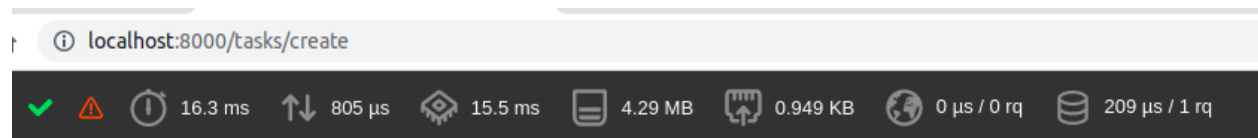


2.3.3. Page de création de tâche

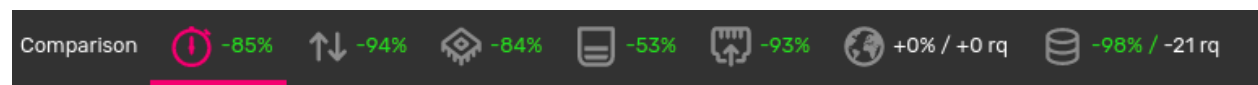
Avant :



Après :

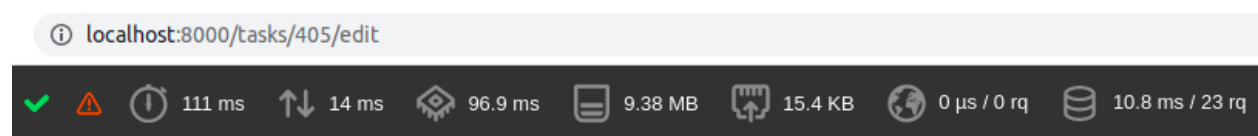


Comparaison :

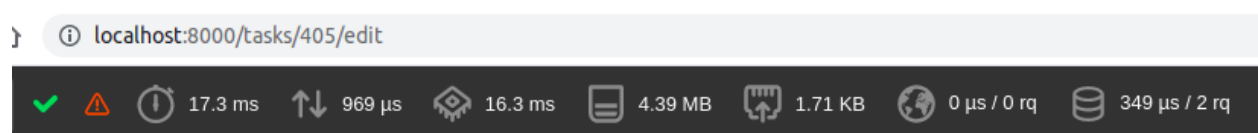


2.3.4. Page d'édition de tâche

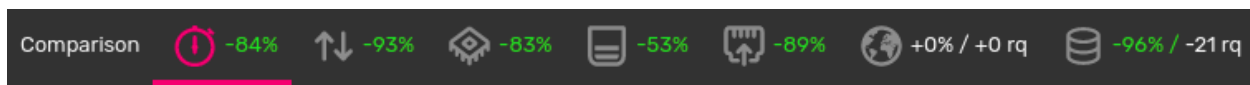
Avant :



Après :

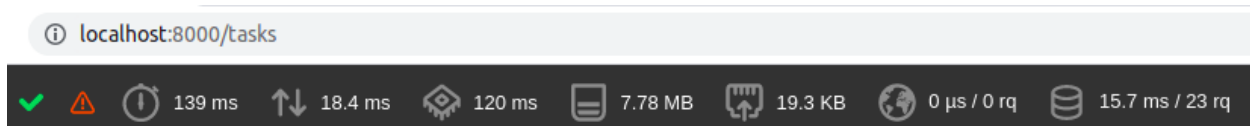


Comparaison :

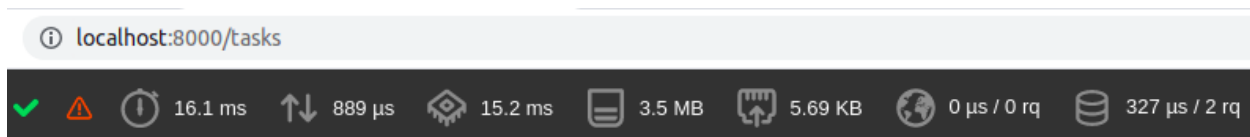


2.3.5. Page de liste des taches

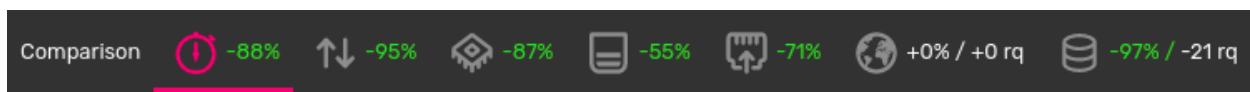
Avant :



Après :

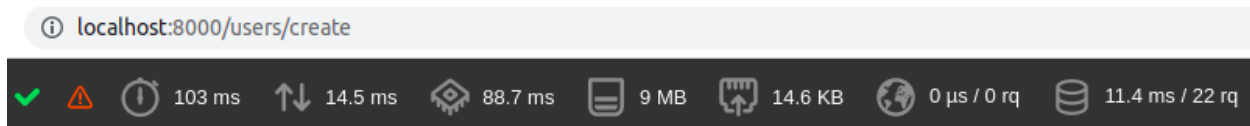


Comparaison :

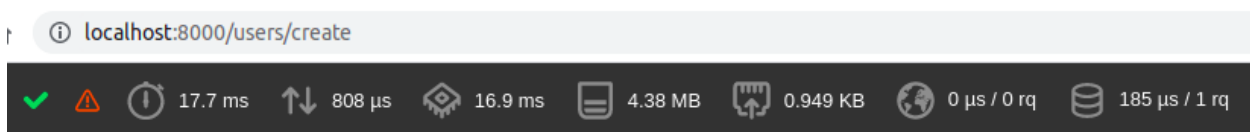


2.3.6. Page de creation d'utilisateur

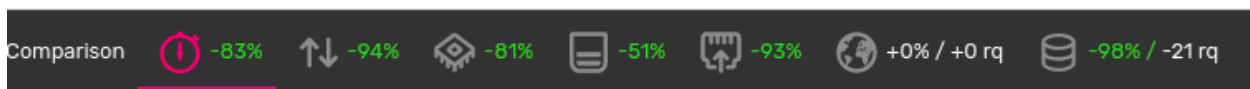
Avant :



Après :

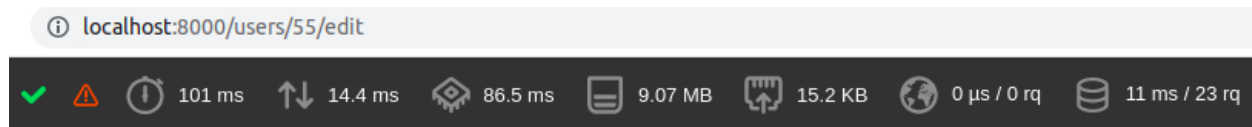


Comparaison :

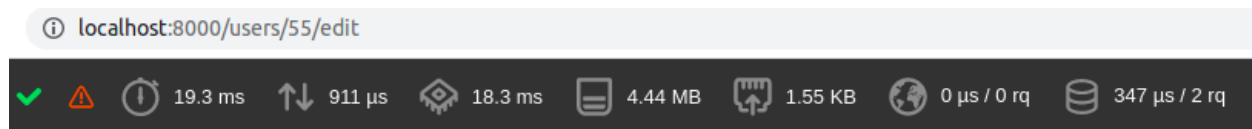


Page d'édition d'utilisateur

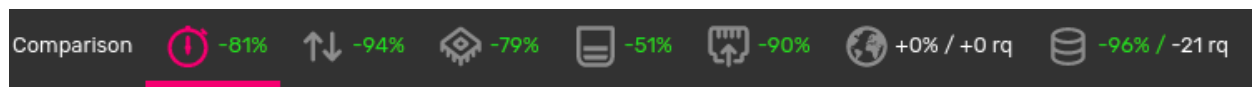
Avant :



Après :

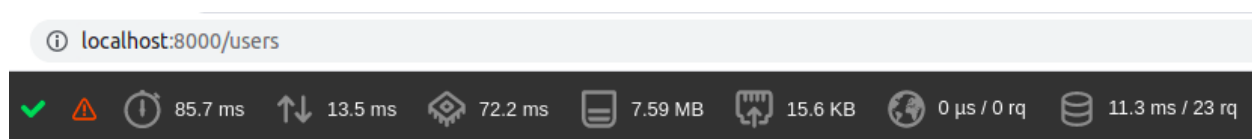


Comparaison :

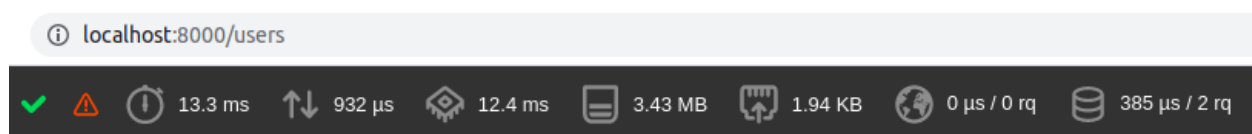


2.3.7. Page de liste d'utilisateurs

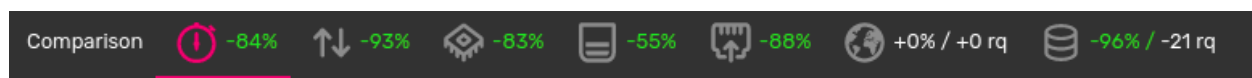
Avant :



Après :



Comparaison :



2.4. Conclusion sur l'optimisation

On peut observer que le passage en mode production ainsi que l'optimisation de l'auto loader ont un effet non négligeable sur toutes les pages du site. Le gain en performance est donc réussi.

3. Audit de qualité et couverture de tests

3.1. Contexte

Plusieurs éléments ont été mis en place afin de garantir une qualité optimale du code, des bonnes pratiques ainsi que de réduire les possibilités de régressions. Permettant à l'application d'être beaucoup plus robuste afin de pouvoir y apporter des modifications beaucoup plus sereinement.

3.2. Qualité du code et des bonnes pratiques



















Afin de garantir la qualité du code et des normes PSR, certains packages sont utilisés, nous pouvons retrouver PHPCS, php-cs-fixer ainsi que PHPStan.

De plus, dans une optique de Continuous Integration (CI), l'exécution de ces packages ainsi que des tests unitaires et fonctionnels est faite de manière automatisée lors des commits (grâce au package GrumPHP) ainsi que lors des pull requests et des push sur GitHub (grâce à GitHub Actions).

Cela permet de surveiller qu'il n'y a pas de régressions lors d'une modification apportée au projet.

3.3. Couverture de tests

Des tests unitaires et fonctionnels ont été ajoutés afin de tester le plus de classes et méthodes possibles, voici la couverture de tests en image pour les tests faits avec PHPUnit :

	Code Coverage									
	Lines				Functions and Methods				Classes and Traits	
Total		95.80%	137 / 143		89.80%	44 / 49			66.67%	8 / 12
■ Controller		94.92%	56 / 59		78.57%	11 / 14			50.00%	2 / 4
■ Entity		92.86%	39 / 42		92.00%	23 / 25			0.00%	0 / 2
■ Form		100.00%	10 / 10		100.00%	2 / 2			100.00%	2 / 2
■ Manager		100.00%	28 / 28		100.00%	6 / 6			100.00%	2 / 2
■ Repository		100.00%	4 / 4		100.00%	2 / 2			100.00%	2 / 2