

ToDo & Co .

Audit technique de l'application.



Table des matières

ToDo & Co	1
Audit technique de l'application.	1
I- Introduction.	3
II- Audit de la qualité du code.	3
1- Introduction à Codacy.	3
2- Analyse du projet par Codacy.....	3
III- Audit des performances de l'application.	4
1- Préface.....	4
2- Travaux d'optimisation réalisés.....	4
3- Analyse de l'application avant et après optimisation.	5
IV- Pistes d'améliorations pour l'application.....	6

I- Introduction.

Cet audit comprend l'analyse du code et celle des performances du projet. Ci-dessous, le processus d'analyse, ainsi que les outils utilisés seront détaillés.

II- Audit de la qualité du code.

1- Introduction à Codacy.

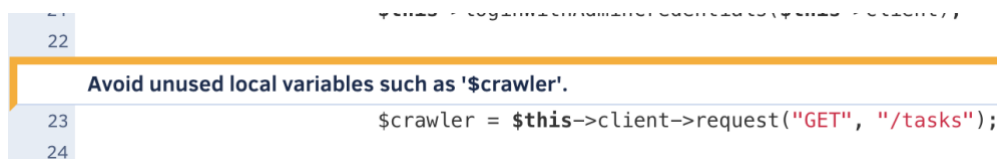
Afin de pérenniser le projet et de vérifier la qualité de celui-ci en minimisant la dette technique de l'application, il convient d'analyser la qualité du code produit.

Pour réaliser cet audit, nous allons nous appuyer sur une solution tierce : **Codacy**.

Codacy est un portail qui permet d'analyser le code produit automatiquement et génère une revue sur ce qu'il doit être apporté comme correctif afin d'avoir le projet le plus pérenne possible.

Le rapport liste les problèmes selon leurs types : sécurité, erreurs, duplication, code inutilisé, performance et selon différents niveaux Infos, Issues (problèmes de convention, code inutilisée et Errors (lignes susceptibles de poser des soucis dans le futur).

Exemple de revu par Codacy :

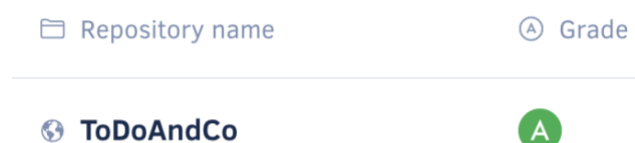


L'analyse est faite automatiquement à chaque push de notre dépôt vers Github. Chaque commit est analysé, revu puis noté.

Note : La configuration de l'analyse exclu donc tous les fichiers de Symfony. Seul notre code produit est analysé. Attention à bien exclure tout autre fichier généré par Symfony.

2- Analyse du projet par Codacy.

Après analyse du repository Github, notre projet a obtenu le **grade A**, soit le plus haut grade possible, ce qui démontre que notre projet ne possède pas d'erreurs et suit les bonnes pratiques de développement.



III- Audit des performances de l'application.

1- Préface.

L'application ToDo lors du début du projet était en Symfony 3.1.6. Cette version n'étant plus maintenue, elle a été portée vers une version LTS (Long Term Support) en Symfony 4.4.21.

Pourquoi choisir une version LTS ?

- Support allongée concernant les mises à jour de bugs pendant 3 ans.
- Support allongée concernant les mises à jour de sécurité pendant 4 ans.

En outre, Symfony 4 apporte des fonctionnalités qui apporteront un plus dans les développements futurs :

- Suppression de la notion de « bundle » : l'arborescence a été repensé afin d'être plus compréhensible. Il y a donc un seul namespace pour toutes les sources (src/).
- **Symfony Flex** : Automatise l'installation de bundles et les configure automatiquement afin de pouvoir les utiliser le plus rapidement possible.
- **Symfony Maker** : possibilité de générer rapidement des controllers, entités, authentification ... via des invites de commandes, avec une configuration de base utilisable dès la création.

2- Travaux d'optimisation réalisés.

Pour analyser les performances de notre application un service tiers a été utilisé :

Blackfire.io.

C'est un profiler plus performant que celui disponible par défaut dans le Web Profiler d'une application en Symfony et permet :

- De trouver les causes exactes des soucis de performances.
- De stocker plusieurs analyses de profils selon différentes situations et de les comparer (comme ce qui a été fait dans la section juste après).
- D'afficher le temps de génération de la page en PHP et la mémoire utilisée.

Attention, l'analyse doit être réalisé en environnement de production, avec l'extension XDebug désactivé.

Suite à l'analyse de la version de production sans optimisations, deux améliorations ont été apportées :

==> **Optimisation de l'autoloader de Composer pour notre projet :**

À la racine du projet, ouvrez le terminal et entrez la commande `composer dump-autoload --no-dev --classmap-authoritative`

→ Par défaut, le gestionnaire de dépendance **Composer** va analyser le projet continuellement pour trouver de nouvelles classes à charger.

→ **En production**, nos fichiers ne changeront pas, c'est pourquoi il faut optimiser l'autoloader en scannant l'entièreté du projet une fois et en construisant un fichier d'autoload optimisé :

- **--no-dev** : Exclu tous les fichiers qui sont utilisés pour le développement.
- **--classmap-authoritative** : créer un fichier classmap regroupe toutes les classes utilisées dans l'application et prévient Composer de scanner les fichiers de classes qui ne sont pas dans la classmap.

==> Activation de l'extension de PHP, Zend OPCache :

Dans un environnement de production, il convient d'activer **OPCache** par défaut, explication :









→ En temps normal, sans l'extension **OPCache**, le script PHP est analysé et transformé en langage machine (opcode) puis détruit à chaque sollicitation.









→ Avec l'extension, vérifie si le binaire correspondant existe et le chargera s'il l'est. Sinon, il le mettra dans une mémoire partagée jusqu'à ce qu'il soit re-sollicité. Ce processus exclut donc la partie analyse du script et allègera le chargement global de notre application en production.









3- Analyse de l'application avant et après optimisation.









Ci-dessous, un échantillon d'analyses par le client Blackfire avant et après optimisation. Le rapport détaillé est disponible en cliquant sur chacune des analyses.

Globalement, après optimisation, on distingue -16 à -75% sur le temps de chargement de chacune des pages, et un gain de -71 à -75% de mémoire partagée allouée en moins, ce qui est non négligeable.

Connexion (path : login_check)			
 login_check (before)	 550 ms	 13.9 MB	 login_check (after)
			 459 ms
			 3.65 MB
Comparaison :		Comparaison  -16%  -74%	

Liste des tâches (path : task_list)			
 task_list (before)	 174 ms	 15 MB	 task_list (after)
			 63.3 ms
			 4.21 MB
Comparaison :		Comparaison  -64%  -72%	

Supression d'une tâche (path : task_delete)		
 task_delete (before)  152 ms  17.3 MB		 task_delete (after)  37.3 ms  4.3 MB
Comparaison :		Comparaison  -75%  -75%

Modification d'un utilisateur (path: user_edit)		
 user_edit (before)  616 ms  17.1 MB		 user_edit (after)  490 ms  4.95 MB
Comparaison :		Comparaison  -20%  -71%

IV- Pistes d'améliorations pour l'application

Suite à la mise en production de l'application ToDo, voici quelques pistes d'améliorations qui pourrait être envisageables dans une future version :

- Utilisation d'un **Voter** pour vérifier le droit d'utilisation d'une ressource dans notre application ([Voir Doc](#)). Un Voter vérifiera si l'utilisateur authentifié possède un droit sur l'utilisation d'une URI, et le rejettera s'il n'en possède pas.
- Utilisation du composant **Cache** de Symfony ([Voir doc](#)). Le but est de pouvoir mettre en cache des données afin d'améliorer la rapidité de notre application en production. (**Ex** : une longue liste de tâches pourrait ralentir le chargement de la page chez certaines personnes possédant des connexions limitées).
- Utilisation de **Webpack Encore** pour la gestion des assets ([Voir doc](#)). Possibilité d'utiliser des librairies Javascript comme React et Vue, d'utiliser deux preprocesseur CSS, SASS et LESS, et de pouvoir minifier / compacter nos sources en vue d'une version de production.
- Revoir l'UX et l'UI de notre application pour une approche plus « user friendly ».