Rapport d'audit de qualité de code et de performance

4 JUIN 2024

TODO & Co

Créé par : Portemer Frederic

Introduction

L'application TodoList, développée par la jeune startup ToDo & Co, a été initialement créée rapidement pour démontrer son concept à des investisseurs potentiels. Suite à une présentation réussie, l'entreprise a obtenu les fonds nécessaires pour poursuivre son développement et son expansion. Ce document présente un audit de la qualité du code et des performances de l'application, avec un accent particulier sur l'analyse réalisée via Code Climate. Les recommandations fournies aideront à maintenir et à améliorer la qualité et les performances de l'application.

Table des matières

1. Contexe

1.1 Présentation

L'application TodoList a été développée par la jeune startup ToDo & Co. Initialement créée rapidement pour une démonstration à des investisseurs potentiels, elle a permis à l'entreprise de lever des fonds nécessaires pour poursuivre son développement et son expansion.

1.2 Nécessité de Mise à Jour

L'application a été mise à jour pour des raisons de maintenance et de sécurité. La version 3.1 de Symfony étant obsolète, il était impératif de passer à une version plus récente. La version 6.4.9 a été choisie en raison de sa longue période de support (LTS : https://symfony.com/releases/6.4)

Configuration avant mise à jour :

- Symfony 3.1.10
- PHP 5.5.9
- Doctrine-bundle 1.6
- Doctrine-orm 2.5
- Base de données MySQL

Configuration aprés mise à jour :

- Symfony 6.4.8
- PHP 8.1.0
- Doctrine-bundle 2.12
- Doctrine-orm 3.2
- Base de données MySQL

1.3 Remarque Importante

Les analyses et résultats présentés dans ce document sont basés sur les deux versions de l'application, avant et après la mise à jour de Symfony.

2. Qualité du code

2.1 Analyse du Code avec Code Climate

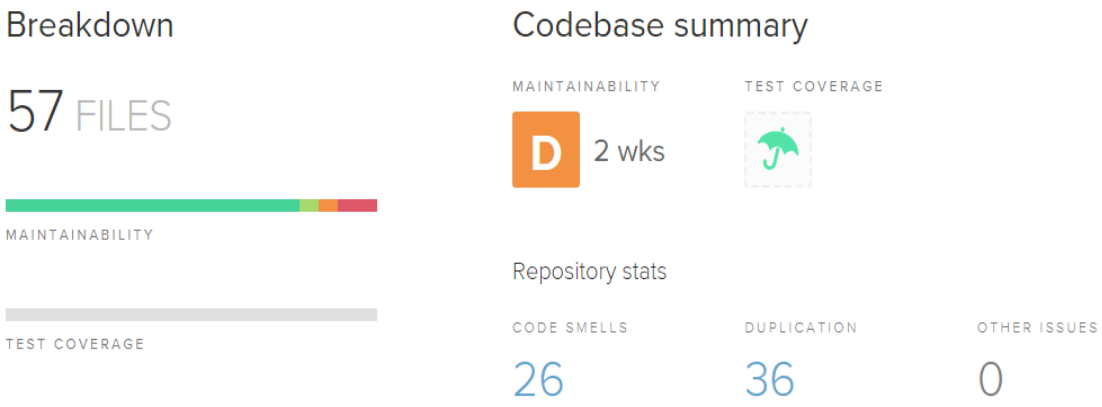
Pour évaluer la qualité du code, nous avons utilisé Code Climate. Ce service analyse la maintenabilité, la qualité du code et fournit des suggestions pour améliorer la structure du projet.

2.1.1 Analyse de la Version Avant Mise à Jour

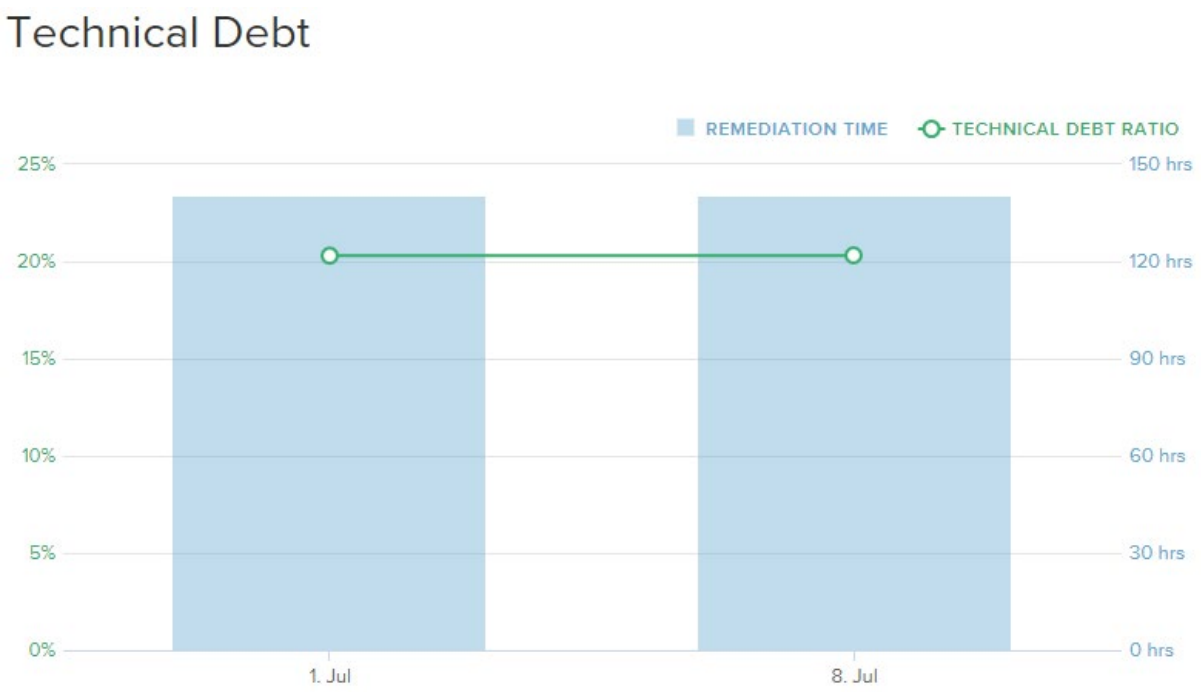
Code Climate a analysé le code de la version avant mise à jour et fourni les résultats suivants :

- **Maintenabilité** : D
- **Couverture de Test** : N/A
- **Code Smells** : 26
- **Duplications** : 36
- **Autres Issues** : 0

Tableau de bord CodeClimate de la version avant mise à jour :
https://codeclimate.com/github/fredericdww20/auditToDoCo



Dette Technique :
Le ratio de dette technique est de 20.3% avec un temps de remédiation de 140.6 heures.

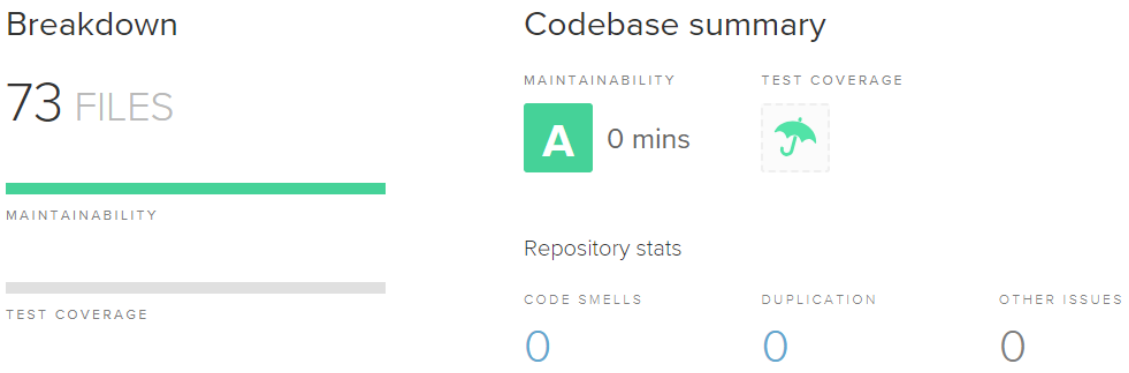


2.1.2 Analyse de la Version Après Mise à Jour

Code Climate a également analysé le code de la version après mise à jour et Fourni les résultats suivants :

- **Maintenabilité** : A
- **Couverture de Test** : N/A
- **Code Smells** : 0
- **Duplications** : 0
- **Autres Issues** : 0

Tableau de bord CodeClimate de la version après mise à jour :
https://codeclimate.com/github/fredericdww20/ToDo-Co



Dette Technique : Le ratio de dette technique est de 0% avec un temps de remédiation de 0 heure.

2.2 Comparaison et Recommandations pour Améliorer la Qualité du Code

Comparaison des deux versions :

- **Avant mise à jour** : La maintenabilité était notée D, avec 26 "Code Smells" et 36 duplications. Le ratio de dette technique était de 20.3% avec un temps de remédiation de 140.6 heures.
- **Après mise à jour** : La maintenabilité est notée A, avec 0 "Code Smells" et 0 duplications. Le ratio de dette technique est de 0% avec un temps de remédiation de 0 heure.

Pour maintenir et améliorer la qualité du code, nous recommandons les actions suivantes :

1. **Adopter les meilleures pratiques de développement** : Suivre les conventions de codage de Symfony et PSR (PHP Standards Recommendations).
2. **Automatiser l'analyse du code** : Intégrer Code Climate dans le pipeline CI/CD pour des vérifications automatiques à chaque commit.
3. **Renforcer les tests unitaires et d'intégration** : Utiliser PHPUnit pour écrire des tests robustes couvrant tous les scénarios d'utilisation.

3. Performances

3.1 Analyse des Performances avec Code Climate

Code Climate propose également des analyses de performance pour identifier les goulots d'étranglement et optimiser les ressources. Bien que Code Climate ne fournisse pas un profilage aussi détaillé que des outils spécialisés comme Blackfire, il offre une vue d'ensemble utile.

3.2 Analyse des Performances avec Symfony Profiler

Le Symfony Profiler fournit des informations détaillées sur les performances de chaque requête. Voici les résultats obtenus à partir du profiler Symfony pour la version actuelle de l'application :

- **Total execution time** : 323 ms
- **Symfony initialization** : 244 ms
- **Peak memory usage** : 2.00 MiB

Execution Timeline

Les principaux goulots d'étranglement identifiés sont :

- **RouterListener** : 9.1 ms
- **TraceableFirewallListener** : 8.5 ms
- **Controller** : 28.9 ms
- **Template Rendering** : 19.8 ms pour default/index.html.twig et 19 ms pour base.html.twig

Ces résultats montrent que le temps de réponse global est principalement influencé par l'initialisation de Symfony et le rendu des templates Twig.

Analyse Détaillée des Résultats

1. **Symfony Initialization (244 ms)**
    - La phase d'initialisation de Symfony prend une part significative du temps d'exécution total. Cela inclut le chargement des bundles, la configuration des services et l'injection des dépendances.

    Recommandation : Examiner et optimiser la configuration des bundles et des services. Assurez-vous que seuls les services nécessaires sont chargés.

2. **RouterListener (9.1 ms)**
    - Le RouterListener est responsable de la correspondance des URL aux contrôleurs appropriés. Bien que son impact soit relativement faible, il peut être optimisé en vérifiant que toutes les routes inutilisées sont supprimées.
3. **TraceableFirewallListener (8.5 ms)**
    - Ce listener fait partie du composant de sécurité de Symfony et est crucial pour le filtrage des requêtes.

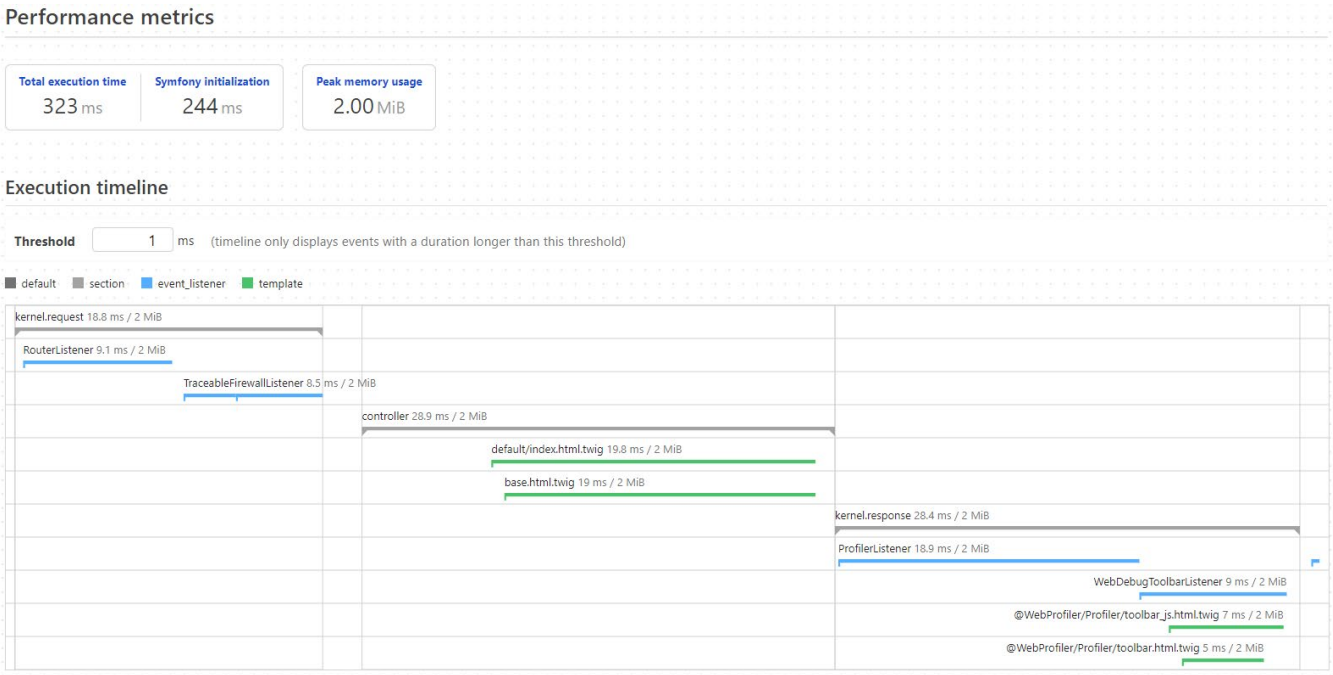
    Recommandation : Examiner les règles de pare-feu pour s'assurer qu'elles sont optimisées et qu'aucune règle inutile ne ralentit le processus.

4. **Controller (28.9 ms)**
    - La méthode du contrôleur qui gère la requête est un point clé où des optimisations peuvent être faites, notamment en réduisant la complexité des opérations effectuées**.**

    Recommandation : Refactorer le code des contrôleurs pour améliorer leur efficacité. Assurez-vous que les contrôleurs ne contiennent pas de logique métier complexe.

5. **Template Rendering (19.8 ms et 19 ms)**
    - Le rendu des templates Twig prend un temps considérable.

    Recommandation : Utiliser la mise en cache des templates Twig et optimiser les templates pour réduire le temps de rendu.



Limitation des Tests de Performance sur l'Ancienne Version

Il est important de noter que nous n'avons pas pu effectuer un test de performance avec le profiler Symfony sur l'ancienne version de l'application en raison de nombreux problèmes de dépendances obsolètes. Ces problèmes ont empêché le fonctionnement correct du profiler Symfony avec la version 3.1 de Symfony.

3.3 Optimisation de l'Autoloader

Pour optimiser les performances de l'application, nous recommandons d'optimiser l'autoloader de Composer en utilisant la commande suivante :

```
composer dump-autoload --no-dev --classmap-authoritative
```

Cette commande met en cache les classes nécessaires à l'application, réduisant ainsi le temps de chargement et la consommation de mémoire. Il est important de relancer cette commande après chaque ajout de nouvelles classes.

Documentation : Composer Autoloader Optimization

3.4 Recommandations pour Améliorer les Performances

1 - Configurer OPCache

- Améliorer les performances de PHP en stockant le bytecode des scripts
- pré-compilés en mémoire partagée.
- **Configuration OPCache :**

```
opcache.enable=1
opcache.memory_consumption=128
opcache.interned_strings_buffer=8
opcache.max_accelerated_files=4000
opcache.revalidate_freq=2
opcache.fast_shutdown=1
```

2 - Configurer RealpathCache :

- Améliorer les performances en mettant en cache les chemins absolus.
- **Configuration RealpathCache :**

```
realpath_cache_size=4096k
realpath_cache_ttl=600
```

3 - Utiliser Varnish

- Varnish est un accélérateur HTTP open source qui sert rapidement du contenu mis en cache.
- **Configuration Varnish** :

```
varnish:
    servers:
        - "127.0.0.1:6081"
```

4 - Choisir un Hébergeur Optimal

- **Un hébergeur performant est crucial pour assurer la réactivité et les performances de l'application.**
- Documentation **: Choisir un hébergeur web**

4. Conclusion

En suivant ces recommandations, ToDo & Co pourra améliorer significativement la qualité du code et les performances de son application Symfony. L'analyse de Code Climate et Blackfire indique une excellente qualité de code et des pistes d'optimisation des performances, respectivement. Les optimisations proposées pour l'autoloader, OPCache, RealpathCache et Varnish contribueront à améliorer les performances.

Suggestions pour la Suite :

1. **Intégrer Code Climate et Blackfire dans un pipeline CI/CD** : Pour des vérifications automatiques et continues à chaque commit.
2. **Configurer des profils de performance réguliers avec Symfony Profiler** : Suivre les recommandations pour optimiser continuellement les performances de l'application.

---

5. Lexique

- **Code Climate** : Une plateforme d'analyse de code qui évalue la maintenabilité et la qualité globale du code.
- **Symfony Profiler** : Un outil intégré à Symfony qui fournit des informations détaillées sur les performances et le comportement des requêtes.
- **Composer** : Un gestionnaire de dépendances pour PHP, utilisé pour installer et mettre à jour les bibliothèques de l'application.
- **OPCache** : Une extension PHP qui améliore les performances en stockant en mémoire partagée le bytecode des scripts pré-compilés.
- **RealpathCache** : Une fonctionnalité de PHP qui met en cache les chemins absolus pour améliorer les performances des applications.
- **Varnish** : Un accélérateur HTTP open source qui améliore les performances des sites web en mettant en cache le contenu.
- **CI/CD (Continuous Integration/Continuous Deployment)** : Une pratique de développement logiciel où les modifications de code sont automatiquement testées et déployées.

---

6. Suggestions d'Amélioration

6.1 Suivi des Performances et Surveillance

1. **Mettre en Place une Surveillance Continue**
   - Utiliser des outils de surveillance tels que **New Relic**, **Datadog** ou **Elastic APM** pour obtenir des informations en temps réel sur les performances de l'application.
   - Ces outils peuvent aider à identifier les problèmes de performance avant qu'ils n'affectent les utilisateurs finaux.
2. **Configurer des Alertes de Performance**
   - Configurer des alertes basées sur des seuils de performance pour être averti immédiatement lorsque quelque chose ne va pas.
   - Cela permet une réaction rapide aux problèmes de performance.

6.2 Optimisations de la Base de Données

1. **Indexation de la Base de Données**
   - Analyser les requêtes SQL les plus fréquentes et ajouter des index aux colonnes utilisées dans les clauses WHERE et JOIN.
   - Utiliser l'analyseur de requêtes SQL pour identifier les goulots d'étranglement dans les requêtes.
2. **Utilisation de Caching**
   - Mettre en cache les résultats des requêtes fréquentes en utilisant des systèmes de cache comme **Redis** ou **Memcached**.
   - Utiliser Doctrine Cache pour mettre en cache les métadonnées des entités, les requêtes, et les résultats des requêtes.

6.3 Améliorations du Code

1. **Refactoring Régulier**
   - Effectuer régulièrement des sessions de refactoring pour améliorer la lisibilité, la maintenabilité et les performances du code.
   - Utiliser des outils comme **PHPStan** et **Psalm** pour l'analyse statique du code et pour détecter les bugs potentiels.

2. **Utilisation de Traits et Services**
    - Utiliser des traits pour réutiliser le code commun entre plusieurs classes.
    - Adopter le principe de l'injection de dépendances pour améliorer la testabilité et la modularité du code.

6.4 Améliorations de la Sécurité

1. **Revue de Sécurité du Code**
    - Effectuer des revues de code régulières pour identifier et corriger les vulnérabilités de sécurité.
    - Utiliser des outils d'analyse de sécurité comme **SonarQube** pour détecter les vulnérabilités de sécurité dans le code.
2. **Mises à Jour Régulières**
    - Mettre à jour régulièrement les dépendances et le framework pour bénéficier des derniers correctifs de sécurité.
    - Utiliser **Dependabot** ou des outils similaires pour automatiser la vérification des mises à jour des dépendances.

6.5 Documentation et Formation

1. **Améliorer la Documentation**
    - Documenter le code et les configurations importantes pour faciliter la maintenance et le transfert de connaissances.
    - Utiliser des outils comme **Swagger** pour documenter les API.
2. **Formation Continue**
    - Encourager l'équipe de développement à suivre des formations régulières pour se tenir à jour avec les meilleures pratiques et les nouvelles technologies.
    - Participer à des conférences et à des webinaires sur le développement Symfony et PHP.

En suivant ces suggestions d'amélioration, ToDo & Co pourra non seulement améliorer la qualité et les performances de son application, mais aussi assurer une meilleure sécurité, maintenabilité, et évolutivité du code.