2020

Rapport d'audit TODO LIST

Alexandre Manteaux

TODO & CO

31/08/2020

Table des matières

Introduction 3

Analyse préliminaire 4

1. Les bibliothèques utilisées 4
   a. Le framework : Symfony 4
   b. Les bibliothèques tierces 4
2. Les anomalies 4
   a. Anomalies rapportées à corriger 4
   b. Autres anomalies identifiées 5
3. Tests 5
   a. Tests unitaires 5
   b. Tests fonctionnels 5
4. Qualité du code 6
   a. Respect des bonnes pratiques 6
   b. Sécurité 6
   c. Analyse avec Codacy 6
5. Analyse de la performance avec Blackfire 6

Solutions apportées 9

1. Mise à jour de Symfony et des bibliothèques 9
   a. Mise à jour de Symfony 9
   b. Mise à jour des bibliothèques externes. 10
2. Correction des anomalies 11
   a. Correction des anomalies rapportées 11
   b. Correction des anomalies identifiées après audit 11
3. Ajout des fonctionnalités 12
4. Ajout des tests 13
   a. Les tests unitaires 13
   b. Les tests fonctionnels 15
   c. Résultats des tests et rapport de couverture 16

   Au total, ce sont 117 assertions pour 65 tests qui ont été implémentés : 16
5. Respect des bonnes pratiques, analyse Codacy 16
   a. Respect des bonnes pratiques 16

c. Analyse Codacy 18
6. Optimisation de la performance 18

Introduction

TODO LIST est une application Web de gestion des tâches quotidiennes.
Réalisée dans l'urgence, celle-ci a été développée à l'aide du framework PHP Symfony et n'est, à l'heure actuelle, qu'un concept de type MVP.
Suite à une levée de fond, l'application va pouvoir être développée.
En tant que développeur expérimenté, il est demandé de réaliser un audit de la qualité et de la performance de l'application.
Si certains problèmes ont déjà été répertoriés, il convient de relever toutes les anomalies et de les corriger. De plus, la direction souhaite que certaines fonctionnalités soient implémentées.

Analyse préliminaire

1. Les bibliothèques utilisées

a. Le framework : Symfony

L'application utilise la version 3.1.10 du framework. Cette version est sortie en mai 2016 et n'est plus maintenue depuis juillet 2017. Il est indispensable d'effectuer une mise à jour du framework.

Par ailleurs, une analyse rapide permet de voir qu'elle comporte de nombreuses dépréciations qu'il convient de corriger.

b. Les bibliothèques tierces

Tous les composants de Symfony et les bibliothèques tierces sont elles aussi calquées sur la version de Symfony et ne sont donc pour la plupart, plus à jour.

2. Les anomalies

a. Anomalies rapportées à corriger

Plusieurs anomalies ont été rapportées par les développeurs du concept. Il nous est demandé de les corriger, voici le détail des modifications à apporter :

- Rattacher la tâche à l'utilisateur qui l'a créée.
- Lors de la modification de la tâche, l'auteur ne peut pas être modifié.
- Pour les tâches déjà créées, elles doivent être rattachées à un utilisateur « anonyme ».
- Choisir un rôle (utilisateur ou administrateur) lors de la création et de la modification d'un compte utilisateur.

b. Autres anomalies identifiées

De nombreuses anomalies ont été identifiées après examen :

- Absence de lien vers la page d'accueil.
- Absence de page pour les tâches marquées comme « Terminées » (mais le lien existe).
- Absence de lien vers la page d'édition des utilisateurs (mais page existe).
- Plusieurs problèmes de validation sur les entités *User* et *Task*
  Ex : Les comptes utilisateurs peuvent avoir le même username et la même adresse email.
- A la création d'un compte, la redirection s'effectue sur la page de gestion des utilisateurs (accès normalement réservé aux administrateurs).
- Problème de traduction des formulaires.
- Traduction des messages de validation et des erreurs.
- Absence de message flash sur certaines actions.
- Absence de pages d'erreurs en production.

3. Tests

a. Tests unitaires

Pas de tests unitaires.

b. Tests fonctionnels

Seulement un seul test présent sur l'accès à la page d'accueil.

4. Qualité du code

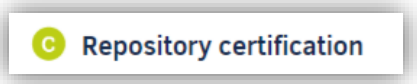
a. Respect des bonnes pratiques

Les bonnes pratiques sont dans la documentation de Symfony. Cette dernière reprend en partie les PSR. En l'espèce, celles-ci n'ont pas été respectées dans le concept. Effectivement, on relève notamment un nombre important de sauts de lignes inutiles mais aussi peu de cohérence dans la mise en forme du code.

b. Sécurité

Si l'ensemble des formulaires de l'application sont protégés contre la faille CSRF, ce n'est pas le cas du formulaire d'authentification. Il convient donc de remédier de manière urgente à cette vulnérabilité.

c. Analyse avec Codacy



Le service *Codacy* permet de données une note générale sur la qualité du code du point de vue des bonnes pratiques mais aussi de la sécurité. Pour cette application, la note obtenue est de « C » ce qui est passable.

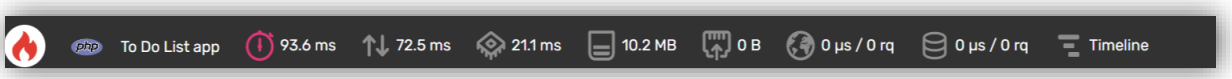
5. Analyse de la performance avec Blackfire.io

Blackfire permet de réaliser des tests de performance très complets sur l'exécution de l'application, page par page.

La barre supérieure du tableau de bord nous donne un résumer des métriques de performance globale de l'exécution d'une page.

Voici les performances de la page de login :

https://blackfire.io/profiles/0d4a6ac2-b394-440d-a054-a8b8a8140b1b/graph



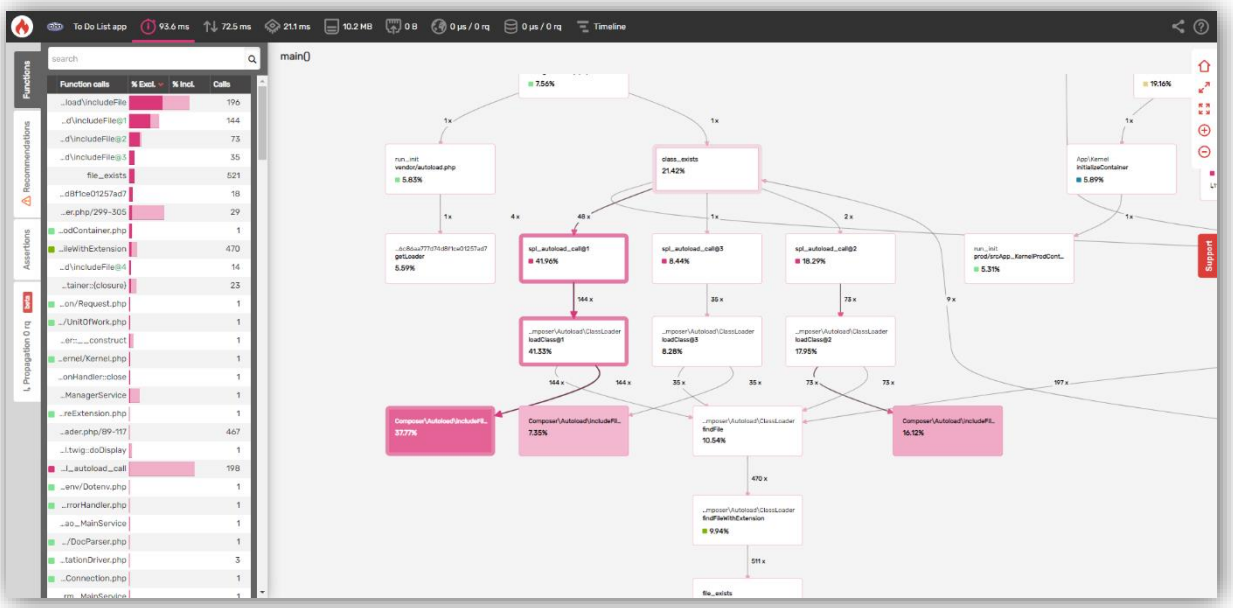
On a de gauche à droite:

- Le temps total de génération de la page.
- Le temps des entrées/sorties.
- Le temps d'exécution du processeur.
- La mémoire vive consommée.
- Les données réseaux.
- Les sous-requêtes HTTP.
- Le nombre de requêtes vers la base de données et le temps.

Blackfire fonctionne à l'échelle des fonctions et ou des fichiers. Concrètement, que ce soit en représentation sous forme de graphe (au centre) ou sous forme de liste (à gauche), pour chaque fonction, Blackfire nous donne :

- Le nombre de fois qu'elle est appelée et par qu'elles autres fonctions : les « callers ».
- Le nombre de fonction qu'elle appelle et lesquelles : les « callees ».
- Le temps d'exécution :
    - exclusif (de la fonction seule)
    - inclusif (de la fonction et des autres fonctions appelées)

Voici par exemple la représentation de l'audit de la page de login :



On constate que c'est la fonction *includeFile()* de Composer qui est la fonction la moins performante de la page. De plus, cette fonction est aussi la moins performante sur l'ensemble des pages de l'application. C'est l'autoloader de Composer qui est à l'origine de cette consommation de ressources importante. Il convient donc en priorité de s'intéresser à ce composant et de découvrir s'il existe une solution pour le solliciter moins fréquemment.

Solutions apportées

1. Mise à jour de Symfony et des bibliothèques

a. Mise à jour de Symfony

Pour garantir une bonne stabilité à l'application, il est préférable d'utiliser une version LTS de Symfony.

La version 3 de Symfony LTS, la 3.4.43 possède une date de fin de support pour novembre 2021.

La version 5 de Symfony est la dernière en date mais son manque de stabilité dû à son manque de maturité incite à opter pour une solution de compromis.

La version LTS de Symfony 4, la 4.4.10 est une version éprouvée dont le support est assuré jusqu'en novembre 2023.

Symfony a donc été mis à jour dans sa version 4.4.10.

Toutes les dépendances ont été mise à jour dans des versions supportées et d'autres ont été installées, par exemple :

- symfony/browser-kit
- symfony/css-selector
- symfony/maker-bundle
- symfony/phpunit-bridge

Les dépréciations actuelles ainsi que celles qui anticipent le passage à Symfony 5 ont aussi été corrigées pour faciliter une éventuelle mise à jour de l'application.

b. Mise à jour des bibliothèques externes.

Les bibliothèques externes ont été mises à jour pour fonctionner avec Symfony 4. Certaines ont été installées essentiellement pour le développement :

- doctrine/doctrine-fixtures-bundle
- fzaninotto/faker
- liip/test-fixtures-bundle

2. Correction des anomalies

a. Correction des anomalies rapportées

Les anomalies qui étaient déjà identifiées ont été corrigées :

- Chaque tâche créée est rattachée à son créateur.
- Lors de la modification de la tâche, l'auteur ne peut pas être modifié.
- Pour les tâches déjà créées, elles sont rattachées à un utilisateur « anonyme ».
- Il est possible de choisir le rôle de l'utilisateur lors de la création du compte ou lors de son édition.

b. Correction des anomalies identifiées après audit

Les anomalies découvertes ont été corrigées :

- Ajout des liens vers la page d'accueil.
- Création de la page des tâches terminées.
- Les contraintes de validation ont été ajoutées, il n'est notamment plus possible de créer des comptes avec le même Login ou la même adresse email. (Contrainte d'unicité).
- A la création d'un compte, l'utilisateur est redirigé vers la page de login.
- Des traductions sur les formulaires ont été ajoutées ou modifiées.
- Ajout des messages flash de confirmation pour les actions qui n'en disposaient pas.
- Ajout de pages d'erreurs pour l'environnement de production.
- Les boutons pour changer le statut des tâches indiquent le bon statut.
- La date de création et le créateur de la tâche est affiché pour chaque tâche.

3. Ajout des fonctionnalités

Les fonctionnalités souhaitées par la direction ont été réalisées :

- Seuls les administrateurs ont accès à la gestion des utilisateurs.
- Les tâches ne peuvent être supprimées que par leurs auteurs.
- Pour les tâches anonymes, seuls les administrateurs peuvent les supprimer.

Il s'agit essentiellement de fonctionnalités liées à l' *access control* du composant *Security* de *Symfony*.

Le système d'annotation a été préféré à la rubrique *access_control* du fichier *security.yaml* pour une meilleure compréhension du code.

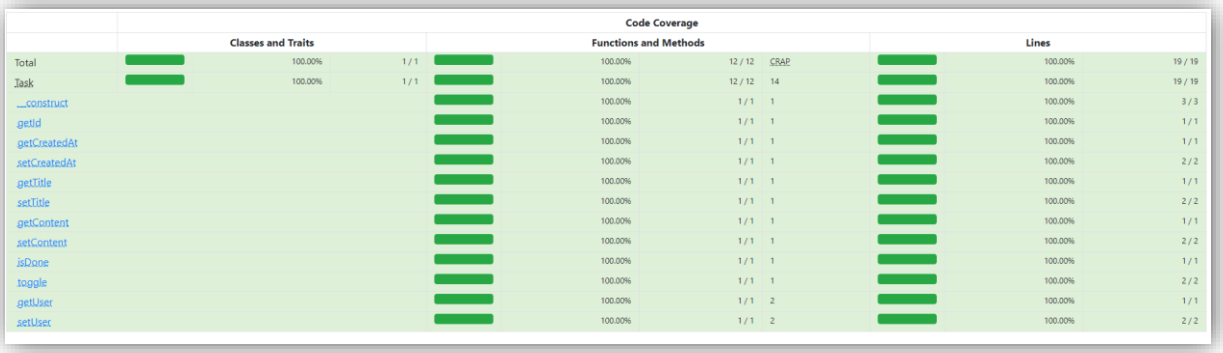
Pour la suppression des tâches par leur auteur et pour la suppression des tâches anonymes par les administrateurs, il a été nécessaire de mettre en place un système de *Voter*.

4. Ajout des tests

a. Les tests unitaires

Les tests unitaires ont été réalisés sur les entités. Il s'agit de tests classiques sur les *setters* et sur les *getters*. De plus, des tests sur les contraintes de validation ont été également implémentés.

Exemple pour la classe *Task* :



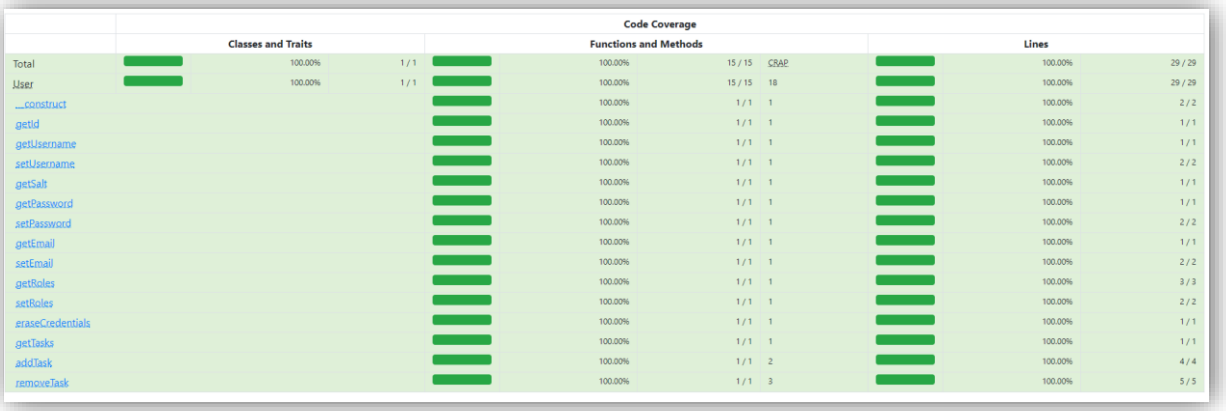
	Code Coverage								
	Classes and Traits			Functions and Methods			Lines		
Total	100.00%	1 / 1		100.00%	12 / 12	CRAP	100.00%	19 / 19	
Task	100.00%	1 / 1		100.00%	12 / 12	14	100.00%	19 / 19	
__construct				100.00%	1 / 1	1	100.00%	3 / 3	
getId				100.00%	1 / 1	1	100.00%	1 / 1	
getCreatedAt				100.00%	1 / 1	1	100.00%	1 / 1	
setCreatedAt				100.00%	1 / 1	1	100.00%	2 / 2	
getTitle				100.00%	1 / 1	1	100.00%	1 / 1	
setTitle				100.00%	1 / 1	1	100.00%	2 / 2	
getContent				100.00%	1 / 1	1	100.00%	1 / 1	
setContent				100.00%	1 / 1	1	100.00%	2 / 2	
isDone				100.00%	1 / 1	1	100.00%	1 / 1	
toggle				100.00%	1 / 1	1	100.00%	2 / 2	
getUser				100.00%	1 / 1	2	100.00%	1 / 1	
setUser				100.00%	1 / 1	2	100.00%	2 / 2	

Tests de validation :

```
1.  public function testValidEntity()
2.    {
3.        $error = self::$container->get('validator')->validate($this->task);
4.        $this->assertCount(0, $error);
5.    }
6.
7.    public function testInvalidEmptyTitle()
8.    {
9.        $this->task->setTitle('');
10.       $error = self::$container->get('validator')->validate($this->task);
11.       $this->assertCount(1, $error);
12.   }
13.
14.   public function testInvalidUsedTitle()
15.   {
16.       $this->loadFixtures([AppFixtures::class]);
17.       $this->task->setTitle('titre');
18.       $error = self::$container->get('validator')->validate($this->task);
19.       $this->assertCount(1, $error);
20.   }
21.
22.   public function testInvalidEmptyContent()
23.   {
24.       $this->task->setContent('');
25.       $error = self::$container->get('validator')->validate($this->task);
26.       $this->assertCount(1, $error);
27.   }
28.
29.   public function testDefaultIsDoneFalse()
30.   {
```

```
31.         $this->assertSame(false, $this->task->isDone());
32.     }
```

Exemples pour la classe *User*:



Code Coverage

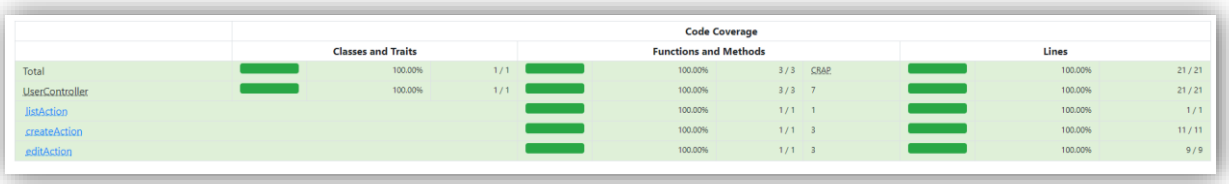
	Classes and Traits		Functions and Methods			Lines	
Total	100.00%	1 / 1	100.00%	15 / 15	CRAP	100.00%	29 / 29
User	100.00%	1 / 1	100.00%	15 / 15	18	100.00%	29 / 29
__construct			100.00%	1 / 1	1	100.00%	2 / 2
getId			100.00%	1 / 1	1	100.00%	1 / 1
getUsername			100.00%	1 / 1	1	100.00%	1 / 1
setUsername			100.00%	1 / 1	1	100.00%	2 / 2
getSalt			100.00%	1 / 1	1	100.00%	1 / 1
getPassword			100.00%	1 / 1	1	100.00%	1 / 1
setPassword			100.00%	1 / 1	1	100.00%	2 / 2
getEmail			100.00%	1 / 1	1	100.00%	1 / 1
setEmail			100.00%	1 / 1	1	100.00%	2 / 2
getRoles			100.00%	1 / 1	1	100.00%	3 / 3
setRoles			100.00%	1 / 1	1	100.00%	2 / 2
eraseCredentials			100.00%	1 / 1	1	100.00%	1 / 1
getTasks			100.00%	1 / 1	1	100.00%	1 / 1
addTask			100.00%	1 / 1	2	100.00%	4 / 4
removeTask			100.00%	1 / 1	3	100.00%	5 / 5

Tests de validation :

```
1.  public function testNoTasks()
2.  {
3.      $this->assertEmpty($this->user->getTasks());
4.  }
5.
6.  public function testAddTasks()
7.  {
8.      $taskStub = $this->createMock(Task::class);
9.      $this->user->addTask($taskStub);
10.     $this->assertInstanceOf(Collection::class, $this->user->getTasks());
11. }
12.
13. public function testRemoveTask()
14. {
15.     $taskStub = $this->createMock(Task::class);
16.     $this->user->addTask($taskStub);
17.     $taskStub->method('setUser');
18.     $taskStub->method('getUser')->willReturn($this->user);
19.     $this->user->removeTask($taskStub);
20.     $this->assertEmpty($this->user->getTasks());
21. }
22.
23. public function testValidEntity()
24. {
25.     $error = self::$container->get('validator')->validate($this->user);
26.     $this->assertCount(0, $error);
27. }
```

b. Les tests fonctionnels

Les tests fonctionnels ont pour vocation à tester le bon fonctionnement des fonctionnalités de l'application, comme un utilisateur pourrait le faire. Il faut donc émuler une requête, grâce à l'implémentation d'un client, et veiller à obtenir la réponse attendue. Ces tests utilisent la base de données.



	Code Coverage								
	Classes and Traits			Functions and Methods			Lines		
Total	100.00%	1 / 1		100.00%	3 / 3	CRAP	100.00%	21 / 21	
UserController	100.00%	1 / 1		100.00%	3 / 3	7	100.00%	21 / 21	
listAction				100.00%	1 / 1	1	100.00%	1 / 1	
createAction				100.00%	1 / 1	3	100.00%	11 / 11	
editAction				100.00%	1 / 1	3	100.00%	9 / 9	

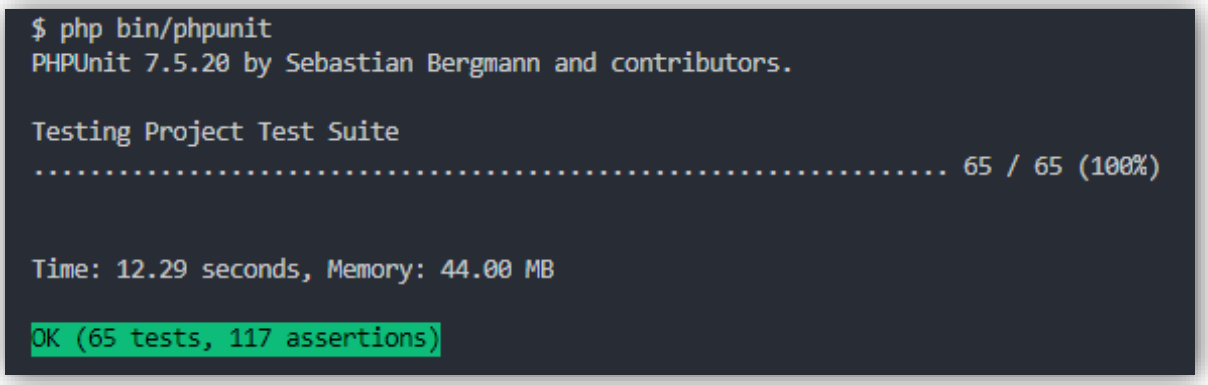
Exemple de tests sur le controller *UserControllerTest* :

```php
public function testUserNotLoggedList()
{
    $client = static::createClient();
    $client->request('GET', '/users');
    $this->assertResponseStatusCodeSame(Response::HTTP_FOUND);
    $client->followRedirect();
    $this->assertResponseStatusCodeSame(Response::HTTP_OK);
    $this->assertSelectorExists('label', 'Mot de passe');
}

public function testUserAuthentifiedList()
{
    $client = static::createClient([], [
        'PHP_AUTH_USER' => 'ludo06',
        'PHP_AUTH_PW'   => 'password'
    ]);
    $client->request('GET', '/users');
    $this->assertResponseStatusCodeSame(Response::HTTP_FORBIDDEN);
}

public function testAdminAuthentifiedList()
{
    $client = static::createClient([], [
        'PHP_AUTH_USER' => 'alex06',
        'PHP_AUTH_PW'   => 'password'
    ]);
    $client->request('GET', '/users');
    $this->assertResponseStatusCodeSame(Response::HTTP_OK);
}

public function testCreateUser()
{
    $client = static::createClient();
    $client->followRedirects();
    $crawler = $client->request('GET', '/users/create');
    $form = $crawler->selectButton('Ajouter')->form([
        'user[username]' => 'fred06',
        'user[roles]' => 'ROLE_USER',
        'user[password][first]' => 'password',
        'user[password][second]' => 'password',
        'user[email]' => 'fred06@gmail.com'
```

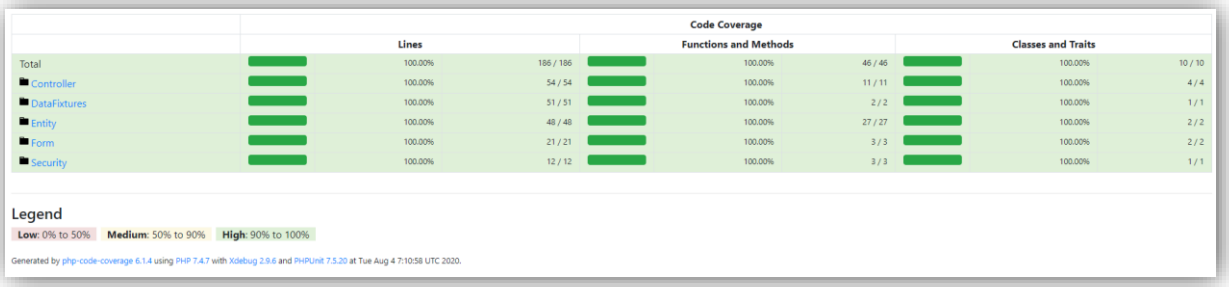
```
42.     ]);
43.     $client->submit($form);
44.     $this->assertResponseStatusCodeSame(Response::HTTP_OK);
45.     $this->assertSelectorExists('.alert-success');
46.     $this->assertSelectorExists('label', 'Mot de passe');
47. }
```

c. Résultats des tests et rapport de couverture

Au total, ce sont 117 assertions pour 65 tests qui ont été implémentés :



Comme indiqué ci-dessous, les tests ont un rapport de couverture de 100% sur la couche métier de l'application.



5. Respect des bonnes pratiques, analyse Codacy

a. Respect des bonnes pratiques

Le code a été en grande partie remanié :

- **La mise en page a été améliorée.**
- **Les noms de variables trop cours ont été modifiés.**
- **Du code inutile a été supprimé.**
- **Les sauts de ligne inutiles ont été supprimés et une refactorisation du code a été effectuée.**

Exemple de code avant mise en forme :

```
1.  /**
2.   * @Route("/tasks/create", name="task_create")
3.   */
4.  public function createAction(Request $request)
5.  {
6.      $task = new Task();
7.      $form = $this->createForm(TaskType::class, $task);
8.
9.      $form->handleRequest($request);
10.
11.     if ($form->isValid()) {
12.         $em = $this->getDoctrine()->getManager();
13.
14.         $em->persist($task);
15.         $em->flush();
16.
17.         $this->addFlash('success', 'La tâche a été bien été ajoutée.');
18.
19.         return $this->redirectToRoute('task_list');
20.     }
21.
22.     return $this->render('task/create.html.twig', ['form' => $form-
    >createView()]);
23. }
```

Le même code respectant les bonnes pratiques de *Symfony* :

```
1.  /**
2.   * @Route("/tasks/create", name="task_create")
3.   * @IsGranted("ROLE_USER")
4.   */
5.  public function createAction(Request $request, EntityManagerInterface $entityManager
    )
6.  {
7.      $task = new Task();
8.      $form = $this->createForm(TaskType::class, $task);
9.      $form->handleRequest($request);
10.
11.     if ($form->isSubmitted() && $form->isValid()) {
12.         $task->setUser($this->getUser());
13.         $entityManager->persist($task);
14.         $entityManager->flush();
15.         $this->addFlash('success', 'La tâche a été bien été ajoutée.');
16.
17.         return $this->redirectToRoute('task_list');
18.     }
19.
20.     return $this->render('task/create.html.twig', ['form' => $form-
    >createView()]);
21. }
```

b. Sécurité

Le formulaire d'authentification est désormais protégé contre la faille CSRF.

c. Analyse Codacy

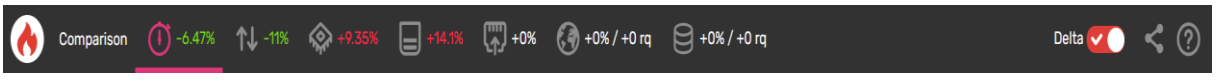
Une fois le code remanié et refactorisé, l'analyse Codacy donne une note de B pour la couche métier du projet.

6. Optimisation de la performance

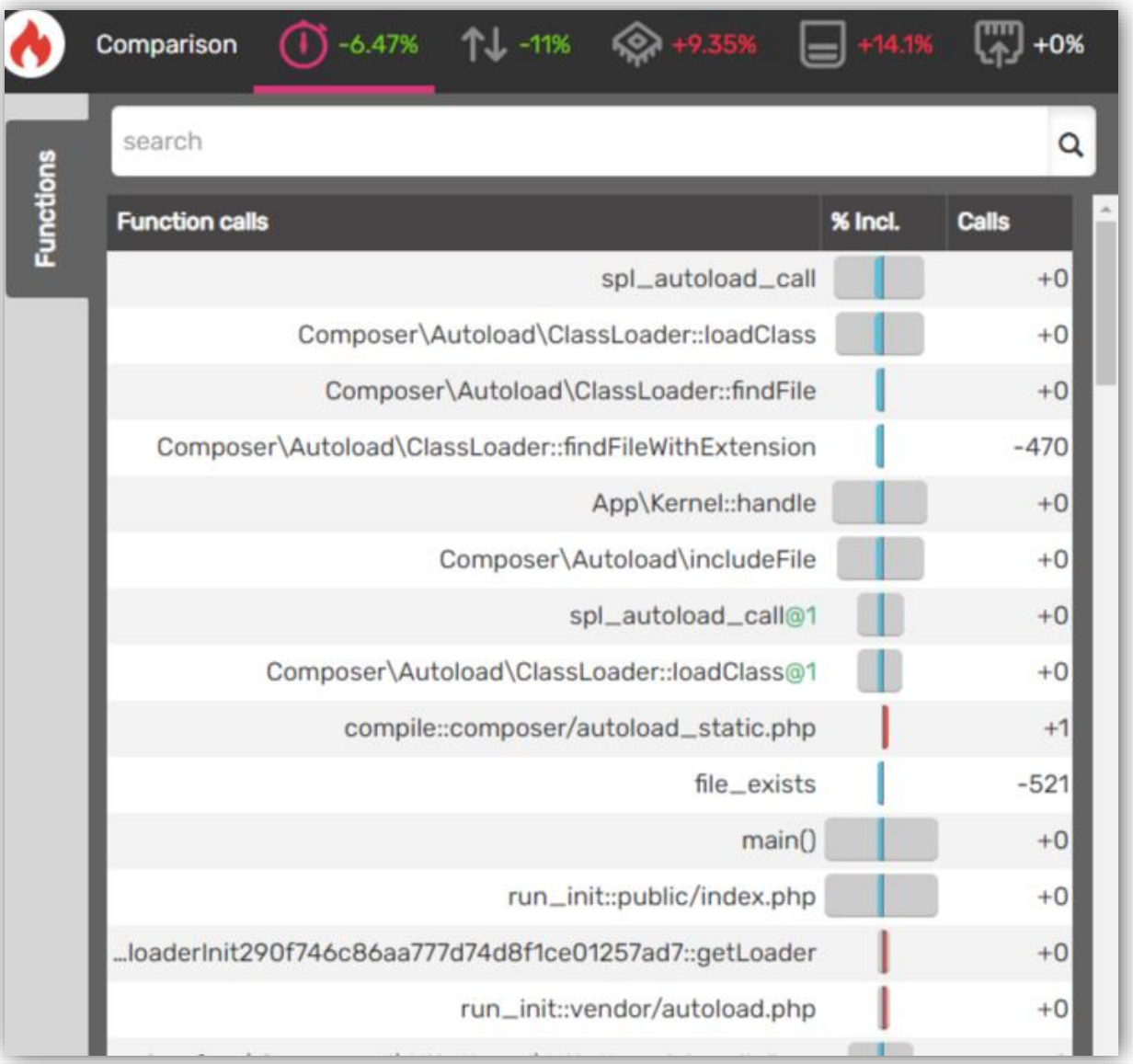
Lors de l'analyse effectuée sur l'application, il nous est apparu que le système d'autoloading de Composer (notamment l'utilisation de la fonction *includeFile()*), était un des principaux consommateurs de ressources. Après quelques recherches, il est indiqué qu'il est possible d'améliorer les performances de ce dernier grâce à la commande :

```
composer dump-autoload -o
```

Celle-ci permet de regrouper dans un fichier l'ensemble des classes et réduit les multiples appels à certaines de ses fonctions. Ainsi, lorsque Composer aura à utiliser une classe, il n'ira plus chercher si le fichier contenant la classe existe, il consultera la liste des classes dans un fichier. En effet, après un nouveau test de la page *login*, en utilisant le comparateur de Blackfire, on a un gain de performance relativement substantiel :

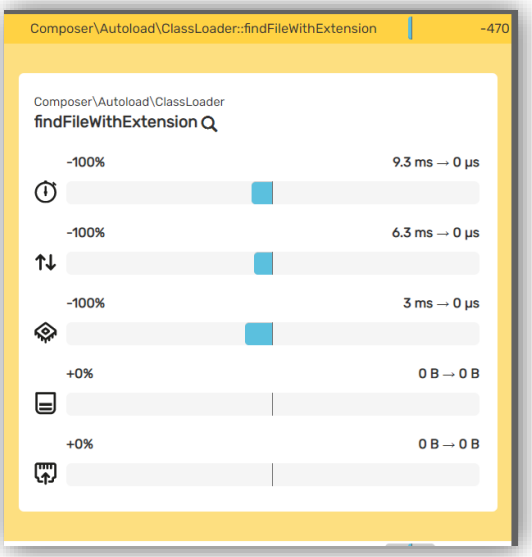


Les performances font un léger bon de 6% même si c'est au prix d'une augmentation de la mémoire vive consommée et de l'utilisation du processeur.

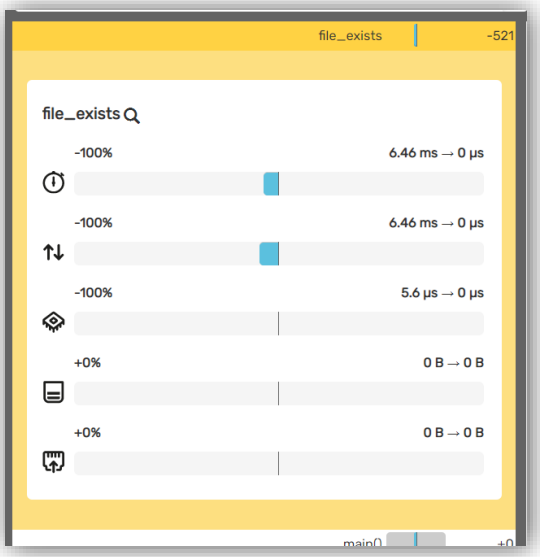


En se penchant sur la liste de fonctions représentée ci-dessus, on peut voir le différentiel dans les appels de plusieurs fonctions liées à Composer :

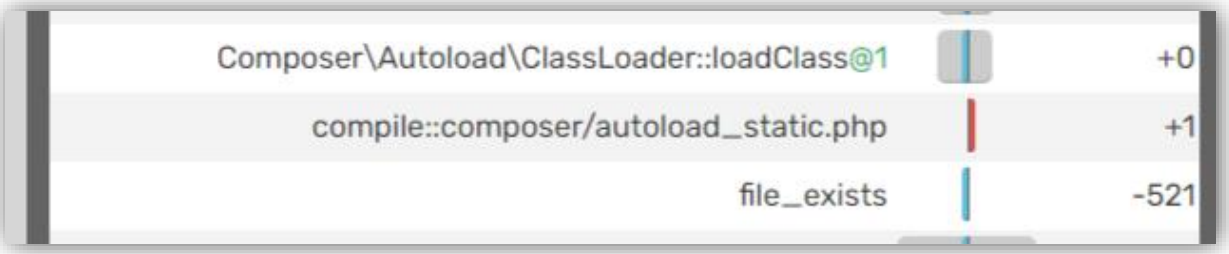
- Pour la fonction *findFileWithExtension()* de la classe *ClassLoader* de Composer, l'analyse montre 470 appels de moins par rapport à avant. On constate sur les détails que celle dernière n'est en faite plus appelée par le système :



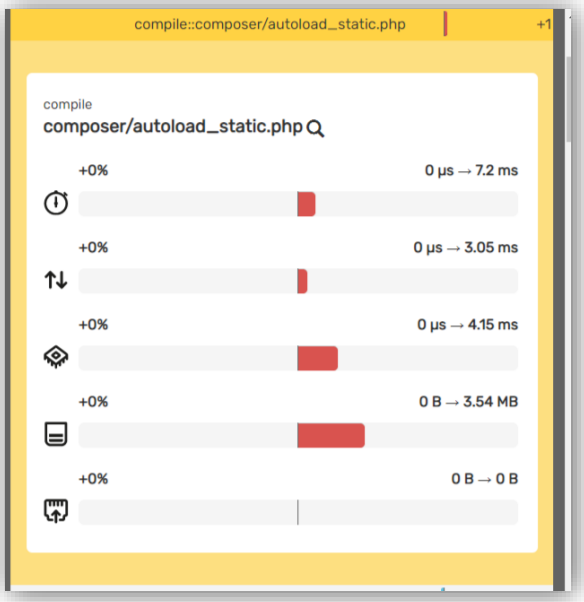
- La fonction système *file_exists()* a 521 appels de moins. Là encore cette dernière n'est plus appelée par le système :



En revanche, le différentiel montre un appel supplémentaire à un nouveau fichier :



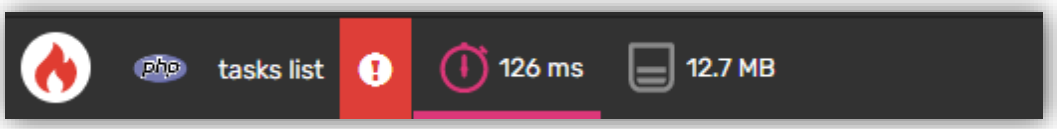
C’est le fichier nouvellement créé par l’optimisation de Composer. Celui-ci contient la liste de toutes les classes.



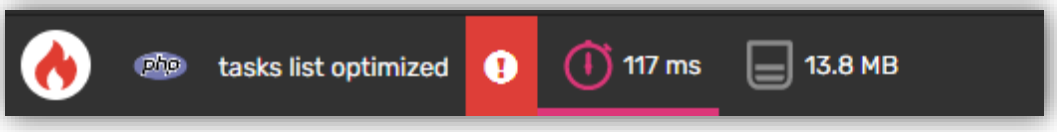
Si on compare d’autres pages, on constate que le gain est bien général sur toute l’application :

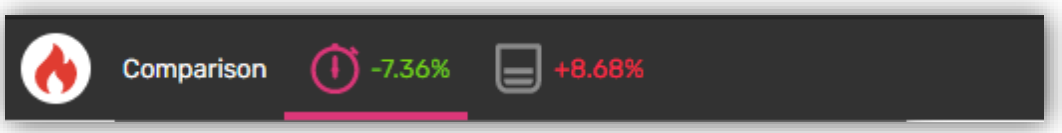
- **Page de la liste de toutes les tâches :**

https://blackfire.io/profiles/6ada4ccf-1a5e-4097-9460-78fa8ce9e3b2/graph :



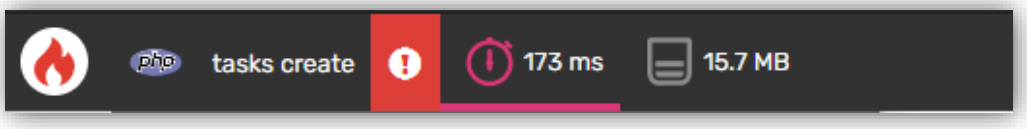
https://blackfire.io/profiles/239e621f-1cc8-47c0-94e5-364db08194a8/graph :



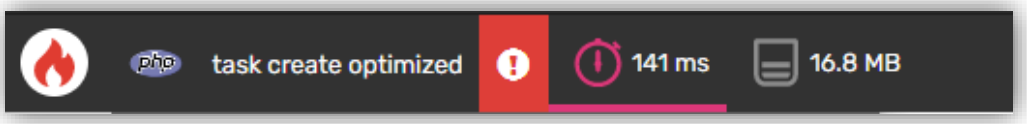


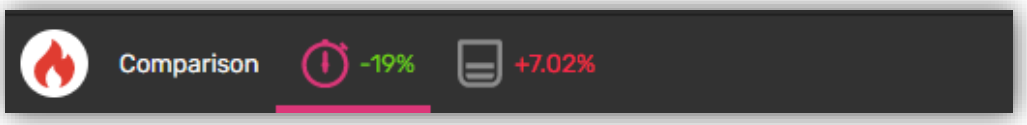
- **Page du formulaire de création d'une tâche :**

https://blackfire.io/profiles/194bdca4-d9c3-4227-bf82-75d1faffdbf5/graph :



https://blackfire.io/profiles/ac22720f-971a-46a8-9ecd-f56658fef1b6/graph :





On constate tout de même un coût en mémoire vive plus important. Toutefois, cette modification permet un gain de performance global, c'est donc un choix payant.