AUDIT TODO & CO

Réalisé par Simon Charbonnier Mai 2024

SOMMAIRE

1- Etat initial

a. Version Symfony

A l'origine l'application a été développé à l'aide du framework Symfony 3.1. Cette version n'est actuellement plus entretenue et n'est donc plus à jour.

b. Fonctionnalités

Les fonctionnalités initiales sont les suivantes :

- En tant que visiteur, je peux accéder à la page d'accueil.
- En tant que visiteur, je peux accéder à la liste des tâches.
- En tant que visiteur, je peux créer une tâche (sans y être associé).
- En tant que visiteur, je peux modifier n'importe quelle tâche.
- En tant que visiteur, je peux toggle n'importe quelle tâche.
- En tant que visiteur, je peux supprimer n'importe quelle tâche.
- En tant que visiteur, je peux m'inscrire.
- En tant que visiteur, je peux accéder à la page de gestion des utilisateurs.
- En tant que visiteur, je peux modifier n'importe quel utilisateur.
- En tant que visiteur, je peux me connecter.

c. Problèmes

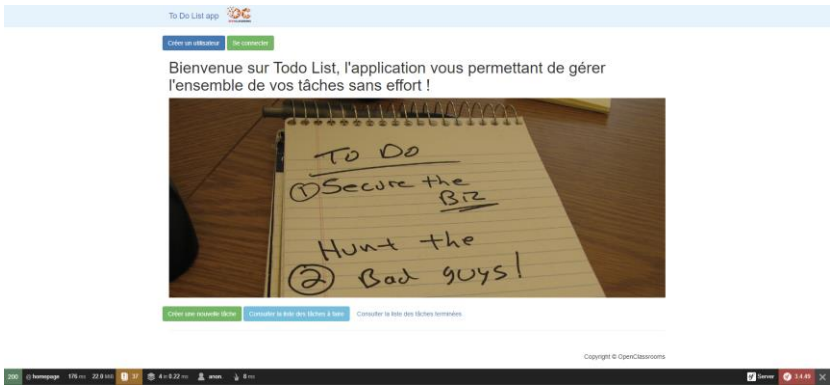
Ces fonctionnalités posent 2 problèmes importants, il n'y a aucun système d'autorisation au sein de l'application (tout le monde peut tout faire) et on ne peut pas savoir à qui appartient chaque tâche.

2- Améliorations

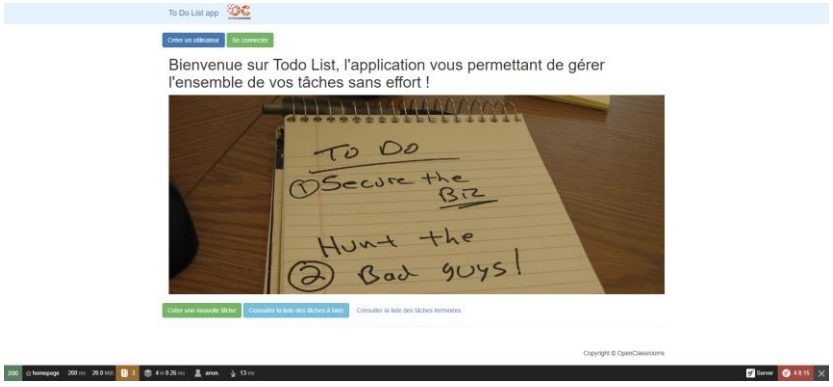
a. Version Symfony

La version Symfony de l'application n'étant plus d'actualité, il a été décidé de migrer le site vers la dernière version LTS (Long-Term Support). J'ai donc augmenté la version du Framework de version majeure en version majeure jusqu'à atteindre Symfony 5.4 tout en adaptant les dépendances et le code à chaque fois.

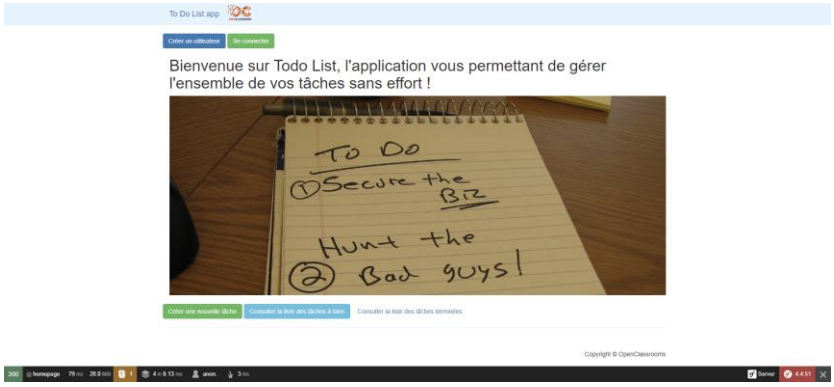
Symfony version 3.4 :



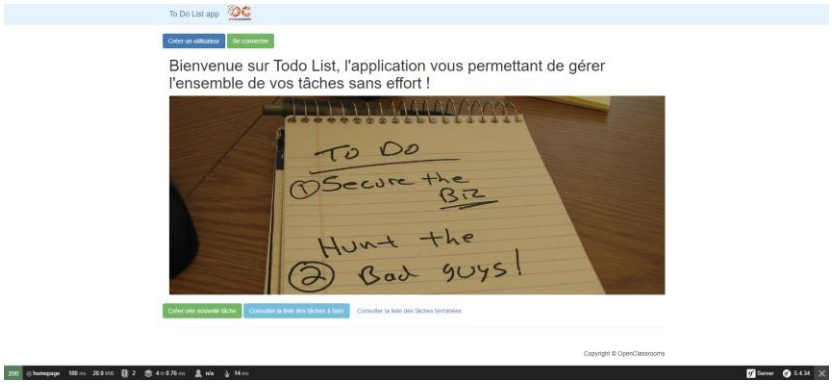
Symfony version 4.0 :



Symfony version 4.4 :



Symfony version 5.4 :



Une fois la migration de Symfony faite, il faut corriger les problèmes évoqués plus tôt.

b. Une tâche doit être attachée à un utilisateur

Premièrement, une tâche n'est pas associée à l'utilisateur qui la crée.
Cela signifie que l'entité *task* n'a pas de lien avec l'entité *user.* J'ai donc mis à jour les fichiers *src/Entity/Task.php* et *src/Entity/User.php*.
Puis j'ai généré et exécuté la migration de base de données. Ensuite on doit rajouter l'utilisateur à la tâche lors de la création de celle-ci, donc dans *src/Controller/TaskController.php.*

https://github.com/Simoncharbonnier/OCP8/commit/c888718a88f73d11a5f4804b30660cf64d24fa7c

c. Choisir un rôle pour un utilisateur

Maintenant qu'une tâche est associée à l'utilisateur qui la crée, il faut gérer l'autorisation des utilisateurs. Actuellement un utilisateur n'a pas de rôles. Donc il faut mettre à jour le fichier *src/Entity/user.php*, générer et exécuter la migration. Afin de choisir les rôles, on peut ajouter un champ au formulaire concernant l'inscription et la modification d'un utilisateur dans *src/Form/UserType.php*.

https://github.com/Simoncharbonnier/OCP8/commit/c60edf313e933205b7126d6c1b6478a5c32298bb

https://github.com/Simoncharbonnier/OCP8/commit/1482aed5c025b6219b3d8eac128312fb13c3bf28

d. Autorisation

Chaque utilisateur possède ses rôles, "ROLE_USER" pour un utilisateur par défaut et "ROLE_ADMIN" pour un administrateur. Symfony propose l'utilisation de "voters" pour gérer l'autorisation d'une application. C'est ce que j'ai choisi d'utiliser.
Dans *src/Security/TaskVoter.php* & *src/Security/UserVoter.php* gèrent respectivement les autorisations concernant les tâches et les utilisateurs.

En ajoutant une ligne de code faisant appel au voter correspondant dans chaque route, on obtient une application refusant l'accès à certains utilisateurs pour certaines fonctionnalités, en fonction de ce que l'on a défini dans les "voters".
J'ai également mis à jour les templates pour afficher ou non les boutons en fonction des droits de l'utilisateur.
Les voters renvoient des "AccessDeniedError" lors des refus d'accès à certaines fonctionnalités, il est donc nécessaire de mettre en place un comportement correct lors de ces erreurs. C'est-à-dire que l'application redirige l'utilisateur avec une petite notification lui indiquant qu'il n'a pas les permissions nécessaires. Pour faire cela, on peut utiliser un "EventListener" qui va intercepter l'erreur avant que l'utilisateur ne la voie (*src/EventListener/AccessDeniedListener.php*) et reproduire le comportement voulu.

https://github.com/Simoncharbonnier/OCP8/commit/870c1c0b712258a80fd04cfa0729f80b4c97eaa0

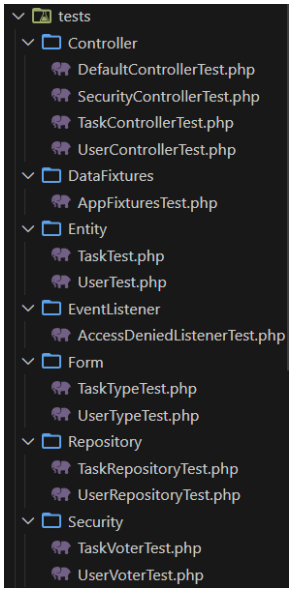
https://github.com/Simoncharbonnier/OCP8/commit/40ba41ec89523356f862e6a069335583f5c3d5c8

https://github.com/Simoncharbonnier/OCP8/commit/a09bdc0ab576c7bee804e2abb2e22e40fd3a5351

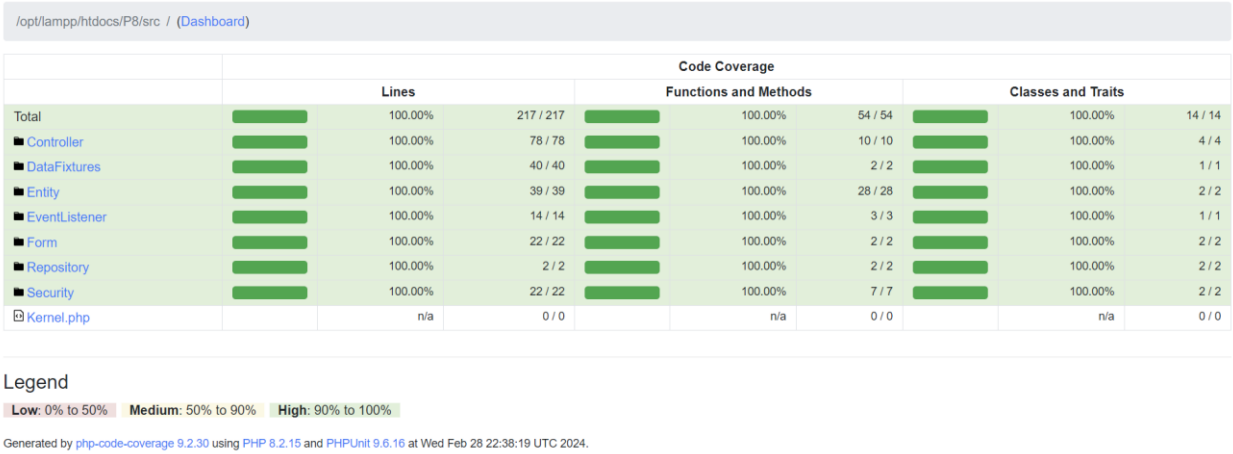
e. Tests automatisés

L'application possède maintenant des fonctionnalités correctes, un système d'authentification, un système d'autorisation, il faut donc être certain que tout cela fonctionne comme on le souhaite, et ce même par la suite. L'ajout de tests automatisés semble être un bon choix. J'ai choisi d'utiliser PHPUnit pour les tests unitaires comme fonctionnels.
Les tests se situent dans le dossier "tests", puis "Controller" pour les contrôleurs, "Entity" les entités, etc. Étant donné que le cœur de l'application se situe dans le dossier "src", il y a un fichier de test par fichier dans "src".

Voici les différents fichiers de test.



Ainsi que le “code coverage” réalisé par PHPUnit.



/opt/lampp/htdocs/P8/src / (Dashboard)

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total	100.00%	217 / 217		100.00%	54 / 54		100.00%	14 / 14	
Controller	100.00%	78 / 78		100.00%	10 / 10		100.00%	4 / 4	
DataFixtures	100.00%	40 / 40		100.00%	2 / 2		100.00%	1 / 1	
Entity	100.00%	39 / 39		100.00%	28 / 28		100.00%	2 / 2	
EventListener	100.00%	14 / 14		100.00%	3 / 3		100.00%	1 / 1	
Form	100.00%	22 / 22		100.00%	2 / 2		100.00%	2 / 2	
Repository	100.00%	2 / 2		100.00%	2 / 2		100.00%	2 / 2	
Security	100.00%	22 / 22		100.00%	7 / 7		100.00%	2 / 2	
Kernel.php	n/a	0 / 0		n/a	0 / 0		n/a	0 / 0	

Legend

Low: 0% to 50% Medium: 50% to 90% High: 90% to 100%

Generated by php-code-coverage 9.2.30 using PHP 8.2.15 and PHPUnit 9.6.16 at Wed Feb 28 22:38:19 UTC 2024.

Une fois les tests réalisés, il a été mis en place une automatisation de ces tests avec Travis, il a suffi d’ajouter le repository sur le site Travis, ainsi qu’un fichier *travis.yml* avec les informations concernant l’environnement ainsi qu’un script pour créer la base de données et exécuter les tests.

```yaml
language: php
php:
  - '8.1'
env:
  - DATABASE_URL=mysql://root@127.0.0.1/travis_todolist
services:
  - mysql
script:
  - composer install
  - php bin/console doctrine:database:create --env=test
  - php bin/console doctrine:schema:update --force --env=test
  - vendor/bin/phpunit
```

Travis nous indique ici que le “build” des tests de ce repository passe sans erreur.

f. Documentation

Les tests devront être adaptés par les développeurs qui contribueront au développement de l'application.
C'est pourquoi la documentation est importante.

Chaque application a besoin de sa documentation, ici nous avons :

- Le fichier README pour l'installation et le lancement des tests.
- Le fichier CONTRIBUTING pour l'organisation autour de la contribution d'autres développeurs.
- La documentation technique pour mieux comprendre le fonctionnement technique de l'application et faciliter la participation d'autres développeurs.
- L'audit pour expliquer les changements effectués et voir où en est le projet fonctionnellement, qualité de code, performances.

3- Audit

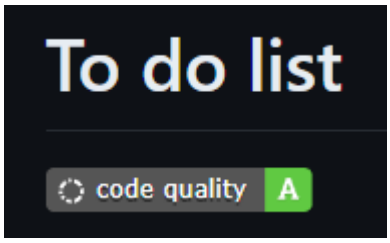
a. Analyse de qualité de code

Après avoir améliorer l'application, j'ai effectué un audit de qualité de code avec Codacy.

La note attribuée était un C, j'ai donc corrigé le style du code à l'aide des issues fournies par Codacy. Les corrections effectuées sont les suivantes :

- Ajout de commentaires de variables et de fonctions
- Ajout de déclaration de type retourné par les fonctions
- Correction d'indentation.

Voici le badge ajouté dans le README du projet.



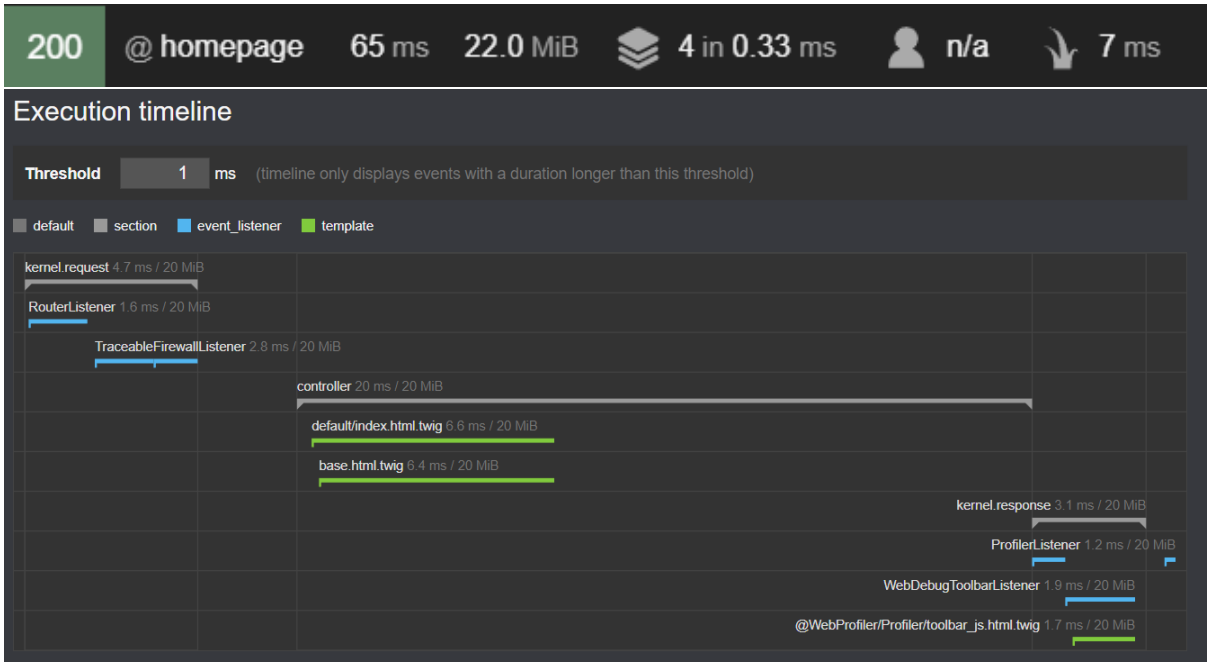
b. Analyse de performances

Une fois l'audit de qualité de code réalisé, il est intéressant de mesurer la vitesse de l'application, soit ses performances. Optimiser l'utilisation des ressources permet d'améliorer la performance, et ainsi rendre l'application plus rapide.

De plus, plus une application est lente, moins elle est recommandée (par le navigateur comme par les utilisateurs).
Afin de réaliser un audit de performance, Blackfire est un très bon outil mais est devenu payant, j'ai donc fait le choix d'utiliser le profiler de Symfony.

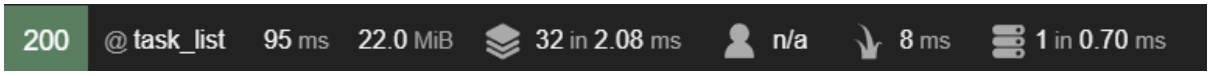
Page d'accueil

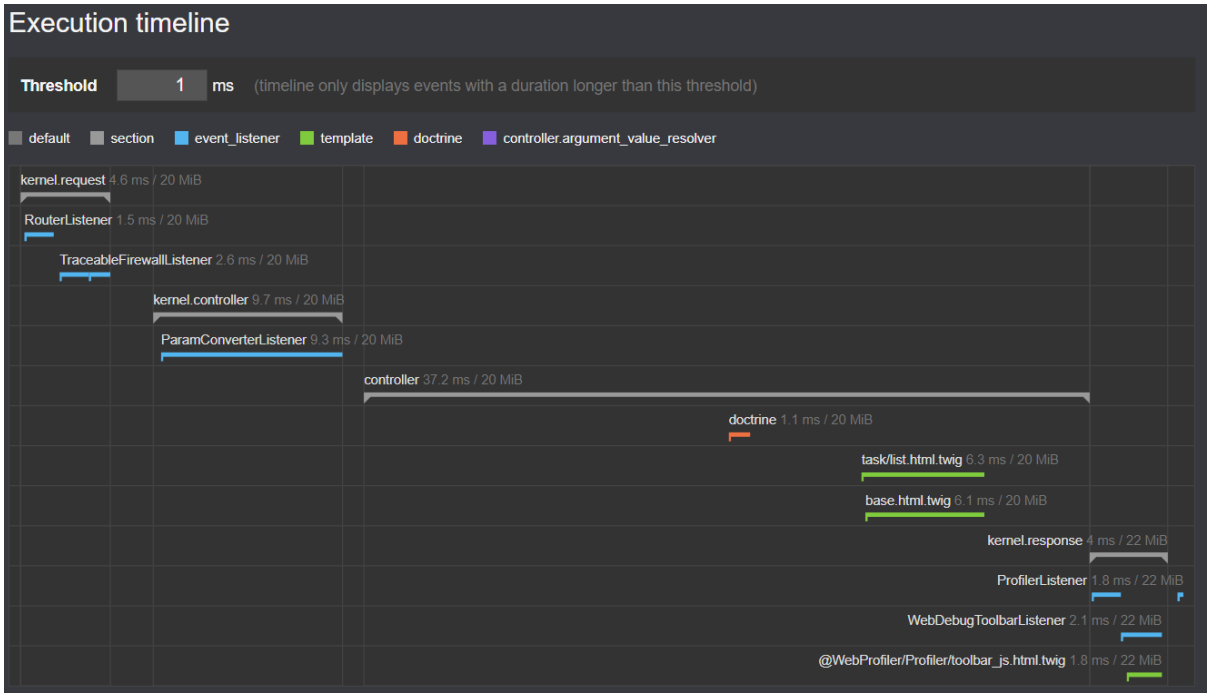
On voit ici que la page d'accueil s'affiche en 65 ms, ce qui est rapide, utilise 22 MiB de mémoire et fait 4 appels au cache en 0.33 ms.



Le contrôleur est l'élément qui prend le plus de temps, on voit en vert qu'il affiche le template twig.

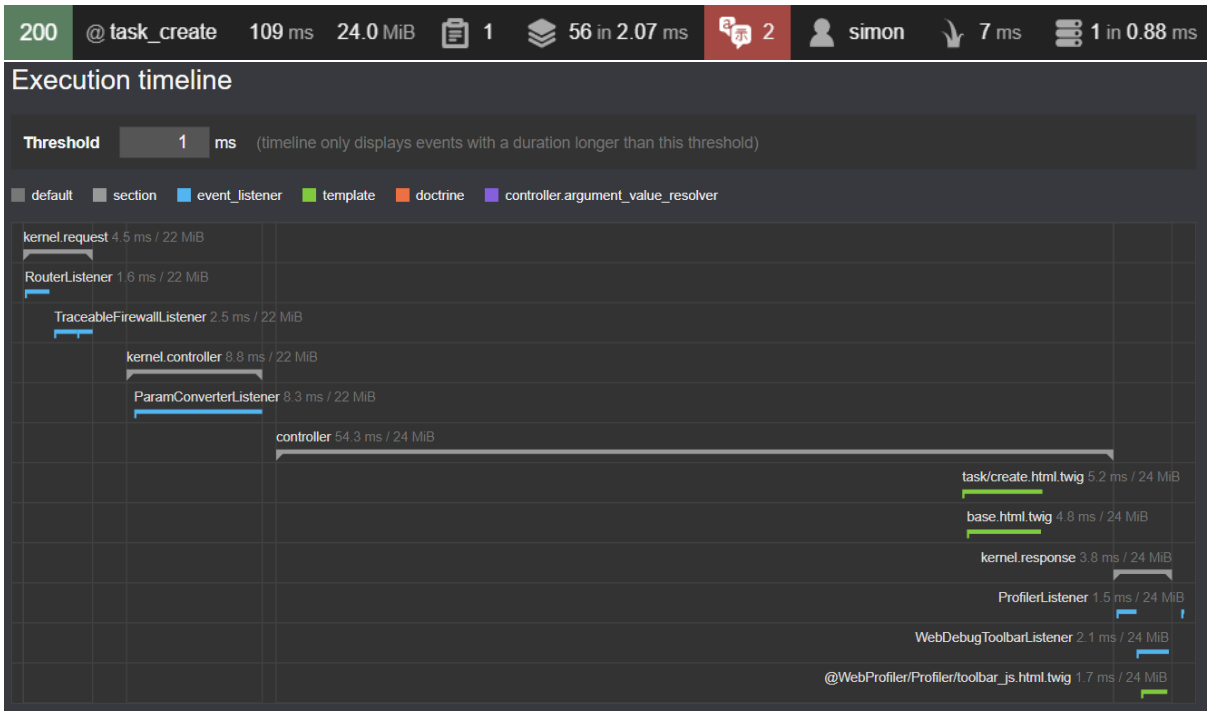
Liste des tâches





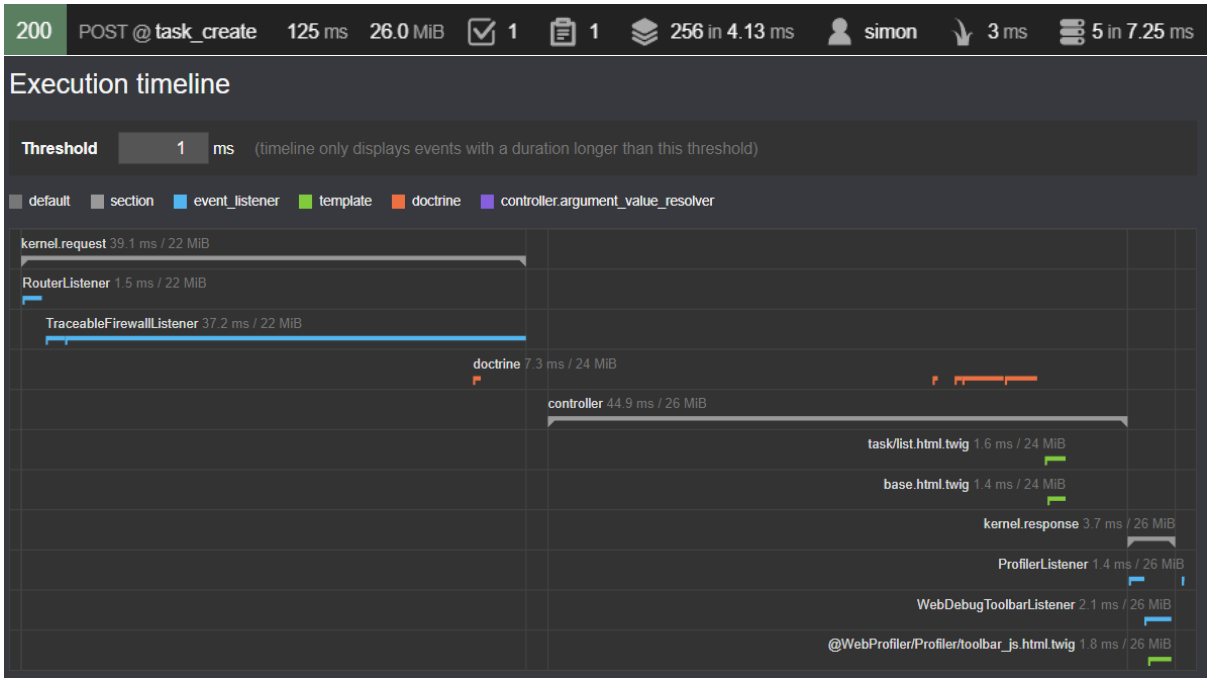
Le contrôleur est l'élément qui prend le plus de temps, on voit en orange qu'il fait une requête à la base de données avec doctrine puis en vert affiche le template.

Page de création d'une tâche



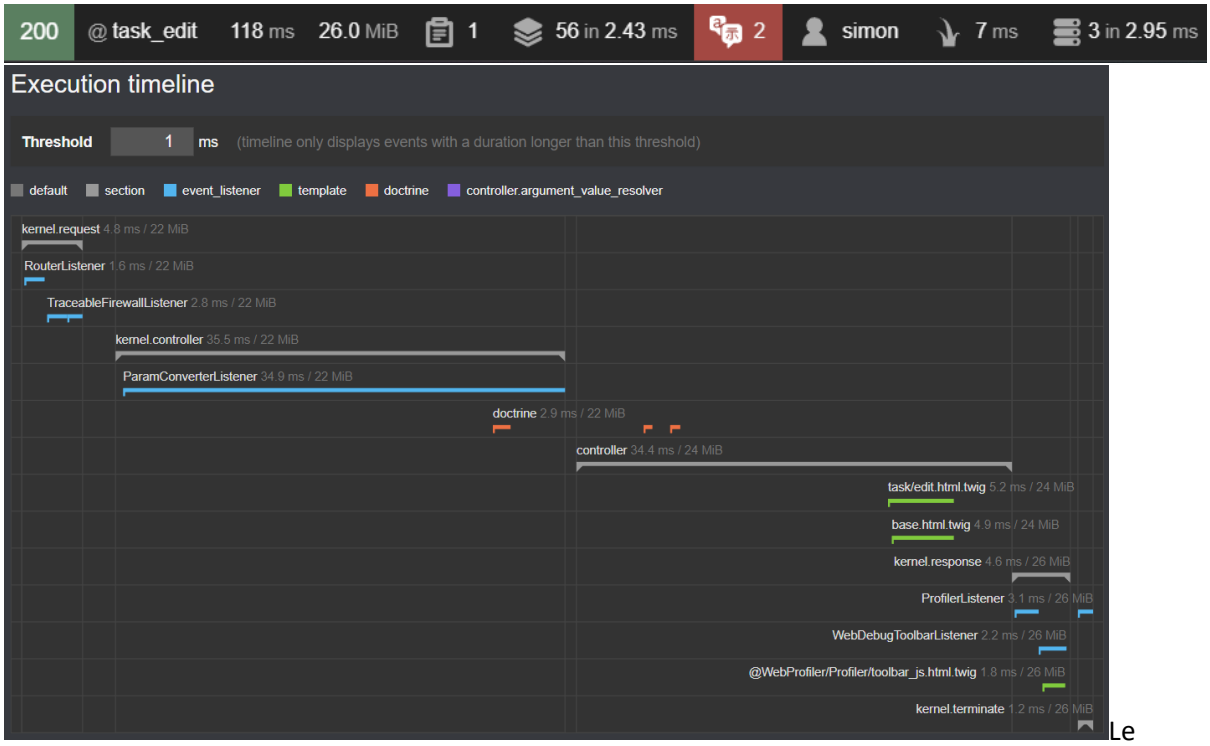
Le contrôleur est l'élément qui prend le plus de temps, on voit en vert qu'il affiche le template.

Création d'une tâche



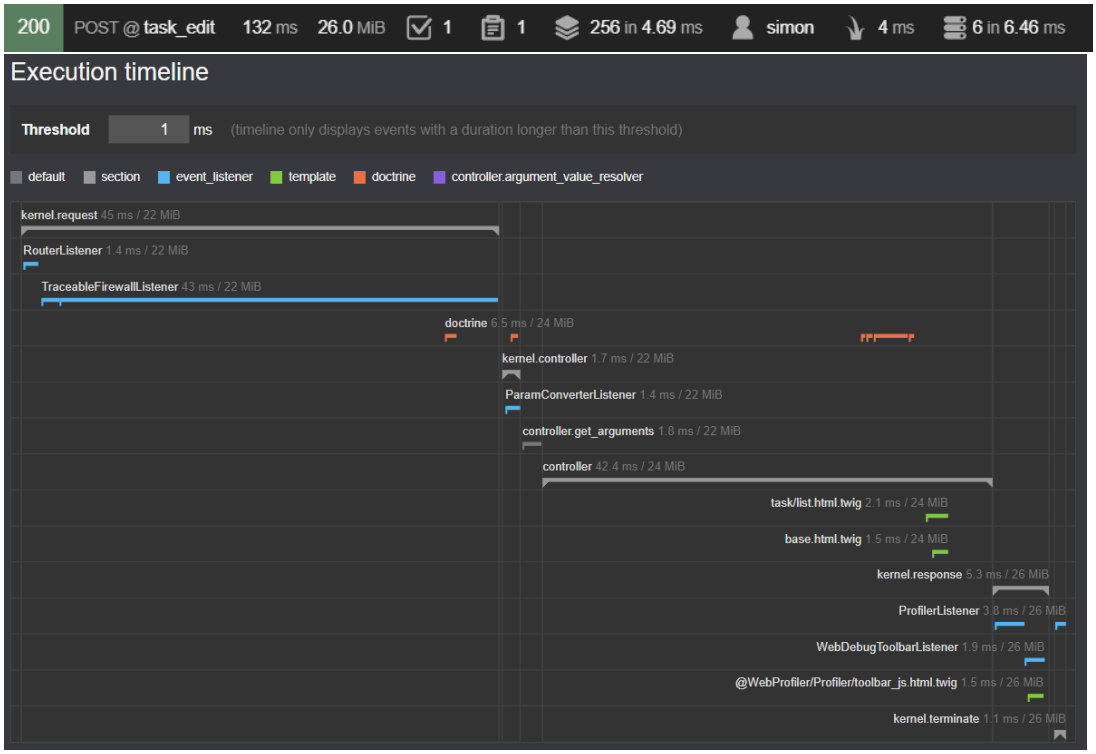
Le firewall et le contrôleur sont les éléments qui prennent le plus de temps. Le firewall vérifie les données et fait un appel à la base de données en orange. Le contrôleur crée la tâche en orange avec doctrine et affiche le template en vert.

Page de modification d'une tâche



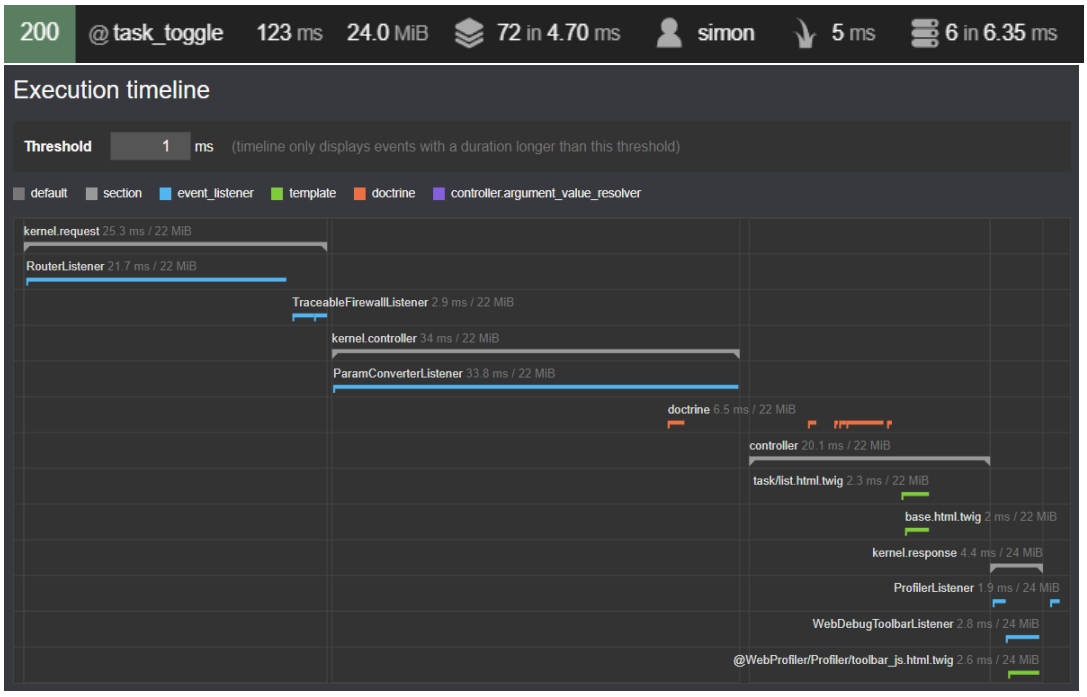
Le contrôleur Kernel et le contrôleur sont les éléments qui prennent le plus de temps, le premier identifie et récupère la tâche avec ParamConverterListener en bleu, le second utilise doctrine en orange et affiche le template en vert.

Modification d’une tâche



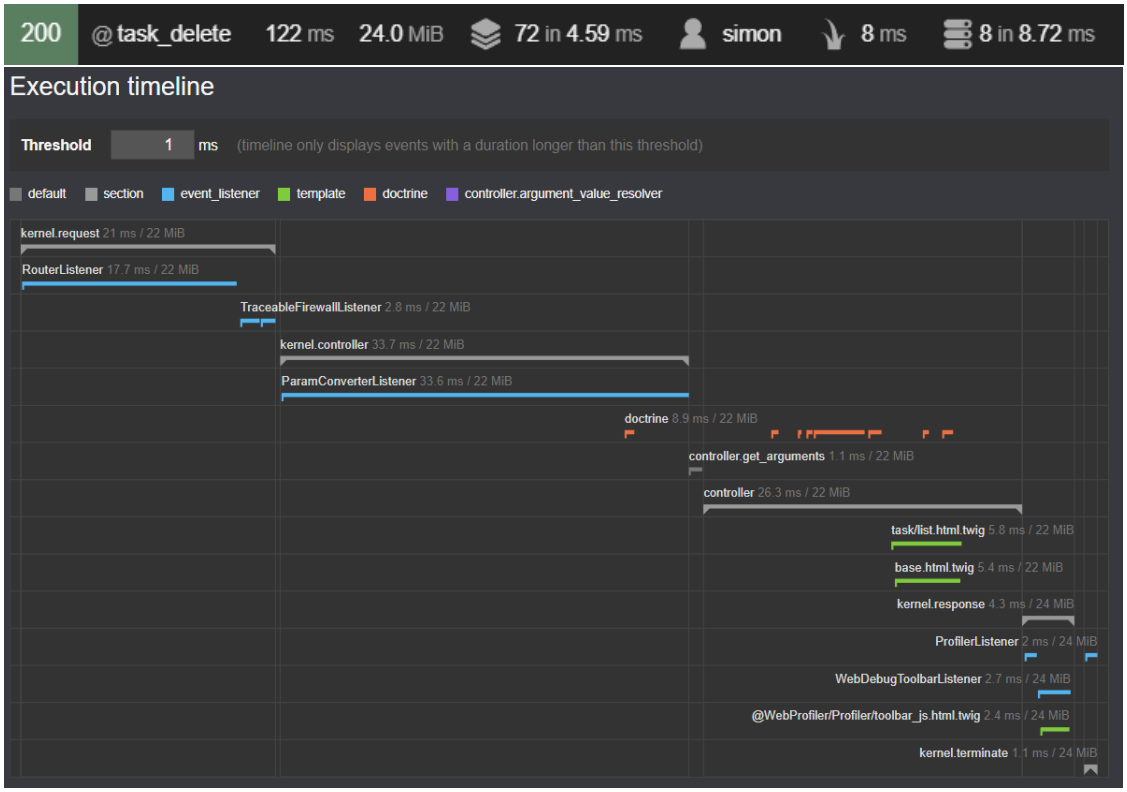
Le firewall et le contrôleur sont les éléments qui prennent le plus de temps. Le firewall utilise doctrine en orange et le contrôleur modifie la tâche avec doctrine en orange et affiche le template en vert.

Toggle d’une tâche



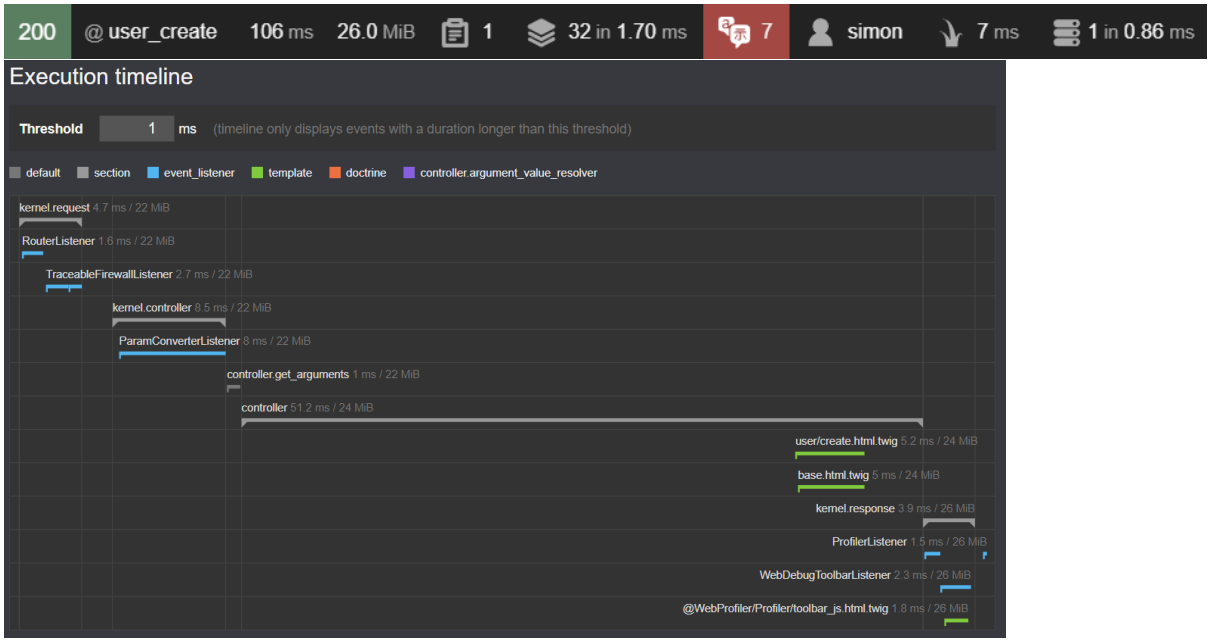
Le contrôleur Kernel et le contrôleur sont les éléments qui prennent le plus de temps. Le premier identifie et récupère la tâche en base de données avec ParamConverterListener en bleu. Le second modifie la tâche et récupère les tâches ensuite avec doctrine en orange, puis affiche le template en vert.

Suppression d’une tâche



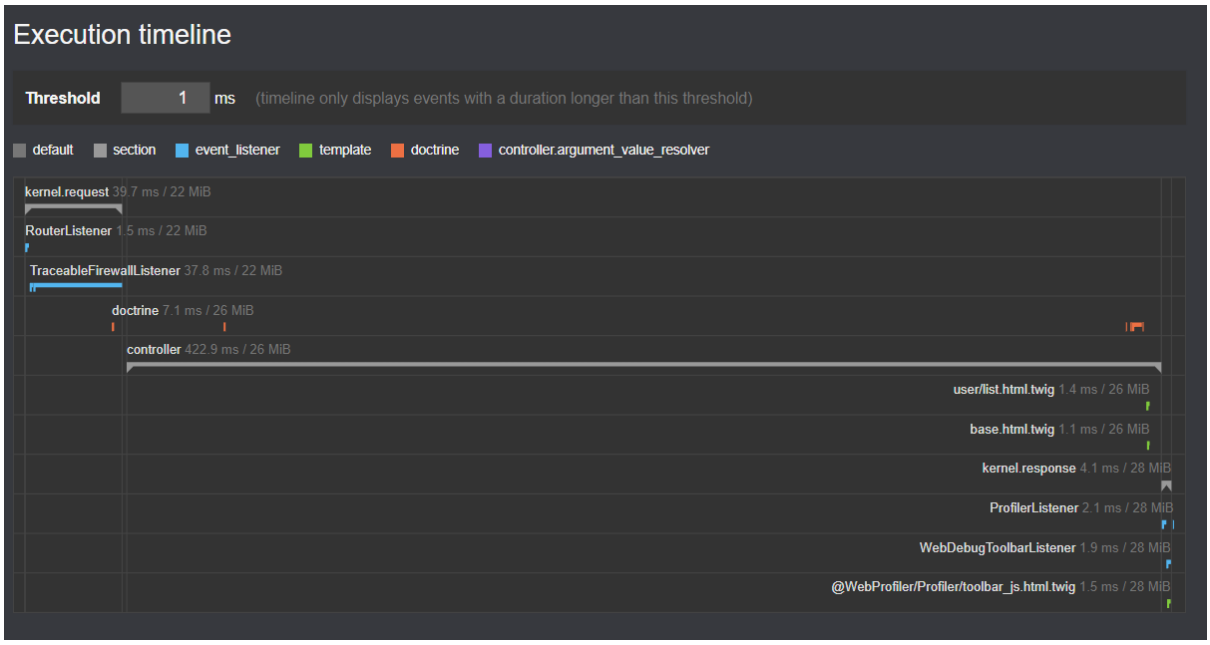
Le contrôleur Kernel et le contrôleur sont les éléments qui prennent le plus de temps. Le premier identifie et récupère la tâche en base de données avec doctrine. Le second supprime la tâche, récupère les tâches avec doctrine en orange et affiche le template en vert.

Page de création d’un utilisateur



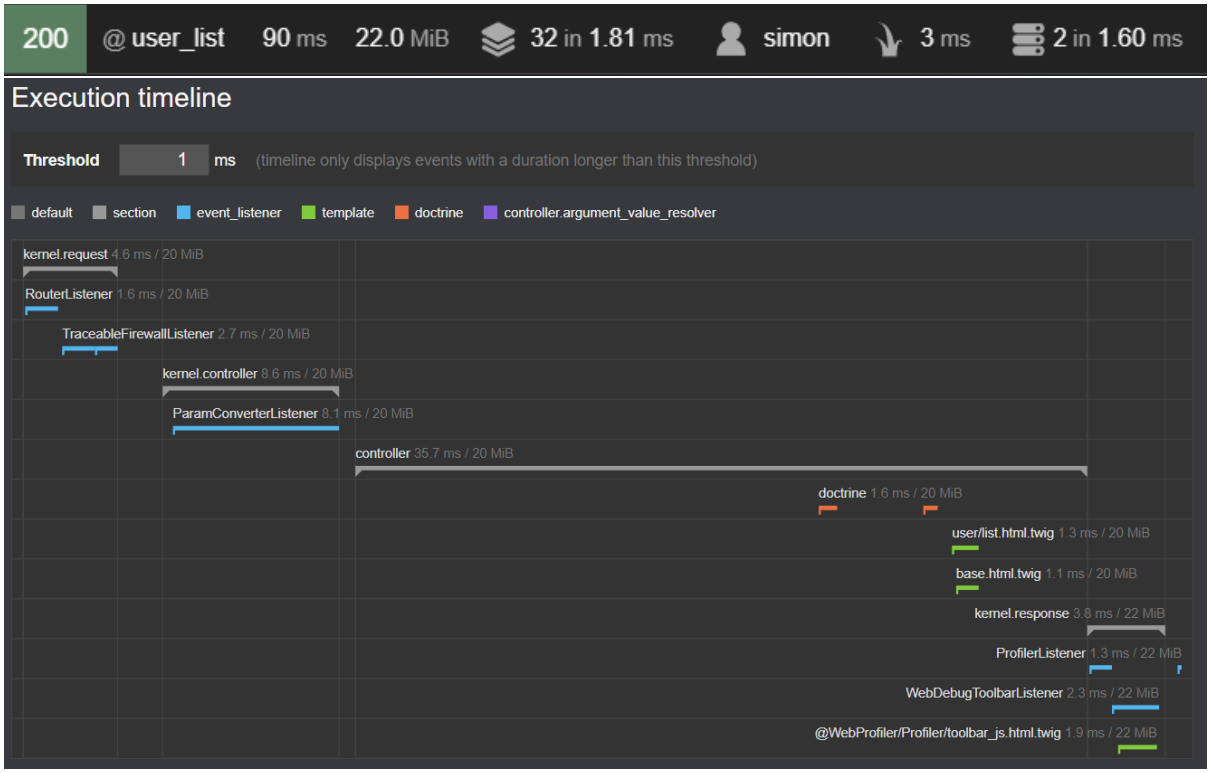
Le contrôleur est l’élément qui prend le plus de temps, il affiche le template en vert.

Création d’un utilisateur



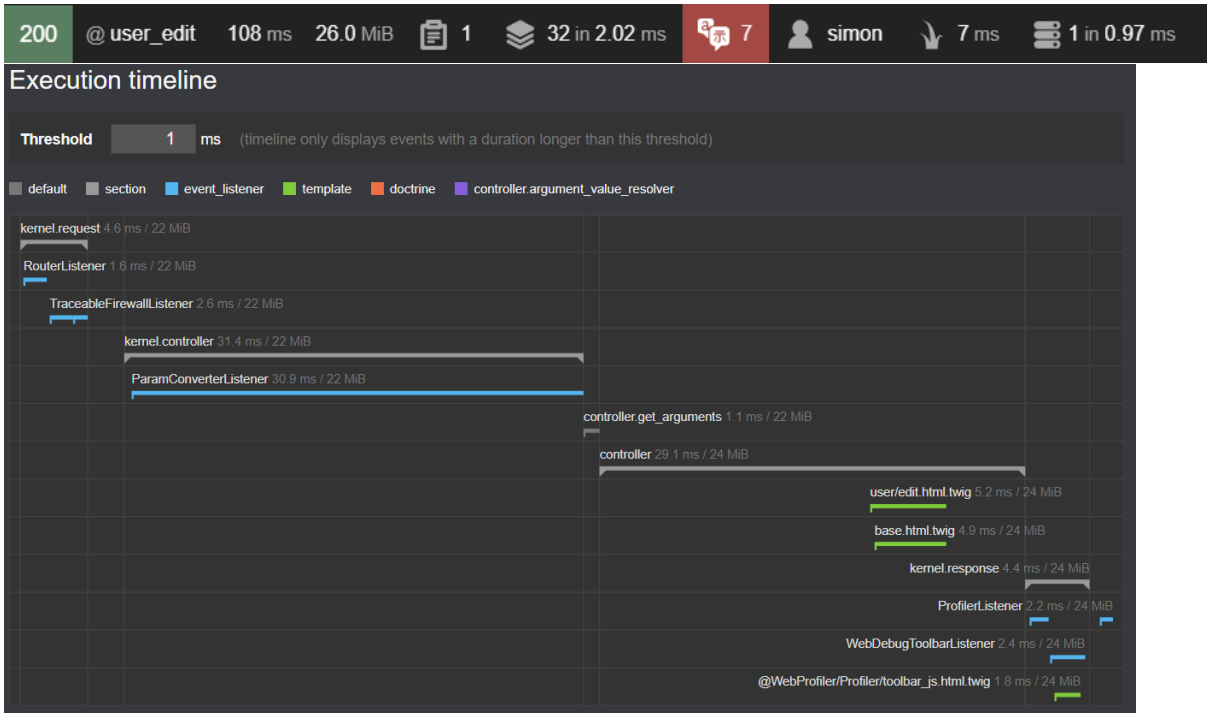
Le contrôleur est l’élément qui prend le plus de temps, il crée l’utilisateur et récupère les utilisateurs avec doctrine en orange et affiche le template en vert.

Liste des utilisateurs



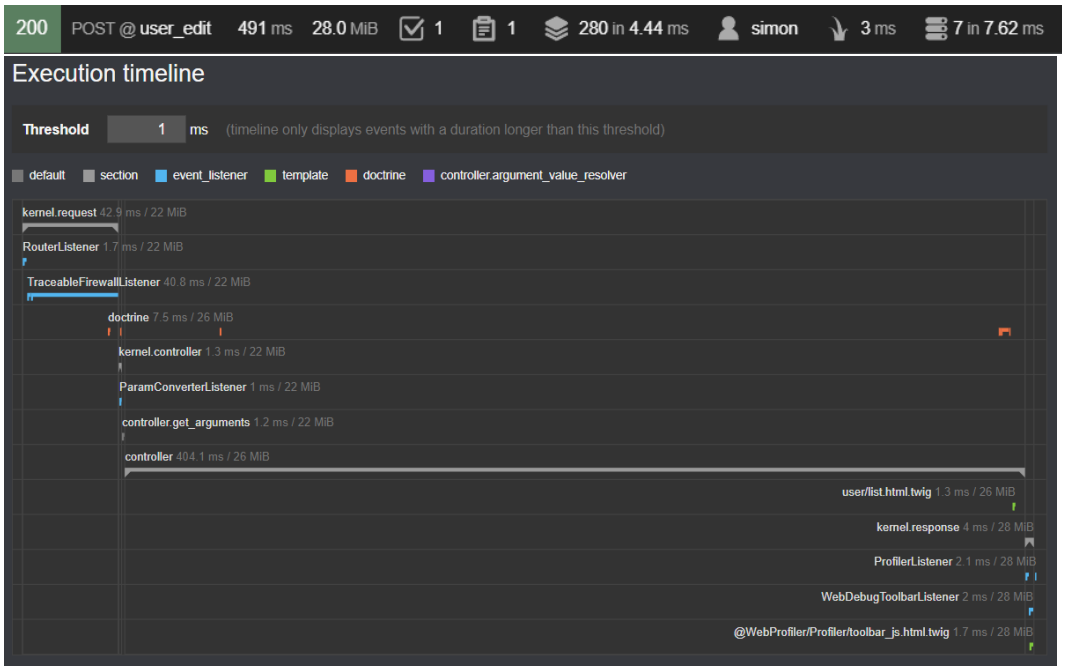
Le contrôleur est l’élément qui prend le plus de temps, il récupère les utilisateurs avec doctrine en orange et affiche le template en vert.

Page de modification d’un utilisateur



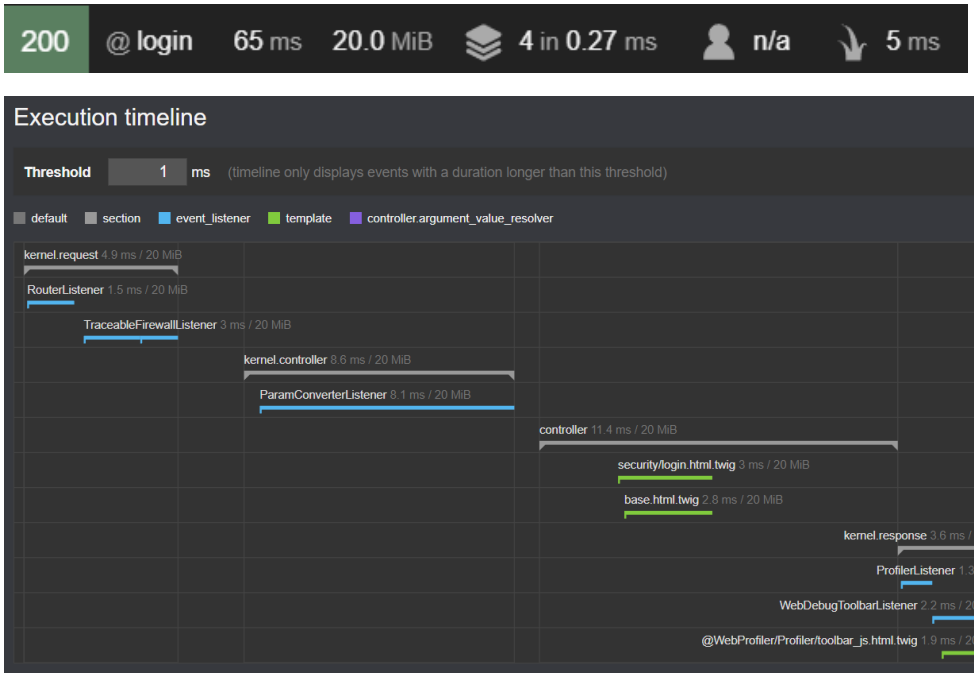
Le contrôleur Kernel et le contrôleur sont les éléments qui prennent le plus de temps. Le premier identifie et récupère l'utilisateur avec ParamConverterListener en bleu. Le second affiche le template en vert.

Modification d'un utilisateur



Le contrôleur est l'élément qui prend le plus de temps, il modifie l'utilisateur et récupère les utilisateurs en base de données avec doctrine en orange et affiche le template en vert.

Page de connexion



Le contrôleur Kernel et le contrôleur sont les éléments qui prennent le plus de temps. Le premier identifie l'utilisateur s'il y en a un avec ParamConverterListener en bleu et le second affiche le template en vert.

En conclusion, les fonctionnalités du site peuvent être optimisées. On voit que le contrôleur est l'élément qui prend le plus de temps pour chacune d'entre elles. C'est le noyau de l'application dans une architecture MVC, il doit donc faire les relations entre les "modèles" (base de données, doctrine) et les "vues" (template, twig).

Nous pouvons améliorer les performances du site de différentes manières :

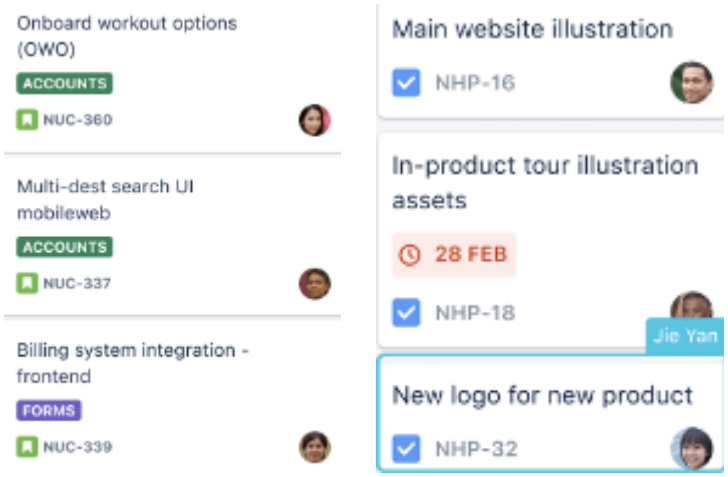
- Utiliser un hébergeur web de qualité.
- Ajouter un système de cache.
- Réduire les requêtes externes en remplaçant Jquery par du JavaScript classique.
- Compresser les images.
- Migrer l'application vers une version plus récente de Symfony.

4- Propositions d'améliorations

a. Fonctionnalités

L’application n’est pas parfaite pour autant, nous pouvons toujours l’améliorer. Par exemple avec l’analyse précédente, on voit que les contrôleurs peuvent être plus performants.
Voici différentes idées de fonctionnalités qui pourraient être implémentées :

- Ajouter et afficher plus d’informations concernant les tâches (utilisateur, date, étiquette, deadline).
  Voici des exemples.



- Ajouter et afficher un avatar pour chaque utilisateur.
- Ajouter une page par utilisateur pour afficher les tâches de celui-ci.
  Ou alors on peut ajouter des filtres pour la liste de tâches, pour pouvoir filtrer par utilisateur, par date ou encore par étiquette.

b. Design

Nous pouvons aussi corriger le design de l’application :

- Soigner le design global du site à partir d’une charte graphique.
- Mieux gérer les espaces entre les éléments (padding/margin).
- Ajouter un menu.
- Personnaliser la page d’erreur (404).