Todo & Co - TodoList

Audit de performance

—

Héligon Sébastien
OpenClassrooms - Développeur d'application - PHP / Symfony
Projet 8 - Améliorez une application existante de Todo & Co

Sommaire

1. Le Projet

1.1. Présentation

L’objectif fondamental de cette application TodoList est de fournir à l'entreprise ToDo & Co un outil efficace pour la gestion de ses tâches quotidiennes.

1.2. Technologies

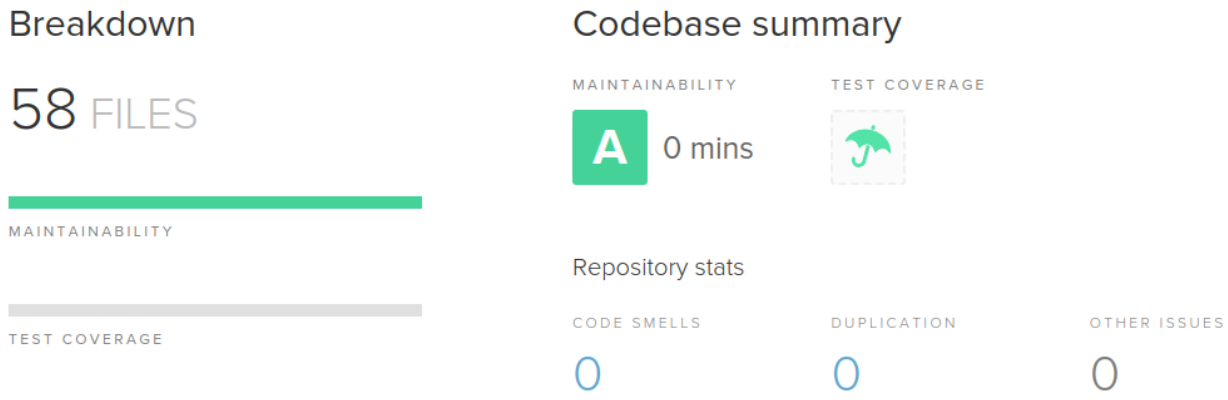
L’application a été récemment mise à jour vers la dernière version de Symfony, la 6.3, afin de bénéficier des toutes dernières fonctionnalités, optimisations et sécurités offertes par ce framework de développement. Cette mise à jour a également nécessité le passage à PHP 8, la dernière version du langage de programmation, pour garantir une performance optimale et une sécurité renforcée.

Cette mise à jour vers Symfony 6.3 et PHP 8 représente un saut significatif en termes de performances, de sécurité et de fonctionnalités pour l’application.

2. Qualité de code

2.1. CodeClimate

Tout d'abord, j'ai fait appel à CodeClimate pour évaluer la qualité du code une fois le projet achevé. Voici le résultat de cette analyse (https://codeclimate.com/github/heligonSeb/TodoList):

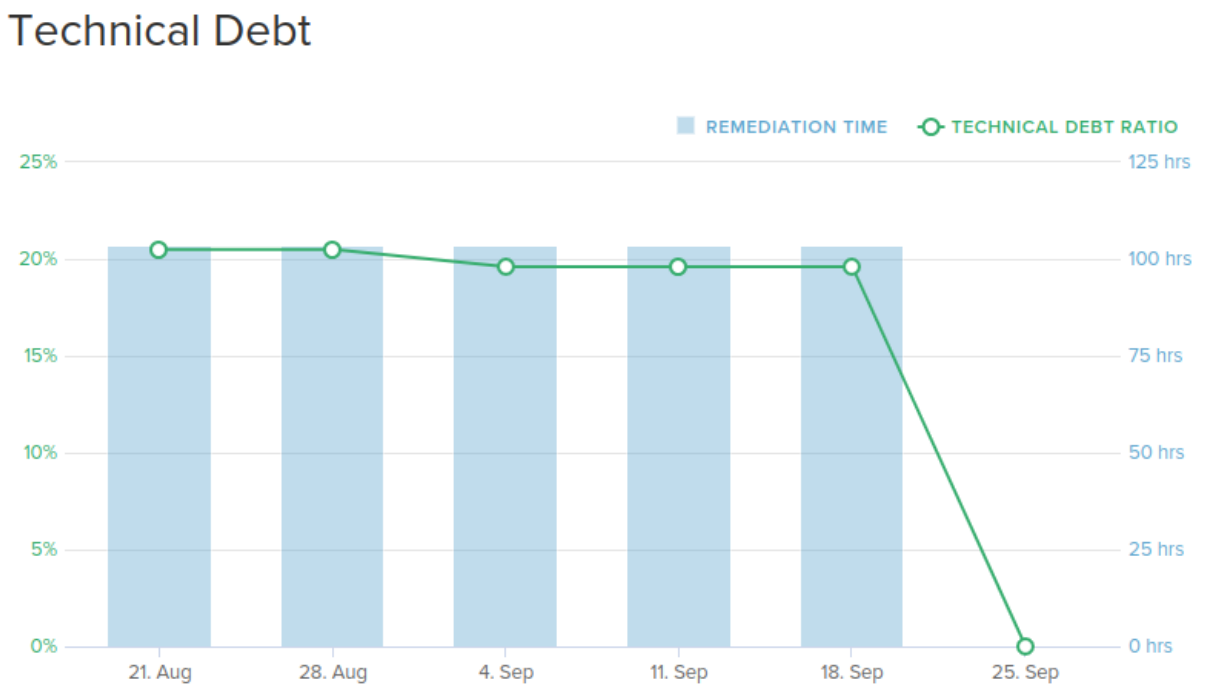


Selon CodeClimate la maintenabilité du code est classée dans la catégorie "vert" avec une note A et une barre de progression entièrement remplie de vert. Cette évaluation témoigne de l'optimisation de l'application, car elle révèle l'absence de mauvaises pratiques, de duplications de code et d'autres problèmes notables.

Cette évaluation est essentielle pour plusieurs raisons. Premièrement, elle démontre que le code source de l'application est soigneusement conçu et organisé, ce qui facilitera grandement sa maintenance à long terme. L'obtention d'une note A signifie que le code répond aux normes élevées en matière de lisibilité, de performance et de structure, ce qui est crucial pour garantir que l'application continue de fonctionner de manière fiable.

De plus, l'absence de mauvaises pratiques et de duplications de code suggère que l'application est développée de manière efficace et par conséquent évitera des erreurs et des problèmes futurs. Cela contribue également à réduire la dette technique, ce qui signifie qu'il sera moins coûteux et plus rapide d'apporter des modifications ou des améliorations à

l'application à l'avenir.



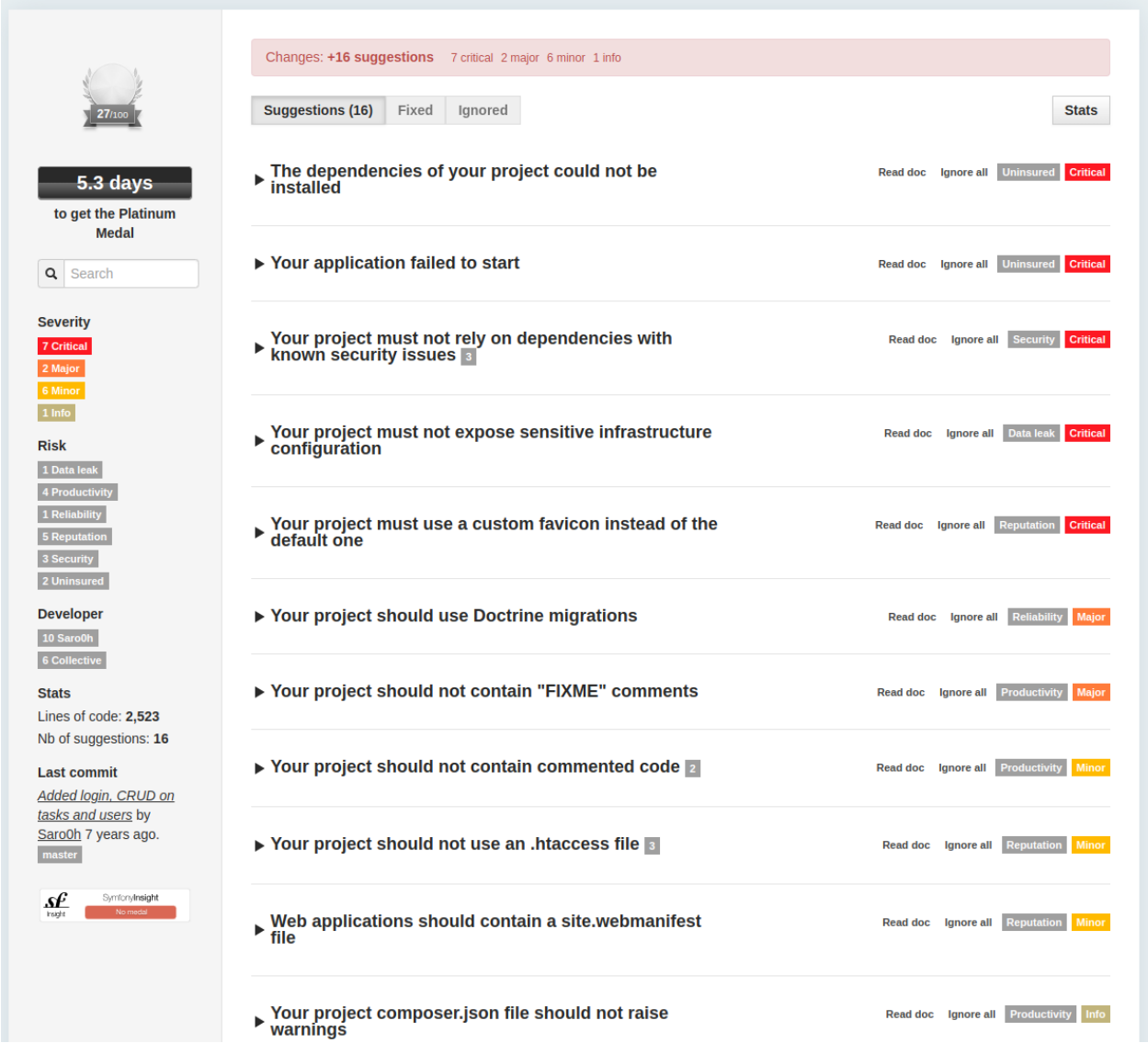
On peut voir sur le schéma qui représente l'évolution de la dette technique de notre application. Jusqu'au 18 septembre, le ratio de la dette technique était de 20%, ce qui signifie que 20% du code de l'application nécessitait des améliorations ou des corrections pour atteindre des normes optimales. Cependant, à partir du 25 septembre, ce ratio a chuté brusquement pour atteindre 0%. Cette descente témoigne du moment où j'ai entrepris des mesures correctives majeures sur l'application en utilisant SymfonyInsight, que je vais vous présenter dans la partie suivante.

Cette réduction drastique de la dette technique est un indicateur très positif pour l'application. Elle signifie que nous avons pris des mesures efficaces pour résoudre les problèmes et les lacunes qui pouvaient exister dans le code source de l'application. Une dette technique élevée peut entraîner des ralentissement, des bugs et une complexité accrue lors de la maintenance, ce qui peut devenir coûteux à long terme. En réduisant la dette technique à zéro, nous assurons que l'application est à son meilleur niveau en termes de qualité et de performance.

En somme, grâce à cette évaluation positive de la qualité du code par CodeClimate, il est clair que l'application est bien positionnée pour offrir une expérience utilisateur robuste et fiable, tout en permettant un développement et une maintenance plus aisés à l'avenir. C'est un élément précieux qui garantit la pérennité et le succès du projet.

2.2. Symfony insight

Dans un second temps, conformément à ce que j'ai annoncé dans la section précédente. j'ai utilisé SymfonyInsight pour évaluer la qualité de mon code. Vous pouvez observer le résultat de cette analyse dans l'image ci-dessous, réalisée après avoir "Fork" l'application. Il est notable qu'à ce stade, l'analyse a révélé un certain nombre d'erreurs, dont 7 étaient classées comme critiques, 2 comme majeures et 6 comme mineurs.



Ces résultats initiaux indiquent clairement qu'il y a des problèmes substantiels dans le code source de l'application qui nécessite une attention. Les erreurs critiques sont particulièrement préoccupantes, car elles peuvent entraîner des dysfonctionnements graves dans l'application. Les erreurs majeures et mineures ne sont pas à négliger non plus elles peuvent entraîner des problèmes de moindre envergure, mais qui peuvent s'accumuler et devenir un fardeau lors de la maintenance à long terme.

Pour remédier à cette situation et améliorer la qualité du code ainsi que la gestion de la dette technique, j'ai entrepris la correction complète de l'ensemble des problèmes relevés par SymfonyInsight, comme illustré dans l'image suivante.

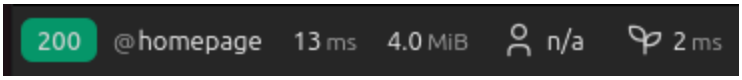
Cette démarche est cruciale pour plusieurs raisons. Tout d'abord, en résolvant les erreurs critiques, majeures et mineures, nous assurons que l'application fonctionne de manière plus stable et fiable. Cela renforce la confiance des utilisateurs et réduit les risques d'incidents imprévus. Ensuite, en investissant dans la correction de ces problèmes nous minimisons la dette technique pour les futures améliorations de l'application. Sans cette correction les erreurs se seraient accumulées et rendraient la maintenance plus difficile et coûteuse.

En résumé, l'utilisation de SymfonyInsight a permis de mettre en lumière les lacunes initiales de notre code, mais elle a également offert la possibilité de les corriger de manière proactive. Cette approche proactive est essentielle pour garantir que notre application continue de répondre aux normes élevées de qualité et de performance, tout en minimisant les coûts et les défis liés à la maintenance à long terme.

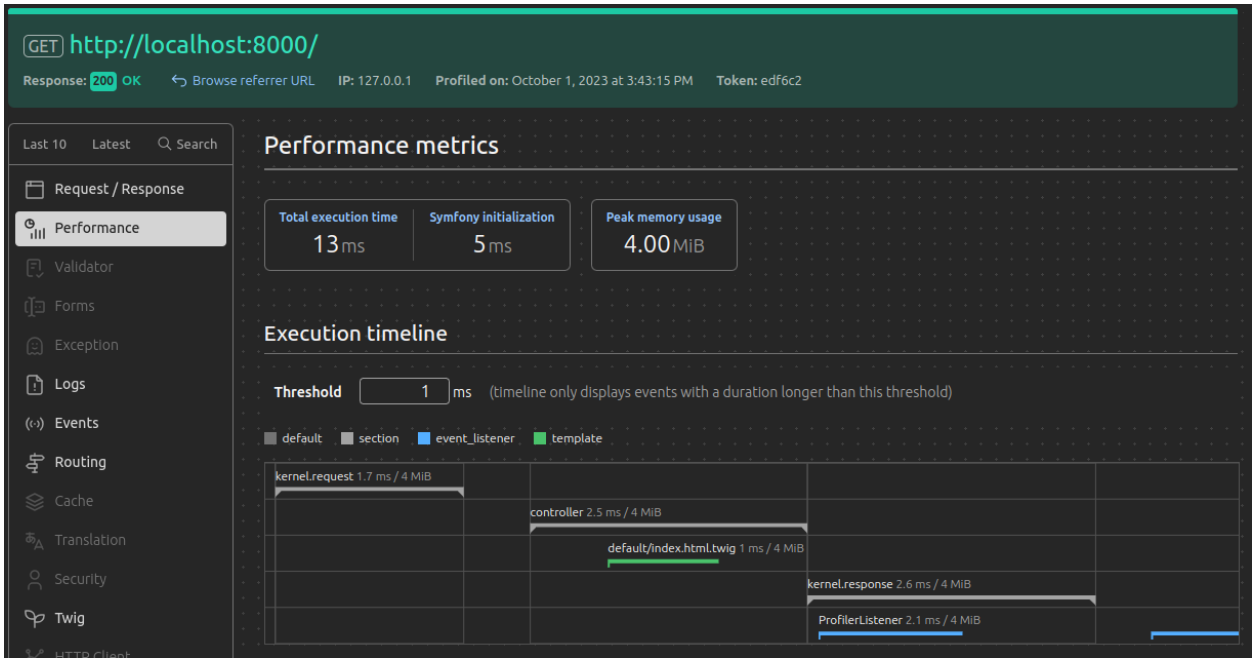
3. Performances

Les tests de performances de l'application ont été réalisés à la fin du projet et la correction du code avec SymfonyInsight et sera réalisé avec le profiler de Symfony.

3.1. Accueil



Ce premier résultat provient de la page d'accueil de l'application. Dans cette analyse, nous pouvons observer que la page a renvoyé un statut 200 indiquant une réponse HTTP réussie, et que le temps d'exécution était de 13 millisecondes, avec une utilisation maximale de la mémoire de 4MiB. L'indicateur "n/a" signifie qu'il n'y avait aucun utilisateur connecté à la page à ce moment-là.

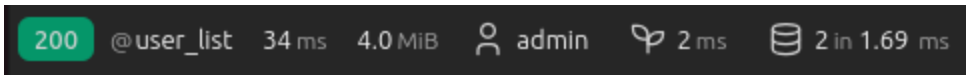


L'image ci-dessus détaille le processus complet au-delà de 1 ms du chargement de la page. On peut constater que la séquence débute par l'évènement "kernel.request" (avec un temps d'exécution de 1,7 ms et une utilisation mémoire de 4MiB). Ensuite, nous avons l'appel au "controller" (avec un temps d'exécution de 2,5 ms et une utilisation mémoire de 4MiB). Pendant le chargement du "controller", il convient de noter qu'il effectue également

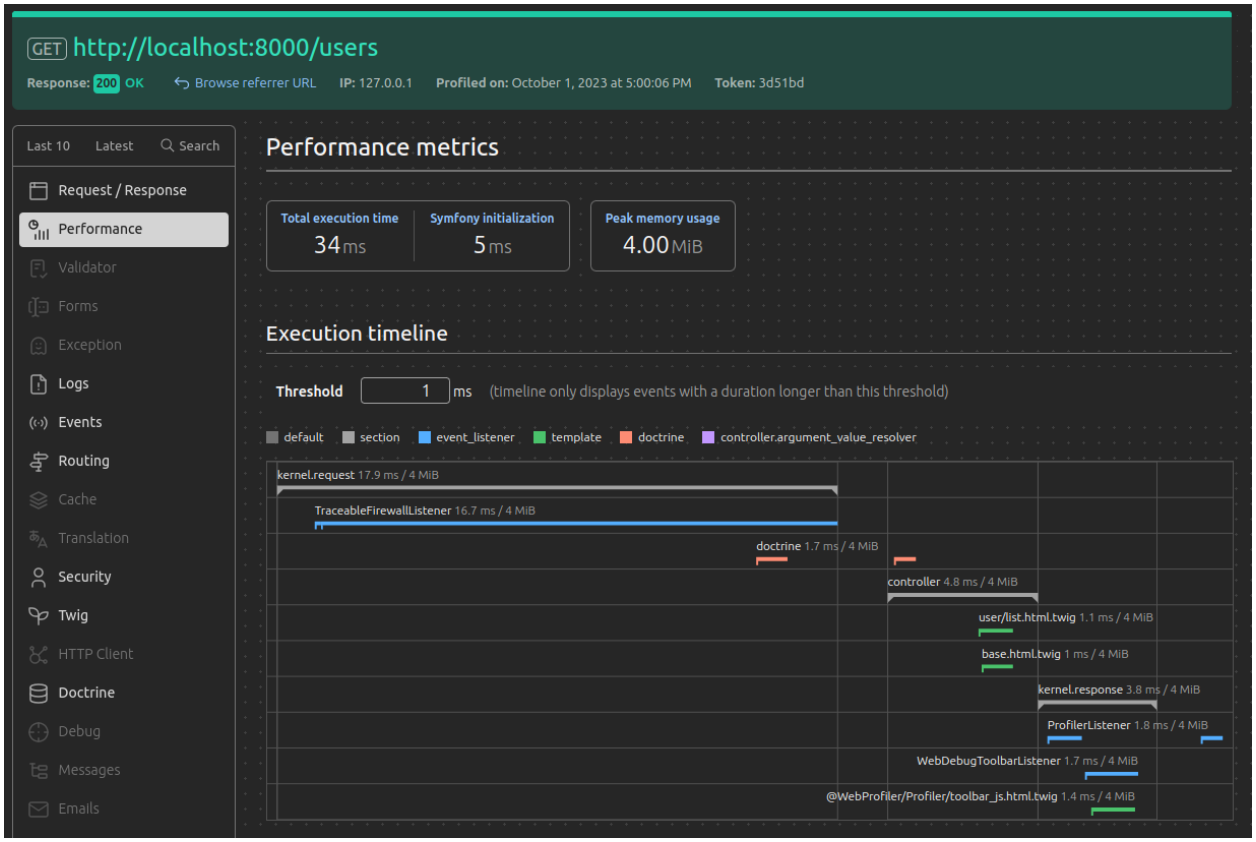
le chargement de la vue (avec un temps d'exécution de 1ms et une utilisation mémoire de 4MiB).

En résumé, la page affiche des performances tout à fait satisfaisantes. Le temps de chargement de la page est rapide, avec seulement 13 millisecondes d'exécution et L'utilisation de la mémoire est minime, à seulement 4MiB.

3.2. Liste des utilisateurs

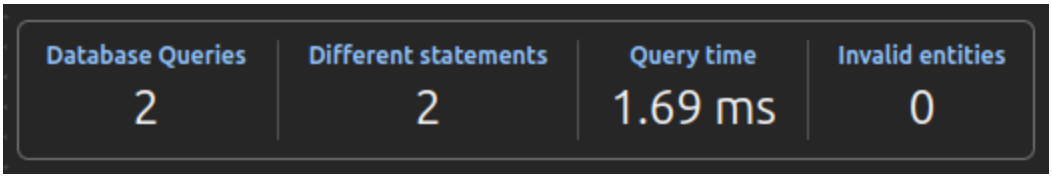


Ce résultat provient de la page de la liste des utilisateurs de l'application. Dans cette analyse, nous pouvons observer que la page a renvoyé un statut 200 indiquant une réponse HTTP réussie, et que le temps d'exécution était de 34 millisecondes, avec une utilisation maximale de la mémoire de 4MiB. On peut voir que nous sommes connectés en temps qu'admin et que la page a aussi passé 2 requêtes à la base de données avec un temps d'exécution de 1,69 ms



L'image ci-dessus détaille le processus complet au-delà de 1 ms du chargement de la page. On peut constater que la séquence débute par l'évènement "kernel.request" (avec un temps d'exécution de 17,9 ms et une utilisation mémoire de 4MiB). Ensuite, nous avons

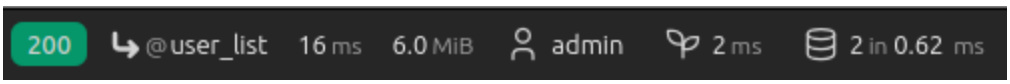
l'appel au "TraceableFirewallListener" (avec un temps d'exécution de 16,7 ms et une utilisation mémoire de 4 MiB). Avant la fin du chargement du "kernel.request" et du "TraceableFirewallListener" on constate l'appel a doctrine qui correspond à nos requête faite à la base de donnée (avec un temps d'exécution de 1,7 ms et une mémoire de 4 MiB). Puis nous avons le "controller" (avec un temps d'exécution de 4,8 ms et une utilisation mémoire de 4MiB). Pendant le chargement du "controller", il convient de noter qu'il effectue également le chargement des vues "user/list.html" (avec un temps d'exécution de 1.1ms et une utilisation mémoire de 4MiB) et "base.html" (avec un temps d'exécution de 1ms et une utilisation mémoire de 4MiB).



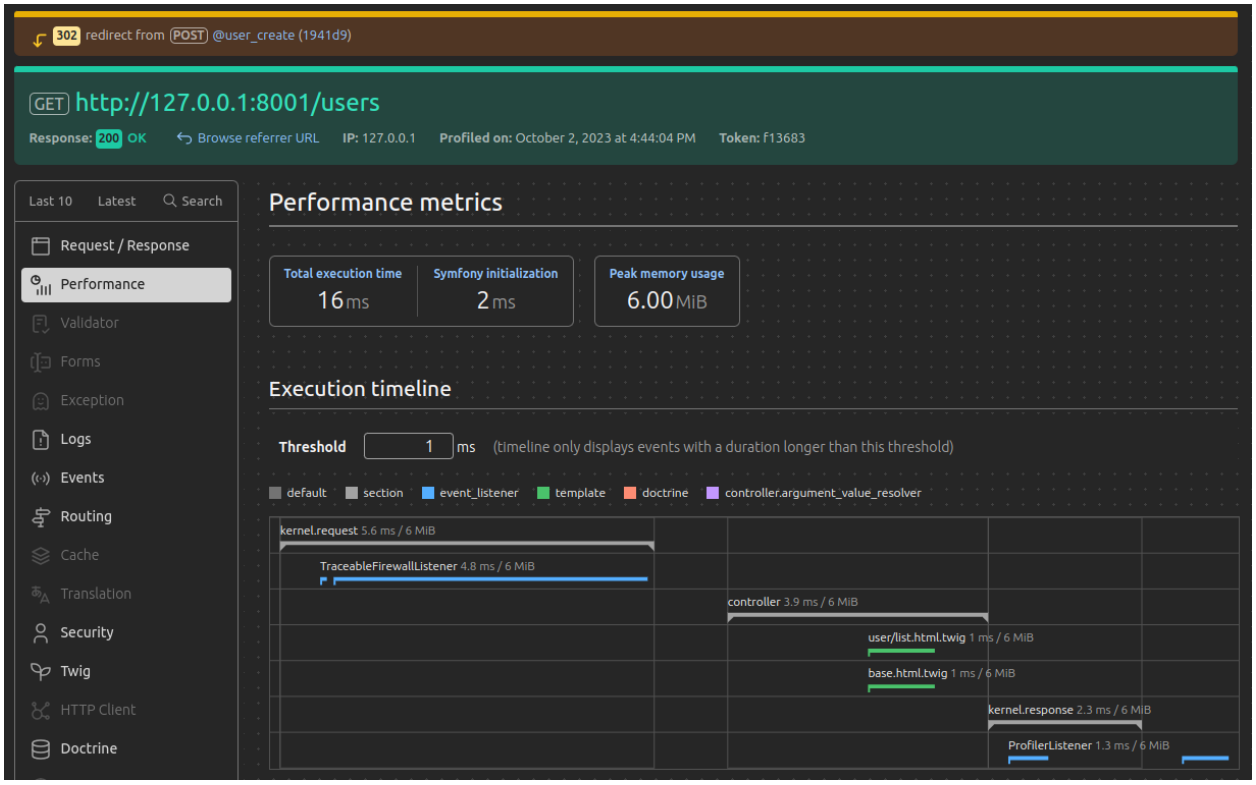
Avec les chiffres ci-dessus on observe qu'il y a 2 requêtes SQL qui ont été effectuées et 2 déclarations SQL distinctes. Cela signifie que 2 requêtes SQL différentes ont été exécutées sur la base de données pour une exécution de 1,69 millisecondes au total. Le profiler nous retourne aussi qu' aucune entité n'a été marquée comme invalide dans cette transaction avec la base de données. Dans le contexte de doctrine, cela pourrait indiquer que toutes les entités liées à la base de données sont valides du point de vue de la validation des données.

En résumé, la page affiche des performances tout à fait satisfaisantes. Le temps de chargement de la page est rapide, avec seulement 34 millisecondes d'exécution et L'utilisation de la mémoire est minime, à seulement 4MiB malgré les 2 requêtes SQL effectuées sur la base de données.

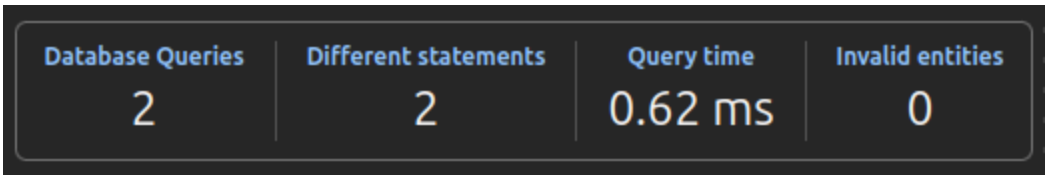
3.3. Création d'un utilisateur



Ce résultat provient de la page de la liste des utilisateurs de l'application qui a été redirigée après la création d'un utilisateur. Dans cette analyse, nous pouvons observer que la page a renvoyé un statut 200 indiquant une réponse HTTP réussie, et que le temps d'exécution était de 16 millisecondes, avec une utilisation maximale de la mémoire de 6MiB. On peut voir que nous sommes connectés en temps qu'admin et que la page a aussi passé 2 requêtes à la base de données avec un temps d'exécution de 0,62 ms.



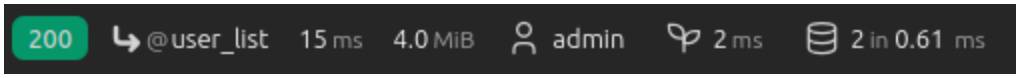
L'image ci-dessus détaille le processus complet au-delà de 1 ms du chargement de la page après la redirection effectuée. On peut constater que la séquence débute par l'évènement "kernel.request" (avec un temps d'exécution de 5,6 ms et une utilisation mémoire de 6MiB). Ensuite, nous avons l'appel au "TraceableFirewallListener" (avec un temps d'exécution de 4,8 ms et une utilisation mémoire de 6 MiB). Puis nous avons le "controller" (avec un temps d'exécution de 3,9 ms et une utilisation mémoire de 6MiB). Pendant le chargement du "controller", il convient de noter qu'il effectue également le chargement des vues "user/list.html" (avec un temps d'exécution de 1ms et une utilisation mémoire de 6MiB) et "base.html" (avec un temps d'exécution de 1ms et une utilisation mémoire de 6MiB).



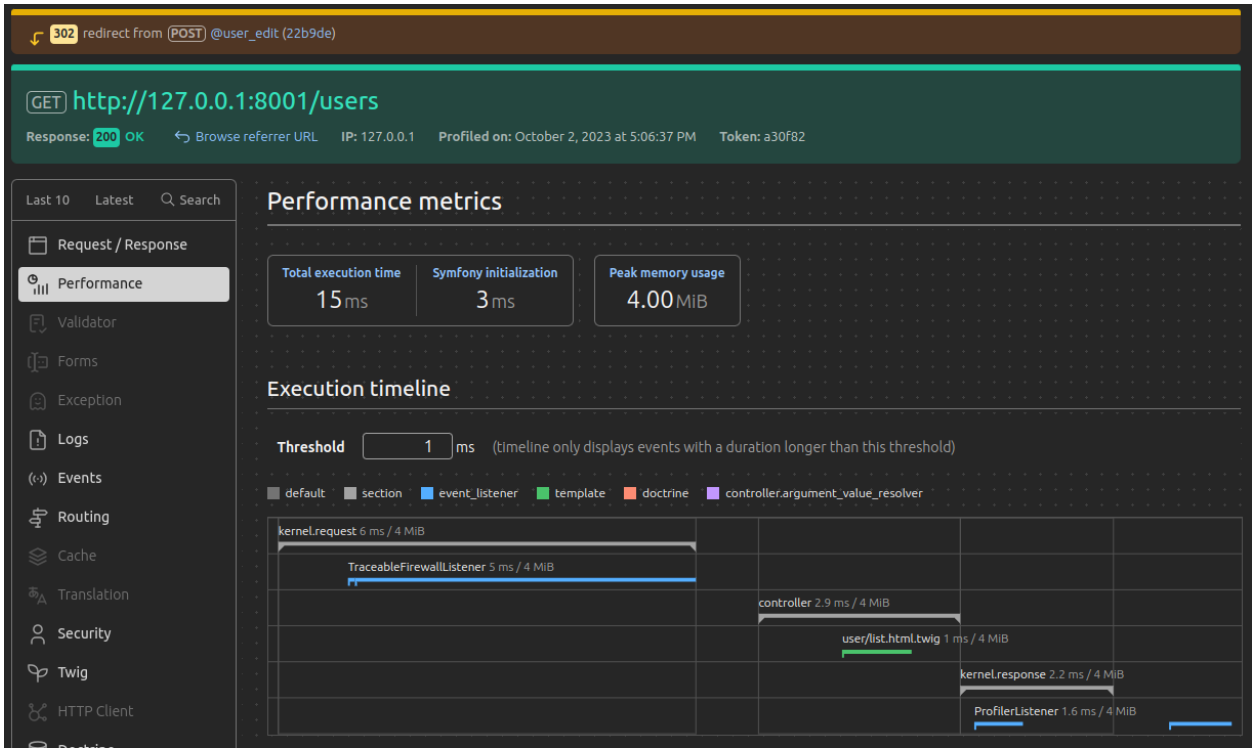
Avec les chiffres ci-dessus on observe qu'il y a 2 requêtes SQL qui ont été effectuées et 2 déclarations SQL distinctes. Cela signifie que 2 requêtes SQL différentes ont été exécutées sur la base de données pour une exécution de 0,62 millisecondes au total. Le profiler nous retourne aussi qu' aucune entité n'a été marquée comme invalide dans cette transaction avec la base de données.

En résumé, la page affiche des performances tout à fait satisfaisantes pour la création d'un utilisateur. Le temps de chargement de la page est rapide, avec seulement 16 millisecondes d'exécution et L'utilisation de la mémoire est minime, à seulement 6MiB malgré les 2 requêtes SQL effectuées sur la base de données.

3.4. Modification d'un utilisateur

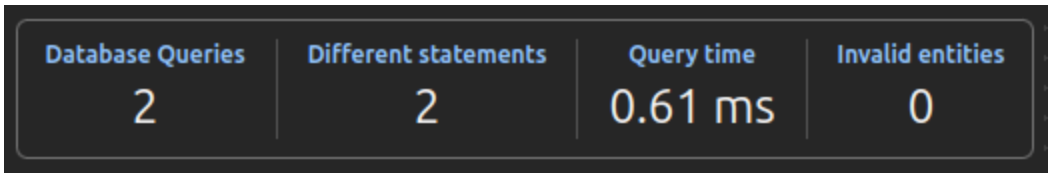


Ce résultat provient de la page de la liste des utilisateurs de l'application qui a été redirigée après la modification d'un utilisateur. Dans cette analyse, nous pouvons observer que la page a renvoyé un statut 200 indiquant une réponse HTTP réussie, et que le temps d'exécution était de 15 millisecondes, avec une utilisation maximale de la mémoire de 4MiB. On peut voir que nous sommes connectés en temps qu'admin et que la page a aussi passé 2 requêtes à la base de données avec un temps d'exécution de 0,61 ms.



L'image ci-dessus détaille le processus complet au-delà de 1 ms du chargement de la page après la redirection effectuée. On peut constater que la séquence débute par l'évènement "kernel.request" (avec un temps d'exécution de 6 ms et une utilisation mémoire de 4MiB). Ensuite, nous avons l'appel au "TraceableFirewallListener" (avec un temps d'exécution de 5 ms et une utilisation mémoire de 4MiB). Puis nous avons le "controller" (avec un temps d'exécution de 2,9 ms et une utilisation mémoire de 4MiB).

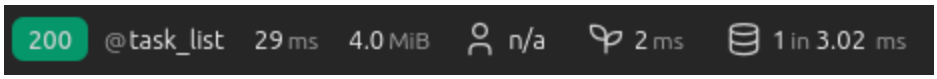
Pendant le chargement du “controller”, il convient de noter qu’il effectue également le chargement de la vue “user/list.html” (avec un temps d'exécution de 1ms et une utilisation mémoire de 4MiB).



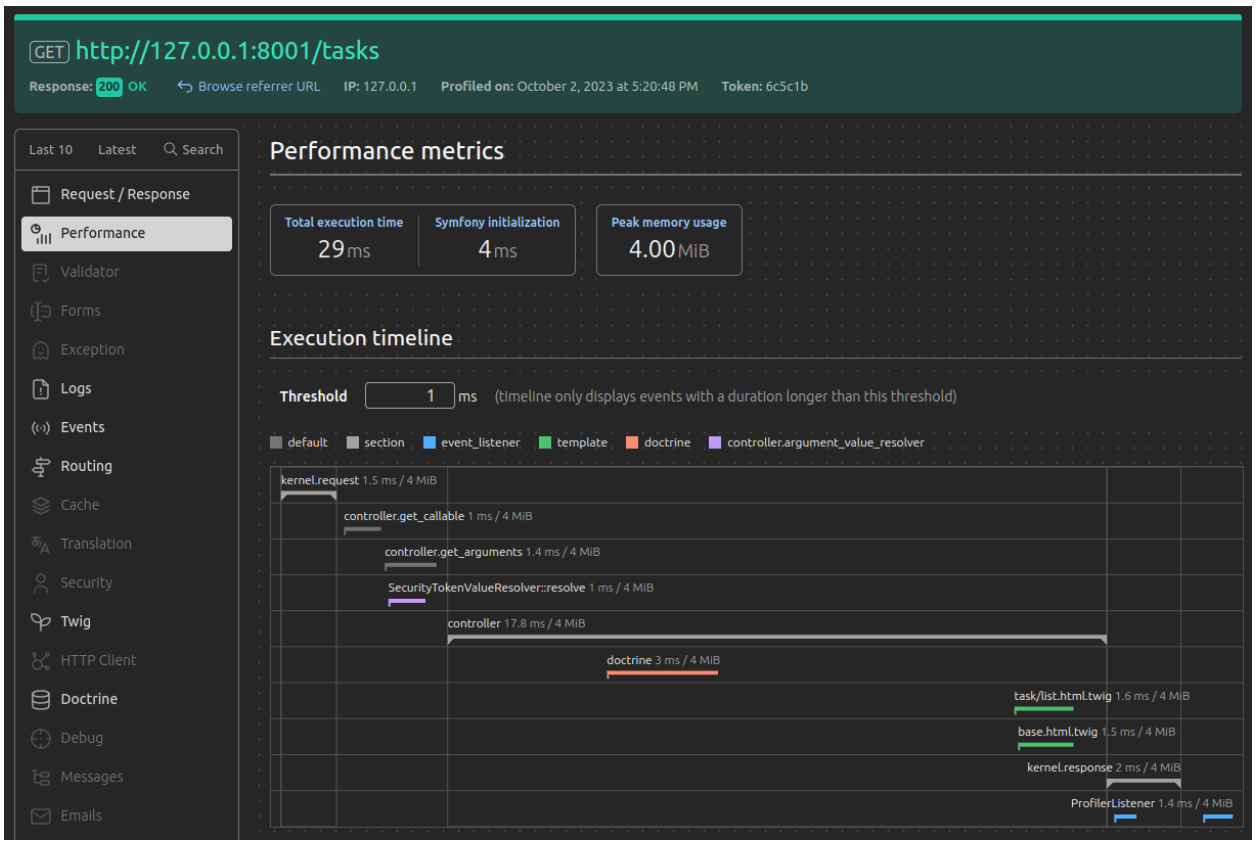
Avec les chiffres ci-dessus on observe qu’il y a 2 requêtes SQL qui ont été effectuées et 2 déclarations SQL distinctes. Cela signifie que 2 requêtes SQL différentes ont été exécutées sur la base de données pour une exécution de 0,61 millisecondes au total. Le profiler nous retourne aussi qu' aucune entité n’a été marquée comme invalide dans cette transaction avec la base de données.

En résumé, la page affiche des performances tout à fait satisfaisantes pour la modification d’un utilisateur. Le temps de chargement de la page est rapide, avec seulement 15 millisecondes d'exécution et L’utilisation de la mémoire est minime, à seulement 4MiB malgré les 2 requêtes SQL effectuées sur la base de données.

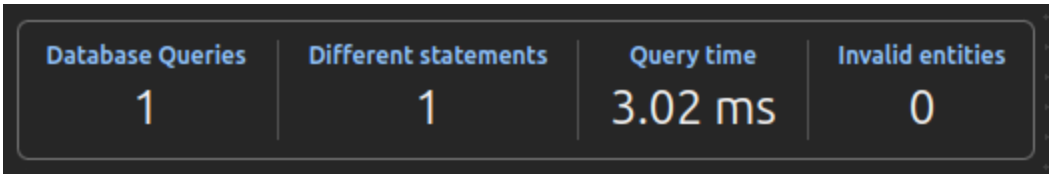
3.5. Liste des tâches



Ce résultat provient de la page de la liste des tâches de l’application. Dans cette analyse, nous pouvons observer que la page a renvoyé un statut 200 indiquant une réponse HTTP réussie, et que le temps d’exécution était de 29 millisecondes, avec une utilisation maximale de la mémoire de 4MiB. On peut voir qu’il n’y a pas d’utilisateur connecté et que la page a aussi passé une requête à la base de données avec un temps d'exécution de 3,02 ms.



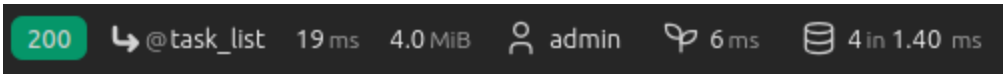
L'image ci-dessus détaille le processus complet au-delà de 1 ms du chargement de la page après la redirection effectuée. On peut constater que la séquence débute par l'évènement "kernel.request" (avec un temps d'exécution de 1,5 ms et une utilisation mémoire de 4MiB). Ensuite, nous avons le "controller.get_callable" (avec un temps d'exécution de 1 ms et une utilisation mémoire de 4MiB) ainsi que le "controller.get_arguments" (avec un temps d'exécution de 1,4 ms et une utilisation mémoire de 4MiB). Puis nous avons l'appel au "SecurityTokenValueResolver::resolve" (avec un temps d'exécution de 1 ms et une utilisation mémoire de 4MiB). Il y a ensuite le "controller" (avec un temps d'exécution de 17,8 ms et une utilisation mémoire de 4MiB). Pendant le chargement du "controller", il convient de noter qu'il effectue également le chargement de doctrine (avec une exécution de 3 ms et une utilisation de mémoire de 4MiB) et des vues "task/list.html" (avec un temps d'exécution de 1,6 ms et une utilisation mémoire de 4MiB) ainsi que "base.html" (avec un temps d'exécution de 1,5 ms et une utilisation mémoire de 4MiB).



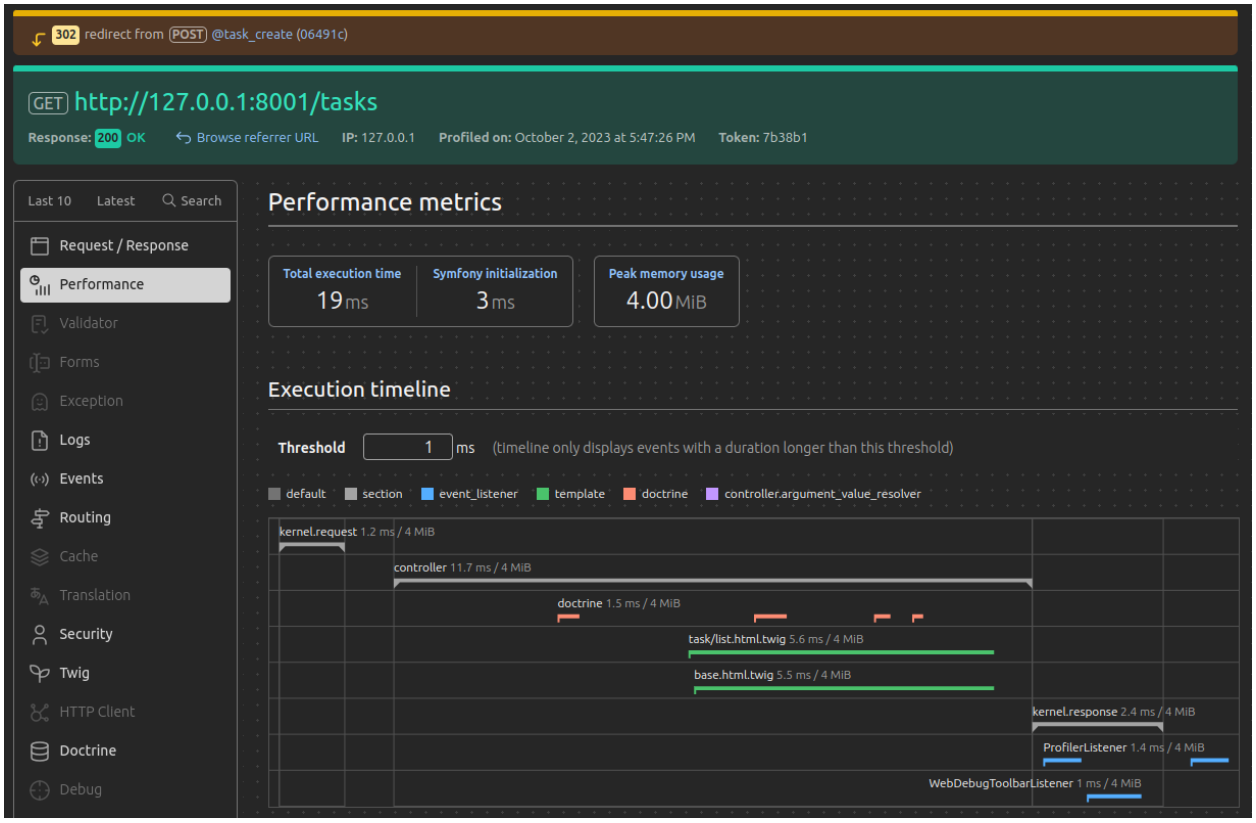
Avec les chiffres ci-dessus on observe qu'il y a une requête SQL qui a été effectuée et une déclaration SQL distincte pour une exécution de 3,02 millisecondes au total. Le profiler nous retourne aussi qu' aucune entité n'a été marquée comme invalide dans cette transaction avec la base de données.

En résumé, la page affiche des performances tout à fait satisfaisantes pour le chargement de la liste de toutes les tâches. Le temps de chargement de la page est rapide, avec seulement 29 millisecondes d'exécution et L'utilisation de la mémoire est minime, à seulement 4MiB.

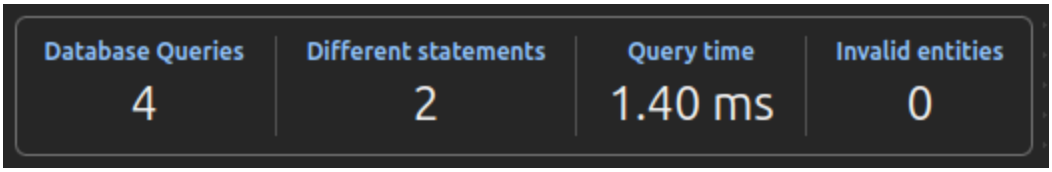
3.6. Création d'une tâche



Ce résultat provient de la page de la liste des tâches de l'application qui a été redirigée après la création d'une tâche. Dans cette analyse, nous pouvons observer que la page a renvoyé un statut 200 indiquant une réponse HTTP réussie, et que le temps d'exécution était de 19 millisecondes, avec une utilisation maximale de la mémoire de 4MiB. On peut voir que nous sommes connectés en temps qu'admin et que la page a aussi passé 4 requêtes à la base de données avec un temps d'exécution de 1,40 ms.



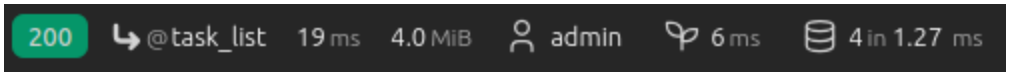
L'image ci-dessus détaille le processus complet au-delà de 1 ms du chargement de la page après la redirection effectuée. On peut constater que la séquence débute par l'évènement "kernel.request" (avec un temps d'exécution de 1,2 ms et une utilisation mémoire de 4MiB). Puis nous avons le "controller" (avec un temps d'exécution de 11,7 ms et une utilisation mémoire de 4MiB). Pendant le chargement du "controller", il convient de noter qu'il effectue également le chargement de doctrine (avec une exécution de 1,5 ms et une utilisation de mémoire de 4MiB) et des vues "task/list.html" (avec un temps d'exécution de 5,6 ms et une utilisation mémoire de 4MiB) ainsi que "base.html" (avec un temps d'exécution de 5,5 ms et une utilisation mémoire de 4MiB).



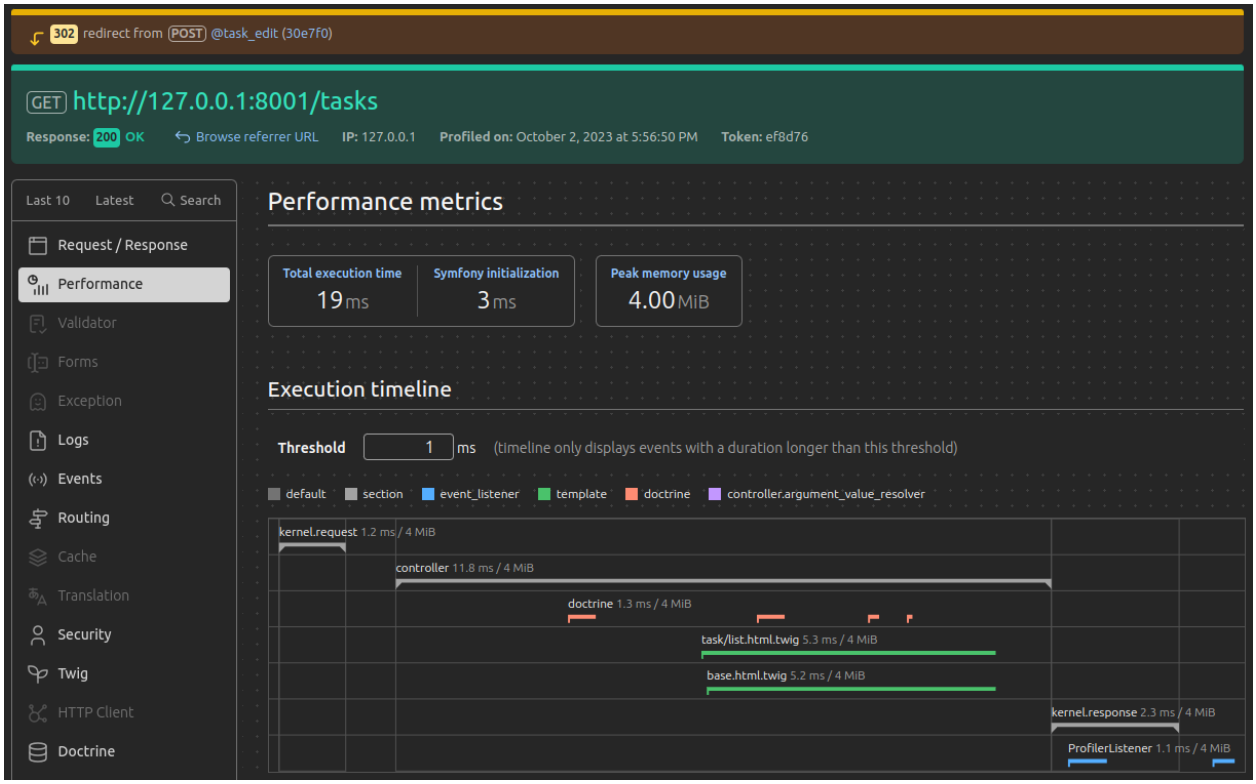
Avec les chiffres ci-dessus on observe qu'il y a 4 requêtes SQL qui ont été effectuées et 2 déclarations SQL distinctes. Cela signifie que 2 requêtes SQL différentes ont été exécutées sur la base de données pour une exécution de 1,40 millisecondes au total. Le profiler nous retourne aussi qu' aucune entité n'a été marquée comme invalide dans cette transaction avec la base de données. Dans le contexte de doctrine, cela pourrait indiquer que toutes les entités liées à la base de données sont valides du point de vue de la validation des données.

En résumé, la page affiche des performances tout à fait satisfaisantes pour la création d'une tâche. Le temps de chargement de la page est rapide, avec seulement 19 millisecondes d'exécution et L'utilisation de la mémoire est minime, à seulement 4MiB malgré les 4 requêtes SQL effectuées sur la base de données.

3.7. Modification d'une tâche

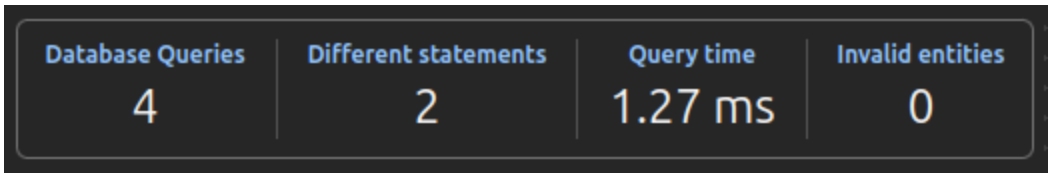


Ce résultat provient de la page de la liste des tâches de l'application qui a été redirigée après la modification d'une tâche. Dans cette analyse, nous pouvons observer que la page a renvoyé un statut 200 indiquant une réponse HTTP réussie, et que le temps d'exécution était de 19 millisecondes, avec une utilisation maximale de la mémoire de 4MiB. On peut voir que nous sommes connectés en temps qu'admin et que la page a aussi passé 4 requêtes à la base de données avec un temps d'exécution de 1,27 ms.



L'image ci-dessus détaille le processus complet au-delà de 1 ms du chargement de la page après la redirection effectuée. On peut constater que la séquence débute par l'évènement "kernel.request" (avec un temps d'exécution de 1,2 ms et une utilisation mémoire de 4MiB). Puis nous avons le "controller" (avec un temps d'exécution de 11,8 ms et une utilisation mémoire de 4MiB). Pendant le chargement du "controller", il convient de

noter qu'il effectue également le chargement de doctrine (avec une exécution de 1,3 ms et une utilisation de mémoire de 4MiB) et des vues "task/list.html" (avec un temps d'exécution de 5,3 ms et une utilisation mémoire de 4MiB) ainsi que "base.html" (avec un temps d'exécution de 5,2 ms et une utilisation mémoire de 4MiB).



Avec les chiffres ci-dessus on observe qu'il y a 4 requêtes SQL qui ont été effectuées et 2 déclarations SQL distinctes. Cela signifie que 2 requêtes SQL différentes ont été exécutées sur la base de données pour une exécution de 1,27 millisecondes au total. Le profiler nous retourne aussi qu'aucune entité n'a été marquée comme invalide dans cette transaction avec la base de données. Dans le contexte de doctrine, cela pourrait indiquer que toutes les entités liées à la base de données sont valides du point de vue de la validation des données.

En résumé, la page affiche des performances tout à fait satisfaisantes pour la modification d'une tâche. Le temps de chargement de la page est rapide, avec seulement 19 millisecondes d'exécution et L'utilisation de la mémoire est minime, à seulement 4MiB malgré les 4 requêtes SQL effectuées sur la base de données. On observe par ailleurs que les chiffres de la modification d'une tâche sont très proches des chiffres de la création d'une tâche, ce qui signifie que nous avons une application assez bien conçue.

3.8. Conclusion

Ces analyses en profondeur du chargement des pages sont cruciales pour comprendre les performances de l'application. Elle nous permet de repérer les éventuels goulots d'étranglement ou les zones de notre code qui pourraient être optimisées. Par exemple, si le temps d'exécution ou l'utilisation mémoire augmentent considérablement à mesure que l'application se développe, cela pourrait indiquer un besoin d'optimisation pour maintenir des performances rapides et efficaces.

En somme, cette évaluation détaillée du chargement des pages nous fournit des informations précieuses pour garantir que l'application continue de fonctionner de manière optimale, avec des temps de réponse rapides et une utilisation de mémoire efficace, ce qui contribue à une meilleure expérience utilisateur et une maintenance plus aisée.