

Audit de performance de code

Application To Do List

Décembre 2020

Caroline Dirat



Formation OpenClassrooms

« Développeur d'application PHP/Symfony »

Projet 8 - **Améliorez une application existante**

SOMMAIRE

Introduction	2
MESURES DE PERFORMANCE DE CODE	3
Page de connexion	3
→ Optimiser l'autoloader de Composer	
→ Configurer les variables d'environnement	
→ Utiliser le pré-chargement de la classe OPcache	
Page d'accueil	5
Page qui liste toutes les tâches	6
→ Trop de créations d'entités	
→ Trop de requêtes SQL	
→ Pagination	
DETTE TECHNIQUE	11
Réduire le nombre de créations d'entités et de requêtes SQL	11
Pagination des listes de ressources	13
En mode production – Liste de contrôle des performances.....	14
Liste de contrôle du serveur de production :	
1. Configuration des variables d'environnement en production	
2. Vider le conteneur de services dans un seul fichier	
3. Utilisez le cache de code d'octet OPcache	
4. Utilisez le pré-chargement de la classe OPcache	
5. Configurez OPcache pour des performances maximales	
6. Ne pas vérifier les horodatages des fichiers PHP	
7. Configurer le cache PHP realpath	
8. Optimiser l'autoloader de Composer	

INTRODUCTION

Comme conseillé par Symfony, la performance de l'application « To Do List » est analysée grâce à l'**application professionnelle** [Blackfire.io](https://blackfire.io).

Je n'ai pas pu exploiter toute la puissance de Blackfire qu'offre un abonnement « Premium » (intégration continue, assertions pour prévenir la régression de performance...). Mais grâce à [Github Student Developer Pack](#), j'ai pu bénéficier des fonctionnalités de l'abonnement « [Profiler](#) » pour analyser le code de l'application.

Aussi, nous analysons l'application en mode production (qui désactive XDebug et le suivi de ressources de configuration Symfony).

PAGE DE CONNEXION

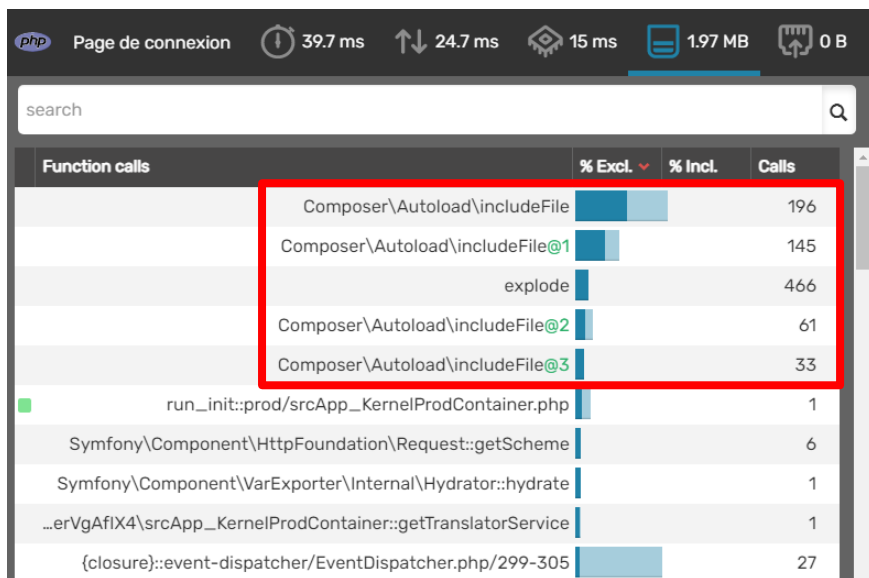
Quand un utilisateur n'est pas connecté, il est automatiquement redirigé sur la page de connexion « /login ».



Le temps de génération de la page est inférieur à 60ms, ce qui est un bon score.

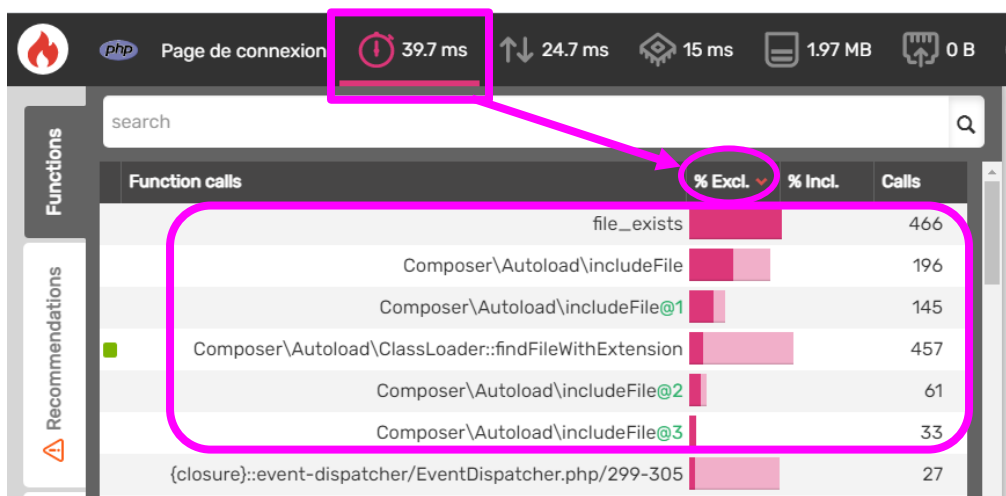
Et la mémoire utilisée proche de 2MB.

Regardons quelles sont les **fonctions les plus couteuses en mémoire** :

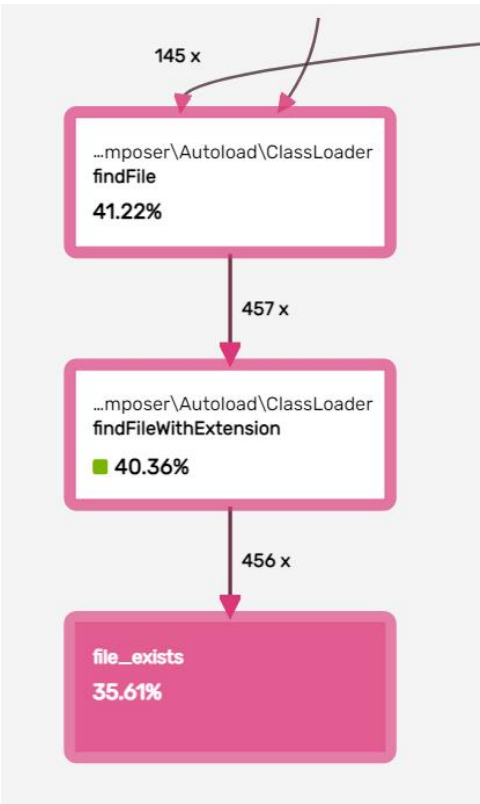


Ce sont des fonctions de l'**autoloader de Composer** (Composer\Autoload\ClassLoader). En effet, l'autoloader de Composer utilisé lors du développement de l'application est optimisé pour rechercher les classes nouvelles et modifiées. Et cela suppose d'aller chercher à chaque requête toutes les classes de l'application.

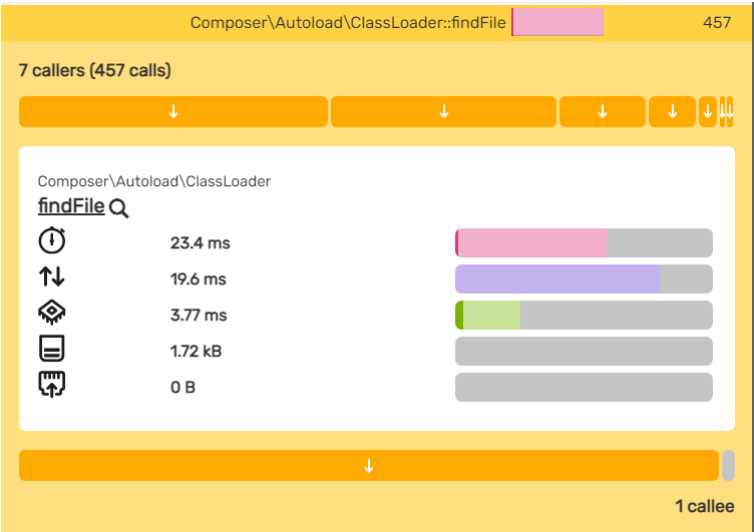
On constate la même prévalence de ces fonctions appelées par l'autoloader de Composer sur la **durées d'accès au système de fichiers (I/O)** :



Les méthodes de l'autoloader de Composer (**Composer\Autoload\ClassLoader**) sont effectivement appelées un très grand nombre de fois :



La fonction PHP **file_exists()** est appelée 456 fois par la méthode **findFileWithExtension()** de la classe **Composer\Autoload\ClassLoader**, qui elle-même est appelée 457 fois par la méthode **findFile()** de la même classe, qui elle-même est appelée 457 fois par la méthode **loadClass()** de la même classe.



La méthode **includeFile()** de **Composer\Autoload\ClassLoader** est appelée $196 + 145 + 61 + 33 + 13 = 448$ fois par la méthode **loadClass()** de la même classe :

Function calls	% Excl. ▾	% Incl.	Calls
Composer\Autoload\includeFile			196
Composer\Autoload\includeFile@1			145
Composer\Autoload\includeFile@2			61
Composer\Autoload\includeFile@3			33
Composer\Autoload\includeFile@4			13

Et la méthode **loadClass()** de **Composer\Autoload\ClassLoader** est appelée $197 + 145 + 5 + 61 + 33 + 14 = 455$ fois par la fonction **spl_autoload_call()** :

Function calls	% Excl. ▾	% Incl.	Calls
Composer\Autoload\ClassLoader::loadClass			197
Composer\Autoload\ClassLoader::loadClass@1			145
Composer\Autoload\ClassLoader::loadClass@5			5
Composer\Autoload\ClassLoader::loadClass@2			61
Composer\Autoload\ClassLoader::loadClass@3			33
Composer\Autoload\ClassLoader::loadClass@4			14

Dans la dernière partie « DETTE TECHNIQUE → LISTE DE CONTRÔLE DES PERFORMANCES » de ce document, on détaillera la procédure pour **optimiser l'autoloader de Composer**. Et retenez que cette optimisation n'a de sens qu'en **environnement de production**.

Aussi Blackfire relève **deux autres recommandations** en rapport avec l'environnement de **production** :

Recommendations

Subscribe to the add-ons

- .env configuration should not be parsed in production**
`metrics.symfony.dotenv.parse.count` `2` `== 0`
- PHP Preloading should be configured**
When
`is_extension_loaded("zend_opcache")`
then:
`runtime.configuration.opcache_preload` `!` `""`
- The Composer autoloader class map should be dumped in production
`metrics.composer.autoload.find_file.count` `457` `<= 50`

- ▶ « **.env configuration should not be parsed in production** » qui demande d'optimiser l'accès aux variables d'environnement.
- ▶ « **PHP Preloading should be configured** » qui demande d'utiliser le pré-chargement de la classe OPcache.

Ces deux points sont aussi traités dans la partie « DETTE TECHNIQUE → LISTE DE CONTRÔLE DE PERFORMANCES ».

PAGE D'ACCUEIL

Une fois l'utilisateur connecté, la page d'accueil est très légèrement plus lente à afficher que la page de connexion, puisqu'il faut le temps de traiter la session utilisateur.

Accueil - To Do List 43.7 ms 25.3 ms 18.5 ms 2.65 MB 114 kB 0 μs / 0 rq 426 μs / 1 rq

D'ailleurs, on peut remarquer qu'il n'y a qu'une seule requête à la base de données, et qu'il s'agit de celle récupérant les données de l'utilisateur connecté.

SQL Queries: 426 μs / 1 rq		
DB Connection: 17.2 ms		
Calls	Time	SQL
1x	426 μs	select ... from user t0 where t0.id = ?

Il n'y a aucune autre requête à la base de données, ce qui est une bonne chose pour minimiser le temps d'affichage d'une page d'accueil qui doit être la plus rapide possible (pour le confort de l'utilisateur).

PAGE QUI LISTE TOUTES LES TACHES

Pour la page qui liste toutes les tâches, Blackfire repère un trop grand nombre de requêtes à la base de données et un trop grand nombre de création d'entités :

LESS ORM ENTITIES SHOULD BE CREATED – MOINS D'ENTITÉS DOIVENT ÊTRE CRÉES

74 entités sont créées (alors qu'il y a 51 tâches à lister et 4 utilisateurs parmi les auteurs des tâches)

Lors de l'utilisation d'un *Object Relation Mapper* (ORM) pour gérer des données avec une base de données relationnelle, un objet appelé « Entité » est créé pour chaque entrée extraite des requêtes adressées à la base de données. Cela inclut généralement les entrées du résultat des relations configurées (One to many, One to one et Many to Many).

Dans certains cas, quelques requêtes SQL peuvent entraîner la création de nombreuses entités. Un exemple typique est la charge d'objets « Article », déclenchant la charge de nombreux objets associés « Comment ».

Dans notre application, les entités User et Task sont reliées par une relation One To Many (**Un utilisateur** peut être l'auteur de **plusieurs tâches**, tandis qu'**une tâche** est reliée à un seul auteur qui correspond à **un seul utilisateur**).

Lors de l’affichage de la liste des tâches, le template `templates/task/list.html.twig` reçoit du contrôleur un tableau d’objets `Tasks`, qui correspond à la création de 51 entités. Et puis pour chaque tâche, on appelle l’objet `User` associé par la relation `OneToMany` pour obtenir la valeur de sa propriété « `username` » pour définir l’auteur de la tâche.

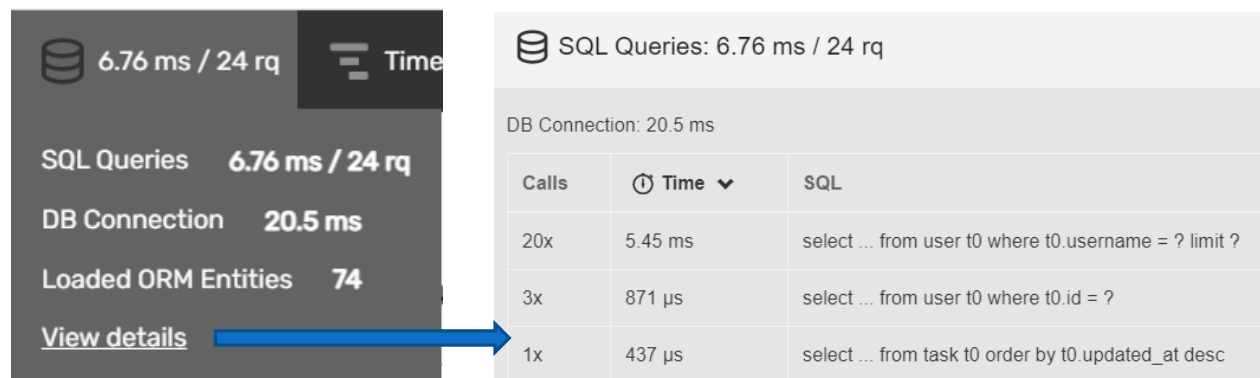
```
# templates/task/list.html.twig
% for task in tasks %}
    <div class="col-md-6 col-lg-4">
        <div class="card my-3">
            <div class="card-body">
                <div class="d-flex justify-content-between">
                    <h4 class="mr-1"><a href="{{ path('task_edit', {'id': task.id }) }}">{{ task.title }}</a></h4>
                    <h4>
                        # icon for “is Done” or “is not Done”
                    </h4>
                </div>
                <p>{{ task.content }}</p>
                <p class="text-primary">{{ task.user.username }}</p>
            </div>
        </div>
    </div>
# ...
```

Une solution à la création d’un trop grand nombre d’entités peut être de récupérer un tableau de tableaux au lieu d’un tableau d’entités. Cela se fait en utilisant la clause `JOIN` dans le constructeur de requête DQL d’une méthode personnalisée du `Repository` qui permettra de récupérer non seulement la liste de toutes les tâches, mais aussi le nom de l’auteur de chaque tâche. Cette méthode personnalisée est alors appelée dans la méthode `listAll()` du `contrôleur TaskController` (au lieu du simple `findBy()`).

Cette solution est présentée dans la partie « DETTE TECHNIQUE → REDUIRE LE NOMBRE DE CREATIONS D’ENTITEES ET DE REQUETES SQL »

YOU SHOULD EXECUTE LESS SQL QUERIES – VOUS DEVRIEZ EXECUTER MOINS DE REQUETES SQL

En fait, en cliquant sur « View details » sur l’onglet de la métrique qui donne le nombre de requêtes SQL, on peut voir que 23 requêtes sont effectuées pour récupérer les utilisateurs (alors qu’il n’y que 5 utilisateurs dans la base de données).

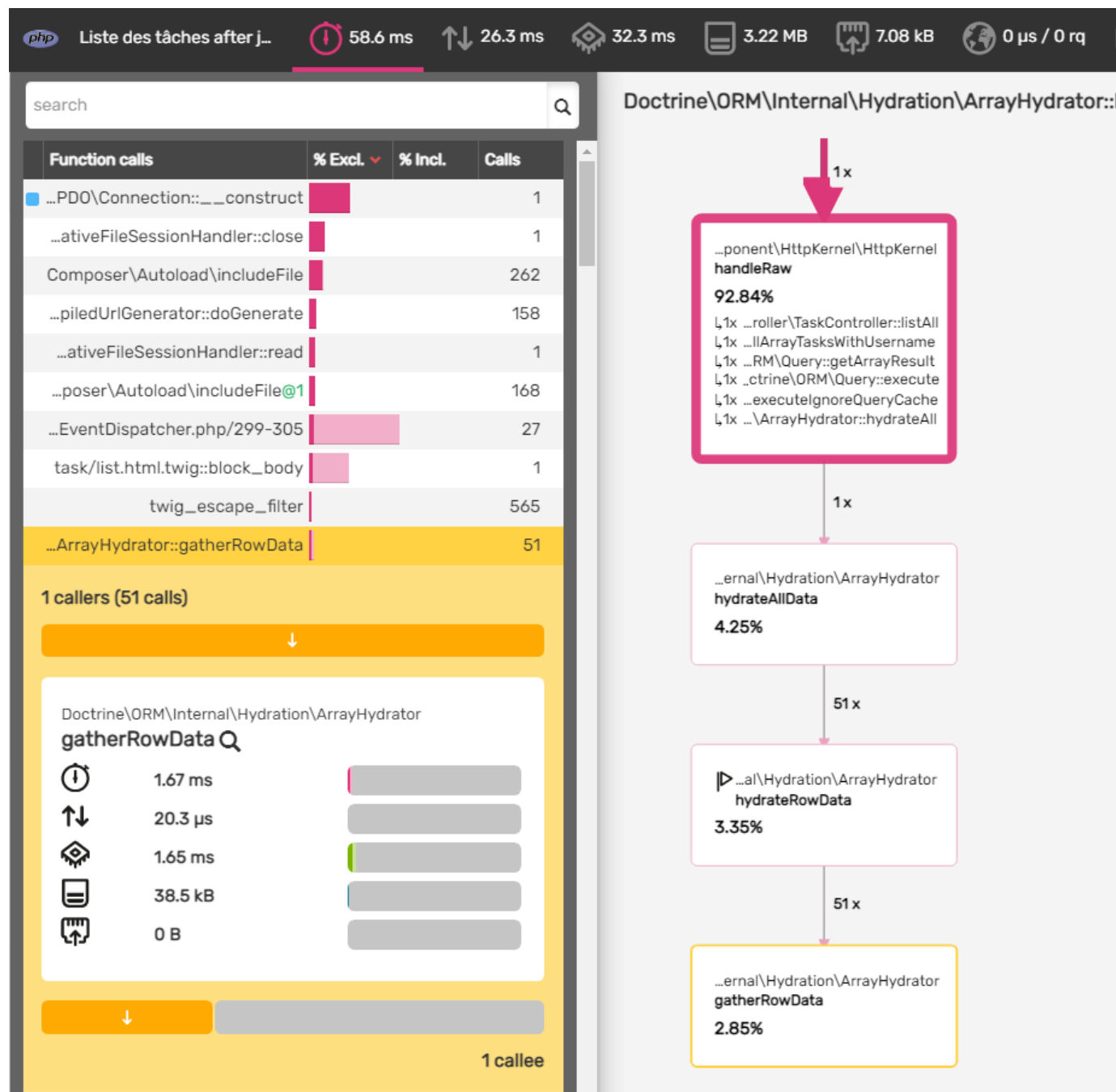


SQL Queries: 6.76 ms / 24 rq		
DB Connection: 20.5 ms		
Calls	Time	SQL
20x	5.45 ms	select ... from user t0 where t0.username = ? limit ?
3x	871 µs	select ... from user t0 where t0.id = ?
1x	437 µs	select ... from task t0 order by t0.updated_at desc

Une solution à cette recommandation est présentée dans la partie « DETTE TECHNIQUE → REDUIRE LE NOMBRE DE CREATIONS D’ENTITEES ET DE REQUETES SQL »

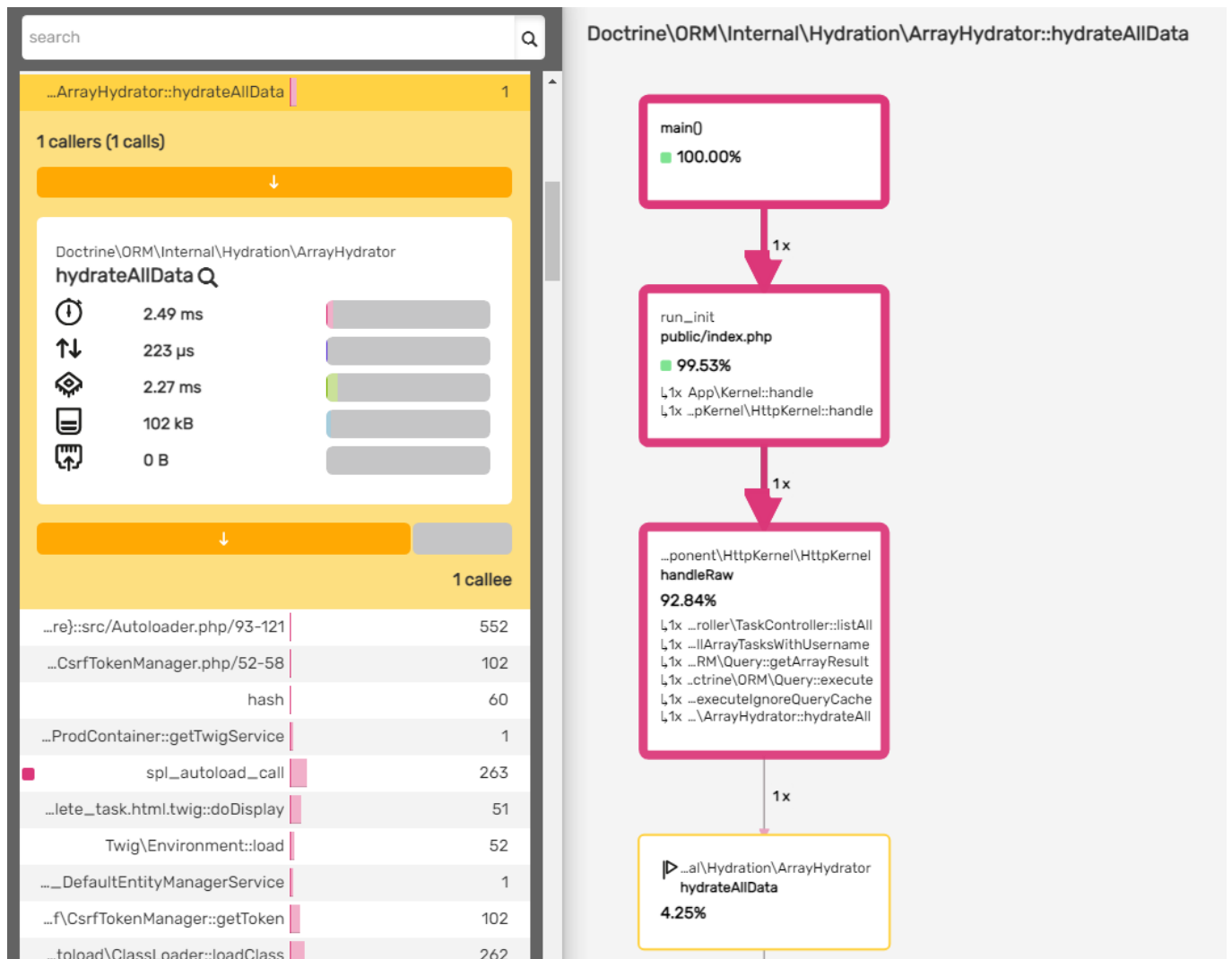
PAGINATION

Observons à nouveau l'analyse de Blackfire de la page qui liste la totalité des tâches :

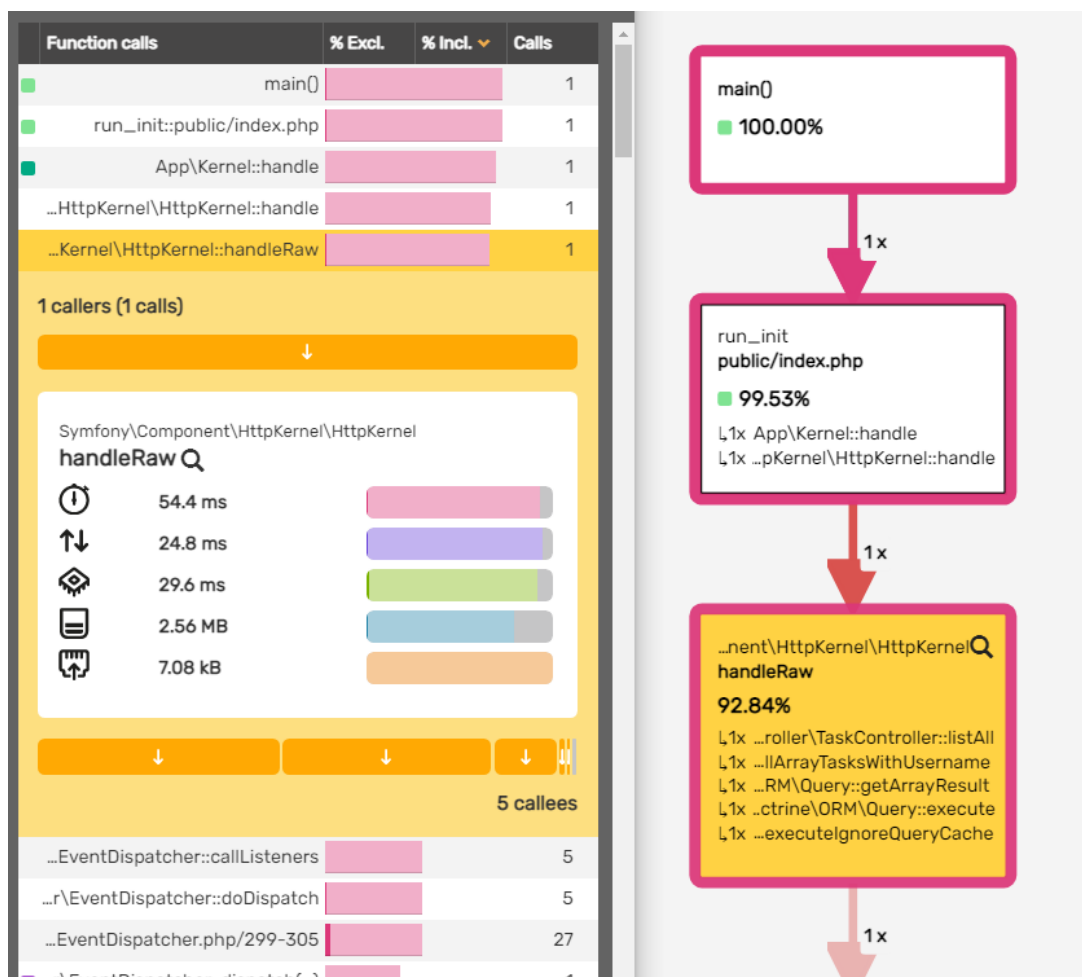


À la 10^{ème} position, sur la liste des fonctions triées selon l'impact qu'elles ont sur la performance par leur coût exclusif, on trouve la méthode **gatherRowData()** de la classe **Doctrine\ORM\Internal\Hydration\ArrayHydrator**. Cette méthode est appelée **51 fois** par la méthode **hydrateRowData()** de la même classe, qui est aussi appelée **51 fois** par la méthode **hydrateAllData()** de la même classe.

« 51 » c'est le nombre total de tâches disponibles dans la base de données, et que l'application va chercher pour afficher la liste de la totalité des tâches.

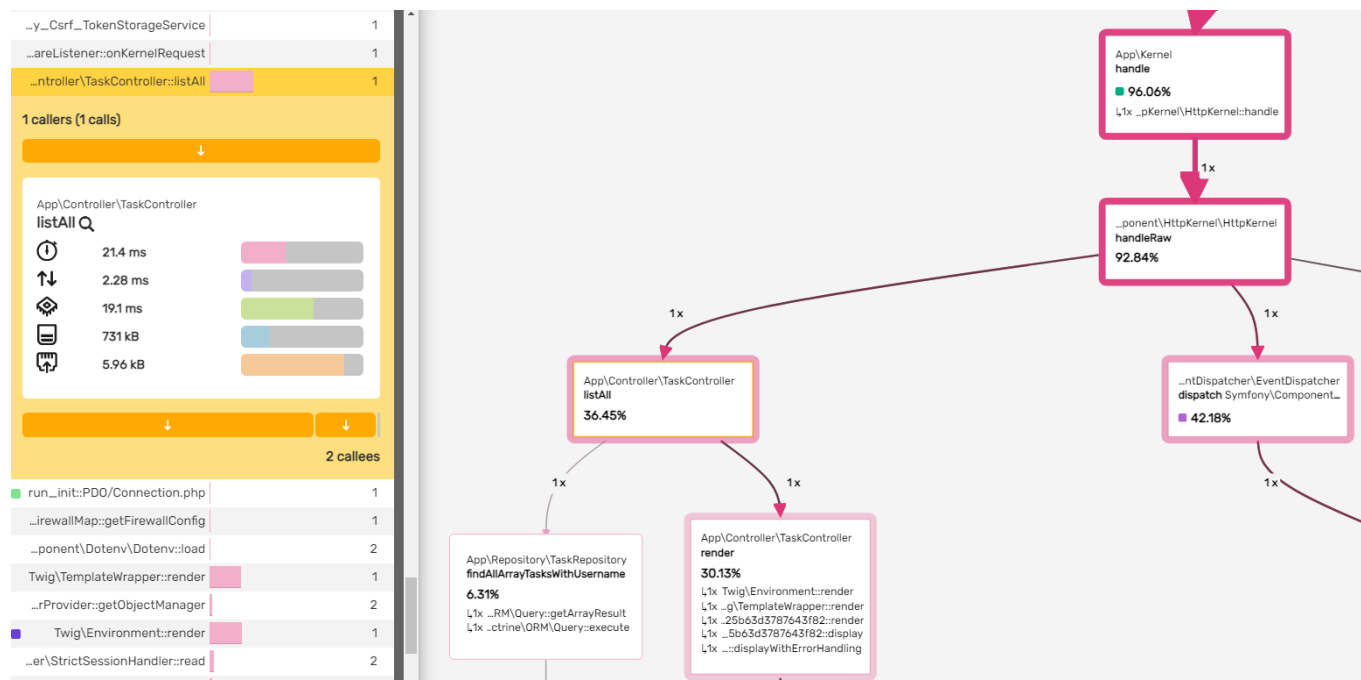


La méthode **hydrateAllData()**, de la classe **Doctrine\ORM\Internal\Hydration\ArrayHydrator**, est appelée par la méthode **handleRaw()** de la classe **Symfony\Component\HttpKernel\HttpKernel** dont le **coût** inclusif de **92,84%** est très important.

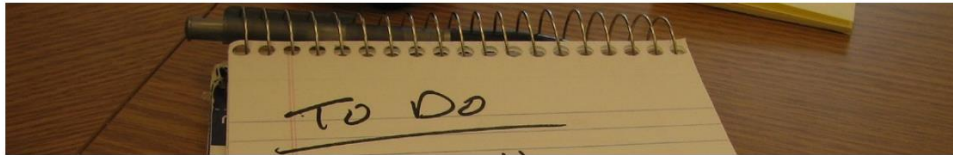


D'ailleurs, la méthode **handleRaw()** du **HttpKernel** est en **5^{ème} position** sur la liste des fonctions triées selon l'impact qu'elles ont sur la performance par **leur coût inclusif**.

On comprend ainsi comment le nombre de tâches à aller chercher dans la base de données va impacter sur la performance de la page qui liste les tâches.



La requête http avec l'URI « /tasks » exécute la méthode **listAll()** du controller **App\Controller\TaskController** qui va aller chercher TOUTES les tâches, alors qu'au maximum six tâches sont initialement visibles sur la page :



Créer une tâche

Consulter la liste des tâches à faire

Consulter la liste des tâches terminées

Liste des tâches

Tâche n°51 <input type="checkbox"/> Texte du contenu de la tâche n°51 user2 Modifié le 04/12/2020 à 06:01 Marquer comme faite Supprimer	Tâche n°50 <input checked="" type="checkbox"/> Texte du contenu de la tâche n°50 user2 Modifié le 04/12/2020 à 05:01 Marquer non terminée Supprimer	Tâche n°49 <input type="checkbox"/> Texte du contenu de la tâche n°49 user2 Modifié le 04/12/2020 à 04:01 Marquer comme faite Supprimer
Tâche n°48 <input checked="" type="checkbox"/> Texte du contenu de la tâche n°48	Tâche n°47 <input type="checkbox"/> Texte du contenu de la tâche n°47	Tâche n°46 <input checked="" type="checkbox"/> Texte du contenu de la tâche n°46

Pour améliorer la performance de la page qui liste toute les tâches, il faudrait récupérer moins de tâches. On peut par exemple ne récupérer que les six premières tâches de la liste, puis afficher les tâches suivantes à la demande de l'utilisateur comme par exemple :

- ✓ En cliquant sur un bouton pour afficher n tâches suivantes.
- ✓ En scrollant la page vers le bas pour déclencher l'affichage des n tâches suivantes.
- ✓ ...

Il s'agit donc de décider du comportement de l'utilisateur qui déclenchera une requête Ajax qui récupèrera les tâches suivantes à afficher sur la page.

DETTE TECHNIQUE

REDUIRE LE NOMBRE DE CREATIONS D'ENTITEES ET DE REQUETES SQL

UTILISER LA CLAUSE JOIN POUR AJOUTER DES CHAMPS SUPPLEMENTAIRES A LA REQUETE

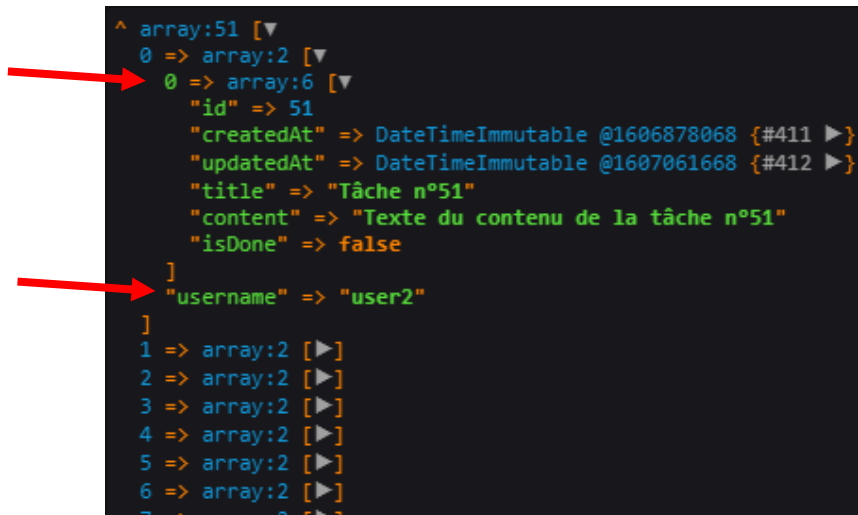
Voici une proposition d'optimisation de code pour diminuer le nombre de créations d'entités en utilisant la clause **JOIN** dans le constructeur de requête d'une méthode personnalisée, par exemple nommées **findAllArrayTasksWithUsername()**, qui récupère la liste des tâches.

```
// App\Repository\TaskRepository
/**
 * findAllArrayTasksWithUsername
 *
 * @return array<int, array<mixed, mixed>
 */
public function findAllArrayTasksWithUsername(): array
{
    return $this->createQueryBuilder('task')
        ->leftJoin('task.user', 'user')
        ->addSelect('user.username as username')
        ->orderBy('task.updatedAt', 'DESC')
        ->getQuery()
        ->getArrayResult()
    ;
}
```

Cette méthode est alors appelée dans le Controller App\Controller\TaskController (à la place de la méthode **findBy()** de l'EntityRepository) pour récupérer la liste des tâches sous la forme, non plus d'un tableau d'entités, mais d'un tableau dont deux items contiennent le tableau des propriétés de la tâche pour l'un et le nom de l'auteur correspondant pour l'autre :

```
/**
 * list all tasks.
 *
 * @Route("/tasks", name="task_list_all")
 *
 * @return Response
 */
public function listAll(TaskRepository $taskRepository): Response
{
    return $this->render(
        'task/list.html.twig',
        [
            'tasks' => $taskRepository->findAllArrayTasksWithUsername(),
        ]
    );
}
```

Donc ce controller envoie au template un tableau de tableaux, où chaque tableau contient un item 0 dont la valeur est un tableau des propriétés d'une tâche et un item « username » dont la valeur associée est le nom de l'auteur de la tâche :



```
^ array:51 [▼
  0 => array:2 [▼
    0 => array:6 [▼
      "id" => 51
      "createdAt" => DateTimeImmutable @1606878068 {#411 ▶}
      "updatedAt" => DateTimeImmutable @1607061668 {#412 ▶}
      "title" => "Tâche n°51"
      "content" => "Texte du contenu de la tâche n°51"
      "isDone" => false
    ]
    "username" => "user2"
  ]
  1 => array:2 [▶]
  2 => array:2 [▶]
  3 => array:2 [▶]
  4 => array:2 [▶]
  5 => array:2 [▶]
  6 => array:2 [▶]
  7 => array:2 [▶]
```

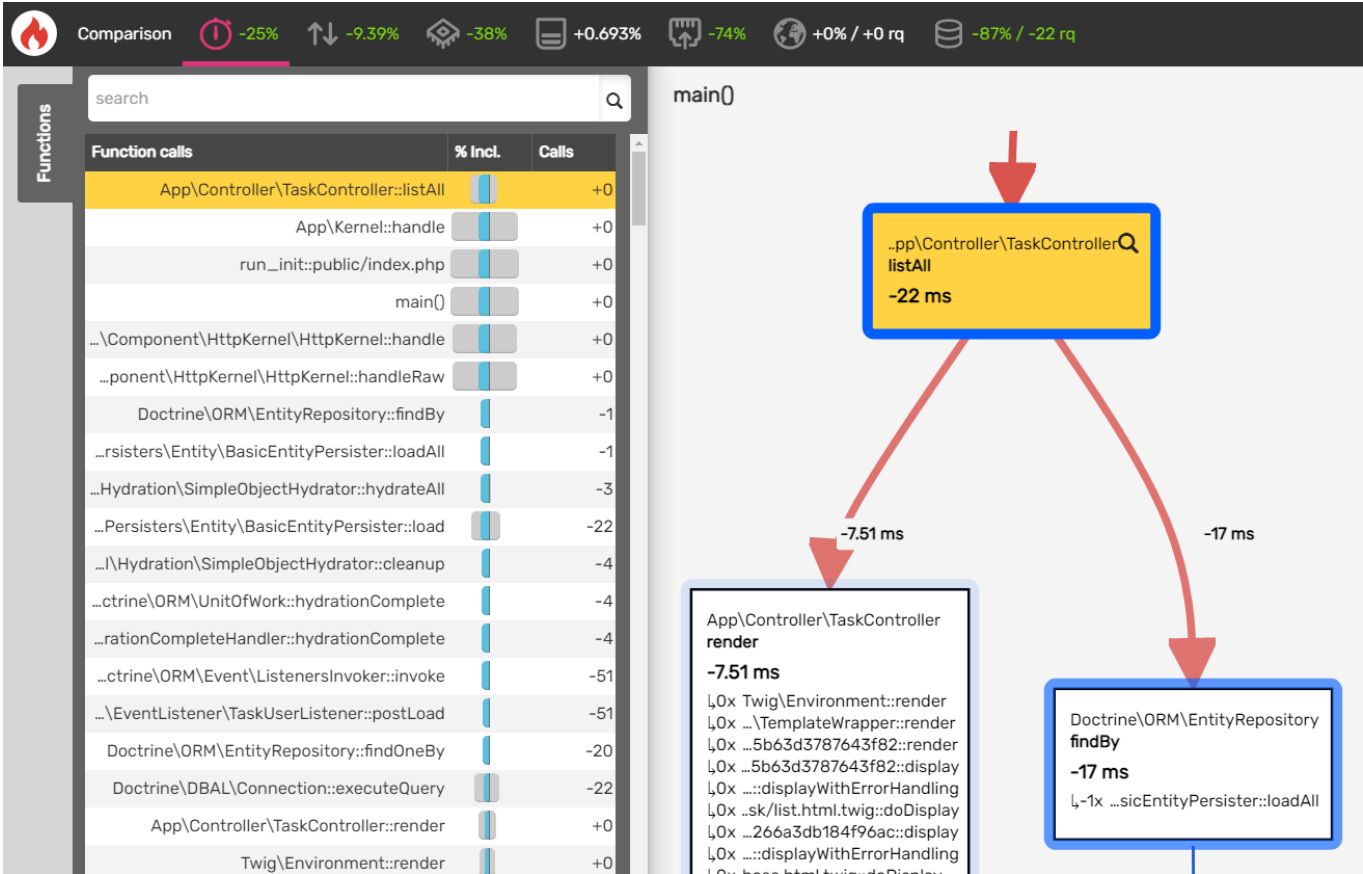
Puis le template **template/task/list.html.twig** qui affiche la liste des tâches est modifié en conséquence :

```
{# template/task/list.html.twig #}
{% for task in tasks %}
    {% set author = task.username %}
    {% set task = task.0 %}
    <div class="col-md-6 col-lg-4">
        <div class="card my-3">
            <div class="card-body">
                <div class="d-flex justify-content-between">
                    <h4 class="mr-1"><a href="{{ path('task_edit', {'id': task.id }) }}">{{ task.title }}</a></h4>
                    <h4>
                        {# icon for "is done" or "is not done" #}
                    </h4>
                </div>
                <p>{{ task.content }}</p>
                <p class="text-primary">{{ author ?? 'Anonymous' }}</p>
            </div>
        </div>
    </div>
{{ ... }}
```

**** MAIS** comme il n'a plus d'entités Task créées, l'EventListener App\EventListener\TaskUserListener n'est plus appelé pour associer les tâches orphelines à l'utilisateur « Anonymous » de la base de données. On se contente alors de remplacer les valeurs nulles de la clé « username » par le nom d'auteur « Anonymous ».

Cette optimisation nous permet de ne plus créer d'entités, on passe de 74 entités créées à zéro ! En outre, on passe de 24 requêtes SQL à seulement 2, avec une diminution de 25% de la durée d'affichage de la page.

Donc au final, cette optimisation permet d'agir sur les deux recommandations de Blackfire qui avait repéré trop de requêtes SQL et trop de créations d'entités.



SQL Queries: -5.9 ms / -22 rq

Calls	⌚ Time ▼	SQL
20x Gone	-5.38 ms	select ... from user t0 where t0.username = ? limit ?
1x -2	267 µs -604 µs	select ... from user t0 where t0.id = ?
1x New	+489 µs	select ... from task t0_ left join user u1_ on t0._user_id = u1._id order by t0._updated_at desc
1x Gone	-382 µs	select ... from task t0 order by t0.updated_at desc

PAGINATION DES LISTES DE RESSOURCES

Pour améliorer la performance de l'application, il faut **paginer les listes de ressources** (des tâches et des utilisateurs).

Lorsqu'une page liste des ressources (des tâches ou des utilisateurs), la requête HTTP appelle un contrôleur qui va appeler une méthode du Repository de l'entité correspondant au type de ressource pour récupérer les données.

Par exemple, la page pour l'URI « /users » qui liste les utilisateurs appelle la méthode `list()` du contrôleur `App\Controller\UserController` :

```
public function list(): Response
{
    return $this->render(
        'user/list.html.twig',
        ['users' => $this->getDoctrine()->getRepository(User::class)->findAll()]
    );
}
```

On peut voir que la méthode `findAll()` du repository `App\Repository\UserRepository` est appelée pour récupérer tous les utilisateurs.

La pagination des ressources consiste alors à créer une méthode dans le Repository de l'entité concernée pour récupérer un nombre donné de ressources (appelé « limit ») et à partir d'un « offset ».

Par exemple, on peut créer la méthode `getPaginatedUsers(int $limit = 6, int $offset = 0)` qui récupère \$limit utilisateurs à partir de celui indiqué par la valeur de \$offset.

Ce qui donne dans le controller :

```
public function list(UserRepository $userRepository): Response
{
    return $this->render(
        'user/list.html.twig',
        ['users' => $userRepository->getPaginatedUsers($limit, $offset)];
    );
}
```

Les valeurs par défaut de \$limit et \$offset (6 et 0 ici) peuvent être définies dans un fichier de configuration, tandis qu'elles pourront être précisées en fonction de la requête.

Ce point de performance demandant de paginer les listes de ressources concerne plusieurs pages de l'application :

- La page de la liste de toutes les tâches sur l'URI « /tasks »
- La page de la liste des tâches terminées sur l'URI « /tasks/true »
- La page de la liste des tâches à faire sur l'URI « /tasks/false »
- La page de la liste des utilisateurs sur l'URI « /users »

EN MODE PRODUCTION – LISTE DE CONTROLE DES PERFORMANCES

La liste de contrôle des performances d'une application Symfony et de son serveur de production est disponible dans la documentation officielle de Symfony

(<https://symfony.com/doc/4.4/performance.html>).

On énumère ci-après cette liste. Et on détaille les points que l'on a précédemment identifiés lors des analyses avec Blackfire :

- ✓ Configuration des variables d'environnement en production
- ✓ Utiliser le pré-chargement de la classe OPcache
- ✓ Optimiser l'autoload de Composer

LISTE DE CONTROLE DE L'APPLICATION SYMFONY

- Installez APCu Polyfill si votre serveur utilise APC au lieu d'OPcache – [Documentation Symfony](#)

LISTE DE CONTROLE DU SERVEUR DE PRODUCTION

1. Configuration des variables d'environnement en production

En production, les fichiers `.env` sont également analysés et chargés à chaque requête. Le moyen le plus simple de définir des variables d'environnement consiste donc à déployer un fichier `.env.local` sur vos serveurs de production avec vos valeurs de production.

Pour améliorer les performances, vous pouvez éventuellement exécuter la commande `dump-env` (disponible dans [Symfony Flex](#) 1.2 ou version ultérieure):

```
# parses ALL .env files and dumps their final values to .env.local.php
composer dump-env prod
```

Après avoir exécuté cette commande, Symfony chargera le fichier `.env.local.php` pour obtenir les variables d'environnement et ne passera pas de temps à analyser les fichiers `.env`.

REMARQUE

Mettez à jour vos outils de déploiement et workflow pour exécuter la commande « `dump-env` » **après chaque déploiement** afin d'améliorer les performances de l'application.

2. Vider le conteneur de services dans un seul fichier – [Documentation Symfony](#)

3. Utilisez le cache de code d'octet OPcache – [Documentation Symfony](#)

4. Utilisez le pré-chargement de la classe OPcache – [Documentation Symfony](#)

À partir de PHP 7.4, OPcache peut compiler et charger des classes au démarrage et les rendre disponibles à toutes les requêtes jusqu'au redémarrage du serveur, améliorant ainsi considérablement les performances.

Lors de la compilation du conteneur (par exemple lors de l'exécution de la commande `cache:clear`), Symfony génère un fichier appelé `preload.php` dans le répertoire `config/` avec la liste des classes à pré-charger.

La seule exigence est que vous devez définir dans `src/Kernel.php` les paramètres `container.dumper.inline_factories` et `container.dumper.inline_class_loader` sur `true`. Ensuite, vous pouvez configurer PHP pour utiliser ce fichier de pré-chargement :

```
; php.ini
opcache.preload="/path/to/project/config/preload.php"
```

REMARQUE

Le pré-chargement d'OPcache n'est pas supporté sur Windows. ([Doc PHP](#))

5. Configurez OPcache pour des performances maximales – [Documentation Symfony](#)

6. Ne pas vérifier les horodatages des fichiers PHP – [Documentation Symfony](#)

7. Configurer le cache PHP realpath – [Documentation Symfony](#)

8. Optimiser l'autoloader de Composer :

Par défaut, le chargeur automatique des classes (l'autoloader) de Composer s'exécute relativement rapidement. Cependant, en raison de la façon dont les règles de chargement automatique PSR-4 et PSR-0 sont configurées, il doit vérifier tout le système de fichiers avant d'identifier définitivement un nom de classe. Cela ralentit un peu les choses : c'est pratique dans les environnements de développement car lorsque vous ajoutez une nouvelle classe, elle peut être automatiquement repérée et utilisée.

Cependant, **en production**, on veut que les choses se passent aussi vite que possible. Aussi, on peut construire la configuration à chaque fois que l'on déploie et les nouvelles classes n'apparaissent pas au hasard entre les déploiements. C'est pourquoi on peut optimiser le chargeur automatique de

Composer pour analyser une seule fois l'application entière et créer une « carte des classes » optimisée de l'application.

Pour cela, Composer propose quelques stratégies pour optimiser son autoloader en mode production (voir la [documentation de Composer](#) pour optimiser son autoloader).

Pour une application Symfony utilisant Opcache, nous allons activer les niveaux 1 et 2.A des stratégies disponibles (détaillées dans la [documentation de Composer](#)).

Une fois l'application terminée, et pour le mode production (où les fichiers PHP ne devraient jamais changer, sauf si une nouvelle version d'application est déployée), on va créer une « carte de classe » optimisée, qui est un grand tableau des emplacements de toutes les classes, et elle est stockée dans **vendor/composer/autoload_classmap.php**.

Pour générer cette « carte de classe », exécutez la commande suivante (le mieux est de l'intégrer à votre processus de déploiement) :

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

- L'option **--no-dev** exclut les classes qui ne sont nécessaires que dans l'environnement de développement (c'est-à-dire les dépendances **require-dev** et les règles **autoload-dev** définies dans le fichier **composer.json**);
- L'option **--classmap-authoritative** crée un mappage de classes pour les classes compatibles PSR-0 et PSR-4 utilisées dans votre application et empêche Composer d'analyser le système de fichiers pour les classes qui ne se trouvent pas dans le mappage de classes.

REMARQUE :

Le bénéfice de l'optimisation de l'autoloader est très important. Il s'agit de diminuer de moitié la durée de génération des pages, comme le montre la comparaison de deux profils de Blackfire (de la page de connexion), avant et après l'optimisation de l'autoloader de Composer :

