

# Code performance audit

To Do List application

2020 December

Caroline Dirat



OpenClassrooms

« Application Developer PHP/Symfony »

Project 8 – **Improve an existing application**

## SUMMARY

Introduction .....	2
<b>CODE PERFORMANCE MEASUREMENTS.....</b>	<b>3</b>
<b>Login page.....</b>	<b>3</b>
→ Optimize Composer autoloader	
→ Configuring environment variables	
→ Use the OPcache class preloading	
<b>Home page.....</b>	<b>5</b>
<b>Page of the list of all tasks .....</b>	<b>6</b>
→ Less ORM entities should be created	
→ To many SQL queries	
→ Paging	
<b>TECHNICAL DEBT.....</b>	<b>11</b>
<b>Reduce the number of created ORM entities and SQL queries.....</b>	<b>11</b>
<b>Resources list paging .....</b>	<b>13</b>
<b>In production mode – Performance checklists.....</b>	<b>14</b>
Production server checklist:	
1. Configuring environment variables	
2. Dump the service container in a single file	
3. Use the OPcache byte code cache	
4. Use the Opcache class preloading	
5. Configure Opcache for maximum performance	
6. Don't check PHP files timestamps	
7. Configure the PHP <i>realpath</i> cache	
8. Optimize the Composer autoloader	

## INTRODUCTION

As advised by Symfony, the **To Do List** application performance is analyzed with the **professional application [Blackfire.io](https://blackfire.io)**.

*I have not been able to harness all the Blackfire power offered by a “Premium” subscription (Continuous integration, assertions to prevent performance regression...). But thanks to the à [Github Student Developer Pack](#), I have been able to take advantage of the features of the “Profiler” subscription to analyze the application code.*

Also, we analyze the application in production mode (which disable XDebug and the Symfony configuration resource tracking).

## PAGE DE CONNEXION

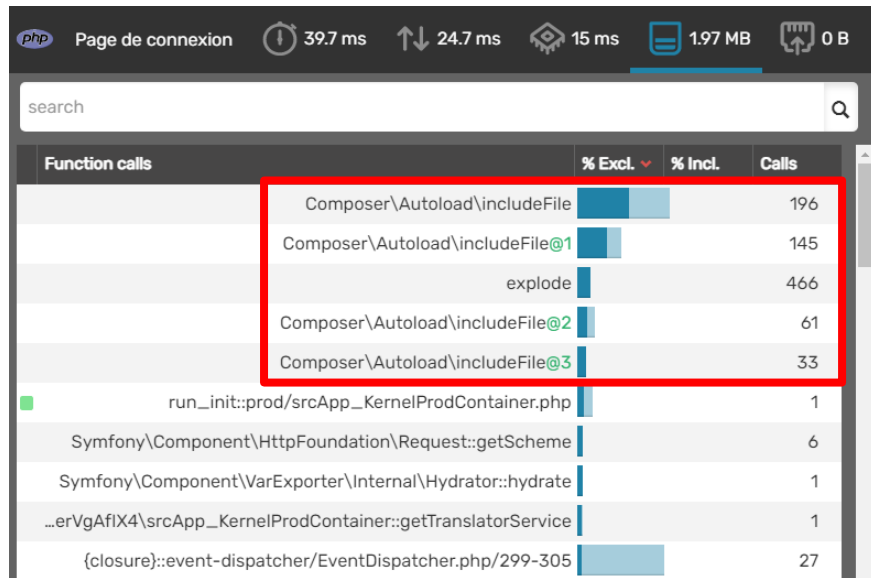
When a user is not connected, he is automatically redirected to the login page on “/login”.



The duration to generate the page is less than 60ms, which is a good score.

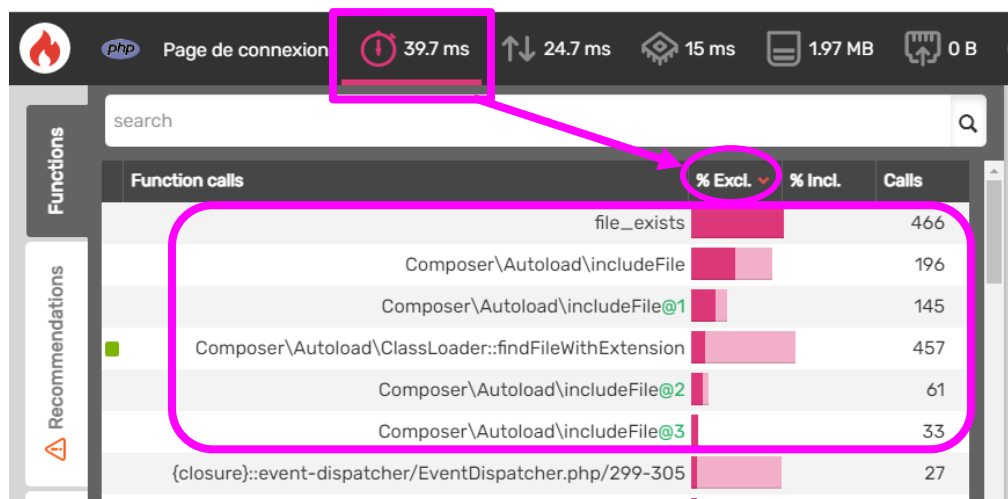
And the used memory is close to 2MB.

Let's look at the most expensive functions in memory:

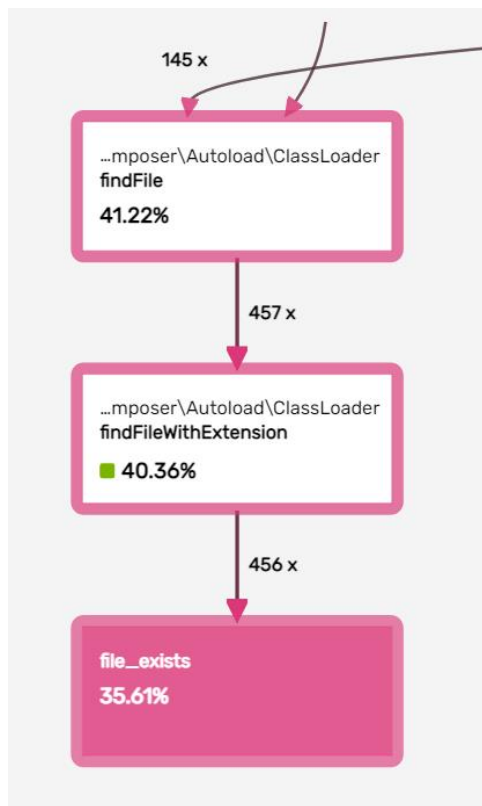


These are the function of the Composer autoloader (Composer\Autoload\ClassLoader). Indeed, the Composer autoloader used during the application development is optimized to search for new and modified classes. And that means picking up query all classes of the application with each query.

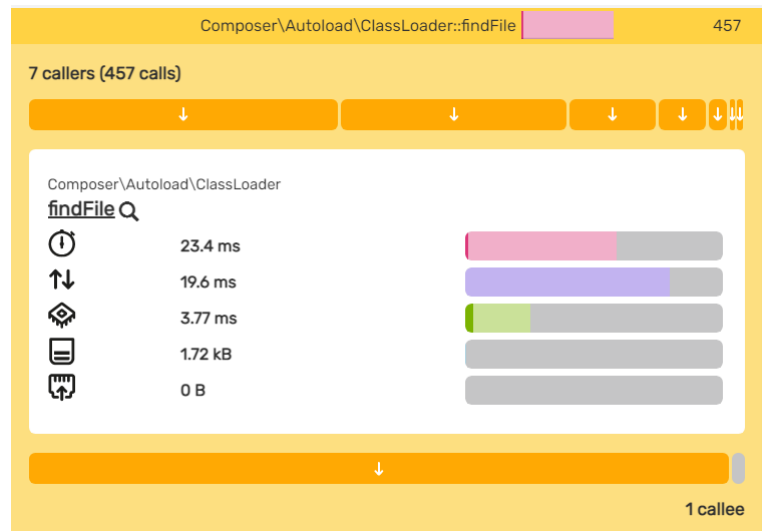
We can see the same prevalence of these functions called by the Composer autoloader on the **duration of access to the files system (I/O)**:



The methods of the Composer autoloader (Composer\Autoload\ClassLoader) are actually called a very large number of times:



The **file\_exists()** PHP function is called 456 times by the **findFileWithExtension()** method of the **Composer\Autoload\ClassLoader** class, which itself is called 457 times by the **findFile()** method of the same class, which itself called 457 times by the **loadClass()** method of the same class.



The **includeFile()** method of **Composer\Autoload\ClassLoader** is called  $196+145+61+33+13 = 448$  times by the **loadClass()** method of the same class:

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

And the **loadClass()** method of **Composer\Autoload\ClassLoader** is called  $197+145+5+61+33+14 = 455$  times by the **spl\_autoload\_call()** function:

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

In the last section « **TECHNICAL DEBT → PERFORMANCE CHECKLISTS** » of this document, we will detail the procedure for optimizing the Composer autoloader. And remember that this optimization only make sense in production environment.

Also Blackfire notes two other recommendations related to the production environment:

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 »
- ▶ « PHP Preloading should be configured »

These two points are also dealt in the section « TECHNICAL DEBT → PERFORMANCE CHECKLISTS ».

## PAGE D'ACCUEIL

Once the user is connected, the home page is a little slower to display than the login page, because there is the duration to process the user session.

php Accueil - To Do List 43.7 ms 25.3 ms 18.5 ms 2.65 MB 1.14 kB 0 µs / 0 rq 426 µs / 1 rq

Besides, we can note there is only one query to the database, and it's the one recovering the data of the connected user.

SQL Queries: 426 µs / 1 rq

DB Connection: 17.2 ms

Calls	Time	SQL
1x	426 µs	select ... from user t0 where t0.id = ?

There are no other query to the database, which is a good thing to minimize the duration to display the home page, which must be as fast as possible (for the user comfort).

## PAGE OF THE LIST OF ALL TASKS

For the page that list all tasks, Blackfire notes too many queries to the database and too many creations of entities.

The image shows the Blackfire toolbar and the Recommendations panel for the page 'Liste de toutes les tâches...'. The toolbar displays various performance metrics: 90.1 ms (red), 50.3 ms (green), 39.9 ms (green), 3.21 MB (green), 27 kB (green), 0 µs / 0 rq (green), and 4.86 ms / 24 rq (red). The Recommendations panel lists four suggestions, with three highlighted by red boxes:

- Less ORM entities should be created**  
metrics.entities.created.count 74 <= 50
- 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 536 <= 50
- You should execute less SQL queries**  
metrics.sql.queries.count 24 <= 10

### LESS ORM ENTITIES SHOULD BE CREATED

The image shows the Blackfire metrics sidebar for the page 'Liste de toutes les tâches...'. It displays the following metrics:

- 6.76 ms / 24 rq (Database)
- SQL Queries 6.76 ms / 24 rq
- DB Connection 20.5 ms
- Loaded ORM Entities 74** (highlighted with a red box)
- [View details](#)

**74** are created (while there are 51 tasks to list and 4 users among the tasks authors)


When using an *Object Relation Mapper* (ORM) to manage data in a relational database, an object named « Entity » is created for each entry extracted from the queries to the database. This usually includes the entries of database relationships also known as associations (Many To One, One To Many, One To One and Many To Many relationships).

In some cases, a few SQL queries can drive to the creation of a lot of entities. One typical example is the loading of « Article » objects, triggering the loading of many associated « Comment » objects.

In our app, the User and Task entities are linked by a One to Many relationship (One user can be the author of several tasks, while one task is linked to a single author corresponding to a single user)

When viewing the all tasks list, the `templates/task/list.html.twig` template receives an array of Task entities from the controller, corresponding to the creation of 51 entities. And for each task, we call the User entity associated by the One To Many relationship to get the value of its « username » property, to define the task's author.

```
# 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>
# ...
```

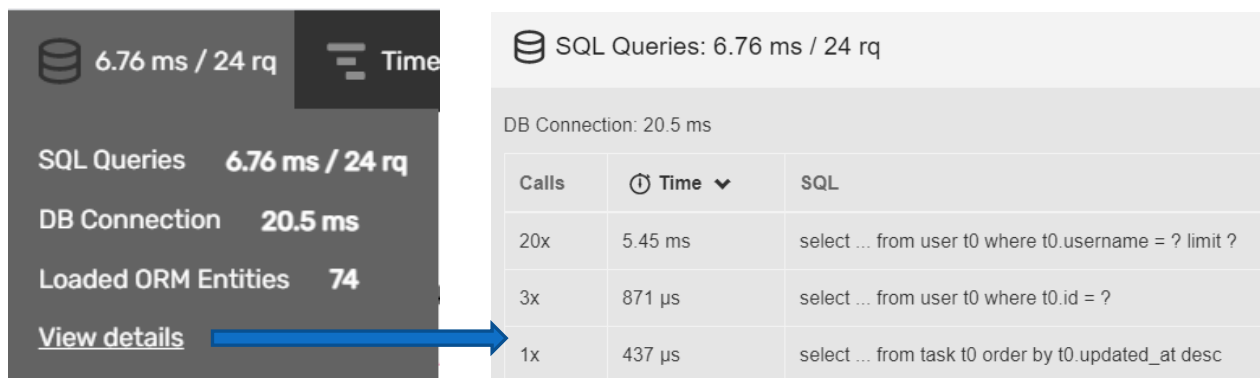


**One solution** to the creation of too many entities can be to collect an array of arrays instead of an array of entities. We can do it using the JOIN cause in the DQL query contractor of a custom Repository method that will collect not only the list of all tasks, but also the author name of each task. This custom method is then called in the `listAll()` method of the [TaskController controller](#) (instead of the generic `findBy()`)

This solution is explained in the section « TECHNICAL DEBT → REDUCE THE NUMBER OF CREATED ORM ENTITIES AND SQL QUERIES »

## YOU SHOULD EXECUTE LESS SQL QUERIES

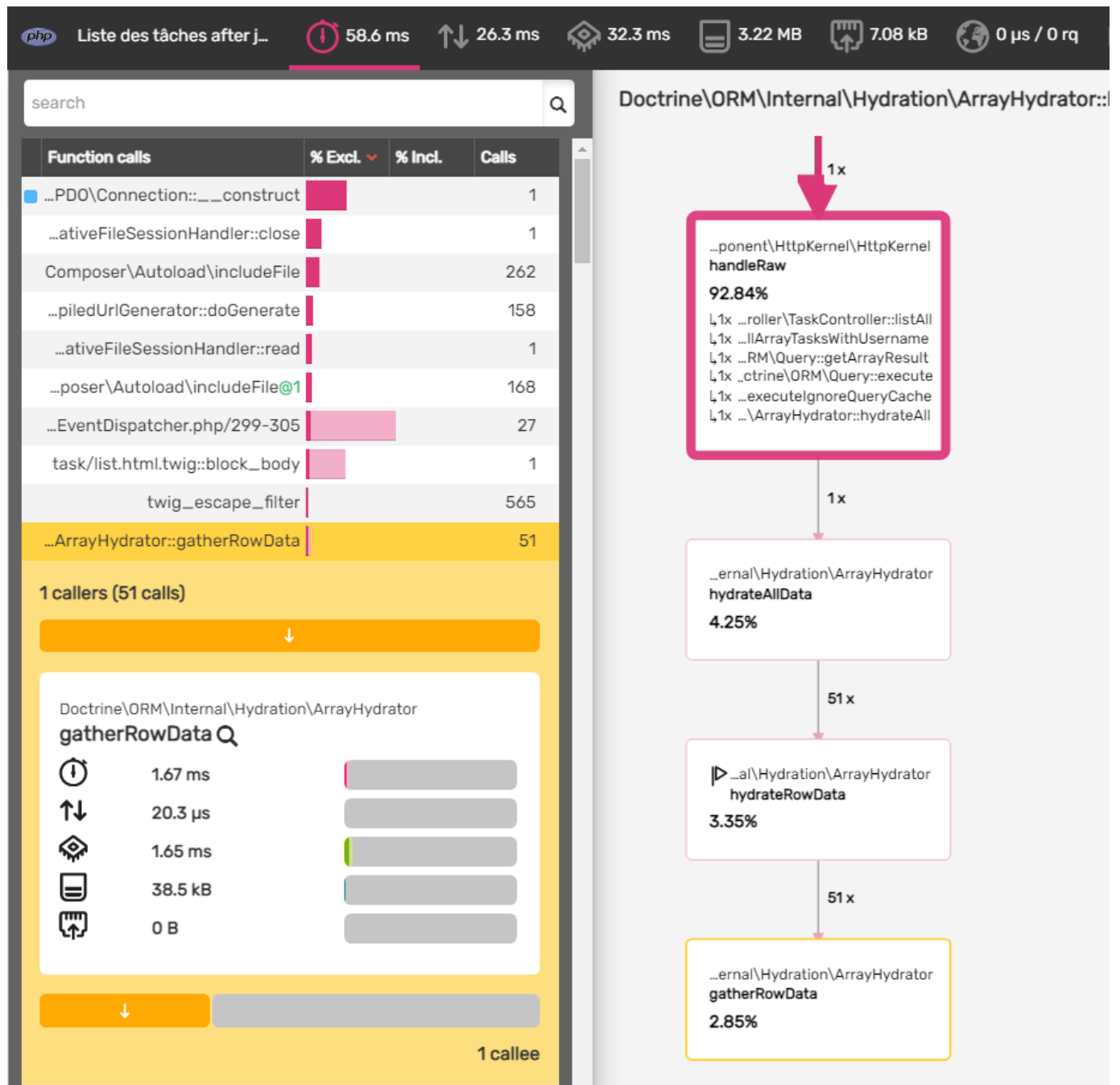
In fact, by clicking on « View details » on the metric tab which give the number of SQL queries, we can see that 23 queries are executed to get users (while there are only 5 users in the database).



A solution to this recommendation is explained in the section « TECHNICAL DEBT → REDUCE THE NUMBER OF CREATED ORM ENTITIES AND SQL QUERIES »

## PAGINATION

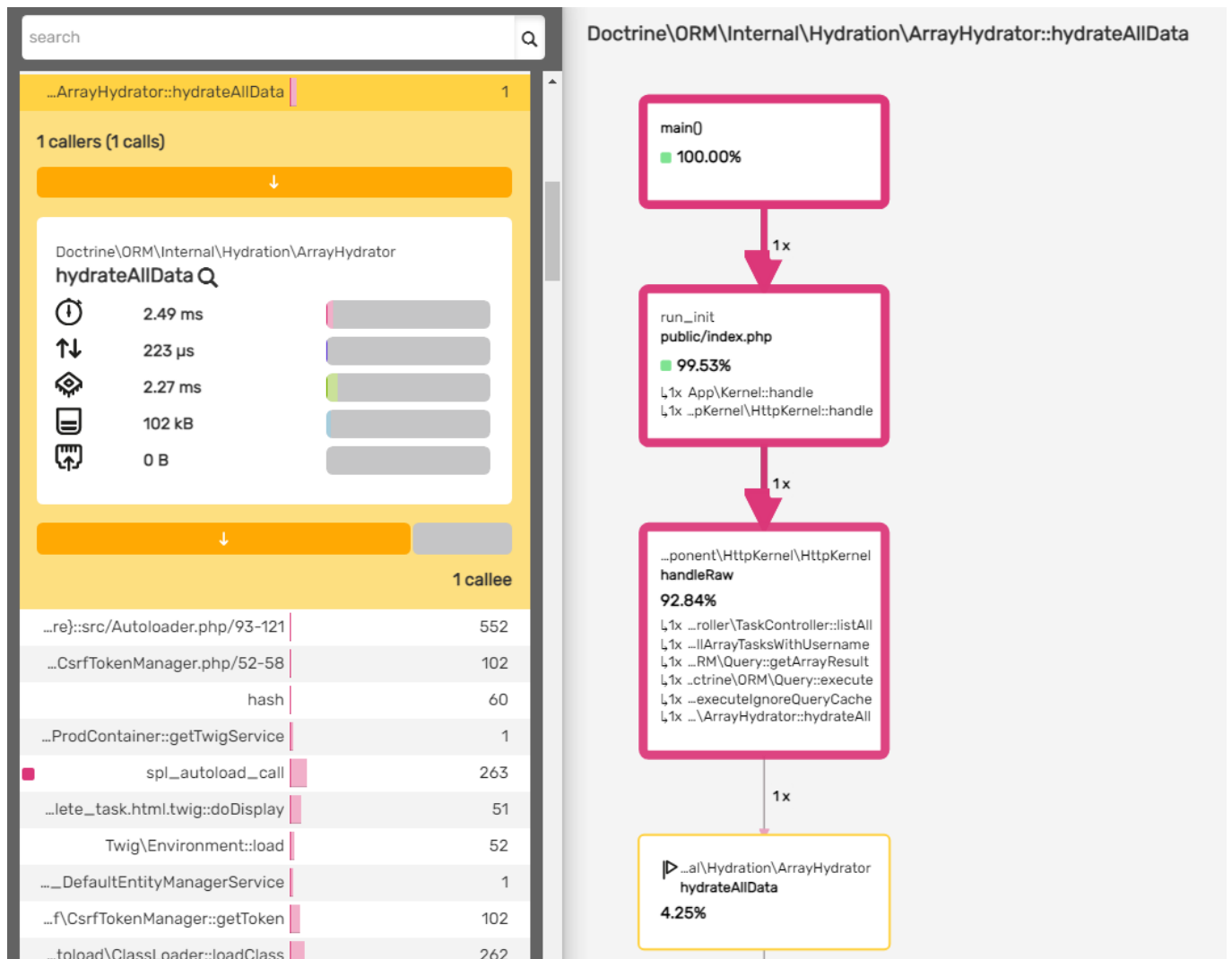
Let's look again the Blackfire analyze of the page that lists all tasks.



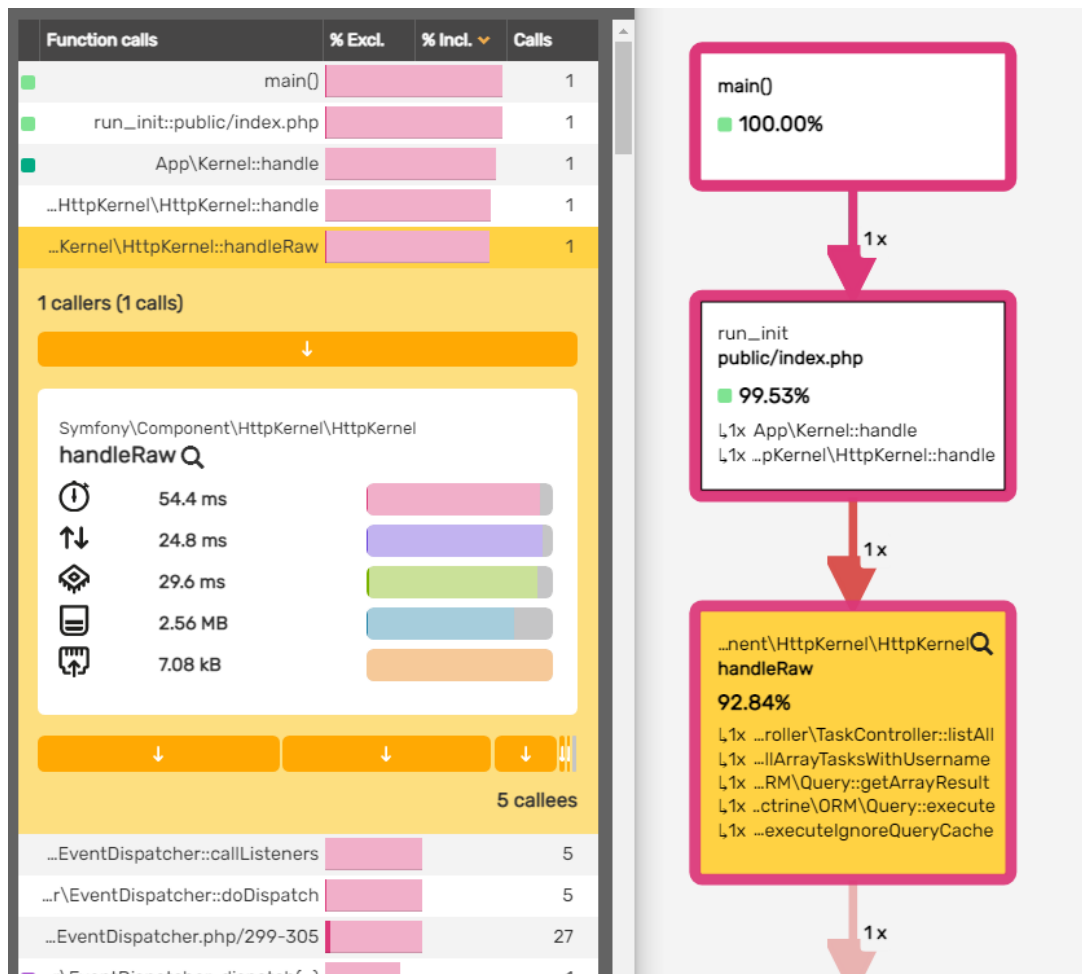
At the 10th position, on the list of sorted function according to the impact they have on the performance by their exclusive cost, we find the **gatherRowData()** method of the **Doctrine\ORM\Internal\Hydration\ArrayHydrator** class. This method is called **51 times** by the **hydrateRowData()** method of the same class, which is also called **51 times** by the **hydrateAllData()** method of the same class.

« 51 » is the total number of tasks available in the database, and that the application goes to search to view the list of all tasks.



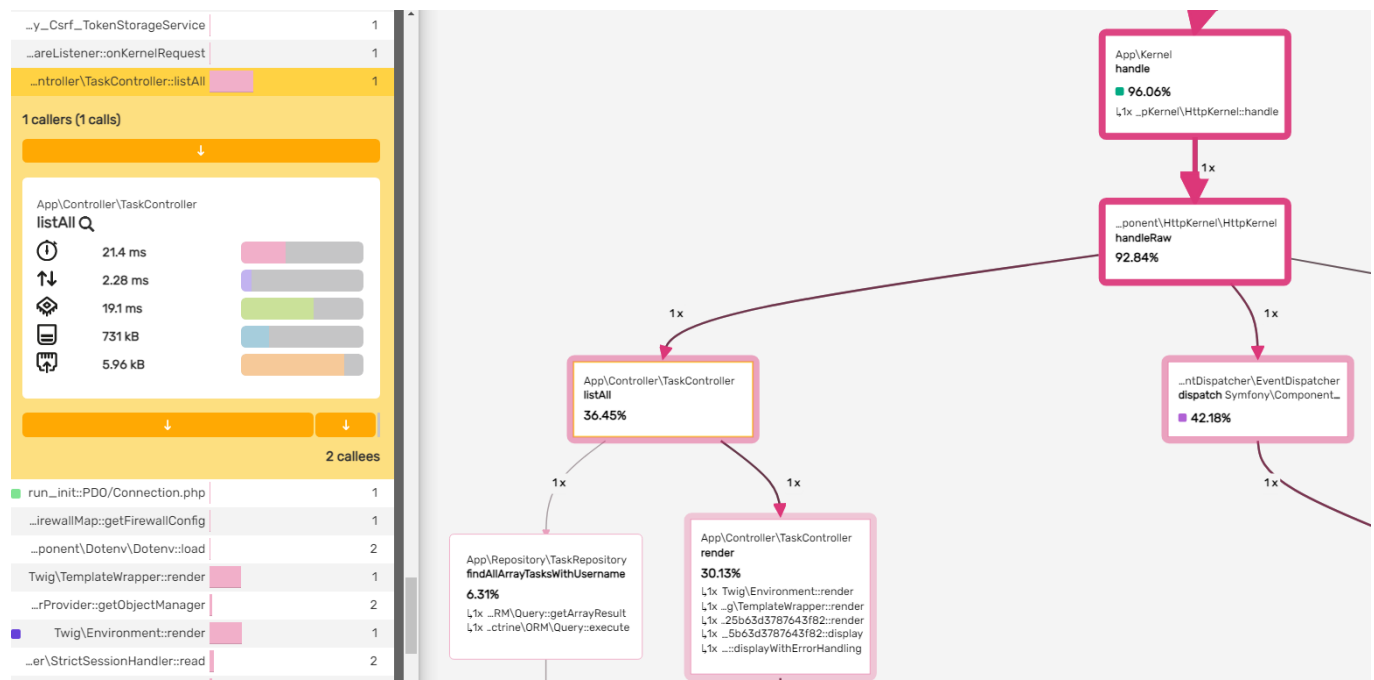


The **hydrateAllData()** method, of the `Doctrine\ORM\Internal\Hydration\ArrayHydrator` class, is called by the **handleRaw()** method of the `Symfony\Component\HttpKernel\HttpKernel` class whose **92,84%** inclusive cost is very high.

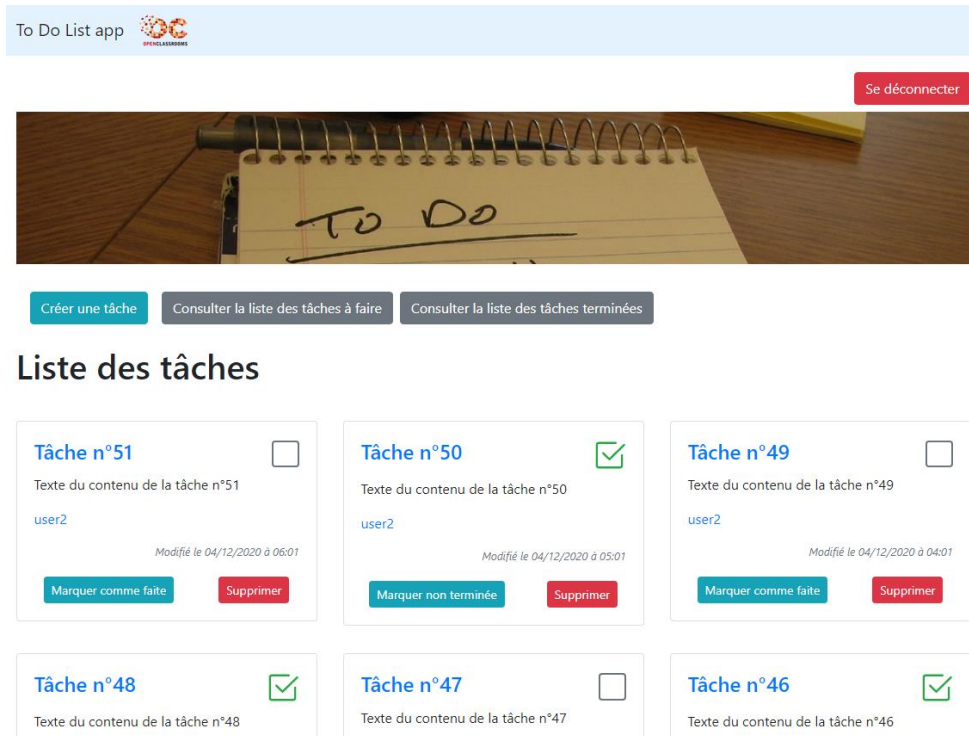


Besides, the `HttpKernel handleRaw()` method is in 5<sup>th</sup> position on sorted functions list according to the impact they have on the performance by their inclusive cost.

This makes it clear how the number of tasks to get from the database will affect the performance of the page that lists all tasks.



The HTTP request with the URI « /tasks » URI execute the listAll() method of the App\Controller\TaskController controller which will pick up ALL tasks, while a maximum of 6 tasks are initially visible on the page.



To improve the performance of the page that list all tasks, we should get fewer tasks. For example, we can get only the first 6 tasks of the list, and then display the following tasks at user's request, such as:

- ✓ By clicking on a button to display n next tasks
- ✓ By scrolling down the page to trigger the display of the next n tasks.
- ✓ ...

It remains to decide of the user's behavior which will trigger an Ajax request that get the following tasks to display on the page.

## TECHNICAL DEBT

### REDUCE THE NUMBER OF CREATED ORM ENTITIES AND SQL QUERIE

#### USE JOIN CLAUSE TO ADD ADDITIONAL FIELDS TO THE QUERY

Here is a proposition of code optimization to reduce the number of created entities, using **JOIN** clause in the query constructor of a custom method, for example named **findAllArrayTasksWithUsername()**, which gets the tasks list.

```
// 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()
        ->getResult();
}
```

```
};
```

Then this method is called in the `App\Controller\TaskController` controller (instead of the `findBy()` method of the Entity Repository) to recover the tasks list as, not an array of entities, but of an array with two items : the first contains the array of the properties of the task, and the second contains the name of the corresponding author.

```
/**
 * 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(),
        ]
    );
}
```

So this controller send to the template an array of arrays, where each array contains an item 0 whose value is an array of the properties of the task, and an item « username » whose associated value is the name of the author of the task:

```
^ 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 [▶]
```

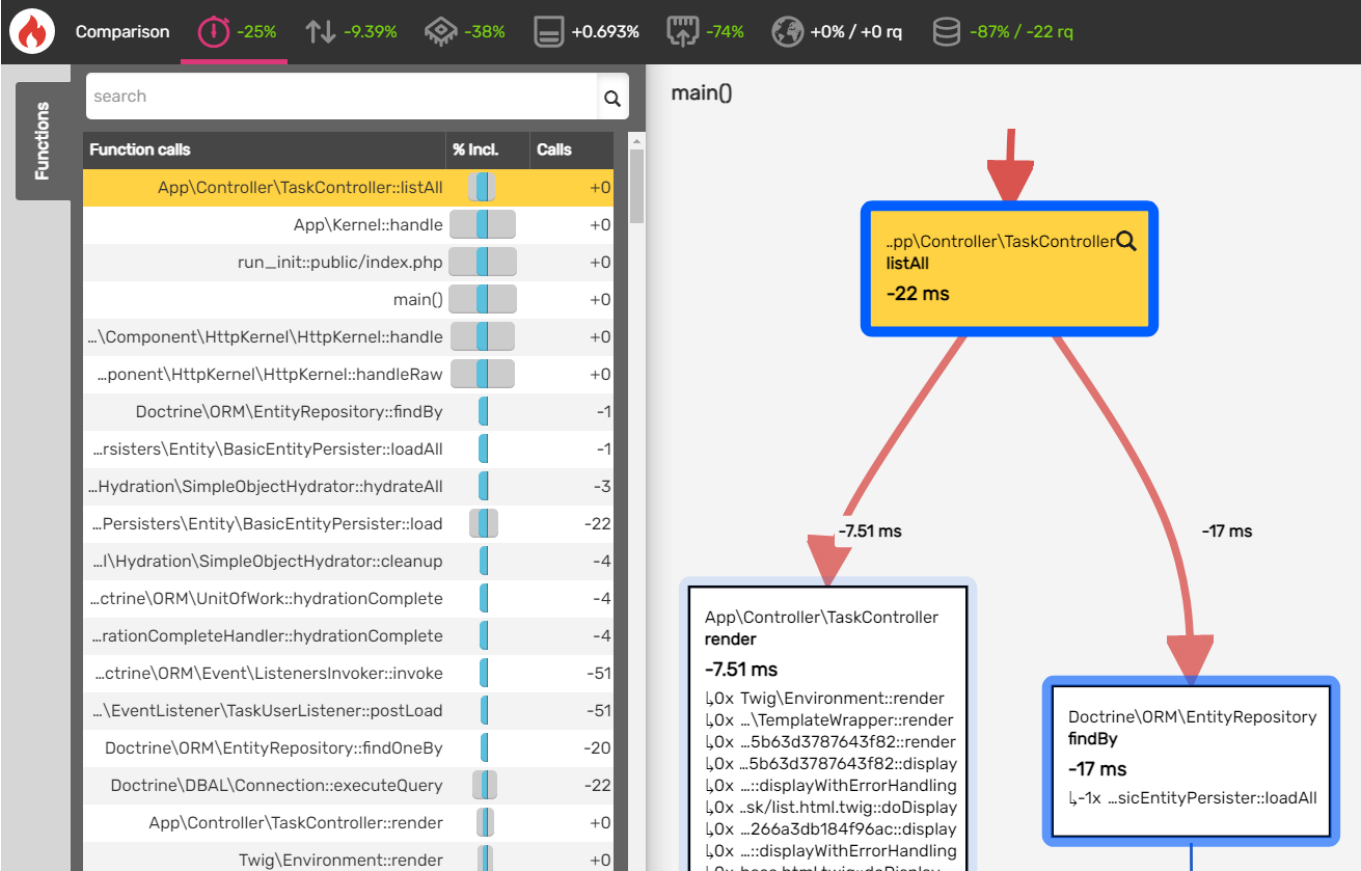
Then the `template/task/list.html.twig` template that display the tasks list is modified consequently:

```
{# 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>
{% endfor %}
{# ... #}
```

**\*\* BUT** since there are no created Task entities anymore, the App\EventListener\TaskUserListener EventListener is no longer called to associate orphan tasks with the « Anonymous » user of the database. We then just replace the null values of the key « username » by the author name « Anonymous ».

This optimization allows us to not anymore create entities, we go from 74 entities to zero ! Furthermore, we go from 24 SQL request to only 2, with a 25% decrease of the page display time.

So in the end, this optimization allows to act on the two recommendations of Blackfire which had note too many SQL queries and too many entity creations.



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

## RESOURCES LIST PAGING

To improve the app performance, it is necessary to paginate the resource lists (tasks and users).

When a page lists resources (tasks or users), the HTTP request calls a controller which calls a Repository's method of the entity corresponding to the resource type to get the data.

For example, the page from the « /users » URI which lists users calls the `list()` method of the `App\Controller\UserController` controller:

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

We can see that the `findAll()` method of the `App\Repository\UserRepository` repository is called to get all users.

The resource pagination then involves creating a method in the Repository of the concerned entity to get a given number of resources (named « limit ») and from an « offset ».

For example, we can create the method named `getPaginatedUsers(int $limit = 6, int $offset = 0)` which get `$limit` users from which one set by the `$offset` value.

That gives in the controller:

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

The default values of `$limit` and `$offset` (here 6 and 0) can be defined in a configuration file, while they can also be specified in the request.

This performance point asking to paginate resource lists concerns several pages of the app:

- The page that lists all tasks on the « /tasks » URI
- The page that lists all done tasks on the « /tasks/true » URI
- The page that lists all open tasks on the « /tasks/false » URI
- The page that lists all users on the « /users » URI

## IN PRODUCTION MODE – PERFORMANCE CHECKLIST

The performance checklist of a Symfony application and its production server is available in Symfony's official documentation (<https://symfony.com/doc/4.4/performance.html>).

This list is listed below. And we detail the points that we have previously identified during the Blackfire analyze:

- ✓ **Configuring environment variables in production**
- ✓ **Use the OPcache class preloading**
- ✓ **Optimize Composer autoloader**

---

## SYMFONY APPLICATION CHECKLIST

- ▶ **Install APCu Polyfill if your server uses APC instead of OPcache** – [Symfony's Documentation](#)

---

## PRODUCTION SERVER CHECKLIST

### 1. Configuring environment variables in production

In production, the `.env` files are also parsed and loaded on each request. So the easiest way to define environment variables is by deploying an `.env.local` file to your production server(s) with your production values.

To improve performance, you can optionally run the `dump-env` command (available in [Symfony Flex 1.2](#) or later):

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

After running this command, Symfony will load the `.env.local.php` file to get the environment variables and will not spend time parsing the `.env` files.

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`.

#### NOTE

Update your deployment tools/workflow to run the `dump-env` command **after each deploy** to improve the application performance.

### 2. Dump the service container into a single file – [Symfony's Documentation](#)

### 3. Use the OPcache byte code cache – [Symfony's Documentation](#)

### 4. Use the OPcache class preloading

Starting from PHP 7.4, OPcache can compile and load classes at start-up and make them available to all requests until the server is restarted, improving performance significantly.

During container compilation (e.g. when running the `cache:clear` command), Symfony generates a file called `preload.php` in the `config/` directory with the list of classes to preload.

The only requirement is that you need to set both `container.dumper.inline_factories` and `container.dumper.inline_class_loader` parameters to true. Then, you can configure PHP to use this preload file:

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

#### REMARQUE

The OPcache preloading is not supported on Windows. ([Doc PHP](#))

### 5. Configure OPcache for maximum performances – [Symfony's Documentation](#)

### 6. Don't check PHP files timestamps – [Symfony's Documentation](#)

### 7. Configure the PHP realpath cache – [Documentation Symfony](#)

### 8. Optimize Composer autoloader:

**By default**, the Composer autoloader runs relatively fast. However, due to the way PSR-4 and PSR-0 autoloading rules are set up, it needs to check the filesystem before resolving a classname conclusively. This slows things down quite a bit, but it is convenient in development environments because when you add a new class it can immediately be discovered/used without having to rebuild the autoloader configuration.

However, **in production**, we want things to happen as quickly as possible. Also, you can build the configuration every time you deploy and the new classes don't appear randomly between deployments. That's why you can optimize Composer's automatic charger to scan the entire app once and create an optimized "class map" of the app.

To do this, Composer offers some strategies to optimize its autoloader in production mode (see [Composer's documentation](#) to optimize its autoloader).

For a Symfony application using Opcache, we will activate levels 1 and 2.A of available strategies (detailed in The [Composer's documentation](#)).

**Once the application is complete, and for production mode** (where PHP files should never change, unless a new application version is deployed), we will create an optimized "class map" which is a large array of locations in all classes, and it is stored in **vendor/composer/autoload\_classmap.php**.

Execute this command to generate the new class map (and make it part of your deployment process too):

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

- The **--no-dev** option excludes the classes that are only needed in the development environment (i.e. require-dev dependencies and autoload-dev rules);
- The **--classmap-authoritative** option creates a class map for PSR-0 and PSR-4 compatible classes used in your application and prevents Composer from scanning the file system for classes that are not found in the class map.

#### REMARQUE :

**The benefit of optimizing the autoloader is very important.** This is to halve the generation time of the pages, as shown by the comparison of two Blackfire profiles (of the login page), before and after the optimization of the Autoloader of Composer:

