DOCUMENTATION TECHNIQUE DE CONTRIBUTION

Étape 1 : Installation des composants nécessaires pour effectuer des tests unitaires et fonctionnels

Installation de phpunit, Xdebug et configuration de phpStorm

Étape 2 : Création d'une base de donnée pour env.test

```
DATABASE_URL=mysql://root:root@127.0.0.1:8889/test_Projet8todolist
```

Étape 3 : Création de test unitaires avec phpunit

Étape 4 : Création de test fonctionnels avec phpunit

Étape 5 : Compte rendu des tests avec le code-coverage :

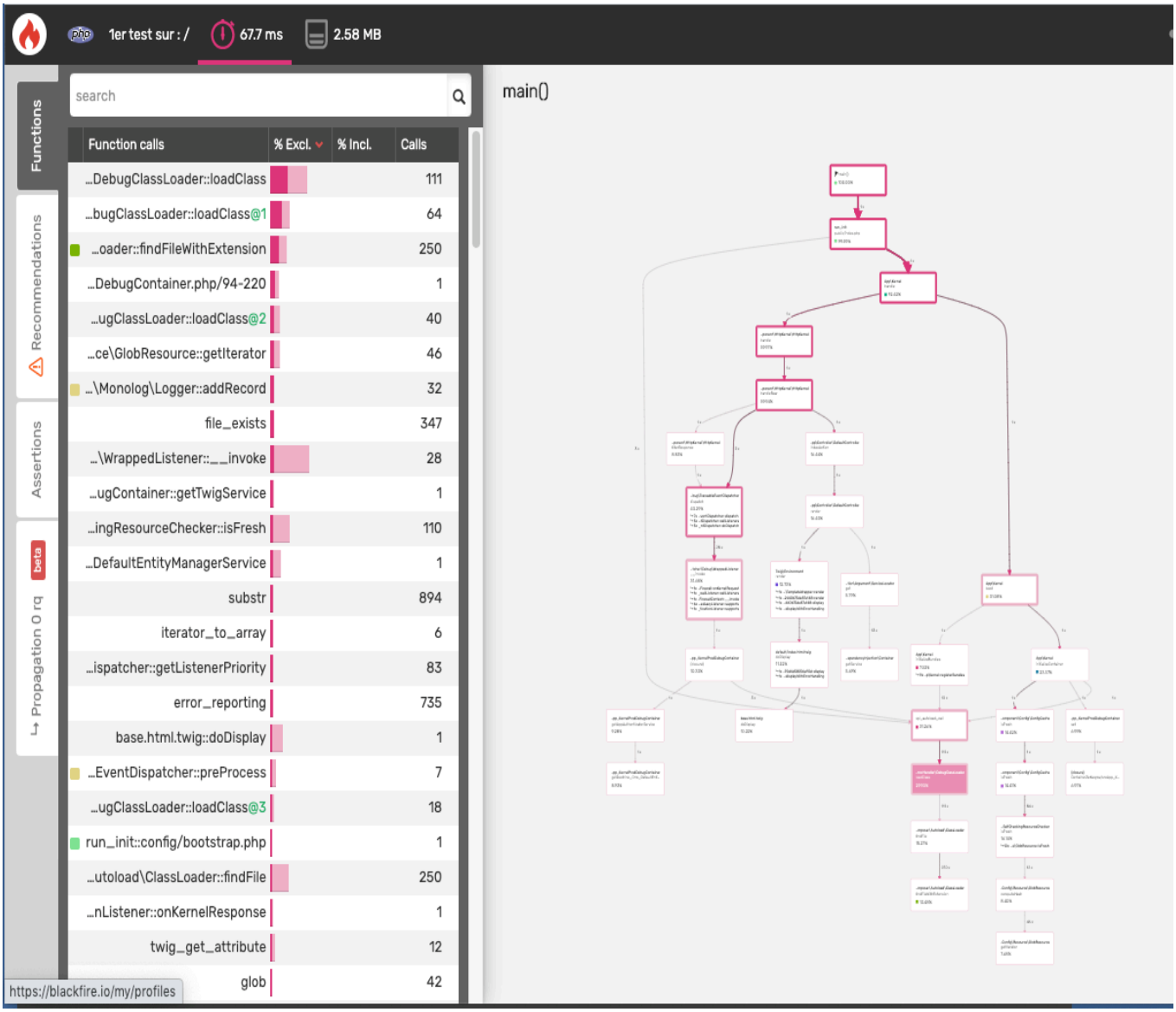
Le fichier se trouve dans le dossier :
SoutenanceP8/projet8-TodoList/public/test-coverage/index.html

Étape 6 : Mise en place des tests d'optimisation avec blackfire

On modifie le .env :
APP_ENV=prod
APP_DEBUG=1
Puis on mets Xdebug en mode non fonctionnel, car ce dernier peut perturber les tests avec blackfire.

Utilisons le Compagnon de blackfire avec GoogleChrome et rendez nous sur http://localhost:8000/ pour commencer les tests.

a)Test de performance sur la page d'accueil :



On peut voir que la fonction loadClass provenant de la classe : Composer\Autoload\ClassLoader, est appelée 111 fois.

Cette fonction est externe à Symfony et provient de composer, trouvons un moyen d'optimiser l'autoloader de Composer.

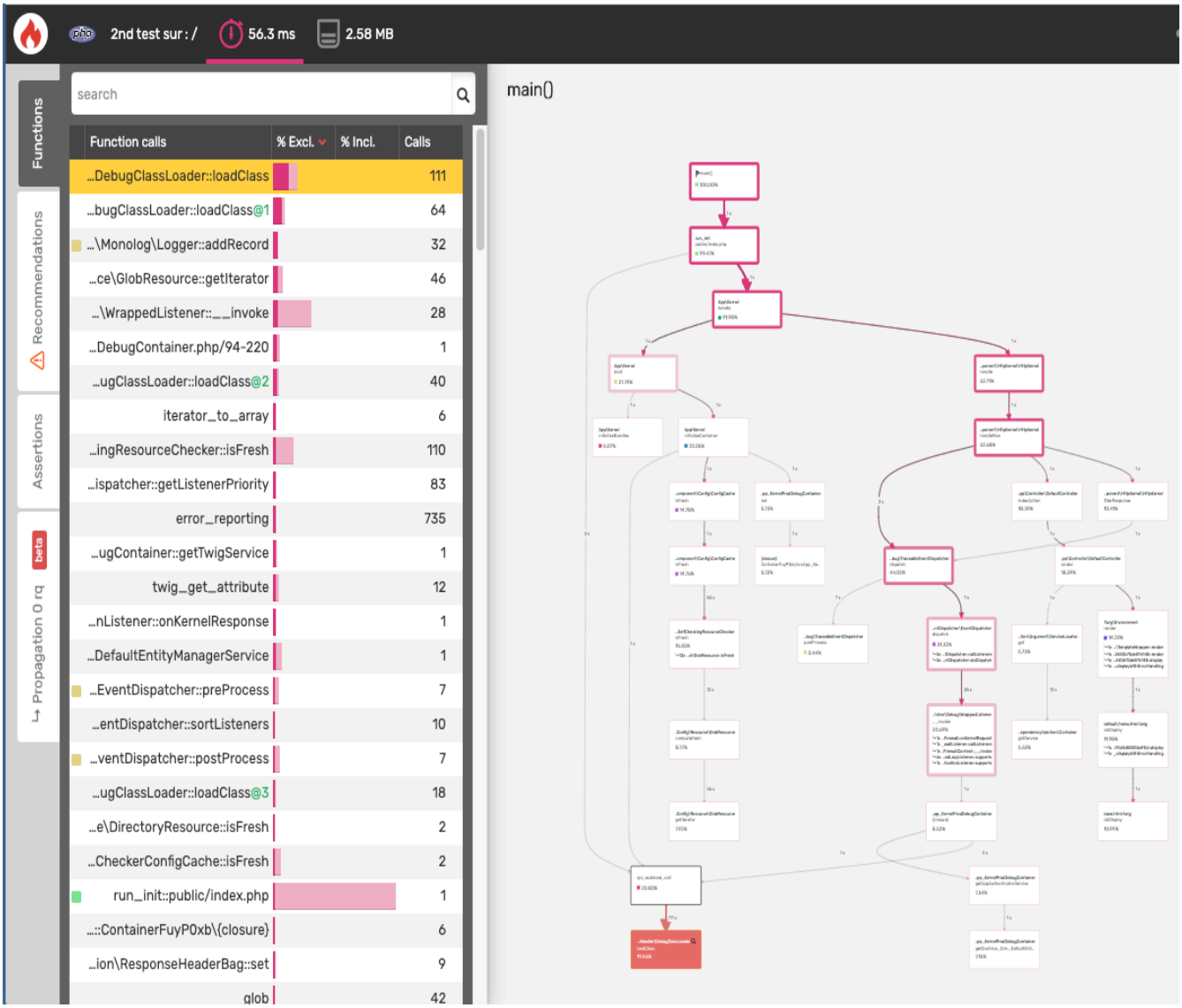
D'après la documentation, il y a plusieurs degrés d'optimisation de l'autoloader de composer.
Commençons par le premier est regardons le résultat avec blackfire.

Optimisation level 1 : Class map generation

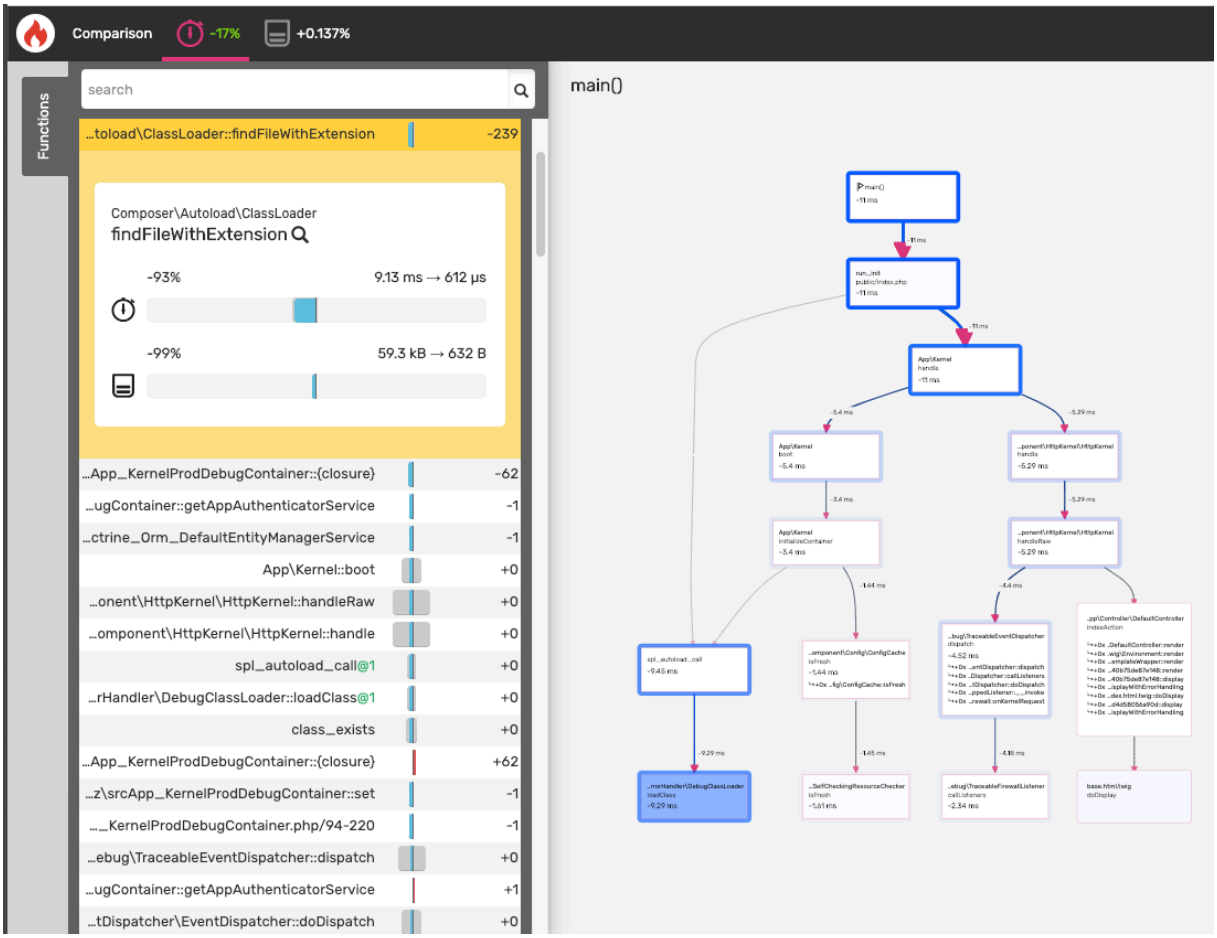
Dans composer.json, mettre la préference *"optimize-autoloader": true* dans la config.
Puis, installer *--optimize-autoloader* avec composer.
Puis faire un `dump-autoload --optimize.`

En quoi cette optimisation va servir ?
=> `La génération Class map, convertit essentiellement les règles PSR-4 / PSR-0 en règles de mappage de classe. Cela rend tout un peu plus rapide car pour les classes connues, la carte de classe retourne instantanément le chemin, et Composer peut garantir que la classe est là, donc aucune vérification du système de fichiers n'est nécessaire.`

b)Second test sur la page d’accueil :



Comparatif entre les deux :



On peut voir une amélioration de 17 % .

Essayons d'améliorer encore plus l'autoloader en configurant une nouvelle optimisation.

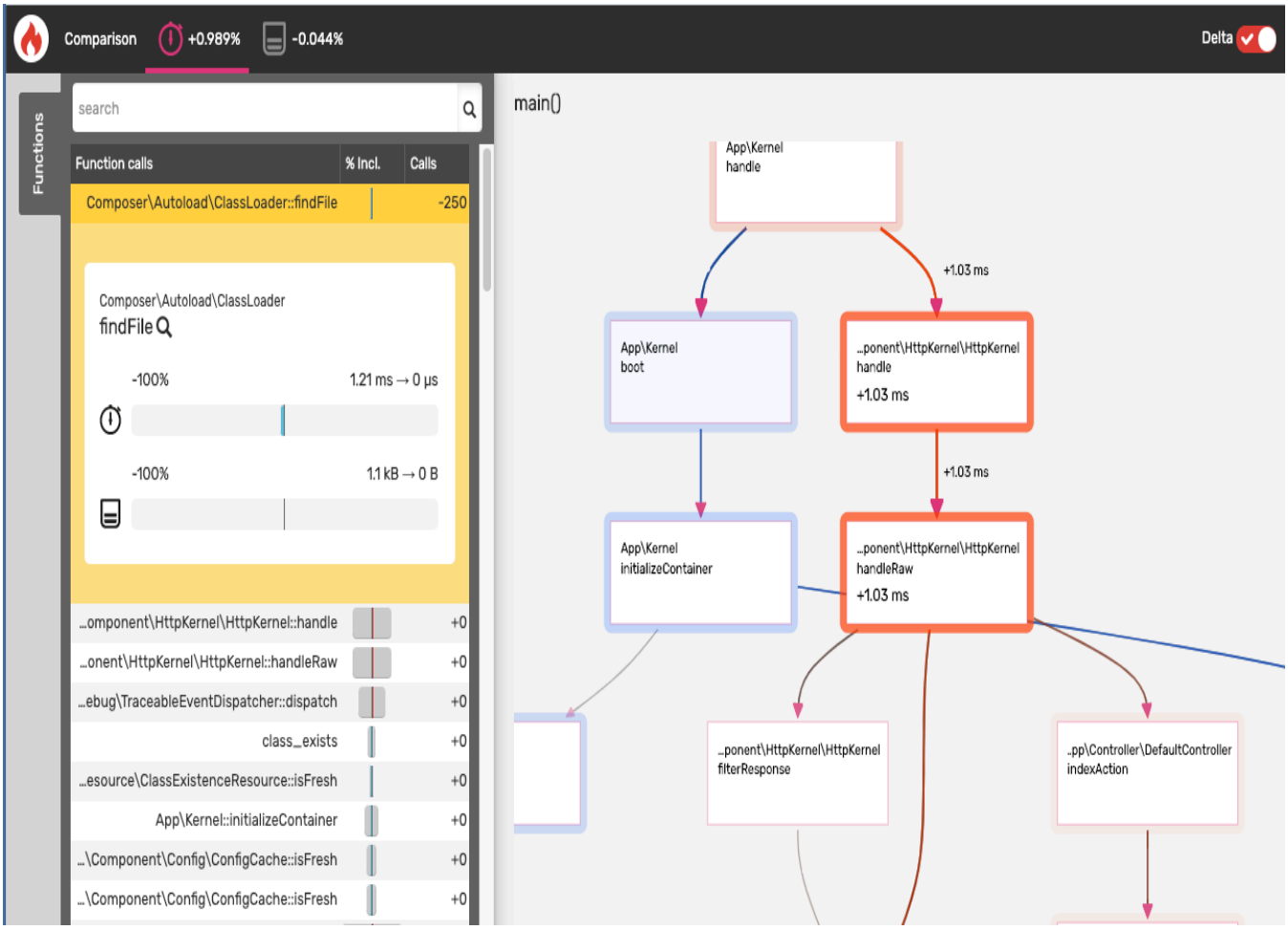
Optimization Level 2/A: Authoritative class maps

Dans composer.json, mettre la préference *"classmap-authoritative": true* dans la config.
Puis, installer *--classmap-authoritative* avec composer.
Puis faire un `dump-autoload` *--classmap-authoritative*

En quoi cette optimisation va servir ?
=> `L'activation active automatiquement les optimisations de mappage de classe de niveau 1.`

`Cette option est très simple, elle dit que si quelque chose n'est pas trouvé dans le classmap, alors il n'existe pas et l'autoload ne devrait pas essayer de regarder le système de fichiers selon les règles PSR-4`

Comparaison entre le deuxième test et le troisième test :



On peut voir une régression de +0.98% du temps de chargement.
Et un gain de mémoire de 0,045 % de mémoire vive consomée.

On voit que les changements ne sont pas significatifs, cependant cette fonctionnalité est optimale.

c)Essayons d'optimiser Composer pour l'environnement Prod de notre application :

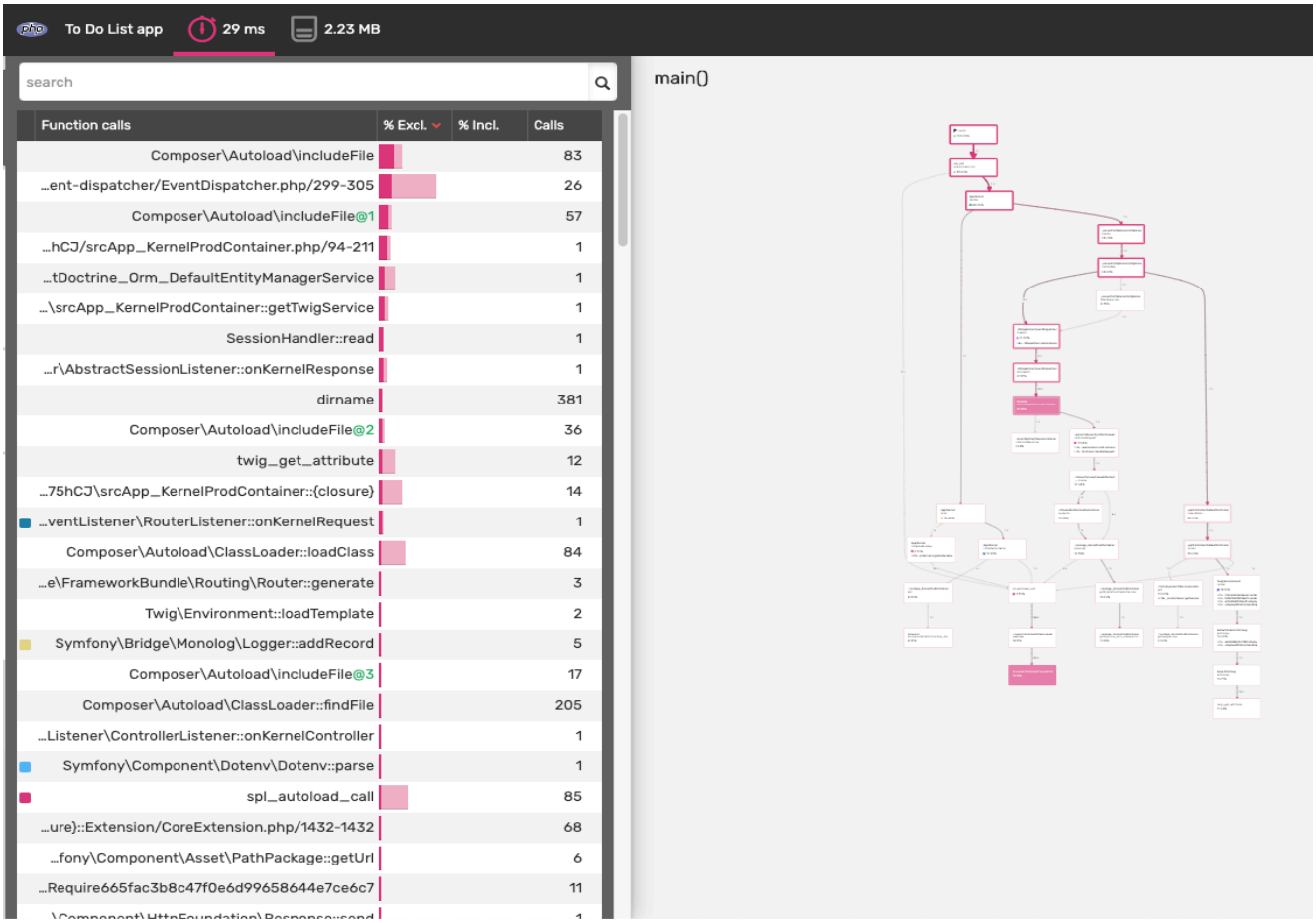
Tapons la commande suivante :

```
composer install --no-dev
```

En quoi cette optimisation va servir ?

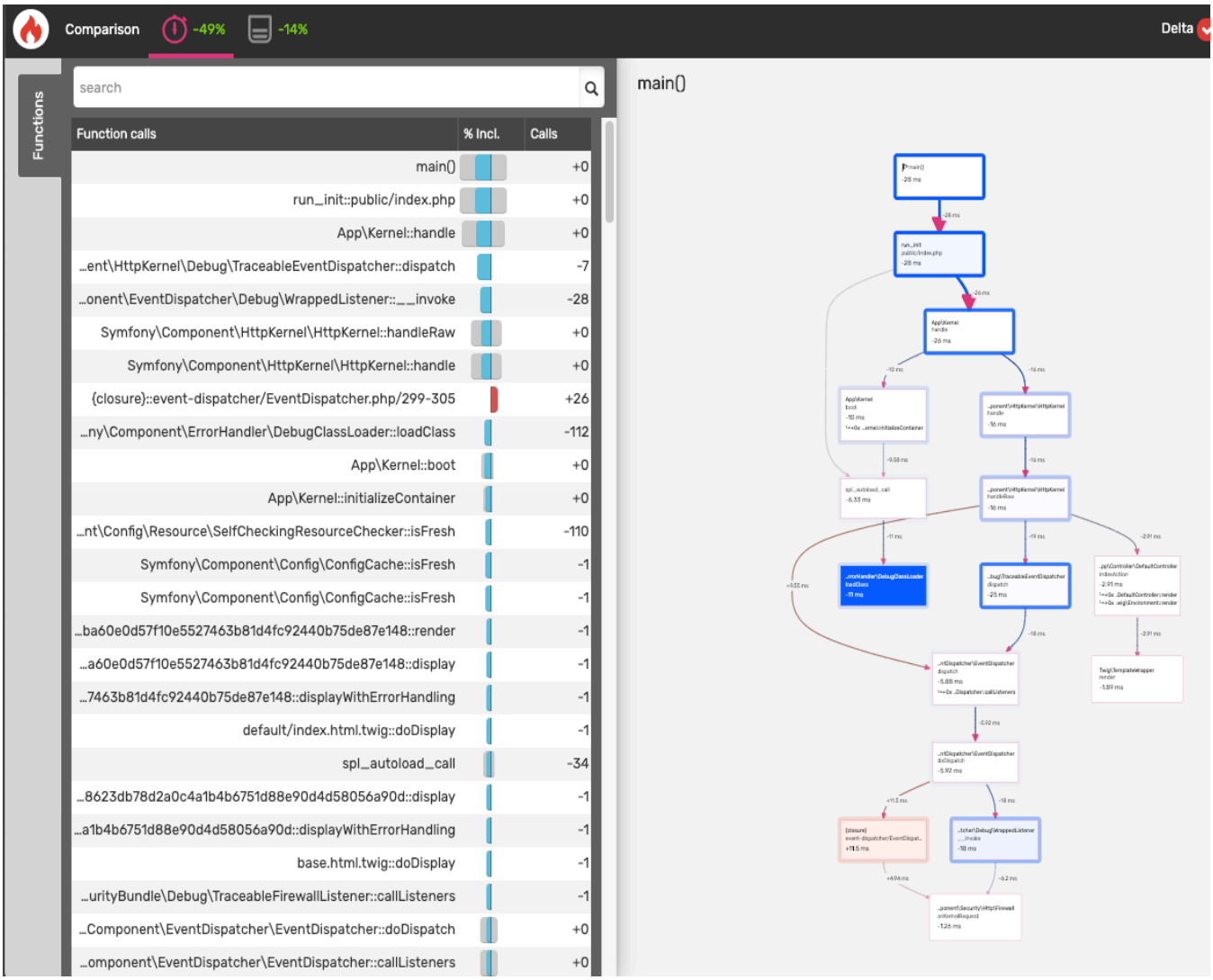
Comme son nom l'indique, cela empêche l'installation de toutes les dépendances de développement, ce qui réduit la taille du dossier fournisseur et la taille du fichier classmap.

Voyons le résultat de cette optimisation :



Waow… sans même faire la comparaison, il est flagrant que le temps de chargement à presque été réduit de moitié.

Faisons la comparaison entre nos deux dernières versions

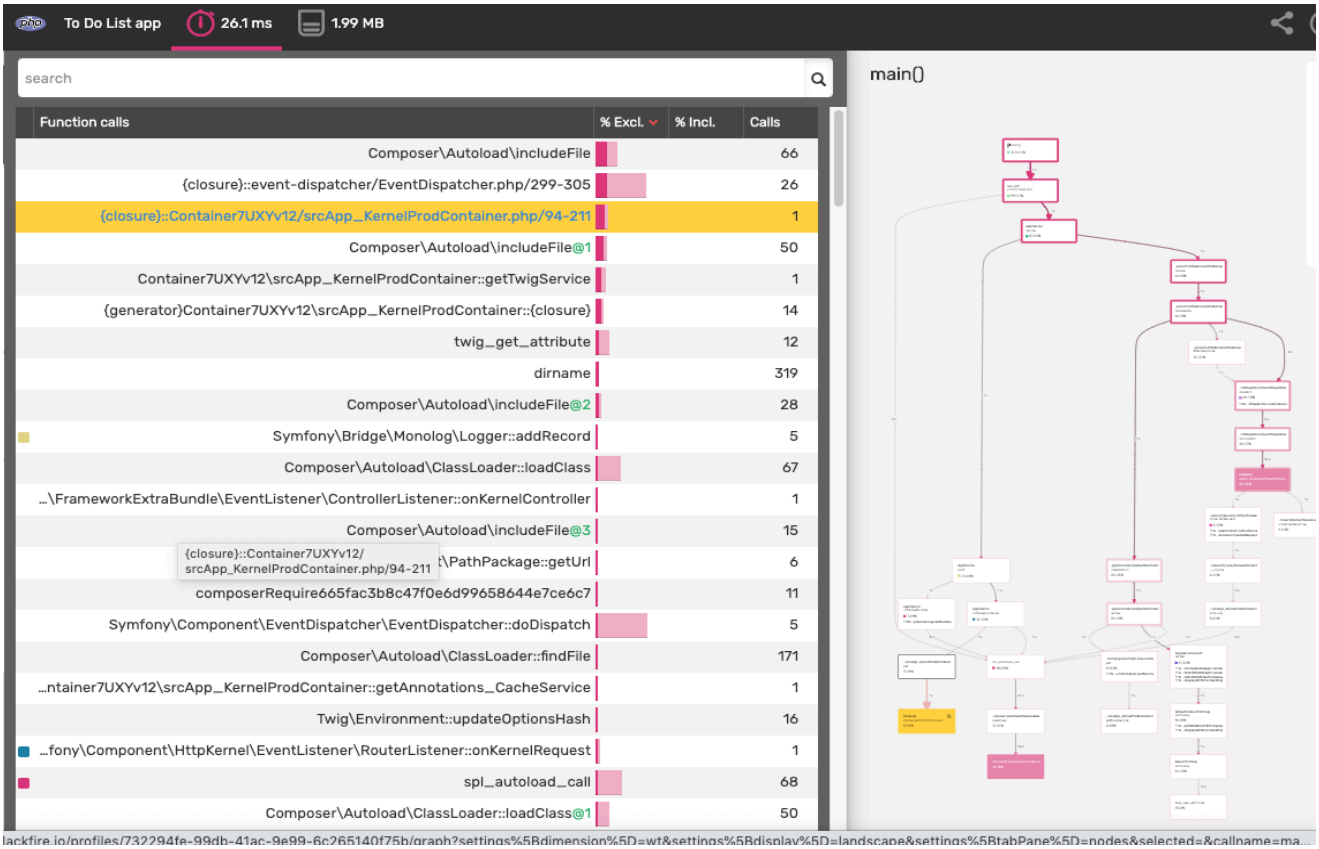


Effectivement, on a gagné 49% de temps de chargement et 14% en mémoire vive consommée.

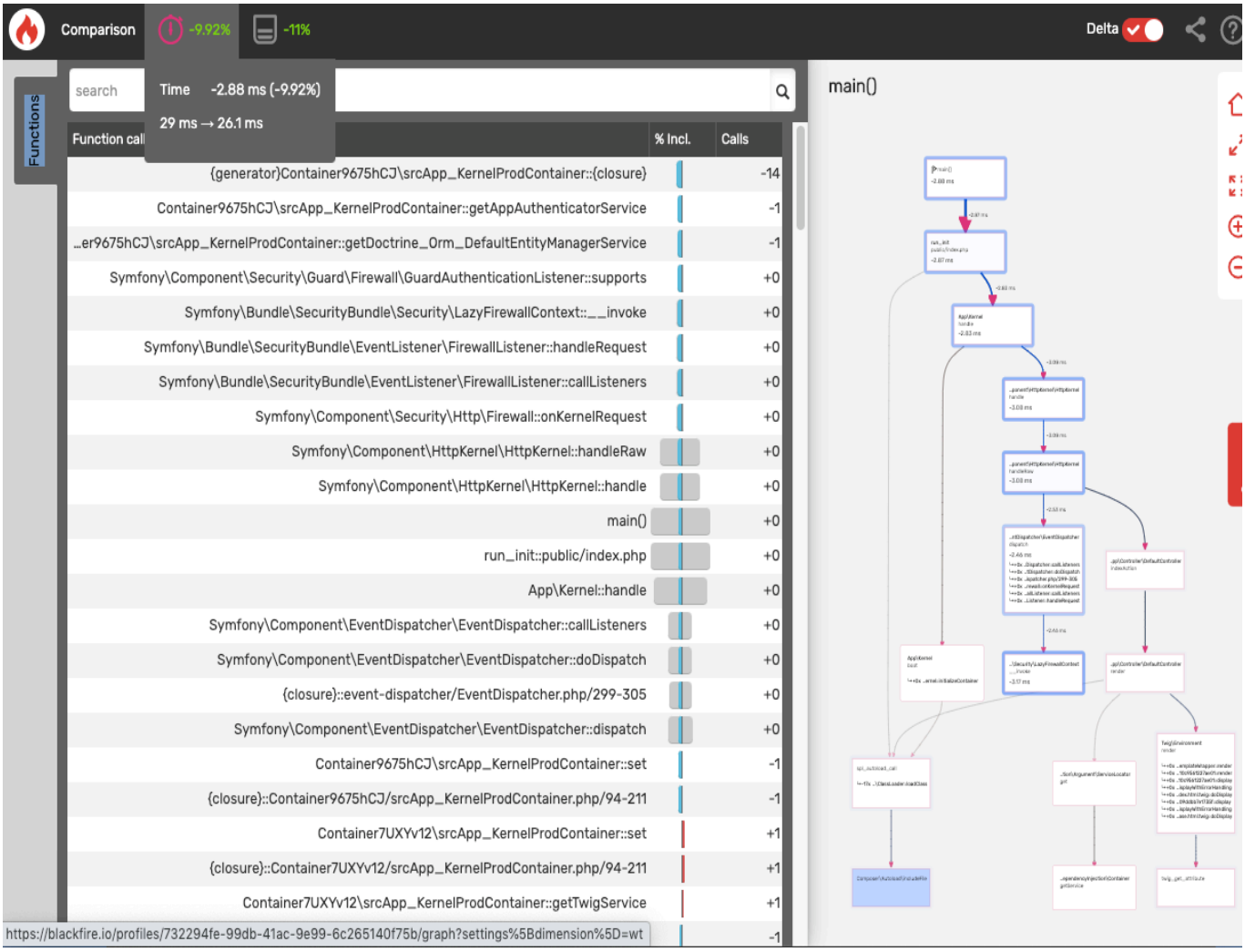
d)Optimisation de twig

On peut voir que twig figure parmis les composants les plus lourd en terme de performance.
Essayons de l'optimiser en utilisant le Lazy service.
Pourquoi cela ?
`L'utilisation de Lazy service vous permet d'injecter des Lazy service dans vos services. Ce qui signifie techniquement que seule une référence à un proxy est injectée dans votre service, à la place une référence au service déjà entièrement amorcé`
Installons avec composer : `symfony/proxy-manager-bridge`.
Puis ajoutons ce code dans les services, dans services.yaml

```
services:
    App\Twig\AppExtension:
        lazy:  true
```



Faisons la comparaison entre nos deux dernières versions :



On peut voir une augmentation de 9,92 % du temps de chargement, ainsi qu'une amélioration de 11% concernant la mémoire vive consommée.

e)Optimisons directement php, en effet la version de php utilisé contient l'extension Opcache, et celle-ci peut être

optimisée afin d'améliorer la performance de notre application :

Je vérifie ma version de php :

```
PHP 7.3.19 (cli) (built: Jun 12 2020 00:08:24)
( NTS )
Copyright (c) 1997-2018 The PHP Group
Zend Engine v3.3.19, Copyright (c) 1998-2018
Zend Technologies
    with Zend OPcache v7.3.19, Copyright (c)
1999-2018, by Zend Technologies
    with blackfire v1.34.3~mac-x64-non_zts73,
https://blackfire.io, by Blackfire
```

OPcache est donc bien installé et je modifie mon php.ini avec la configuration suivante :

[opcache]

```
; enables opcache
opcache.enable=1

; maximum memory that OPcache can use to store
compiled PHP files
opcache.memory_consumption=256

; maximum number of files that can be stored in the
cache
opcache.max_accelerated_files=20000

; don't check for timestamps
opcache.validate_timestamps=0
```

Pourquoi cela ?

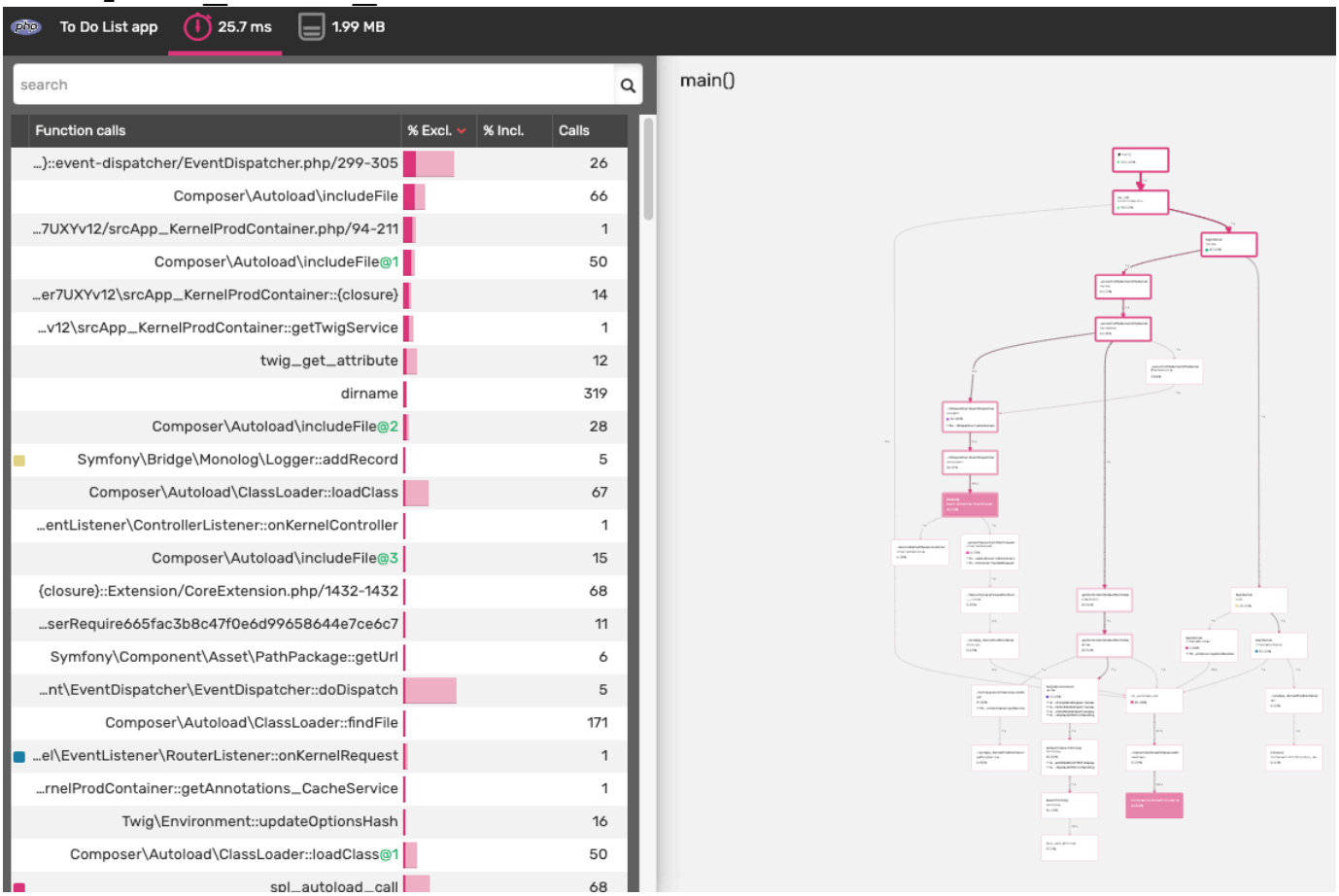
Cela activera non seulement opcache pour notre environnement PHP, mais modifiera également les paramètres pour obtenir les meilleures performances de notre opcache.

Une autre chose que nous pouvons faire est de permettre à PHP de mettre en cache beaucoup plus de chemin relatifs aux mappages.
Cela réduit considérablement le nombre de fois où PHP doit rechercher un mappage pour un chemin relatif vers un chemin absolu, car presque tous les mappages dans une application symfony typique peuvent désormais être mis en cache, améliorant les performances globales de notre site Web.

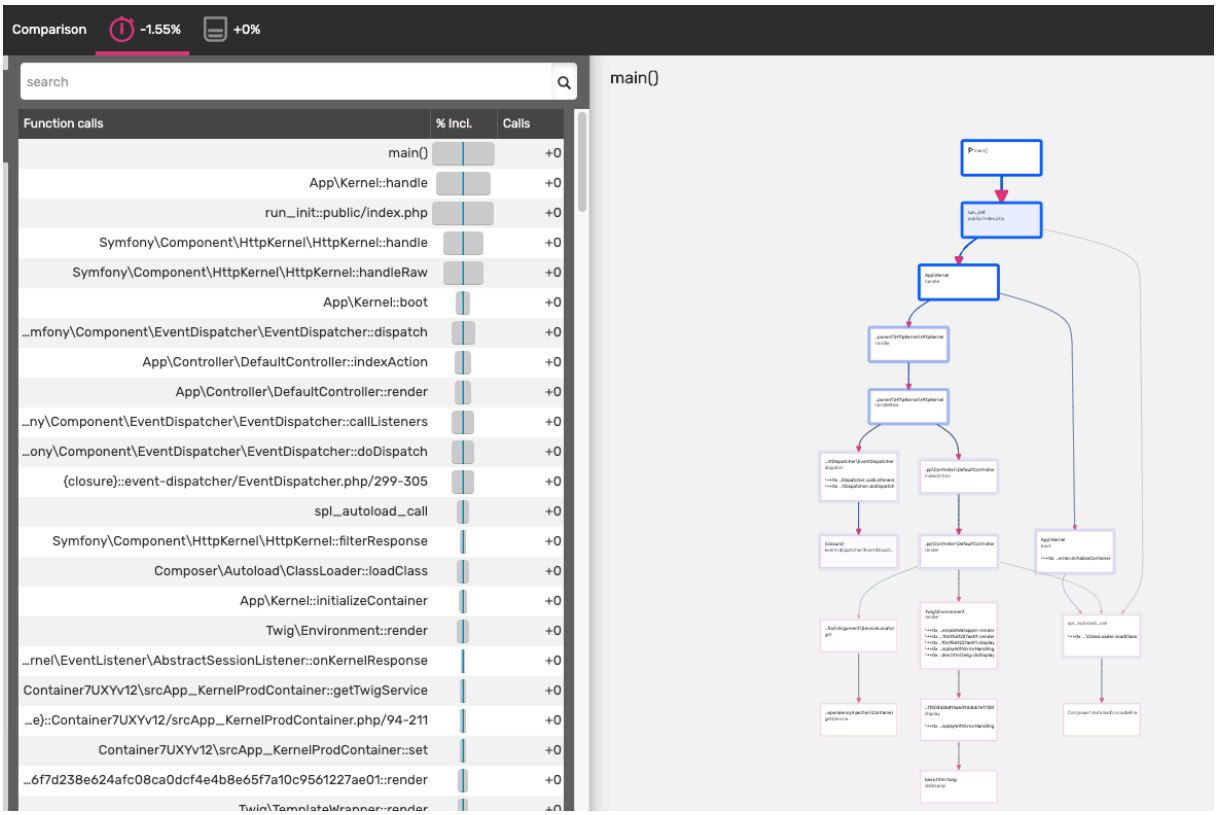
Configurant encore ceci dans notre php.ini :

```
; maximum memory allocated to store the results
realpath_cache_size=4096K

; save the results for 1 hour (3600 seconds)
realpath_cache_ttl=3600
```



Comparons nos deux dernières versions :



On peut constater une amélioration de 1,55% de vitesse de chargement.

Étape 7 : Contrôle de la qualité du code

Un contrôle de qualité du code a été effectué sur le site codacy.com

Lien : https://app.codacy.com/manual/Jean-archibald/projet8_documentation/dashboard?bid=18751695