Rendu audit des performances

Groupe: Amin Nairi et Quentin Hermiteau

Les avantages de la performance pour une application

Chaque site internet est référencé sur le moteur de recherche de Google (sauf exceptions ou lorsque cela est explicitement désactivé). Une fois que ce référencement est fait, Google va procéder à un classement de ces sites selon plusieurs critères :

- performance du site
- performance du référencement naturel
- accessibilité du site
- bonnes pratiques
- et optionnellement si l'application est prête pour être déployée en tant que PWA

La performance est donc un des critères essentiel à tous sites internet qui souhaite être référencé plus haut dans le classement des moteurs de recherches. C'est donc un avantage indéniable surtout que contrairement à de la publicité (référencement payant) cela ne nécessite pas d'investir des sommes d'argent régulières pour être classé plus haut. Néanmoins cela nécessite un audit régulier pour rester haut dans ce fameux classement.

Un site internet doit surveiller différent indicateurs clé de performances, que ce soit au niveau des performances pures mais aussi au niveau de l'efficacité du référencement naturel. Ces indicateurs sont mutuellement liés car ils déteignent chacun l'un sur l'autre.

Par exemple, améliorer la performance de son site, et donc sa rapidité permet mécaniquement d'améliorer son taux d'acquisition (nombre de personnes passant de simple visiteurs à utilisateurs inscrits). Mais il diminue aussi le taux de rebond (nombre d'utilisateur ne restant pas en navigation sur le site assez longtemps pour être transformé car le chargement est trop long ou le site ne répond pas assez vite).

En terme de pertes financières, ce sont des sommes parfois conséquentes qui sont mise en jeu car tout l'enjeu d'un site internet est de pouvoir transformer et fidéliser ses clients via un tunnel qui est spécifique à chaque business. Maîtriser cette acquisition est donc essentiel pour chaque entreprise qui souhaite devenir maintenir une rentabilité et une pérennité.

Etat des lieux des performances de l’application

Côté client

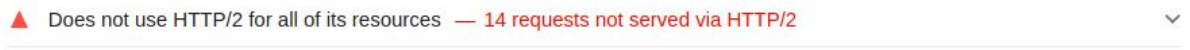
Certaines ressources HTTP peuvent être téléchargées en basse priorité et/ou en parallèle lorsque cela est possible. Cela permet aux pages de se charger plus vite et d’avoir un premier contenu plus tôt.

Les ressources bloquantes d’un site internet sont toutes les ressources HTTP qui sont téléchargées avant que le navigateur puisse analyser le code HTML et construire l’affichage graphique. Cela peut être due à un script ou une feuille de style qui est chargé avant le rendu graphique de la page. Cela diminue très souvent fortement le FCP, ce qui peut donc entraîner une diminution des leads mécaniquement.

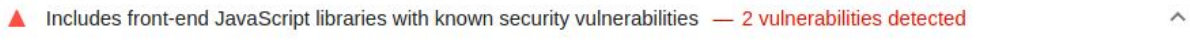
Si on continue dans le chargement des feuilles de styles, on peut souvent être tenté d’utiliser un framework CSS complet qui propose de multiple fonctionnalités sous formes de composants. C’est le cas des librairies comme Bootstrap ou MaterializeCSS pour n’en citer que deux. Néanmoins, bien souvent, la plupart de ces composants ne sont pas du tout utilisé, ce qui entraîne un surplus de style que le client télécharge, mais qui n’est au final jamais utilisé. C’est une perte de temps énorme car non seulement le client doit télécharger du code inutile, ce qui, dans le cas d’une connexion intermittente, peut grandement ralentir l'exécution des feuilles de styles qui doit attendre le téléchargement complet avant l’analyse, mais en plus cela lui empêche d’avoir un rendu de la page plus tôt, dégradant encore plus sont expérience en tant qu’utilisateur du site.

Un site rapide c’est bien, un site qui reste rapide au fil du temps, c’est encore mieux. C’est pour cela qu’il est important de maîtriser le cache des ressources téléchargée par le navigateur. Forte heureusement, la plupart des navigateurs mettent souvent ces fameuses ressources HTTP et les stockent sur le système de fichier local du client. Néanmoins ces fichiers sont volatiles, et ont souvent une durée de vie très limitée par défaut. Dans le cas du site audité, il n’y a absolument aucune politique de mise en cache faite par le serveur. Cela peut dégrader une expérience utilisateur renouvelée car le client devra télécharger de nouveau certaines ressources qui sont, pour la plupart peu ou pas mises à jour.

Le protocole HTTP/2 est le successeur du protocole HTTP/1. C'est un protocole qui apporte bien évidemment tout un lot d'amélioration, dont des améliorations de performances comme la possibilité de pouvoir télécharger des ressources en parallèle, ce qui offre un gain de performance non-négligeable. Néanmoins, le site actuellement audité ne profite pas de ce réel gain de performance et n'est pas HTTP/2 ready.



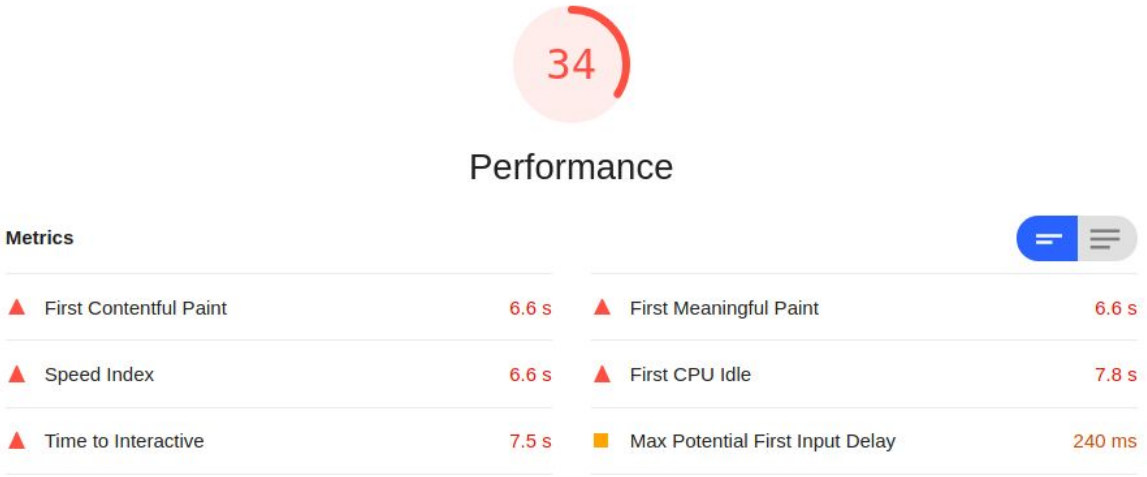
Bootstrap et jQuery sont deux librairies qui sont utilisées au sein du site audité et qui permettent pour l'une d'ajouter un aspect moderne au site internet et pour l'autre d'ajouter toute une couche de dynamisme en proposant notamment de récupérer des images au fur et à mesure que le client évolue sur le site (aussi appelé *Lazy Loading*). Néanmoins ces librairies sont obsolètes. Cela pose deux problèmes. Le premier, c'est qu'une version obsolète va entraîner des ralentissement sur l'analyse et la construction du site internet par le navigateur. La deuxième est un risque de sécurité. Ces deux points critiques peuvent entraîner un sévère malus dans le score de l'audit LightHouse qui est mené par le crawler de Google, ce qui entraîne mécaniquement une baisse dans les scores de référencement naturels, et donc une perte potentielle de clients.



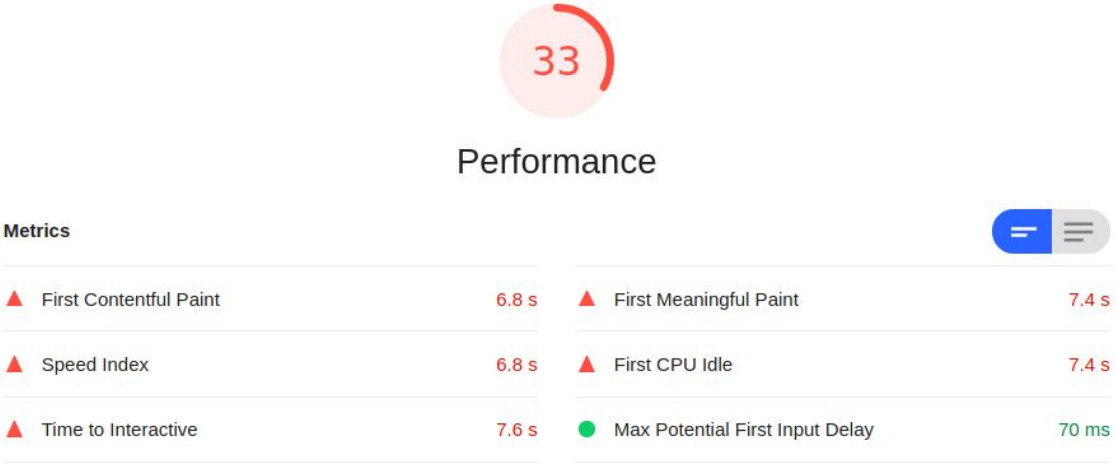
Certaines images utilisées sur le site ne sont pas optimisées pour de l'affichage sur le Web. Cela pourrait entraîner un ralentissement de la page, car l'image nécessite d'être téléchargée, puis analysée par le navigateur avant de pouvoir être affichée graphiquement sur le site internet.

Enfin, la librairie Font Awesome est utilisée pour les icônes du site. Ce qui est dommage car sur les deux pages auditée, seulement deux icônes, sur les 5000 icônes qui sont téléchargées lors de l'importation de la feuille de style.

Finalement, tous ces points mis bout à bout donne un score de performance relativement moyen.



Score de l'audit Lighthouse pour la page d'accueil.



Score de l'audit Lighthouse pour la page des tricks.

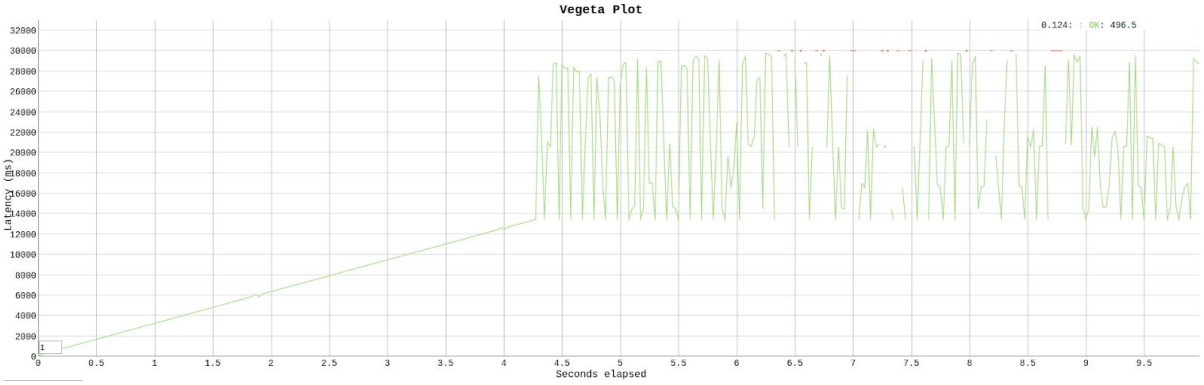
Côté serveur

Côté serveur, le code ne respecte pas les normes de codage de PHP et comporte beaucoup d'erreurs de programmation comme des boucles ou des conditions inutiles qui peuvent ralentir l'application tout comme le fait que certaines variables ou données sont utilisées sans vérifier si elles sont null avant, cela pourrait mener à des crashs inopinés de l'application.

La version de Symfony est également dépréciée et en retard d'une major release donc on ne profite pas des dernières améliorations en terme de sécurité et de performance.

En ce qui concerne les tests de charges, on peut remarquer dans le graphique ci-dessous que la latence sursaute à partir de 4.2 secondes environ, moment à partir duquel la latence s'envole). Sachant que les tests de chargement ont été réalisé avec Vegeta et pour un taux de 40 requêtes à la secondes, cela revient à effectuer environ 160 requêtes toutes les

secondes avec une latence d'environ 12 secondes par requêtes. On peut également noter que les premières erreurs interviennent à partir de 6 secondes où les requêtes prennent 30 secondes à charger et pour certaines échouent.



Explication sur les optimisations générales

Optimisation de Symfony:

- Passage de la version expirée 4.2.12 à 5.0
- Mise en cache des réponses
- Mise en place de Blackfire

Optimisation de PHP:

- Utiliser Phpcbf pour corriger certaines erreur de codage comme des erreurs d'indentation
- Corriger les erreurs de code qui ralentissent le traitement
- Mise du service container sur une seule ligne pour profiter du preloader de PHP 7.4

Optimisation de doctrine:

- mise en place de APCu pour les requêtes

Optimisation des performance du côté du navigateur Web :

- Optimisations des chargements des scripts
- Optimisations des chargements des feuilles de styles
- Mise en place du HTTP/2
- Ajout d'un TTL raisonnable pour chaque ressources
- Élimination des ressources bloquantes pour le chargement du code HTML
- Utilisation d'icônes SVG au cas par cas à la place d'une librairie entière

Description des modifications envisagées

Côté serveur

Symfony est en version 4.2.12, c'est une version dépréciée, il est impératif de passer la version de Symfony à 5.0 qui est la version la plus avancée à ce jour afin d'avoir les dernières mises à jour de sécurité et d'optimisation.

En appliquant un phpcbf, beaucoup de problème de code ont été réglés comme cet exemple:

avant

```
public function buildForm(FormBuilderInterface $builder, array $options)
 {
    $builder
    ->add('email')
 ;
 }
```

après

```
public function buildForm(FormBuilderInterface $builder, array $options)
 {
    $builder
       ->add('email');
 }
```

Cela permet de rendre le code plus lisible et plus facilement maintenable en respectant les normes de codage.

D’autres erreurs de code ne peuvent pas être corrigés par phpcbf et doivent être corrigés à la main en refactorisant le code, exemple:

avant

```
$trick =  $this->validateEdition($form, $trick, 'imgDocs', $storedImages);
$trick = $this->validateEdition($form, $trick, 'videoDocs', $storedVideos);
```

```
private function validateEdition($form, $trick, $identifier, $array)
  {
     if ($form->get($identifier)->getData() == null && isset($array)) {
        $files2Save = $this->uploadedFile->docsInputManager($array);
        ($identifier == 'imgDocs')? $trick->setImgDocs($files2Save): $trick->setVideoDocs($files2Save);
     } else {
        $docs = $form->get($identifier)->getData();
        $files2Save = $this->uploadedFile->docsInputManager($docs);

        ($identifier == 'imgDocs')? $trick->setImgDocs($files2Save): $trick->setVideoDocs($files2Save);
     }

     return $trick;
  }
```

après:

```
$trick->setImgDocs($this->validateEdition($form, 'imgDocs', $storedImages));
$trick->setVideoDocs($this->validateEdition($form, 'videoDocs', $storedVideos));
```

```
private function validateEdition($form, $identifier, $array)
{
   if ($form->get($identifier)->getData() == null && isset($array)) {
      return $this->uploadedFile->docsInputManager($array);
   }

   $docs = $form->get($identifier)->getData();
   return $this->uploadedFile->docsInputManager($docs);
}
```

Ce genre d’optimisations permettent d’accélérer le runtime de l’application tout en ayant le même comportement et la même facilité à comprendre le code.

Par défaut, Symfony compile le service container en plusieurs fichiers, mais avec la version 7.4 et le principe de class preloading, il est devenu plus efficace d'avoir un seul et même fichier, pour cela il suffit de modifier le fichier services.yaml présent dans le dossier config et rajouter la ligne suivante dans parameters:

```
parameters:
  # ...
  container.dumper.inline_factories: true
```

Dans le cas d'une modification d'entity, il est inutile de faire un $entityManager->persist(...), il suffit de faire un flush.

avant

```
$entityManager = $this->getDoctrine()->getManager();
$entityManager->persist($trick);
$entityManager->flush();
```

après

```
$entityManager = $this->getDoctrine()->getManager();
$entityManager->flush();
$entityManager->clear();
```

Ensuite, après chaque flush, il est conseillé de faire un $entityManager->clear() afin de vider le cache de l'entity manager.

Ne pas donner de variables au twig si on ne les affiche pas, pour la page d'accueil, on récupère tous les tricks en bdd alors qu'on ne les affiche pas!

avant

```
return $this->render('trick/index.html.twig', [
    'tricks' => $trickRepository->findAll(),
    'fixed_menu'=> 'enabled'
]);
```

après

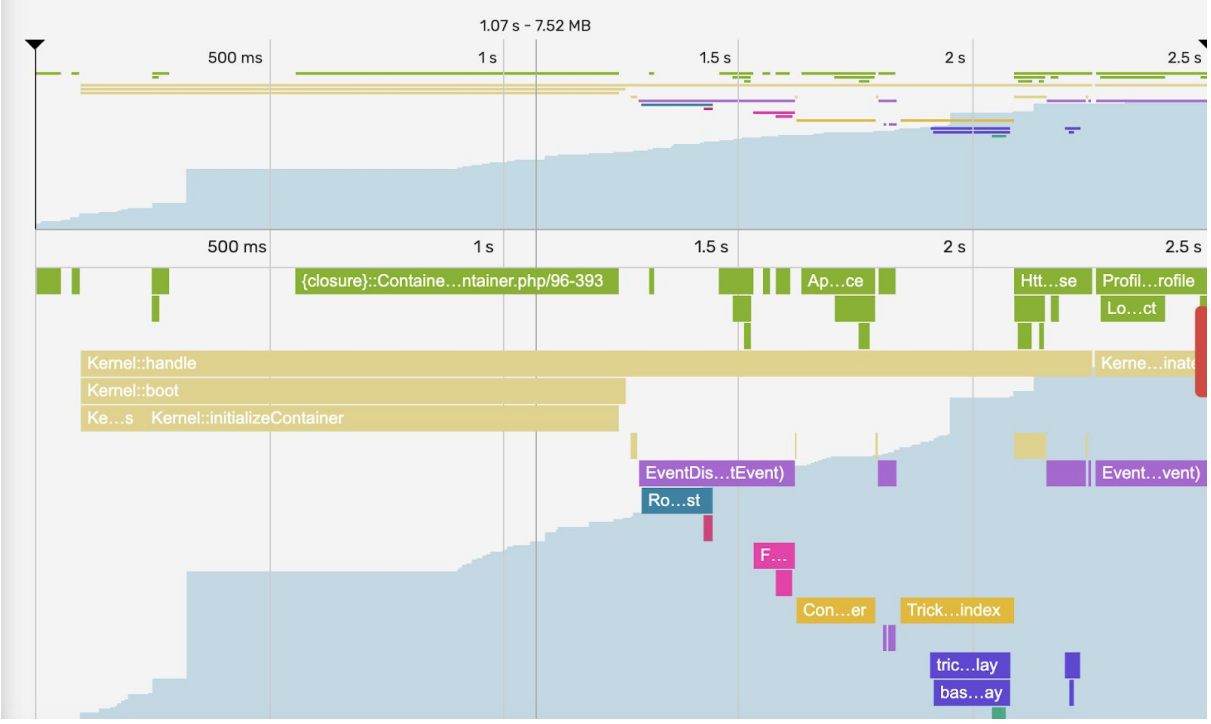
```
return $this->render('trick/index.html.twig');
```

Concernant la mise en cache des templates TWIG, il est tentant de vouloir faire un ->setMaxAge() mais en réalité, Symfony met automatiquement les templates TWIG rendu en cache lorsque nous somment en environnement de production donc il n'est pas utile de les rajouter.

Doctrine a aussi son système de cache, on peut par exemple utiliser “apcu” afin de mettre en cache le résultat de certaines requêtes, pour cela vous devez ajouter les lignes suivantes dans le fichier config/packages/doctrine.yaml.

```
orm:
    # ...
    metadata_cache_driver: apcu
    result_cache_driver: apcu
    query_cache_driver: apcu
    second_level_cache:
       enabled: true
```

En plus de Vegeta pour tester la fiabilité du service en cas de forte charge serveur, nous avons utilisé Blackfire afin d’étudier les performances des réponses serveur afin de cibler les points à améliorer.



Côté client

Il est nécessaire de charger les ressources de type feuilles de style ou scripts après que la page soit devenue interactive afin d'améliorer la responsivité du site. En JavaScript, il est possible de modifier le DOM en ajoutant des éléments HTML. La balise link étant une balise HTML, il est alors possible de la construire en JavaScript. Le langage JavaScript étant un langage mono threadé, mais utilisant un paradigme de programmation orienté événements, il est possible d'utiliser ce langage à notre avantage et de charger des balises de style dynamiquement, sans pour autant bloquer l'affichage ou l'interactivité du site, contrairement à une balise link écrite en HTML classique.

```
'use strict'

/**
 * @description Create a HTML element based off a given URL.
 * @param {string} href The url of the sytlesheet to create.
 * @return {HTMLLinkElement} The HTML element create from the url.
 * @example createStylesheet("/path/to/my/app.css");
 */
function createStylesheet(href) {
  // If the first argument is not a string
  if (typeof href !== 'string') {
    // This means it is a developer mistake, it should be accounted
    throw new TypeError('Expected first argument to be a string.')
  }

  // Creation of the link tag, but in JavaScript
  const stylesheet = document.createElement('link')

  // Tell the browser to load the URL as a stylesheet
  stylesheet.setAttribute('rel', 'stylesheet')

  // Tell the browser to load the stylesheet from this URL
  stylesheet.setAttribute('href', href)

  // Return the HTML Link Tag
  return stylesheet
}
```

Grâce à ce code, et en l'utilisant pour charger nos feuilles de styles, on gagne un temps non-négligeable d'interactivité de l'ordre d'environ 2 secondes sur le chargement de nos pages.

De plus, il nous est possible de nous pré-connecter aux origines externes dont certaines feuilles de styles dépendent. C'est le cas par exemple de Font Awesome qui est une librairie d'icônes. Ce qui nous permet de gagner quelques précieuses millisecondes à notre score de performance.

```
<link rel="preconnect" href="https://use.fontawesome.com/releases/v5.7.0/css/all.css">
```

En utilisant une configuration serveur avancée avec un serveur NGINX qui redirige les requêtes PHP vers un serveur PHP FPM qui traite les données, nous sommes capable de servir le site internet avec un cache optimisé, de la compression, du HTTP+SSL et du HTTP/2. La communication entre NGINX et PHP se fait via un reverse proxy. Ce qui permet de gagner encore en performance ainsi qu'en score de notation par le crawler de Google via LightHouse.

```
server {
    listen 80;

    server_name 127.0.0.1 localhost;

    return 301 https://$host$request_uri;
}

server {
    listen              443 ssl http2 default_server;

    server_name         localhost 127.0.0.1;

    root                /usr/share/nginx/html/public;

    index               index.html index.php;

    ssl_certificate     /etc/nginx/ssl/localhost.pem;
    ssl_certificate_key     /etc/nginx/ssl/localhost-key.pem;

    add_header          X-Frame-Options "SAMEORIGIN";
    add_header          X-XSS-Protection "1; mode=block";
    add_header          X-Content-Type-Options "nosniff";

    charset             utf-8;

    more_clear_headers  "Server";

    gzip on;
    gzip_comp_level 9;
    gzip_min_length 0;
    gzip_types font/woff2 image/gif text/plain application/javascript text/css
application/json application/xml image/jpg image/png image/webp;

    location / {
        expires 1y;

        try_files $uri $uri/ /index.php?$query_string;
    }

    location ~ \.php$ {
        fastcgi_pass    php:9000;
        fastcgi_index   index.php;
        fastcgi_param   SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include         fastcgi_params;
    }

    location ~ /\.(?!well-known).* {
        deny all;
    }
}
```

Nous avons également optimisé les images qui était auparavant servie au format JPEG ou PNG. Nous les avons donc remplacée par des images au format WEBP, ce qui permet un gain d'environ 40 à 50% en terme d'espace de stockage, ce qui est un gain non-négligeable. Tout en gardant la même qualité d'image.

Comme dit précédemment, la librairie Font Awesome n'est pas nécessaire car elle intègre les 5000 icônes de la librairie, ce qui fait 4998 icônes inutilisée. En supprimant la librairie et en téléchargeant les icônes directement au format SVG, on gagne non seulement en performance car le navigateur n'a plus besoin d'analyser toutes les icônes inutilisées.

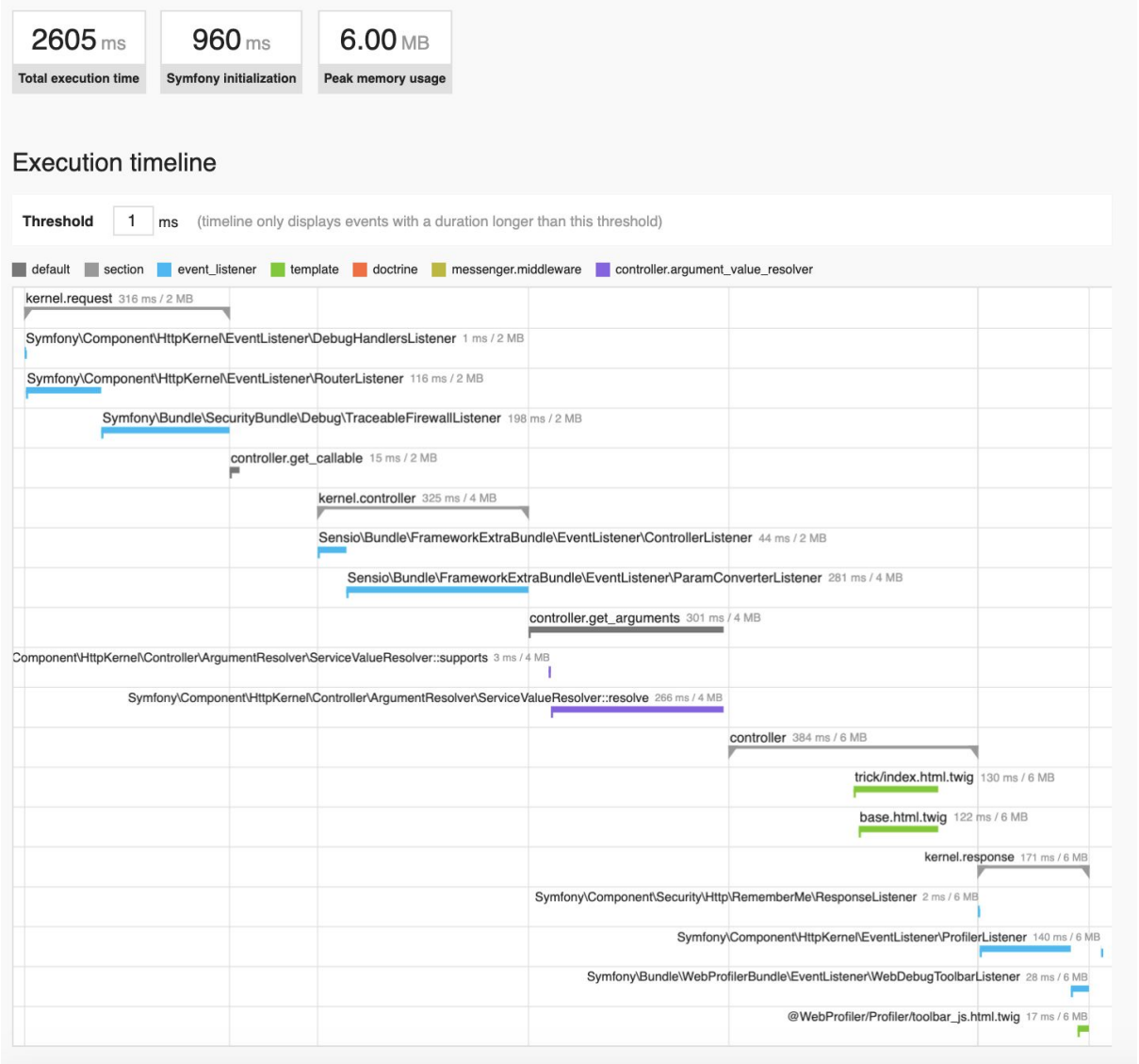
Enfin, si l'on met bout à bout toutes ces petites optimisations, on arrive à un bien meilleur score d'audit LightHouse que ce que nous avions auparavant.

Sachant que le score idéal pour le crawler de google se situe entre 90 et 100, on se rend compte ici que nous ne somme pas loin d'être dans le top car nous avoisinons les 91 de score de performance pour les deux pages.

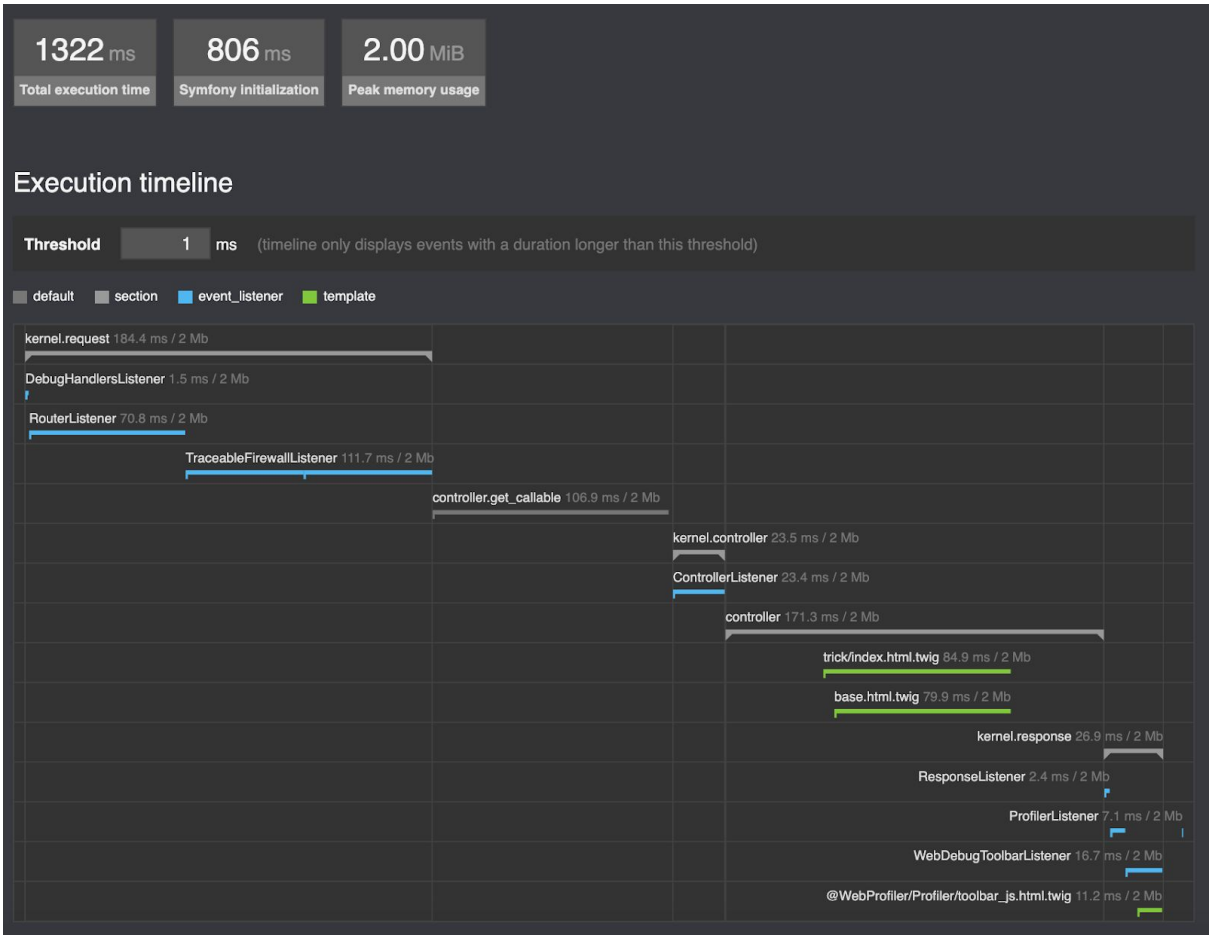
En ce qui concerne l'audit de Vegeta, c'est un gain énorme puisque nous avions auparavant des temps de réponse qui pouvait grimper jusqu'à 32 secondes. Désormais, le maximum est de 11 secondes (pour la page des tricks) et 8 secondes (pour la page d'accueil), et il n'y a plus de déperditions (lancé avec exactement les mêmes paramètres). C'est presque 3 fois moins que ce que l'on avait initialement.

Résultats des gains de performance

Page d’accueil

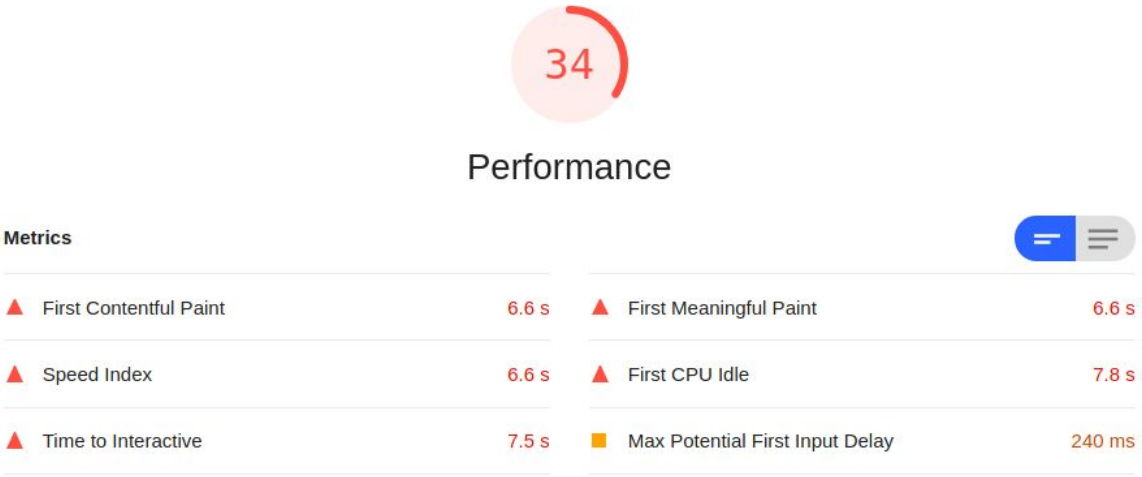


Voici le résultat du profilter Symfony sans modification de notre part.

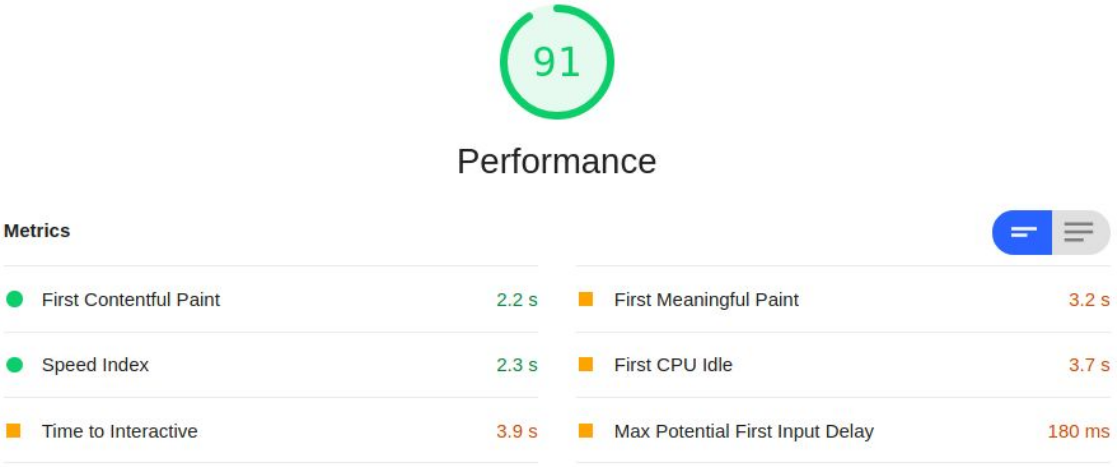


Et voici les résultats de Symfony profiler après les modifications que nous vous avons présentés. C’est une division du temps de charge par 2!

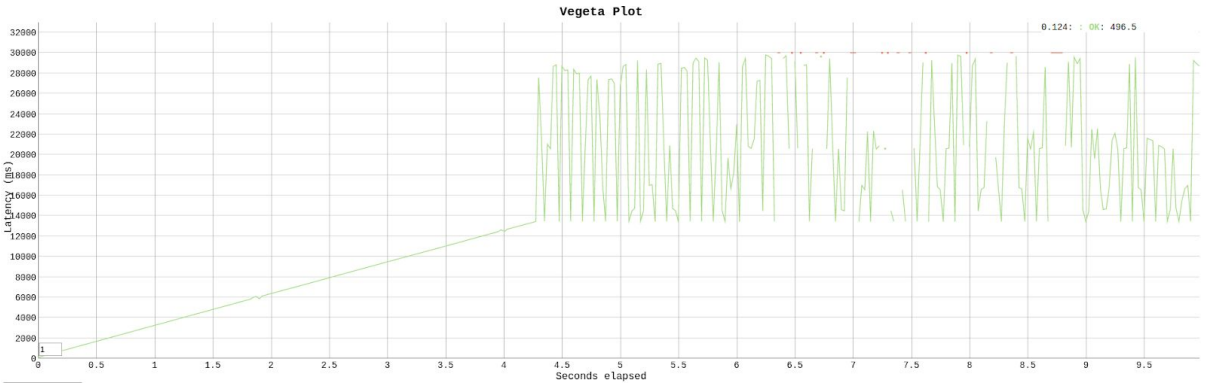
Pour ce qui est du score Lighthouse voici les chiffres:



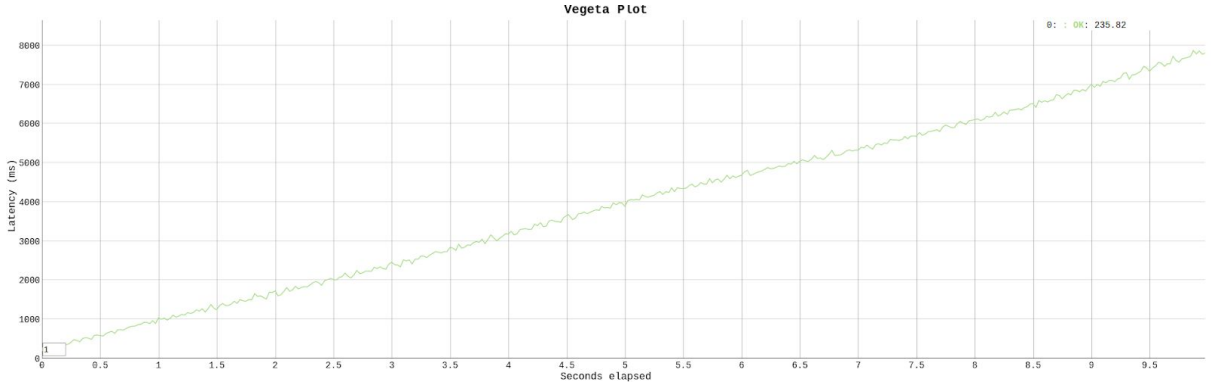
Score Lighthouse avant application des optimisations de la page d'accueil.



Score Lighthouse après application des optimisations de la page d'accueil (duration 10s, rate 40).



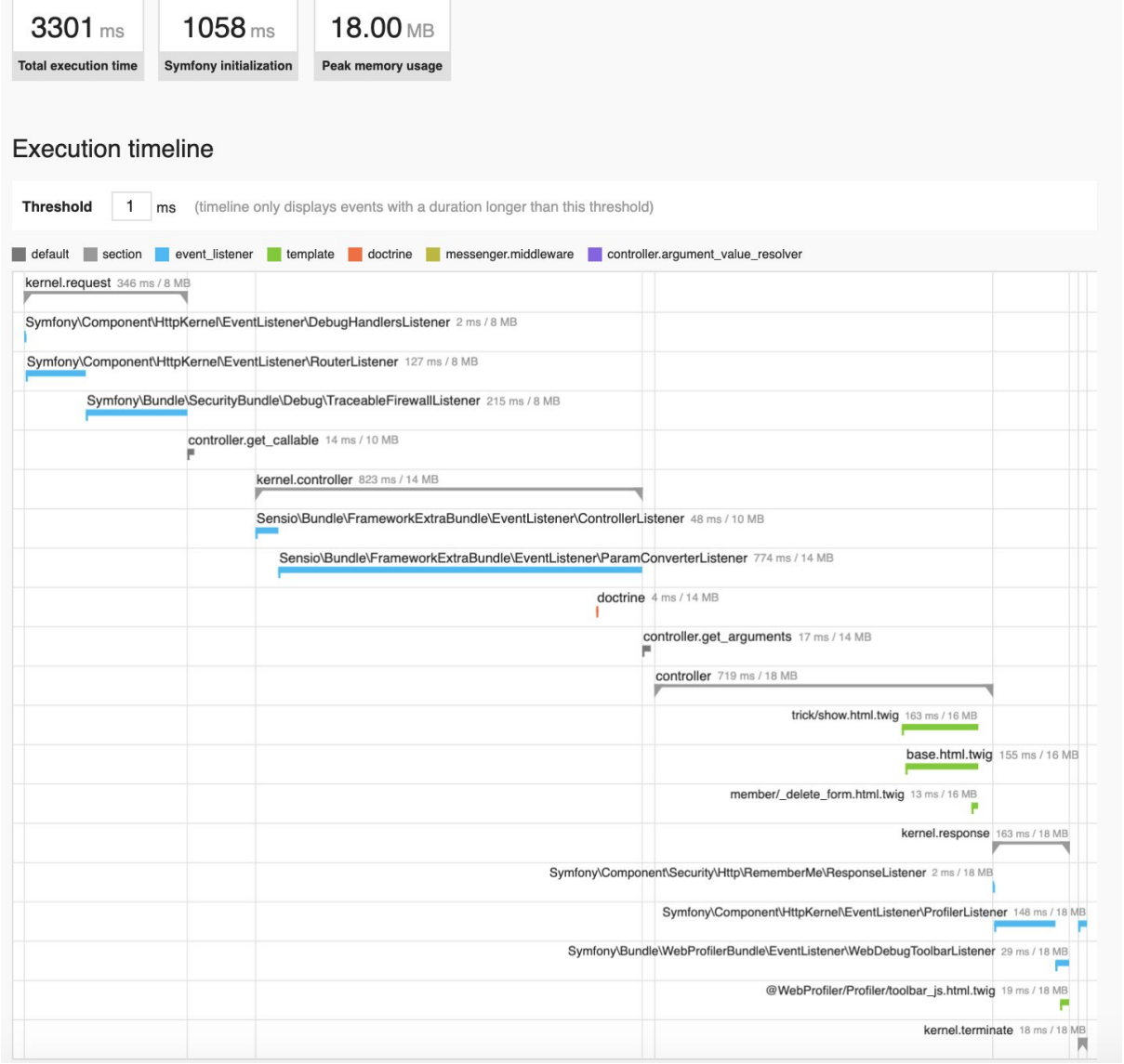
Résultats de l'audit Vegeta sur la page d'accueil avant optimisations (duration 10s, rate 40).



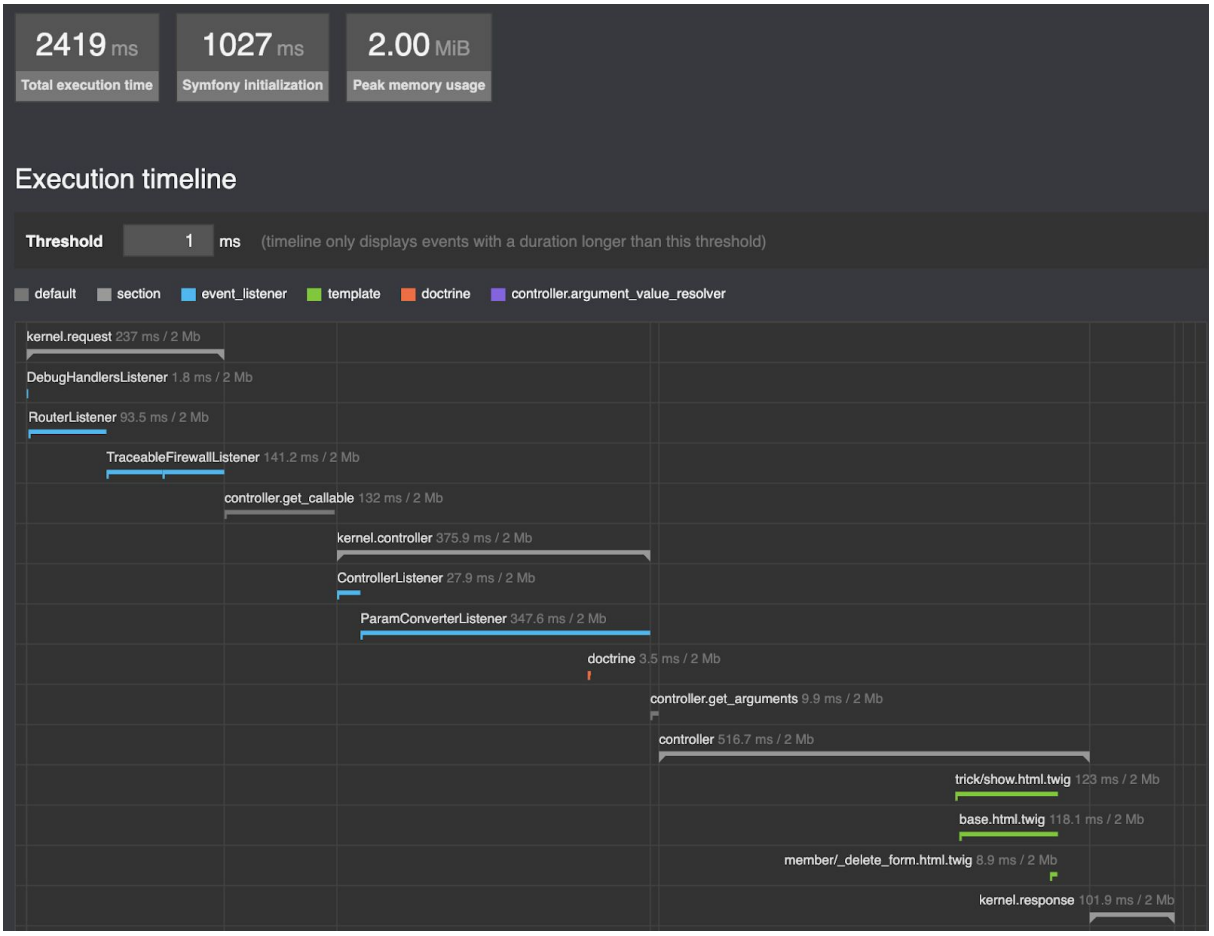
Résultats de l’audit Vegeta sur la page d’accueil après optimisations.

Page d’un trick

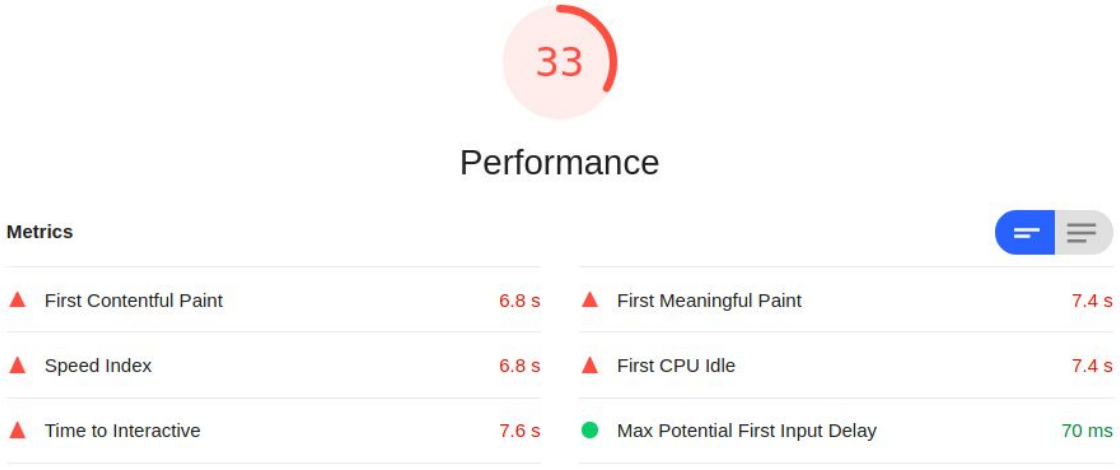
Voici les chiffres du profiler Symfony pour la page d’un trick



Avant modification de la page d’un trick.

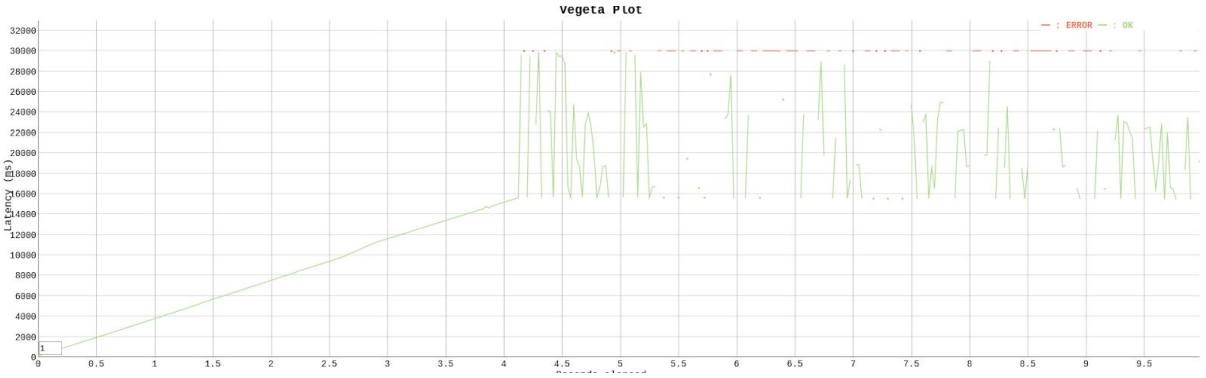


Après modification de la page d'un trick.

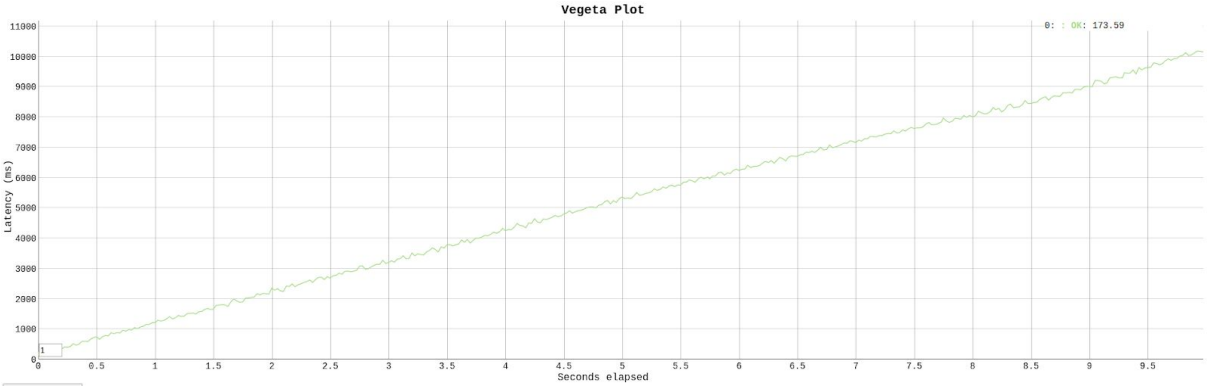


Score de l'audit Lighthouse pour la page des tricks.

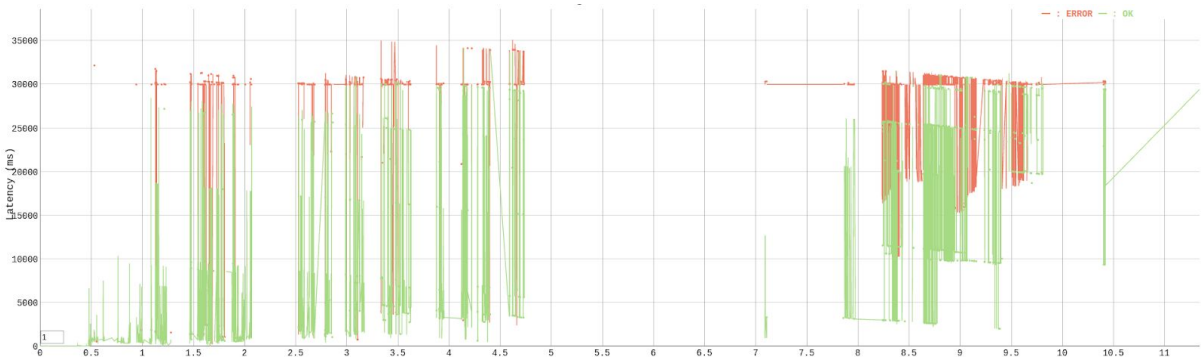
Score Lighthouse après application des optimisations de la page des tricks.



Résultat de l'audit Vegeta avant optimisations (duration 10s, rate 40).



Résultat de l'audit Vegeta après optimisations (duration 10s, rate 40).



Résultat de Vegeta poussé au maximum (duration 10s, rate 5000) à titre indicatif.