

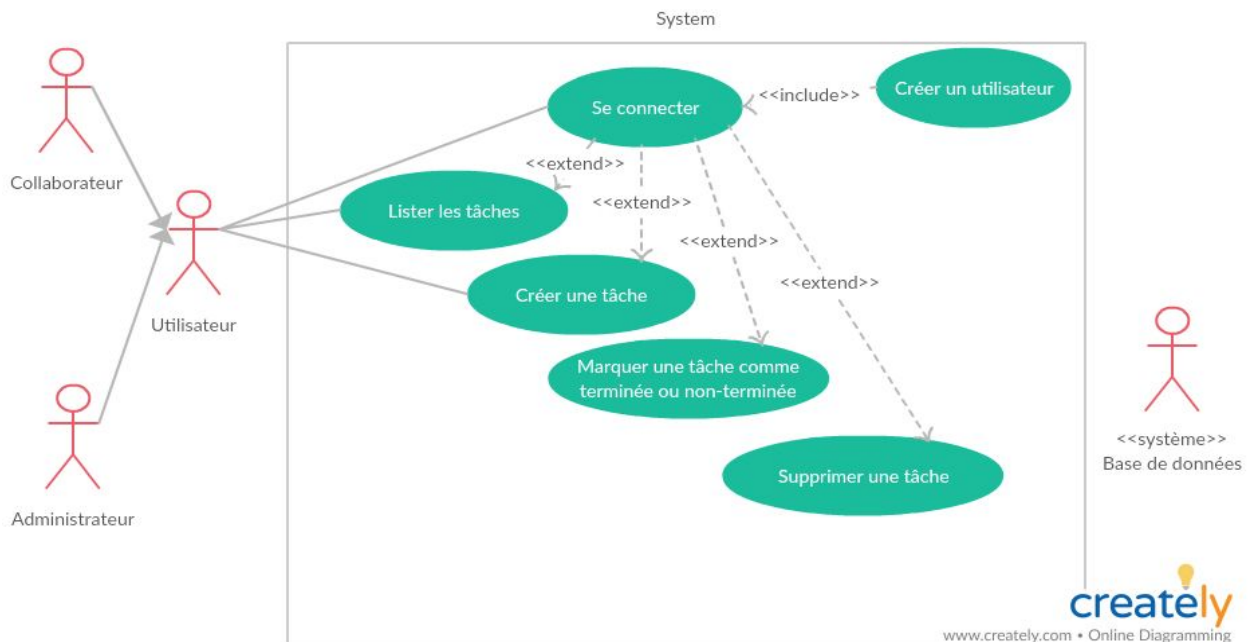
# Audit Qualité et Performance de l'application ToDoList

<b>Contexte</b>	<b>1</b>
<b>Tests</b>	<b>2</b>
<b>Qualité du code</b>	<b>3</b>
<b>Performance de l'application</b>	<b>5</b>
<b>Recommandations</b>	<b>6</b>

# Contexte

L'application ToDoList, réalisé par la société ToDo & Co, a pour but de faciliter l'organisation d'un travail au sein d'une équipe : chacun pour y créer une tâche qui se retrouvera dans la liste des tâches à effectuer, interagir avec elles. Tout membre de l'équipe utilisant ToDoList a la liberté de pouvoir gérer les tâches répertoriées, en indiquant leurs états.

Voici un diagramme résumant les cas d'utilisation de l'application :










Des interventions ont été faites sur le MVP initial, et ont permis :

- de rattacher chaque tâche nouvellement créée à un membre du site
- d'associer 2 types de profil à un membre : administrateur ou "simple" utilisateur
- de rendre possible la gestion des utilisateurs par un administrateur
- d'apporter des contraintes quant à la suppression de tâches

# Tests

L'ensemble des cas d'utilisation a été testé, en incluant pour chaque cas la vérification du pare-feu d'authentification. Dans les cas où un formulaire est utilisé, le bon déroulement de la validation (modélisée par des annotations sur les entités en questions) a été vérifié.

Voici l'aperçu du rapport de couverture du code, réalisé avec PHPUnit :

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total	<div><div></div></div>	93.29%	153 / 164	<div><div></div></div>	88.89%	40 / 45	<div><div></div></div>	76.92%	10 / 13
 Controller	<div><div></div></div>	96.92%	63 / 65	<div><div></div></div>	83.33%	10 / 12	<div><div></div></div>	75.00%	3 / 4
 DataFixtures	<div><div></div></div>	100.00%	32 / 32	<div><div></div></div>	100.00%	1 / 1	<div><div></div></div>	100.00%	1 / 1
 Entity	<div><div></div></div>	91.43%	32 / 35	<div><div></div></div>	91.67%	22 / 24	<div><div></div></div>	50.00%	1 / 2
 Form	<div><div></div></div>	100.00%	10 / 10	<div><div></div></div>	100.00%	2 / 2	<div><div></div></div>	100.00%	2 / 2
 Twig	<div><div></div></div>	100.00%	3 / 3	<div><div></div></div>	100.00%	2 / 2	<div><div></div></div>	100.00%	1 / 1
 Utils	<div><div></div></div>	68.42%	13 / 19	<div><div></div></div>	75.00%	3 / 4	<div><div></div></div>	50.00%	1 / 2
 AppBundle.php	<div><div></div></div>	100.00%	0 / 0	<div><div></div></div>	100.00%	0 / 0	<div><div></div></div>	100.00%	1 / 1

## Legend

Low: 0% to 50%    Medium: 50% to 90%    High: 90% to 100%

Generated by php-code-coverage 4.0.0 using PHP 7.1.16-1+ubuntu17.10.1+deb.sury.org+1 with Xdebug 2.6.0 and PHPUnit 5.4.6 at Wed Apr 18 18:52:45 UTC 2018.

Nous nous apercevons ici que nous approchons un taux de couverture de 100% pour la plupart des classes implémentées dans le fichier source, et mises en jeu dans l'application. Le code non couvert correspond :

- dans le dossier `src/Controller/`, aux méthodes de `SecurityController` permettant d'implémenter la vérification de l'authentification (route `login_check`), ainsi que la déconnexion (route `logout`). La spécificité du *framework* Symfony est de ne jamais atteindre ce code.
- dans le dossier `src/Entity/`, au *getter* `getCreatedAt` et au *setter* `setCreatedAt`, permettant de récupérer et d'affecter la propriété `createdAt`, la date de création.
- dans le dossier `src/Utils/`, à la méthode du service `UserRefresher` permettant de rafraîchir le *token* utilisateur en session dans le cas où l'on viendrait à modifier l'entité associé à l'utilisateur connecté (en modifiant par exemple son rôle).

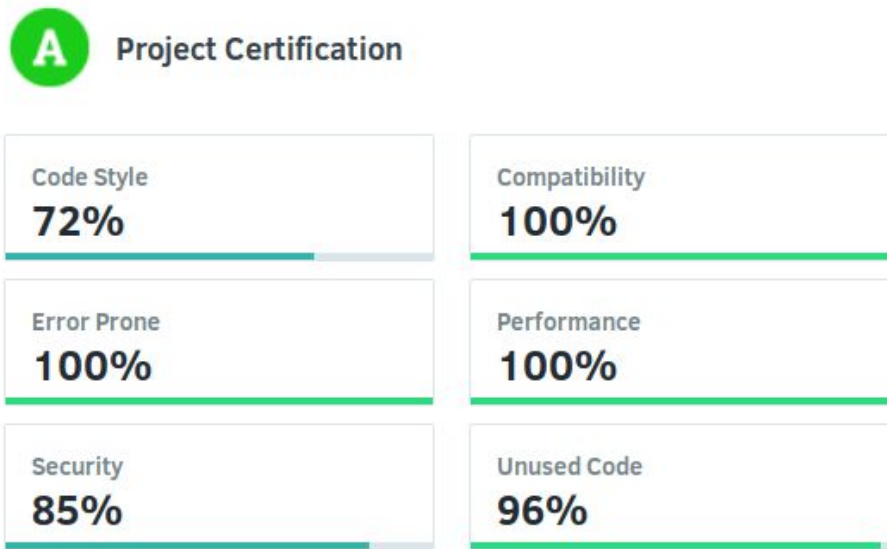
Le profil de cette couverture est standard. Ces tests de la logique critique au bon fonctionnement de l'application `ToDoList` sont satisfaisants. L'ensemble du rapport de couverture est disponible dans le dossier `coverage/` du *repository* `todolistapp`.

# Qualité du code

Les rapports de mesure de qualité, établis pour l'application ToDoList, permettent d'être confiant quant à la qualité du code écrit. Les outils suivant ont été utilisés :

- CodeClimate
- PHP Code Sniffer (PHPCS)
- PHPMetrics

ToDoList est certifié de classe A par CodeClimate :



23 problèmes minimes sont relevé par l'outil, et la plupart est liée au *framework* Symfony. Les seuls problèmes liés au code produit concernent la longueur de certaines variables (propriété `$id`, référence `EntityManager $em`), et le paramètre `$request` inutilisé dans la méthode `loginAction` de la classe `AppBundle/Controller/SecurityController`.

Le rapport complet CodeClimate est disponible à l'URL suivante :

<https://app.codacy.com/app/Nabil001/todolistapp/dashboard>.

PHPCS relève des avertissements prévus relatifs à la documentation du code, et à la longueur de certaines lignes.

PHPMetrics ne relève aucune violation. La complexité et la maintenabilité des classes sont satisfaisantes. L'outil attire cependant l'attention sur la maintenabilité des classes AppBundle\DataFixtures\ORM\Fixtures\Fixtures, et AppBundle\Entity\User, qui contiennent plus de ligne que les autres classes (elles sont signalées en rouge sur le diagramme) :



Ceci est normal : la classe User doit implémenter en plus 4 méthodes de *UserInterface* pour être exploitable par le composant sécurité du *framework*, et la classe Fixtures créer une multitude d'instance afin d'initialiser la base de données. Le rapport complet PHPMetrics est disponible dans le dossier *phpmetrics/* du *repository* *todolistapp*.

# Performance de l'application

[Blackfire](#) a été utilisé pour mesurer les performances de l'application. Cet outil modélise la performance sous forme de diagrammes, en représentant les classes et méthodes appelées, avec leurs nombres d'appels et le temps d'exécutions de chacune. Cela permet de se rendre compte du comportement global d'une application PHP, et de rapidement déceler tout problème de performance, par exemple lié à un appel de méthode trop fréquent.

Deux profils ont été établis pour l'application ToDoList (Symfony est alors dans un environnement de production). Ils représentent tous deux le comportement de l'application au chargement de la page d'authentification :

- le premier est représente le comportement sans cache créé par le *framework* Symfony (le dossier var/ n'existe pas) :  
<https://blackfire.io/profiles/c6901fda-8c4f-4fa7-9545-ed8e8694411d/graph>
- le second représente le comportement avec cache :  
<https://blackfire.io/profiles/5a997e77-31e3-4d6a-a17f-dcc49f302bb9/graph>

Ces profils peuvent être considérés comme normaux. Aucune anomalie n'est à remarquer. Nous observons qu'avec le cache, l'application est plus de 3 fois plus rapide que sans. Cela est dû qu'au premier lancement de l'application, Symfony charge le cache, et au second lancement, le cache contient déjà classes utilisées par le *framework* dans un fichier PHP. Ainsi moins de *namespaces* sont à résoudre.

Dans le cas où l'application aurait beaucoup de tâches à charger, nous pouvons la faire gagner en performance en implémentant une technique de *proxy caching*, un serveur ou un composant du *framework* mettant en cache certaines réponses : par exemple, la liste des tâches serait mise en cache pour toute nouvelle requête vers le server web, jusqu'à création d'une nouvelle. Symfony dispose du composant Cache, qui, à chaque lancement de l'application, vérifie si la réponse est déjà mise en cache. L'inconvénient est que l'application Symfony est tout de même lancée, qu'il y ait la réponse en cache ou non. Une solution plus optimale serait de mettre en place un serveur de cache, tel que [Varnish](#), qui jouerait le rôle d'intermédiaire entre le client et le serveur web.

# Recommandations

Le projet est correctement organisé : il répond aux attentes du *framework* Symfony en terme de structure. Cependant, Nous aurions tout à gagner à implémenter quelques bonnes pratiques, manquantes dans ce projet.

Il serait intéressant d'avoir la configuration suivante, du fichier `app/config/service.yaml` :

```
services:
  _defaults:
    autowire: true
    autoconfigure: true
    public: false
  AppBundle\:
    resource: '../..src/AppBundle/*'
    exclude: '../..src/AppBundle/{Entity, Repository}'
```

Cette configuration importe par défaut toutes les classes situées dans le dossier `src/` en tant que services (à l'exception des classes issues des dossiers `Entity/` et `Repository/`). Couplée avec l'activation de la clé `autowire`, elle au *framework* d'injecter automatiquement les dépendances dans les classes créées, en analysant la signature des constructeurs. On peut toujours effectuer des injections de dépendances, ou écraser cette configuration par défaut manuellement :

```
services:
  # ...
  AppBundle\Updates\SiteUpdateManager:
    argument:
      $manager: doctrine.orm.entity_manager
    public: true
```

Avec l'activation de la clé `autoconfigure`, Symfony configure automatiquement les services créés : `Subscriber`, extension `Twig`, `Guard`, etc. Il n'y a plus à associer manuellement de *tag* au service créé.

Attribuer la valeur `false` à la clé `public` rend les services privés. Il ne sont plus accessible par la méthode `get` du conteneur de services. Pour se servir d'un service dans une méthode de contrôleur, il faut simplement l'indiquer dans la signature de cette méthode :

```
public function someAction(EntityManagerInterface $manager)
{
  # ...
}
```

Une autre des recommandations concerne les actions des contrôleurs. Les méthodes des contrôleurs contiennent relativement beaucoup de lignes, et la logique métier est souvent directement codées dedans. Il serait plus flexible et lisible de créer des services dédiés à l'implémentation de la logique métier. Lorsqu'une méthode a besoin d'un tel service, il suffit de lui injecter en ajoutant le service en paramètres de la méthode (à condition que l'*autowiring* est activé).

Prenons l'exemple du traitement des formulaires. Jusqu'ici, ils sont intégralement traités directement au sein des contrôleurs :

```
public function add(Request $request, EntityManagerInterface $em)
{
    $task = new Task();
    $form = $this->createForm(TaskType::class, $task)
        ->handleRequest($request);

    if ($form->isValid()) {
        if ($this->isGranted('IS_AUTHENTICATED_FULLY')) {
            $task->setAuthor($this->getUser());
        }

        $em->persist($task);
        $em->flush();

        $this->addFlash('success', 'Task has been created.');
```

```
        return $this->redirectToRoute('task_list');
    }

    return $this->render('task/add.html.twig', [
        'form' => $form->createView()
    ]);
}
```

En utilisant un service traitant les formulaires, tel que celui du bundle [TBoileau\FormHandlerBundle](#), développé par [Thomas Boileau](#), nous pouvons nous permettre de n'avoir que très peu de codes dans notre action :

```
public function add(TaskHandler $handler)
{
    return $handler->handle(new Task(), [], 'task/add.html.twig');
}
```