

Documentation technique

Todo & Co - Todolist

Thomas Chauveau

OpenClassrooms - Développeur d'application - PHP / Symfony

Projet 8 - Améliorez une application existante de Todo & Co

16 octobre 2023

Sommaire

1. Le Projet.....	2
1.1. Présentation.....	2
1.2. Technologies.....	2
2. Authentification.....	3
2.1. Les Interfaces.....	3
2.1.1. UserInterface.....	3
2.1.2. PasswordAuthenticatedUserInterface.....	4
2.2. Security.yaml.....	5
2.2.1. Les providers.....	5
2.2.2. Le Hasher.....	6
2.2.3. Le Firewall.....	7
2.2.4. Hiérarchie des rôles.....	8
2.2.5. Contrôle d'accès.....	8
2.3. Le Controller.....	9
2.3.1. Login.....	9
2.3.2. Logout.....	9
3. Autorisation.....	10
3.1. L'attribut IsGranted.....	10
3.2. TaskVoter.....	10
4. Implications.....	11
4.1. Les Interfaces.....	11
4.2. Le fichier Security.yaml.....	11
4.3. Le Controller et le Voter.....	11
5. Conclusion.....	12



1. Le Projet

1.1. Présentation

ToDo & Co est une startup qui a développé un MVP (Todolist) pour la gestion de tâches via une application Symfony. ToDo & Co veut faire évoluer la codebase de Todolist vers des technologies plus récentes et faciles à maintenir. ToDo & Co a donc fait appel à moi pour améliorer la qualité du code, ajouter de nouvelles fonctionnalités et corriger des anomalies. Je dois aussi implémenter des tests automatisés pour assurer la fiabilité de l'application et réaliser un audit de qualité du code et de performance.

1.2. Technologies

L'application est désormais à jour, utilisant désormais la dernière version de Symfony (6.3, afin de bénéficier des toutes dernières fonctionnalités, optimisations et sécurités offertes par ce framework de développement). Cette version a également été choisie pour sa facilité à passer rapidement, au moins sur la version Symfony 6.4, à sortir très prochainement et qui sera en version LTS.

Le passage à Symfony 6.3 nécessite aussi de disposer de PHP 8. Todolist requiert d'ailleurs PHP 8.2 au minimum, car PHP 8.2 est actuellement encore activement maintenu, pendant encore 1 an pour son développement, 2 ans pour les correctifs de sécurité.

La mise à jour vers Symfony 6.3 et PHP 8 représente donc un saut significatif en termes de performances, de sécurité et de fonctionnalités pour l'application.

2. Authentification

L'authentification est un élément crucial de toute application web sécurisée. Dans le cadre de notre application ToDolist, basé sur Symfony, cela est géré par la classe **User** qui implémente par défaut certaines interfaces fournies par Symfony. Ces interfaces aident à standardiser la manière dont les utilisateurs sont gérés et authentifiés. Dans cette section, nous allons explorer en profondeur la classe **User** et les deux interfaces clés : **ManagerInterface** et **AuthenticatedManagerInterface**.

Ces interfaces sont automatiquement ajoutées si le projet utilise les bundles **symfony/security-bundle** et **symfony/maker-bundle**.

2.1. Les Interfaces

2.1.1. ManagerInterface

ManagerInterface est l'interface de base que toute classe **User** doit implémenter pour être compatible avec le système de sécurité de Symfony. Elle définit un ensemble minimal de méthodes que votre classe **User** doit fournir.

Ces méthodes sont les suivantes :

- **eraseCredentials()**: Après l'authentification, cette méthode est appelée pour effacer toute information sensible. C'est une étape de sécurité pour s'assurer qu'aucune donnée sensible ne persiste.
- **getRoles()**: Cette méthode est cruciale pour le système d'autorisation de Symfony. Elle doit retourner un tableau de rôles que l'utilisateur possède. Ces rôles sont ensuite utilisés pour prendre des décisions sur ce que l'utilisateur est autorisé à faire dans l'application.
- **getUserIdentifier()**: Cette méthode doit retourner l'identifiant unique de l'utilisateur, qui peut être un nom d'utilisateur, une adresse e-mail, ou tout autre identifiant unique.

2.1.2. PasswordAuthenticatedUserInterface

PasswordAuthenticatedUserInterface est spécifique aux scénarios où l'utilisateur doit être authentifié à l'aide d'un mot de passe. Elle ajoute une méthode supplémentaire à la classe **User** pour gérer le mot de passe.

Cette méthode est la suivante :

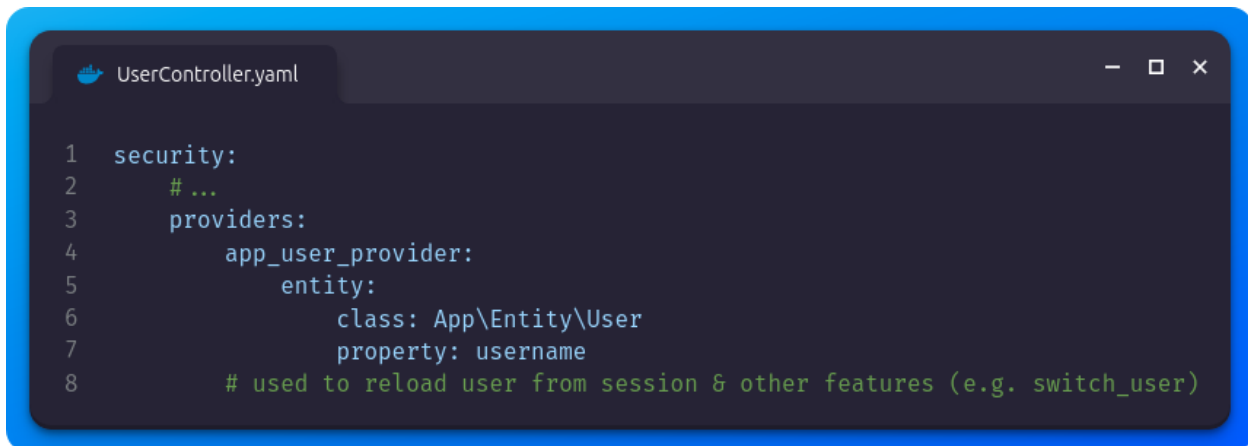
- **getPassword()**: Cette méthode doit retourner le mot de passe haché de l'utilisateur. Le système de sécurité de Symfony utilisera ce mot de passe haché pour valider les tentatives de connexion.

2.2. Security.yaml

Le fichier **security.yaml** sert de pont entre la classe **User** et le système de sécurité de Symfony. Il utilise les méthodes définies dans les interfaces pour configurer comment les utilisateurs sont chargés (**providers**), comment les mots de passe sont vérifiés (**hashers**), et comment le processus d'authentification doit se dérouler (**firewall**).

2.2.1. Les providers

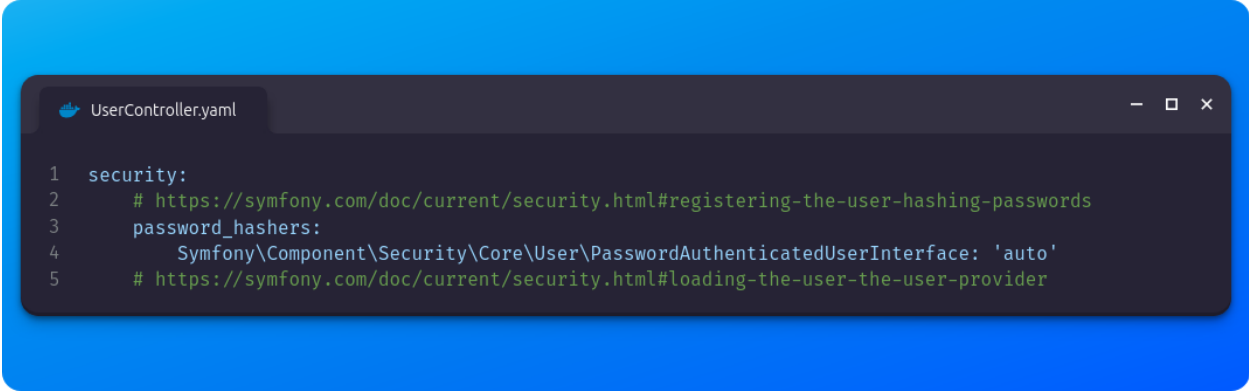
Dans la section **providers**, la source des utilisateurs est spécifiée. Si une base de données est utilisée pour stocker les informations des utilisateurs, c'est ici que la classe **User** est définie.



```
UserController.yaml
1 security:
2   # ...
3   providers:
4     app_user_provider:
5       entity:
6         class: App\Entity\User
7         property: username
8       # used to reload user from session & other features (e.g. switch_user)
```

Par exemple, pour Todolist, **App\Entity\User** est la classe **User** qui implémente les interfaces **UserInterface** et **PasswordAuthenticatedUserInterface**. Le champ **username** est utilisé comme identifiant unique pour chaque utilisateur, conformément à la méthode **getUserIdentifier()** de la classe **User**.

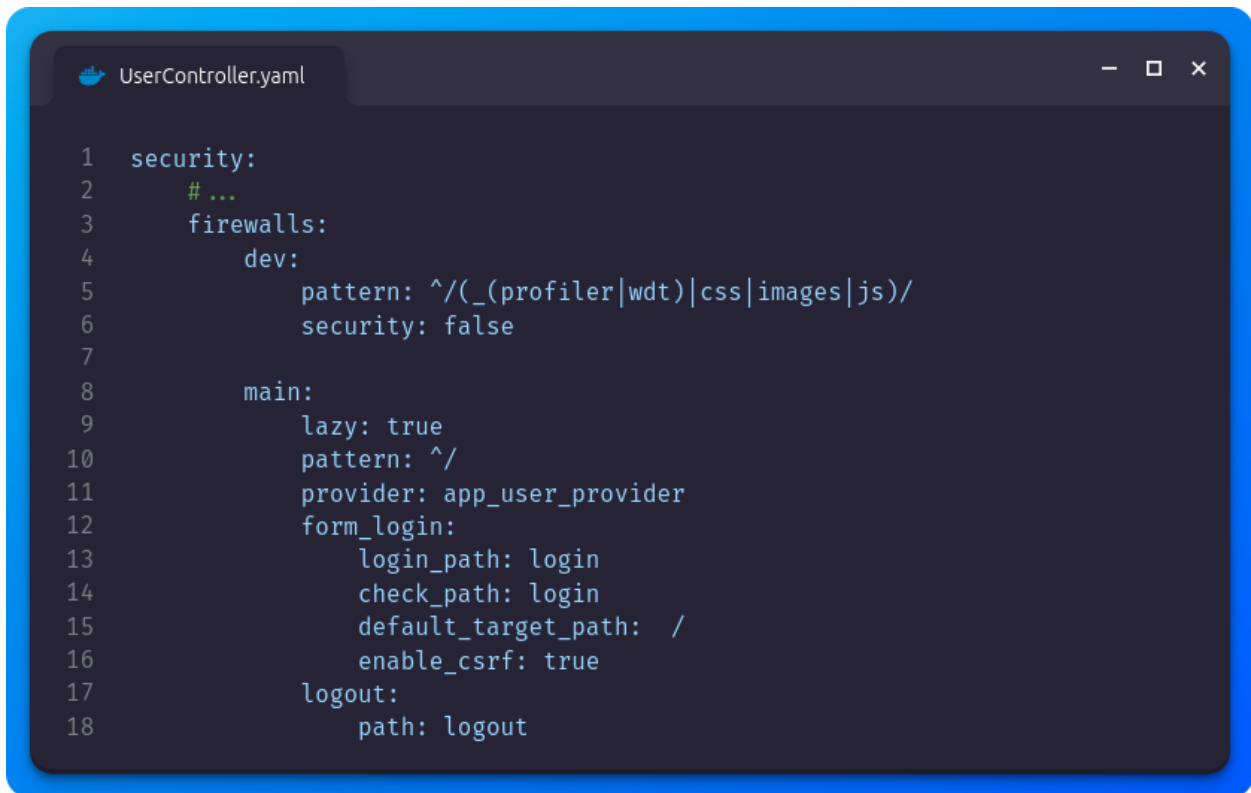
2.2.2. Le Hasher



```
UserController.yaml
1 security:
2   # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
3   password_hashers:
4     Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
5   # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
```

La section **password_hashers** spécifie l'algorithme de hachage utilisé pour les mots de passe. Cela doit être en accord avec ce que retourne la méthode **getPassword()** dans la classe **User**. L'option **'auto'** permet à Symfony de choisir le meilleur algorithme de hachage disponible sur le système. Cette option s'applique à toutes les classes qui implémentent **PasswordAuthenticatedUserInterface**, y compris la classe **User**.

2.2.3. Le Firewall



```
1 security:
2   # ...
3   firewalls:
4     dev:
5       pattern: ^/(_(profiler|wdt)|css|images|js)/
6       security: false
7
8     main:
9       lazy: true
10      pattern: ^/
11      provider: app_user_provider
12      form_login:
13        login_path: login
14        check_path: login
15        default_target_path: /
16        enable_csrf: true
17      logout:
18        path: logout
```

La majorité de la logique d'authentification est configurée dans la section **firewalls**. Il est possible de spécifier quel "provider" utiliser, quel formulaire pour le login, etc. Le pare-feu **main** est responsable de l'authentification et utilise le fournisseur **app_user_provider**. Il est également configuré pour utiliser un formulaire de connexion et pour activer la protection CSRF.

2.2.4. Hiérarchie des rôles

```
UserController.yaml
1 security:
2   # ...
3   role_hierarchy:
4     ROLE_ADMIN: ['ROLE_ADMIN']
5     ROLE_USER: ['ROLE_USER']
```

Une hiérarchie de rôles simple est définie, où les rôles **ROLE_ADMIN** et **ROLE_USER** n'ont pas de rôles enfants. Cette hiérarchie peut être utilisée pour définir des permissions complexes si nécessaire.

2.2.5. Contrôle d'accès

Des règles d'accès sont définies pour différentes routes. Par exemple, seules les personnes ayant le rôle **ROLE_ADMIN** peuvent accéder aux routes commençant par **/users**.

```
UserController.yaml
1 security:
2   # ...
3   access_control:
4     - { path: ^/login, roles: PUBLIC_ACCESS }
5     - { path: ^/users/create, roles: PUBLIC_ACCESS }
6     - { path: ^/users, roles: ROLE_ADMIN }
7     - { path: ^/tasks, roles: ROLE_USER }
8     - { path: ^/, roles: PUBLIC_ACCESS }
```

2.3. Le Controller

Le contrôleur **SecurityController** est responsable de la gestion des actions liées à l'authentification dans l'application Symfony. Il étend **AbstractController**, ce qui lui donne accès à des méthodes pratiques pour manipuler les requêtes, les réponses et d'autres fonctionnalités de Symfony.

2.3.1. Login

La méthode **loginAction** est associée à la route **/login** et est nommée **login**. Elle prend en paramètre un objet **AuthenticationUtils**, qui est un utilitaire fourni par Symfony pour simplifier certaines tâches liées à l'authentification.

- **getLastAuthenticationError()**: Cette méthode récupère la dernière erreur d'authentification, si elle existe. Cela est utile pour informer l'utilisateur de ce qui a mal tourné lors de la tentative de connexion.
- **getLastUsername()**: Cette méthode récupère le dernier nom d'utilisateur qui a été saisi. Cela est utile pour pré-remplir le champ du nom d'utilisateur si l'utilisateur échoue à se connecter, améliorant ainsi l'expérience utilisateur.

La méthode **render** est ensuite appelée pour afficher la vue associée à l'action de connexion, en passant le dernier nom d'utilisateur et l'erreur éventuelle comme variables à la vue.

2.3.2. Logout

La méthode **logoutAction** est associée à la route **/logout** et est nommée **logout**. Cette méthode ne contient pas de logique d'exécution et se contente de lancer une exception **RuntimeException** si elle est appelée directement. En réalité, cette méthode est juste un point d'ancrage pour la configuration de déconnexion dans **security.yaml**. Symfony prend en charge la logique de déconnexion et n'appellera jamais cette méthode directement.

3. Autorisation

3.1. L'attribut IsGranted

L'attribut **IsGranted** est un mécanisme d'autorisation dans Symfony qui permet de vérifier si l'utilisateur courant a les permissions nécessaires pour accéder à une certaine route ou exécuter une certaine action.

Dans le contrôleur **TaskController**, cet attribut est utilisé pour s'assurer que l'utilisateur est le propriétaire de la tâche qu'il essaie de modifier, de voir ou de supprimer.

Cet attribut prends deux paramètres :

- **TaskVoter::OWNER**, représente le type de vérification à effectuer. Cette constante est définie dans le fichier **TaskVoter**.
- **subject: 'task'**, indique que l'objet sur lequel la vérification doit être effectuée est de type **Task**.

3.2. TaskVoter

La classe **TaskVoter** est un Voter personnalisé qui implémente la logique d'autorisation pour les tâches. Elle étend la classe **Voter** de Symfony et doit implémenter deux méthodes : **supports** et **voteOnAttribute**.

- La méthode **supports()** vérifie si le Voter prend en charge l'attribut et le sujet donnés. Dans ce cas, il vérifie si l'attribut est **OWNER** et si le sujet est une instance de la classe **Task**.
- La méthode **voteOnAttribute()** contient la logique de vote. Elle récupère l'utilisateur courant à partir du son token et effectue diverses vérifications pour décider si l'accès doit être accordé ou non.
 - Si l'utilisateur est un administrateur (**ROLE_ADMIN**), il a accès à toutes les actions.
 - Si la tâche n'appartient pas à un utilisateur, l'accès est refusé.
 - Si l'utilisateur est le propriétaire de la tâche, l'accès est accordé.

4. Implications

Dans cette section, nous allons examiner les implications et les recommandations en matière de sécurité pour l'application, en nous concentrant sur les trois piliers principaux : les interfaces, le fichier **security.yaml** et les Controllers et Voters.

4.1. Les Interfaces

Les interfaces (**UserInterface** et **PasswordAuthenticatedUserInterface**) sont le fondement de la manière dont Symfony gère l'authentification des utilisateurs. Leur utilisation correcte garantit que l'application respecte les meilleures pratiques de sécurité, notamment en ce qui concerne le stockage des mots de passe et la gestion des rôles.

4.2. Le fichier Security.yaml

Ce fichier est le cœur de la configuration de sécurité. Il configure de manière détaillée comment les utilisateurs sont authentifiés et autorisés, en s'appuyant sur les méthodes et les interfaces implémentées dans la classe **User**.

4.3. Le Controller et le Voter

Le contrôleur d'authentification (**SecurityController**) et le Voter (**TaskVoter**) sont des exemples de la manière dont une application peut mettre en œuvre des mécanismes d'authentification et d'autorisation robustes.



5. Conclusion

L'application est construite sur une fondation solide en matière de sécurité, en utilisant les fonctionnalités robustes offertes par le framework Symfony, et notamment par la puissance offerte par Symfony 6. Grâce à une implémentation des interfaces **UserInterface** et **PasswordAuthenticatedUserInterface**, l'application assure une authentification sécurisée des utilisateurs.

Le fichier **security.yaml** agit comme le cerveau de la configuration de sécurité, permettant une authentification et une autorisation flexibles mais robustes. Les contrôleurs et les voters ajoutent une couche supplémentaire de sécurité, en permettant des contrôles d'accès granulaires basés sur des rôles et des permissions.

