



ToDo&Co

Documentation technique
Guide d'authentification

Auteur: Charlotte SAURY

Dernière mise à jour: 23/10/2020

Version: 1.0

Contexte

Ce guide a pour objectif d'expliciter le **système d'authentification** mis en place pour l'**application ToDo&Co**, afin d'assurer une correcte compréhension par tous les développeurs qui contribueront au développement de l'application, et de définir quels fichiers peuvent être modifiés et comment afin de pallier aux évolutions de l'application. Cette application est accessible par tous, mais son utilisation nécessite d'être enregistré et authentifié.

Introduction

L'application ToDo&Co est développée sur la base du **Framework Symfony** qui propose un composant performant permettant de gérer l'authentification des utilisateurs: le **composant Security**. Ce composant permet d'une part de gérer l'**authentification**, à savoir d'où proviennent les utilisateurs et comment gérer l'authentification.

D'autre part, ce système permet également de gérer un système d'**autorisations**, à savoir gérer l'accès à certaines ressources selon le rôle défini de l'utilisateur (visiteur non authentifié, utilisateur authentifié ou encore administrateur).

Ces deux aspects seront traités dans les deux parties suivantes. La majorité des configurations pour ces différents volets sont regroupées au sein du fichier **config/packages/security.yaml**.

Authentification

Les utilisateurs

La configuration des **utilisateurs** est gérée intégralement au sein du fichier **security.yaml**.

- Un utilisateur est représenté par l'entité **User**, qui implémente la **UserInterface**.
- La classe User étant une entité Doctrine, les utilisateurs sont stockés en **base de données**. Cette configuration est permise par le paramètre **security.providers**
- L'authentification est réalisée sur la base d'un **nom d'utilisateur (username)**, définie grâce au paramètre **property** et d'un **mot de passe**. Pour des raisons de sécurité évidentes, le mot de passe ne peut être enregistré en clair en base de données. le paramètre **security.encoders** permet d'assurer un encodage de ce mot de passe avant enregistrement de l'utilisateur. Dans notre cas, l'utilisation de la clé **auto** pour l'algorithme d'encodage permet à Symfony de sélectionner le meilleur algorithme. Cet **encoder** sera utilisé pour encoder le mot de passe lors de la création d'un utilisateur en base de données, via l'interface **UserPasswordEncoderInterface**.

```

encoders:
    App\Entity\User: auto

providers:
    doctrine:
        entity:
            class: App\User
            property: username

```

Le système d'authentification

Le système d'authentification du composant Security repose sur la définition de **firewalls** qui définissent comment les utilisateurs sont authentifiés, et notamment du **firewall main**.

```

main:
    anonymous: true
    lazy: true
    pattern: ^/
    form_login:
        login_path: login
        check_path: login
        csrf_token_generator: security.csrf.token_manager
        always_use_default_target_path: true
        default_target_path: /
    logout:
        path: logout

```

- Le 1er paramètre **firewalls.main.anonymous** indique un statut anonyme aux utilisateurs non authentifiés. Le paramètre **lazy** permet d'éviter de créer une nouvelle session s'il n'y a pas besoin d'autorisation particulière, permettant la mise en cache de toutes les urls ne nécessitant pas d'utilisateur, améliorant ainsi les performances.
- L'authentification mise en place pour cette application utilise le **provider d'authentification de Symfony** via un formulaire de login (paramètre **form_login**) dont le nom de route est spécifié par le paramètre **login_path**. Ce formulaire est protégé par jeton CSRF grâce au service **token_manager**, dont la vérification est automatiquement réalisée par Symfony. Finalement, le **default_target_path** permet d'indiquer la route que laquelle seront redirigés les utilisateurs après succès de l'authentification ([Voir la documentation](#)). La configuration actuelle utilise les paramètres par défaut du composant Security, mais si les évolutions de l'application le nécessitent, il sera possible d'adapter la méthodologie par l'implémentation d'un **Guard Authenticator** personnalisé ([Voir la documentation](#)).
- Enfin, le paramètre **firewalls.main.logout** permet de définir le nom de route utilisé pour la déconnexion des utilisateurs. Par défaut les utilisateurs sont redirigés vers la page d'accueil, mais comme celle-ci n'est pas accessible aux utilisateurs non authentifiés, la redirection se fera vers la page de login.

Ici, l'authentification est réalisée basiquement sur le **username** et le **password**. Si les évolutions implémentent d'autres vérifications, comme par exemple la vérification de l'activation du compte par email, il sera nécessaire d'implémenter une vérification adaptée, via la création d'une classe **UserChecker** implémentant la **UserCheckerInterface** ([Voir la documentation](#)).

Autorisations

Alors que le système d'authentification est quasiment intégralement configuré au sein du fichier **security.yaml**, le système d'autorisations, à savoir la définition de restriction d'accès à certaines ressources en fonction des rôles utilisateurs (**ROLE_USER**, **ROLE_ADMIN**) peut être réalisée de différentes façons.

→ Les rôles

Par l'implémentation de l'interface **UserInterface**, la classe **User** possède une méthode **getRoles()** permettant à Symfony de récupérer lors de la connexion le/les rôle(s) de l'utilisateur, stocké sous forme d'un tableau en base de données.

Les différents rôles définis pour notre application sont les rôles utilisateur (**ROLE_USER**) et administrateur (**ROLE_ADMIN**). Symfony propose également d'autres "statuts" permettant de préciser certains accès: anonyme (**IS_AUTHENTICATED_ANONYMOUSLY**) ou encore authentifié (**IS_AUTHENTICATED_FULLY**).

Une hiérarchie peut être mise en place grâce au paramètre **role_hierarchy**, spécifiant dans notre cas que le rôle administrateur possède également le rôle utilisateur.

```
role_hierarchy:
  ROLE_ADMIN:
    - ROLE_USER
```

→ Les restrictions d'accès

Access control

Dans le cas de l'application ToDO&Co, une partie des règles de restriction d'accès sont définies dans le fichier **security.yaml** sous le paramètre **access_control**. Les règles décrites ci-dessous définissent les comportements suivants:

- L'url "/login" est accessibles aux utilisateurs non authentifiés
- Toutes les urls commençant par "/users" ne sont accessibles qu'aux utilisateurs ayant le rôle administrateur.
- L'ensemble des autres urls est accessible aux utilisateurs authentifiés ayant le rôle **ROLE_USER**.

```
access_control:
  - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
  - { path: ^/users, roles: ROLE_ADMIN }
  - { path: ^/, roles: ROLE_USER }
```

Annotation @IsGranted

Comme nous l'avons évoqué précédemment, toutes les routes liées à la gestion des utilisateurs possédant une url commençant par "/users/" ne sont accessibles qu'aux utilisateurs ayant le rôle administrateur. Cette configuration au sein du fichier **security.yaml** aurait également pu être mise en place au sein des contrôleurs grâce à l'annotation **IsGranted("ROLE_ADMIN")** du Sensio/FrameworkExtraBundle (Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted), soit directement au niveau des annotations globales de la classe **UserController**, soit au niveau des annotations des différentes méthodes dont on veut restreindre l'accès. Si certaines pages liées aux utilisateurs devaient devenir accessibles non plus aux seuls administrateurs, cette méthode serait plus adaptée à mettre en place.

Cette annotation est également prise en charge au sein des templates par le moteur de templating Twig, et a été mis en place notamment pour que les boutons de gestion des utilisateurs ("Créer un utilisateur", "Liste des utilisateurs") ne soient visibles que par les personnes possédant le rôle administrateur. ([En savoir plus](#))

```
<div class="row row-button">
  {% if is_granted("ROLE_ADMIN") %}
    <a href="{{ path('user_create') }}" class="btn btn-primary">Créer un utilisateur</a>
    <a href="{{ path('user_list') }}" class="btn btn-primary">Voir les utilisateurs</a>
  {% endif %}
</div>
```

Voters

Finalement, le 3e système proposé par le composant Security pour gérer les restrictions d'accès est le système de **Voters**, dont l'implémentation se fait au niveau des contrôleurs. Ce système a été mis en place pour restreindre la possibilité de supprimer une tâche à l'auteur de la tâche, et pour les tâches liées à "l'utilisateur anonyme", de restreindre leur suppression aux administrateurs. ([En savoir plus](#))

```
* @Route("/tasks/{id}/delete", name="task_delete")
* @IsGranted("TASK_DELETE", subject="task", statusCode=401)
*
* @return Response
*/
public function deleteTaskAction(Task $task)
{
}
```

Pour ce faire, une classe **TaskVoter** a été créée et implémente l'interface **VoterInterface**. Cette classe a pour objectif d'autoriser la suppression de la tâche seulement :

- Si l'utilisateur authentifié est administrateur et que la tâche en question est liée à un utilisateur "anonyme"
- Si l'utilisateur authentifié est bien l'auteur de la tâche

```
private function canDelete(Task $task, User $user)
{
    if ($this->security->isGranted('ROLE_ADMIN') && (null === $task->getAuthor())) {
        return true;
    }

    return $user === $task->getAuthor();
}
```

Pour en savoir plus :

- [Symfony Security Component](#)
- [Symfony authentication managment](#)
- [Symfony authorization managment](#)