DOCUMENTATION TECHNIQUE

Symfony

Projet 8 - To Do List

Implémentation de l'authentification
Symfony 5.4

Riso Nicolas 2023

Contexte

Ce guide a pour objectif d'expliciter le ***système d'authentification*** mis en place pour l'***application ToDoList***, 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 ***ToDoList*** 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.***

```yaml
security:
    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***.

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false

    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
        switch_user: true
```

- 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.

ci, 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)***.

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

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.

Les Restrictions d'Accès

Acces Control

Dans le cas de l'application ToDoList, 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***

```yaml
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 pû 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 différents 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. (Voir la documentation)

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

Voters

le dernier 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 controlleurs. 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. (Voir la documentation)

```
     * @Route("/tasks/{id}/delete", name="task_delete")
     * @IsGranted("TASK_DELETE", subject="task", statusCode=401)
     * @return Response
     */
    no usages   NicoRiso13 *
    public function deleteTaskAction(Task $task): Response
    {
```

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

```
1 usage   NicoRiso13
private function canDelete(Task $task, User $user): bool
{
    if ($this->security->isGranted( attributes: 'ROLE_ADMIN') && (null === $task->getAuthor())) {
        return true;
    }

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

Liens connexes

- Symfony Security Component
- Symfony Authentication Management
- Symfony Authorized Management

Processus de contribution

Clonage du Projet :

• Clonez le dépôt du projet à partir de la source principale.

• Assurez-vous de travailler à partir de la branche appropriée, généralement master ou une branche de fonctionnalité spécifique.

Mise en Place de l'Environnement de Développement :

• Installez les dépendances en utilisant Composer.

• Configurez les paramètres d'environnement nécessaires (connexion à la base de données, services tiers, etc.).

Création d'une Branche de Fonctionnalité :

• Pour chaque nouvelle fonctionnalité ou correction, créez une nouvelle branche à partir de la branche de base (master généralement).

• Nommez la branche de manière descriptive et en suivant une convention, par exemple : feature/nouvelle-fonctionnalite ou bugfix/correction-bug-x.

Développement :

• Suivez les principes de conception de Symfony et les bonnes pratiques de développement.

• Écrivez des tests unitaires et fonctionnels pour couvrir les nouvelles fonctionnalités et les corrections.

• Respectez les standards de codage de Symfony et utilisez les normes PSR lorsque cela est applicable.

Validation Locale :

• Avant de soumettre une demande de fusion (Pull Request), assurez-vous que votre code passe les tests locaux et les outils d'analyse statique.

Demande de Fusion (Pull Request) :

• Soumettez une demande de fusion depuis votre branche de fonctionnalité vers la branche cible (généralement master).

• Fournissez une description claire et concise de ce que fait la modification.

• Vérifiez que les tests automatisés sont réussis sur votre demande de fusion.

• Effectuez une revue de code approfondie pour repérer les erreurs et les améliorations potentielles.

Processus de qualité

Tests Automatisés :

Toute nouvelle fonctionnalité doit être accompagnée de tests unitaires et/ou fonctionnels.

Assurez-vous que les tests existants ne sont pas cassés par vos modifications.

Analyse Statique du Code :

Utilisez des outils d'analyse statique tels que *Codacy* pour garantir que le code suit les normes de codage.

Documentation :

Mettez à jour la documentation du projet pour refléter les nouvelles fonctionnalités ou les changements apportés.

Les commentaires dans le code doivent être clairs et explicites.

Performance :

Évaluez l'impact de vos modifications sur les performances de l'application.

Évitez les requêtes inutiles à la base de données et optimisez les traitements si nécessaire.

Règles de qualité et bonnes pratiques

Cohérence :

Suivez les conventions de nommage et de structure du projet existant.

Modularité :

Favorisez la création de modules réutilisables.

Simplicité :

Évitez la complexité inutile. Privilégiez la simplicité et la lisibilité du code.

Sécurité :

Respectez les meilleures pratiques de sécurité de Symfony et évitez les vulnérabilités connues.

Commentaires :

Commentez votre code lorsque cela est nécessaire pour expliquer des choix complexes ou des solutions non évidente.

En suivant ce guide, vous contribuerez de manière efficace et cohérente au projet Symfony 3.4 tout en maintenant des normes de qualité élevées. Assurez-vous de toujours consulter la documentation officielle de Symfony pour les dernières informations et les meilleures pratiques.