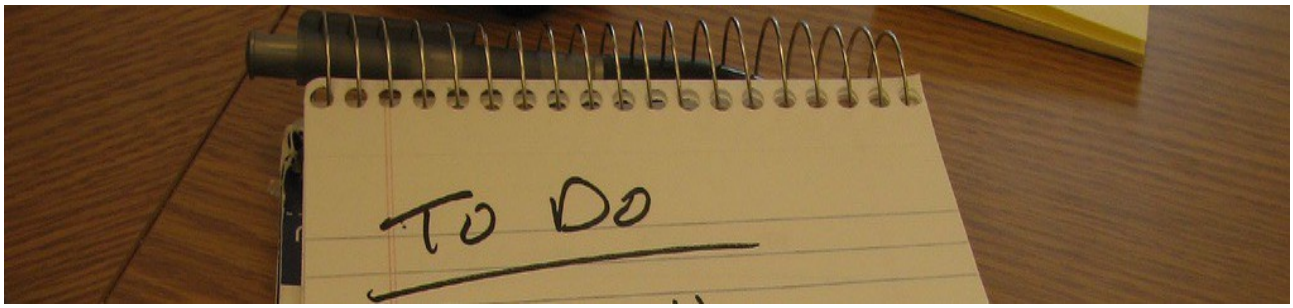

L'AUTHENTIFICATION

Application Web ToDoList

Réalisé par Gallot Benjamin
Développeur d'Application Web PHP/Symfony

SOMMAIRE

INTRODUCTION

Le composant Sécurité Symfony fournit un système de sécurité complet pour votre application Web. Il est livré avec des fonctionnalités d'authentification avec l'authentification de base HTTP, mais vous permet également d'implémenter vos propres stratégies d'authentification.

Nous allons parler des deux étapes importantes dans la sécurité d'une application web. C'est l'authentification d'un utilisateur et les autorisations d'accès à cette application qui en découlent. L'objectif est de vous expliquer à quel niveau s'opère l'authentification avec le framework Symfony. Vous serez sur quels fichiers il faut apporter des modifications afin d'avoir des utilisateurs enregistrés en base de données qui sont membres de l'application web ToDoList et qui ont des rôles spécifiques.

USER INTERFACE

Nous créons une classe User dans le dossier Entity où nous définissons plusieurs propriétés comme le nom d'utilisateur et le mot de passe par exemple. Cette classe doit implémenter l'interface UserInterface. Cela va permettre à Symfony de traiter l'objet User pour les différentes actions liées à la sécurité comme l'authentification. Symfony va pouvoir récupérer les identifiants de l'utilisateur par exemple.

```
<?php

namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\\Validator\Constraints as Assert;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Table("user")
 * @ORM\Entity
 * @UniqueEntity("email")
 */
class User implements UserInterface
{
    private $id;

    private $username;

    private $password;

    # suite du code…
}
```

CONFIGURATION-Security.yml

Le système de sécurité est configuré dans app/config/security.yml. C'est dans ce fichier que nous allons faire les modifications afin de configurer l'authentification.

Encoders

Vous permets de choisir le type d'encodage pour le mot de passe de l'utilisateur.

```
security:
    encoders:
        AppBundle\Entity\User: bcrypt
```

Providers

Le provider vous permet de définir où vous voulez obtenir vos utilisateurs.

```
providers:
        doctrine:
            entity:
                class: AppBundle:User
                property: username
#Ici l'utilisateur est chargé par la propriété username.
```

Firewalls

Le firewall vous permet de définir plusieurs pare-feu que nous souhaitons utiliser pour sécuriser notre application web. Il permet de déclarer un contexte dans lequel certaines routes sont publiques ou privées. Cela lui permet de gérer de manière indépendante la page d'authentification, les redirections, la gestion de la session…

Voici la configuration pour notre application.

```
firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
#Le pare-feu dev n'est pas important. Cela garantit que l'outil de développement symfony
#fonctionne correctement.
        main:   #Pare-feu principal
```

```
            anonymous: ~    #Si non connecté : authentifié comme anonyme
            pattern: ^/
            form_login:    #Création du formulaire de connexion
                login_path: login  #Route pour accéder au formulaire qui sera appelé par
                                   #LoginAction dans le controller.
                check_path: login_check #Le pare-feu intercepte et transforme automatiquement tout
                                        #formulaire soumis en l'url `/ login_check`
                always_use_default_target_path:  true    #Ignore l'url précédemment demandée
                default_target_path:  /     #redirige vers la page par défaut
            logout: ~    #Par défaut logout n'est pas configurée. Lorsque les utilisateurs se
                         #déconnectent  leurs sessions deviennent invalides.
```

La méthode loginAction qui retourne la vue du formulaire.

```
class SecurityController extends Controller
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(Request $request)
    {
        return $this->render('security/login.html.twig', array(
            'last_username' => $this->get('security.authentication_utils')->getLastUsername(),
            'error'         => $this->get('security.authentication_utils')->getLastAuthenticationError(),
        ));
    }
```

Role_Hierarchy

Au lieu d'associer plusieurs rôles aux utilisateurs, vous pouvez définir des règles d'héritage de rôles en créant une hiérarchie de rôles:

```
role_hierarchy:
        ROLE_USER: ROLE_USER
        ROLE_ADMIN: [ROLE_USER, ROLE_ADMIN]
```

Nous avons défini ici deux rôles. Les utilisateurs qui auront le rôle Role_Admin auront également le rôle Role_User.

SECURITÉ-Gestion des Autorisations

Access_control

Access_control permet de sécuriser des Urls.

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

Dans notre application, nous autorisons tous les utilisateurs à accéder à la route /login afin que tout le monde puissent se connecter.

Nous décidons que la gestion des utilisateurs n'est possible que pour les membres ayant le rôle Admin et que les membres ayant le rôle User peuvent accéder aux autres fonctionnalités de notre application web. L'ordre de déclaration des contrôles est très important.

@Security

Pour plus de sécurité, nous avons rajouté une annotation de sécurité dans le UserController.

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
/**
 * @Security("has_role('ROLE_ADMIN')")
 */
class UserController extends Controller
```

Ici toutes les méthodes du UserController sont accessibles uniquement par les utilisateurs ayant le rôle Admin.

Voter

Nous pouvons améliorer la sécurité en utilisant les électeurs de sécurité (Voters). Ils vont permettre aux utilisateurs d'accéder à des fonctionnalités précises en fonction de leurs rôles. Par exemple pour l'application ToDoList, il a été demandé de que les tâches rattachées à l'utilisateur “anonyme” ne peuvent être supprimées uniquement par les

utilisateurs ayant le rôle administrateur (*ROLE_ADMIN*).

Nous avons donc créé un service TaskVoter.php où des permissions ont été définies.

La décision que l'utilisateur puisse réaliser telle action sera prise par une instance de AccessDecisionManagerInterface dans le controller.

```
/**
     * @Route("/tasks/{id}/delete", name="task_delete")
     */
    public function deleteTaskAction(DeleteTaskHandler $handler, Task $task)
    {
        $this->denyAccessUnlessGranted('delete', $task);
        return $handler->handle($task);
    }
#denyAccessUnlessGranted () dans le contrôleur va utiliser le service TaskVoter que nous avons créé.
```

CONCLUSION

Ce document doit vous permettre de comprendre comment l'authentification est gérée pour l'application web ToDoList. Vous êtes désormais en capacité d'amener des modifications à cette application dans le cas d'évolutions futurs. Vous pouvez par exemple ajouter de nouveaux rôles, les sécuriser ou créer des permissions sur des actions spécifiques. Si vous souhaitez approfondir le sujet, vous trouverez les liens nécessaires dans la rubrique Webographie.

WEBOGRAPHIE

https://symfony.com/doc/current/components/security.html

https://symfony.com/doc/current/security/custom_provider.html

https://symfony.com/doc/current/security.html

http://symfony.com/doc/current/security/access_control.html#security-access-control-enforcement-options

https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/security.html

http://symfony.com/doc/current/security/voters.html