Le 20 Juillet 2017
par Antoine Bernay

Todo&co

Implémenter une mécanique d’authentification

sur Symfony 3

Menu

Introduction

L’authentification dans Symfony peut se faire de différentes manières. Dans ce document, nous aborderons l’une de ces manières en passant par le mécanisme d’authentification par défaut de Symfony.

Nous allons apprendre à créer un utilisateur descendant de la UserInterface configurer notre application pour l’utiliser, sécuriser nos routes et faire nos formulaires d’inscription et de connexion.

Un court chapitre additionnel sera par ailleurs dédié à l’autorisation.

Créons notre classe User

Pour être reconnue par l’authenticator de Symfony (la classe qui gère et contrôle l’authentification), notre classe User va devoir étendre la UserInterface. Ajoutons-lui les champs $id, $username et $password, les getters, setters et implémentons les méthodes de la UserInterface : getRoles(), getPassword(), getSalt(), getUsername() et eraseCredentials(). Détaillons à quoi servent ces méthodes :

- **getRoles()** : doit retourner un tableau contenant les rôles de l’utilisateur. Nous n’utilisons pas encore les rôles, donc getRoles() retournera ["ROLE_USER"]. ROLE_USER est le rôle par défaut d’un utilisateur authentifié. Nous verrons cela plus en détail dans le chapitre sur la configuration de notre application.

- **getPassword()** : doit retourner le mot de passe de notre utilisateur.

- **getSalt()** (optionnel): retourne le salt. Il s’agit d’une chaîne de caractères utilisée dans certaines méthodes de hashage. Nous allons utiliser bcryt, une de ces méthodes, qui n’a pas besoin de salt donc pas besoin de retourner quoi que ce soit avec getSalt().

- **getUsername()** : doit retourner, non pas forcément le nom d’utilisateur, mais le champ qu’il va devoir utiliser pour se connecter (en plus du mot de passe). C’est généralement soit un nom d’utilisateur, soit une adresse email.

- **eraseCredentials() (optionnel)** : utile si à un moment donné, il y a un risque de des données sensibles soient enregistrées en base de données (comme le mot de passe en clair). Nous n’en aurons pas besoin.

Et voila ! Notre classe User est prête à l’emploi ! Pensez à mettre à jour le schéma de la base de données (./bin/console doctrine:schema:update par défaut).

Configurons notre application

Ouvrez le fichier app/config/security.yml.

C'est ici que nous allons spécifier comment notre application doit gérer l'authentification.

Pour ce faire, il va nous falloir lui dire quelle classe est utilisée pour authentifier un utilisateur. Ajoutez-donc lui les lignes suivantes :

```
    providers:
        doctrine:
            entity:
                class: AppBundle:User
                property: username
```

Elles signifient que nous utilisons Doctrine, que nous utilisons une entité pour l'authentification, que cette entité est la classe AppBundle\Entity\User (Symfony va chercher directement dans le dossier Entity) et que la propriété qui nous sert à retrouver notre utilisateur est username (c'est la même que nous retournons avec getUsername()).

Ensuite, nous avons besoin de spécifier par quel moyen nous allons hasher le mot de passe. Ajoutez les lignes suivantes :

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

L'algorithme utilisé est bcrypt. C'est un puissant algorithme fortement résilient face aux attaques de type brute force. Si vous voulez en savoir plus, une page wikipédia lui est consacrée : https://fr.wikipedia.org/wiki/Bcrypt.

La configuration suivant est un peu plus verbeuse mais tout aussi simple. Elle doit se trouver sous la clé firewalls :

```
        main:
            anonymous: ~
            pattern: ^/
            form_login:
                login_path: login
                check_path: login_check
                always_use_default_target_path:  true
                require_previous_session: false
                default_target_path:  /
            logout: ~
```

Ici, nous spécifions notre firewall. Sous form_login, nous lui indiquons :

- **login_path** : la route utilisée pour s'authentifier (tout utilisateur non authentifié et tentant de se rejoindre une page protégée se verra redirigé vers cette route).

- **check_path** : la route que Symfony utilisera pour authentifier l'utilisateur une fois le formulaire de connexion soumis. Nous n'avons pas besoin de coder cette logique car Symfony la prend entièrement en charge.

- **always_use_default_target_path** : par défaut, Symfony redirige un utilisateur fraîchement authentifié vers la dernière page qu'il consultait. Cette option permet de contourner cette règle et de rediriger l'utilisateur authentifié vers la route spécifié dans default_target_path.

- **require_previous_session** : utilisée dans les tests fonctionnels et unitaires, cette option permet de valider une authentification sans qu'une session n'ait été créée.

- **default_target_path** : spécifie l'url de redirection de l'utilisateur fraîchement authentifié (si always_use_default_target_path est à true).

Pour en savoir plus sur ces paramètres, consultez la documentation de Symfony à cette adresse : http://symfony.com/doc/current/security/form_login.html

Ici nous ne spécifions pas d'informations sur la déconnexion et laissons Symfony la gérer entièrement.

Votre fichier security.yml devrait ressembler à ça :

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

    providers:
        doctrine:
            entity:
                class: AppBundle:User
                property: username

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: ~
            pattern: ^/
            form_login:
                login_path: login
                check_path: login_check
                always_use_default_target_path:  true
                require_previous_session: false
                default_target_path:  /
            logout: ~
```

Sécurisons nos routes

Il est temps de dire à Symfony quelles routes sont visitables lorsqu'on n'est pas authentifié et lesquelles le sont !

Restez dans votre fichier security.yml. C'est ici que ça se passe.

Nous allons lui ajouter les lignes suivantes :

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

Ici, nous spécifions pour chaque chemin (path) le ou les rôles autorisés.

- path : c'est une regex. Dans notre exemple ci-dessus, tous les urls commençant par /login seront visitables par les utilisateurs possédant le « rôle » IS_AUTHENTICATED_ANONYMOUSLY qui n'est autre qu'un utilisateur non authentifié. Tandis que le tout autre url commençant par / n'est visitable que par les utilisateurs possédant le rôle ROLE_USER.

- roles : vous pouvez spécifier un ou plusieurs rôles. Jusque là, nous n'avons spécifié qu'un seul rôle dédié aux utilisateurs authentifiés : ROLE_USER. Cependant, Symfony comporte deux rôles par défaut :

    - IS_AUTHENTICATED_ANONYMOUSLY : il s'agit des utilisateurs non authentifiés (et donc anonymes).

    - IS_AUTHENTICATED_FULLY: il s'agit des utilisateurs authentifiés en général

Attention ! L'ordre de déclaration des règles d'accès a son importance car Symfony les lira de haut en bas jusqu'à trouver un path qui correspond à l'url courante !

Et voila ! Nos routes sont sécurisées ! Désormais, tout utilisateur non authentifié tentant d'accéder à autre chose que /login y sera redirigé.

Créons notre formulaire d'inscription

Ce document n'ayant pas pour but d'enseigner la création et gestion des formulaires sur Symfony, les classes suivantes ne seront pas détaillées. Si vous ne maîtrisez la création de formulaire et la persistance d'entités avec Doctrine, voici quelques liens pour les apprendre :

- créer un formulaire à l'aide du composant Form : http://symfony.com/doc/current/forms.html#creating-form-classes

- persister une entité avec Doctrine : http://symfony.com/doc/current/doctrine.html#persisting-objects-to-the-database

Créons notre UserType :

```php
<?php

namespace AppBundle\Form;

use AppBundle\Entity\User;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;

class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('username', TextType::class, ['label' => "Nom d'utilisateur"])
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
                'invalid_message' => 'Les deux mots de passe doivent
correspondre.',
                'required' => true,
                'first_options'  => ['label' => 'Mot de passe'],
                'second_options' => ['label' => 'Tapez le mot de passe à
nouveau'],
            ])
        ;
    }
}
```

Notre méthode de contrôleur :

```
/**
 * @Route("/register", name="register")
 */
public function createAction(Request $request)
{
    $user = new User();

    $form = $this->createForm(UserType::class, $user);
    $formHandler = new UserFormHandler($this->get("router"), $this-
>get("doctrine.orm.entity_manager"), $this→get("security.password_encoder"));
    $response = $formHandler->handle($form, $request);

    if ($response instanceof RedirectResponse) {
        $this->addFlash('success', "L'utilisateur a bien été ajouté.");
        return $response;
    }

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

Ce qui change de nos formulaires classiques, cette fois, est l'injection du service security.password_encoder dans notre FormHandler.

Si vous ne savez pas à quelle classe correspond tel ou tel services, ou quels services sont accessibles, ouvrez votre terminal à la racine du projet et entrez ./bin/console debug:container.

Puis notre gestionnaire de formulaires :

```php
<?php

namespace AppBundle\Form\Handler;

use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\RouterInterface;
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;

class UserFormHandler
{
    /**
     * @var RouterInterface
     */
    private $router;
    /**
     * @var EntityManagerInterface
     */
    private $em;
    /**
     * @var UserPasswordEncoderInterface
     */
    private $passwordEncoder;

    public function __construct(RouterInterface $router, EntityManagerInterface
$em, UserPasswordEncoderInterface $passwordEncoder)
    {
        $this->router = $router;
        $this->em = $em;
        $this->passwordEncoder = $passwordEncoder;
    }

    public function handle(FormInterface $form, Request $request)
    {
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $user = $form->getData();
            $password = $this->passwordEncoder->encodePassword($user, $user-
>getPassword());
            $user->setPassword($password);

            $this->em->persist($user);
            $this->em->flush();

            return new RedirectResponse($this->router->generate('login'));
        }
        return $form;
    }
}
```

Ce qui rend notre FormHandler différent des autres est l'utilisation du UserPasswordEncoder (injecté depuis le contrôleur).
Nous l'utilisons dans la méthode handle pour encoder le password. Dans les coulisses, le UserPasswordEncoder va regarder quel algorithme de hashage utiliser. Il va voir dans notre config qu'il faut utiliser bcypt et dans notre entité, qu'il n'a pas besoin de salt (car getSalt() ne retourne rien).

Enfin, nous persistons l'entité et retournons une RedirectResponse vers la page de connexion.

Et enfin notre vue :

```
{% extends base.html.twig' %}

{% block header_title %}<h1>Créer un utilisateur</h1>{% endblock %}

{% block header_img %}{% endblock %}

{% block body %}
    <div class="row">
        {{ form_start(form, {'action' : path('register')}) }}
            {{ form_widget(form) }}
            <button type="submit" class="btn btn-success pull-
right">Ajouter</button>
        {{ form_end(form) }}
    </div>
{% endblock %}
```

Créons notre formulaire de connexion

Contrairement à l'inscription, la connexion laisse Symfony gérer bien plus de logique en coulisse.

Créons notre SecurityController :

```php
<?php

namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class SecurityController extends Controller
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(Request $request)
    {
        $authenticationUtils = $this->get('security.authentication_utils');
        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render('security/login.html.twig', array(
            'last_username' => $lastUsername,
            'error'         => $error,
        ));
    }

    /**
     * @Route("/login_check", name="login_check")
     */
    public function loginCheck()
    {
        // This code is never executed.
    }

    /**
     * @Route("/logout", name="logout")
     */
    public function logoutCheck()
    {
        // This code is never executed.
    }
}
```

Pas de panique, détaillons pas à pas ce qui s'y passe :

- la loginAction est ce que nous utiliserons pour renvoyer le formulaire de connexion. Le service security.authentication_utils permet de récupérer non seulement les erreurs lors de la soumission du

formulaire (telle que « cet utilisateur n’existe pas »), mais aussi le nom d’utilisateur entré afin de préremplir le champ en cas de besoin.

- loginCheck et logoutCheck sont là pour créer les routes mais ne seront jamais appelées car Symfony gèrera lui-même la validation de la connexion et la déconnexion.

Ajoutons la vue :

```
{% extends 'base.html.twig' %}

{% block body %}

    {% if error %}
        <div class="alert alert-danger" role="alert">{{ error.messageKey|
trans(error.messageData, 'security') }}</div>
    {% endif %}

    <form action="{{ path('login_check') }}" method="post">
        <label for="username">Nom d'utilisateur :</label>
        <input type="text" id="username" name="_username"
value="{{ last_username }}" />
        <label for="password">Mot de passe :</label>
        <input type="password" id="password" name="_password" />
        <button class="btn btn-success" type="submit">Se connecter</button>
    </form>

{% endblock %}
```

Prenez bien soin de spécifier le chemin vers login_check comme cible du formulaire, ainsi que l’erreur et le nom d’utilisateur entré si le formulaire a retourné une erreur.

Et c’est tout !

Symfony gère le reste en coulisse. C’est là que notre configuration dans security.yml entre en scène. Elle va permettre à Symfony de savoir quelle entité récupérer, sur quel propriété se baser, sur quelle route faire la validation, vers quelle route rediriger en cas d’erreur et vers quelle route rediriger en cas de succès.

Conclusion

Nous avons vu les différentes étapes permettant la mise en place d'une mécanique d'authentification sur Symfony 3.

La méthode que nous avons utilisée est probablement la plus simple. Nous aurions pu tout aussi bien passer par le Guard et créer notre propre authenticator (https://symfony.com/doc/current/security/guard_authentication.html) ou passer par le FOSUserBundle. Pour en savoir plus, n'hésitez pas à consulter la documentation complète à l'adresse suivante : https://symfony.com/doc/current/security.html.

Cette documentation vous est fournie par Antoine Bernay, développement d'applications PHP&Symfony.

Chapitre additionnel : l'Autorisation

Si l'authentification consiste à vérifier qui est l'utilisateur, l'autorisation vérifie, elle, ce qu'il a le droit de faire.

Pour ce faire, Symfony utilise les rôles. Vous vous rappelez ? Cette méthode getRoles() de la UserInterface.

La première chose à faire pour intégrer une mécanique d'autorisation est d'ajouter un champ à notre classe User : $role, et de faire en sorte que getRoles retourne [$this→role].

Ensuite, rendez-vous dans votre fichier security.yml pour y définir la hiérarchie de vos rôles :

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

La hiérarchie des rôles consiste à dire quel rôle comprend tel autre rôle. Ici, notre ROLE_ADMIN est aussi un ROLE_USER, ce qui fait qu'il aura tous les droits d'un ROLE_USER en plus des siens.

Ensuite, mettez à jour votre UserFormHandler pour faire en sorte que le ROLE_USER soit attribué à l'utilisateur juste avant sa persistance, lors de son inscription :

```
$user->setRole('ROLE_USER');
```

Enfin, vous avez deux moyens de contrôler l'accès à une route :

- soit vous ajoutez une règle à votre access_control en spécifiant votre nouveau rôle.

- soit vous ajoutez une annotation à votre méthode de contrôleur, comme tel :

```
/**
 * @Route("/users/{id}/edit", name="user_edit")
 * @Security("is_granted('ROLE_ADMIN')")
 */
public function editAction(User $user, Request $request)
{
    // De la logique
}
```

Et voila ! Vous avez mis en place une mécanique d'autorisation.

Pour plus d'informations, rendez-vous sur la documentation de Symfony : http://symfony.com/doc/current/components/security/authorization.html#roles.