ToDo&Co

Documentation technique: l'implementation de l'authentification (Symfony 6)

Auteur: Ching Yi, Pelgrims Chen
Dernière mise à jour: 14/03/2022
Version:1.0

Sommaire

Contexte

à quoi sert ce document ?

Ce document explique comment l'authentification fonctionne dans le framework Symfony (Symfony version : 6) et comment elle est appliquée dans notre application: ToDo&Co. L'objectif de ce guide est d'aider tous ceux qui doivent modifier l'application en termes d'authentification à l'avenir. Il est également utile à ceux qui souhaitent mieux comprendre l'authentification dans le framework Symfony avec un exemple concret.

Avec ce guide, vous pourriez :

- comprendre quel(s) fichier(s) il faut modifier et pourquoi ;
- comment s'opère l'authentification ;
- et où sont stockés les utilisateurs.

Introduction

Qu'est-ce que l'authentification ?

Elle permet de poser la question "Qui êtes-vous ?" de plusieurs manières. Autrement dit, lorsqu'un utilisateur fait une certaine demande sur l'application, l'application va vérifier s'il est autorisé . La demande n'est-elle accessible qu'à l'utilisateur connecté ? Si l'utilisateur est connecté, quel rôle a-t-il ? La demande n'est-elle accessible qu'à certains rôles ?

le document l'expliquera sous 3 aspects :

- le processus de connexion
- recherche de l'utilisateur
- autorisation

Security.yaml :

le fichier de configuration de la sécurité

Security.yaml est le fichier de configuration de la sécurité de l'application. Dans certains cas, la configuration est générée automatiquement (vous le comprendrez mieux dans les chapitres suivants), cependant il est important de la comprendre afin de pouvoir la modifier si nécessaire.

Security (ligne 1):

enable_authenticator_manager : true

Il doit être mis à ‘true’ afin d'autoriser la fonction d'authentification dans l'application.

password_hashers

Il permet de hacher le mot de passe avant de l'enregistrer dans la base de données et il permet à l'application de "traduire" le mot de passe renseigné pendant le processus de connexion et de vérifier s'il correspond au mot de passe haché.

Providers (ligne7):

class: App/Entity/User

Le ‘class’ est utilisé pour charger les utilisateurs de la base de données sur la base d'un "user identifier".

property: username

La configuration ci-dessus utilise Doctrine pour charger l'entité User en utilisant la propriété 'username' comme "user identifier". Dans notre cas, l'utilisateur de l'application ‘ToDo&Co’ utilise son nom d'utilisateur (pas son adresse e-mail) et son mot de passe pour se connecter.

Main (ligne 17):

Provider: app_user_provider

les données fournies par “app_user_provider"(voir ligne 9) sont utilisées pour vérifier si l'utilisateur est authentifié.

form_login

Il existe plusieurs façons pour le système d'essayer de trouver un utilisateur correspondant au visiteur de la page web. ‘ToDo&Co' utilise l'authentificateur de formulaire de connexion (form_login), c'est-à-dire un moyen traditionnel de fournir un formulaire de connexion où les utilisateurs s'authentifient en utilisant un identifiant (nom d'utilisateur dans notre cas) et un mot de passe.

login_path:login

check_path: login

La route nommée 'login' (voir ici) est utilisée pour la connexion et vérifie si l'utilisateur est connecté avec succès.

enable_csrf: true

il est défini comme "true", de sorte que nous pouvons ajouter des jetons CSRF cachés dans le formulaire de connexion afin d'empêcher les attaques CSRF de connexion.(voir ici)

logout:

  path: app_logout

La route nommée 'app_logout' est utilisée pour la déconnexion.(voir ici)

Access_control (ligne 27):
Nous le configurons pour contrôler l'accès à certaines URL et le limiter au rôle utilisateur (ROLE_USER).

Pour des cas plus complexes, par exemple, dans notre cas, le contenu de la page de 'consultation des utilisateurs' est affiché différemment en fonction du rôle de l'utilisateur, il est configuré dans le dossier 'Controller'. (voir ici)

Voter est également utilisé dans certaines URL (dans notre cas, pour la modification et la suppression de tâches) lorsque le rôle de l'utilisateur n'est pas le seul critère permettant de déterminer si l'utilisateur est autorisé à y accéder. (voir ici)

Role_hierarchy (ligne 31):
`ROLE_AMIN: ROLE_USER`
Ce paramètre spécifie dans notre cas que le rôle administrateur possède aussi le rôle utilisateur.

Quel est le processus de connexion ?

User.php : création de l'utilisateur

Les permissions dans Symfony sont toujours liées à un objet utilisateur, donc une classe utilisateur doit être créée pour 'ToDo&Co'. Il s'agit d'une classe qui implémente 'UserInterface'. Dans notre cas, elle est aussi une entité Doctrine, ,qui nous permet de communiquer avec la base de données en ce qui concerne un utilisateur.

Comment le créer?
Grâce au Maker Bundle, il suffit de lancer les commandes :
`php bin/console make:user`

Les réponses suivantes ont été données lors de la création de l'"Entity: user":

```
Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
 > yes

 Enter a property name that will be the unique "display" name for the user (e.g
email, username, uuid) [email]:
 > username
Does this app need to hash/check user passwords? (yes/no) [yes]:
 > yes
```

l'opération nous permet de modifier le security.yaml (e.g. providers, password_hasher) et de créer Entity/User.php avec des propriétés et méthodes et correspondant automatiquement à la réponse ci-dessus.

Dans le fichier Entity/User.php, on peut trouver:

Private $roles= [' ROLE_USER']

```
32      #[ORM\Column(type: 'json')]
33      #[Assert\NotBlank( message: 'Vous devez choisir un rôle.')]
34      private $roles = ['ROLE_USER'];
35
82      /**
83       * @see UserInterface
84       */
85      public function getRoles(): array
86      {
87          $roles = $this->roles;
88          // guarantee every user at least has ROLE_USER
89
90          return array_unique($roles);
91      }
92
93      public function setRoles(array $roles): self
94      {
95          $this->roles = $roles;
96
97          return $this;
98      }
```

Il est nécessaire de garantir que chaque utilisateur possède au moins ROLE_USER. Dans notre cas, chaque utilisateur est soit ROLE_USER soit ROLE_ADMIN .

Public function getUserIdentifier()

```
71
72      /**
73       * A visual identifier that represents this user.
74       *
75       * @see UserInterface
76       */
77      public function getUserIdentifier(): string
78      {
79          return (string) $this->username;
80      }
```

Le nom d'utilisateur est utilisé pour représenter un utilisateur grâce à cette fonction.

PasswordAuthenticaterUserInterface

```
100     /**
101      * @see PasswordAuthenticatedUserInterface
102      */
103     public function getPassword(): string
104     {
105         return $this->password;
106     }
107
108     public function setPassword(string $password): self
109     {
110         $this->password = $password;
111
112         return $this;
113     }
114
```

L'interface :PasswordAuthenticaterUserInterface est implémentée afin que getPassword() puisse vérifier le mot de passe de l'utilisateur pour voir si le mot de passe correspond au mot de passe haché.

Private $plainPassword

```
36      #[ORM\Column(type: 'string', length: 64)]
37      #[Assert\Length(min: 6, max: 64)]
38      private $password;
39
40      private $plainPassword;
41
```

La propriété $plainPassword est crée et utilisée uniquement pour obtenir le mot de passe du formulaire d'inscription et n'est pas enregistrée dans la base de données. Pour des raisons de sécurité, la propriété $password sert à enregistrer le mot de passe haché dans la base de données. La propriété $plainPassword est utilisée pour obtenir le mot de passe du formulaire d'inscription afin que l'application puisse effectuer le processus de hachage avant de le sauvegarder.

UserService.php : la service pour hacher le mote de passe

```php
namespace App\Service;

use App\Entity\User;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasher;

class UserService
{

    private $userPasswordHasher;
    private $entityManager;
    public function __construct(UserPasswordHasherInterface $userPasswordHasher, EntityManagerInterface $entityManager)
    {
        $this->userPasswordHasher = $userPasswordHasher;
        $this->entityManager = $entityManager;
    }
    public function saveUser($plainpassword, $user)
    {
        $user->setPassword(
            $this->userPasswordHasher->hashPassword(
                $user,
                $plainpassword
            )
        );
        $this->entityManager->persist($user);
        $this->entityManager->flush();

    }
```

Nous avons créé un service et utilisé l'injection de dépendance pour injecter 'UserPasswordHasherInterface' afin que $plainPassword puisse être haché avant d'être enregistré dans la base de données.

LoginController.php : Gérer la formulaire de connexion

LoginController permet à retourne le formulaire de connexion et permet à l'authentificateur form_login de gérer la soumission du formulaire automatiquement.

Comment le créer?
Grâce au Maker Bundle, il suffit de lancer les commandes :
php bin/console make:controller Login

src > Controller > LoginController.php > ...

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class LoginController extends AbstractController
{
    #[Route('/login', name: 'login')]
    public function index(AuthenticationUtils $authenticationUtils): Response
    {
        if ($this->getUser()) {
            return $this->redirectToRoute('homepage');
        }
        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render('default/index.html.twig', [
            'controller_name' => 'LoginController',
            'last_username' => $lastUsername,
            'error'=> $error,
        ]);
```

Avec le paramètre dans la route normée ‘login’, il stocke l'erreur lorsque le mot de passe/nom d'utilisateur rempli est incorrect. Dans notre cas, lorsqu'une telle erreur se produit, elle déclenche le message flash du template (login/login.html.twig qui est inclus dans default/index.html.twig). En outre, lorsqu'un utilisateur non authentifié tente d'accéder à un accès protégé, il sera redirigé vers cette route.

l'opération nous permet aussi de modifier le security.yaml (e.g. form_login) automatiquement.

login.html.twig : le template du formulaire de connexion

templates > login > login.html.twig

```twig
    <form action="{{ path('login') }}" method="post">
        <label for="username">Nom d'utilisateur:</label>
        <input type="text" id="username" name="_username" {% if last_username is defined %}value="{{ last_username }}" {% endif %} requied/>

        <label for="inputPassword">Mot de passe:</label>
        <input type="password" id="password" name="_password" required/>

        <input type="hidden" name="_target_path" value="/"/>
        <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">

        <button class="btn btn-sm btn-success"  type="submit">Se connecter</button>
    </form>
{% endblock %}
```

<input type=”hidden” name=”_target_path” value =”/”>
il permet à l'application d'être redirigée vers l'URL "/" lorsque la connexion est réussie.

<input type="hidden" name="_csrf_token" value={{ csrf_token ("authenticate")}}">
Il est ajouté afin d'empêcher les attaques CSRF de connexion.

Comment l'application sait-elle si l'utilisateur est connecté et quel est son rôle ?

Controller : $this->getUser()

La méthode $this->getUser() ne peut être utilisée que par le contrôleur. $this->getUser() permet à l'application de savoir si un utilisateur est connecté. $this->getUser()->getRoles() permet à l'application de connaître le rôle de l'utilisateur connecté.

Voici un exemple tiré de notre application

src > Controller > UserController.php > ...

```
59
60      #[Route('/users/{id}/edit', name: 'user_edit')]
61      #[IsGranted('ROLE_USER')]
62      public function editAction(User $user, Request $request)
63      {
64          $current_user=$this->getUser();
65          if ($current_user->getRoles()!==['ROLE_ADMIN']) {
66              if ($user!==$current_user){
67          $this->addFlash('error', "Désolé. Vous n'avez pas le droit d'y accéder.");
68          return $this->redirectToRoute('homepage');
```

Service : Security Service

Symfony\Component\Security\Core\Security peut être utilisé dans un service par injection de dépendances.

Voici un exemple tiré de notre application

src > Security > Voter > TaskVoter.php > TaskVoter

```php
class TaskVoter extends Voter
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    protected function supports(string $attribute, $subject): bool
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, ['TASK_EDIT', 'TASK_DELETE'])
            && $subject instanceof \App\Entity\Task;
    }

    protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
    {
        $user = $token->getUser();
        // if the user is anonymous, do not grant access
        if (!$user instanceof UserInterface) {
            return false;
        }

        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case 'TASK_EDIT':
              if($user===$subject->getAuthor()){
                  return true;
                }
               if($subject->getAuthorName()==='anonyme' && $this->security->isGranted('ROLE_ADMIN')){
```

Dans notre cas, $this->security->getUser() permet le service 'Voter' de savoir si un utilisateur est connecté. $this->security->isGranted('ROLE_ADMIN') est pour savoir si l'utilisateur a le role de 'ROLE_ADMIN'.

Template:

le template est écrit dans le langage de template Twig. app.user est utilisé dans notre template pour que le template soit modifié selon que l'utilisateur est connecté ou non.

```twig
        {% if app.user %}
          <a href="{{ path('app_logout') }}" class=" btn btn-danger m-1">
            Se déconnecter
          </a>
        {% endif %}

        {% if not app.user and 'homepage' != app.request.attributes.get('_route') %}
          <a href="{{ path('login') }}" class="btn btn-success">Se connecter</a>
        {% endif %}
        </div>

      <div class="row">
        <div class="col-md-12">
```

Contrôle d'accès ? L'utilisateur avec son rôle spécifique est-il autorisé à faire une telle demande ?

il y a plusieurs façons de configurer l'autorisation, dans notre application, ils sont situés :

1. **Security.yaml** (e.g. Role_hierarchy, access_control) (voir ici)

2. Controller

Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted est utilisé pour que nous puissions utiliser une annotation pour limiter l'accès à certains rôles pour les routes.

Voici un exemple tiré de notre application

```php
#[Route('/tasks/create', name: 'task_create')]
#[IsGranted('ROLE_USER')]
public function createAction(Request $request)
{

    $task = new Task();
    $task->setAuthor($this->getUser());
    $form = $this->createForm(TaskType::class, $task);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $this->taskService->saveTask($task);
        $this->addFlash('success', 'La tâche a été bien ajoutée.');
        return $this->redirectToRoute('task_list');
    }
    return $this->render('task/create.html.twig', ['form' => $form->createView()]);
}

#[Route('/tasks/{id}/edit', name: 'task_edit')]
#[IsGranted('ROLE_USER')]
public function editAction(Task $task, Request $request)
{
  if (!$this->isGranted('TASK_EDIT', $task)) {
    $this->addFlash('error', "Désolé. Vous n'avez pas le droit d'y accéder.");
    return $this->redirectToRoute('task_list');
   }
```

3. Voter

Le système de voter est également mis en œuvre dans certaines routes pour centraliser la logique d'autorisation. Dans notre cas, elle est mise en œuvre lorsque le rôle de l'utilisateur n'est pas le seul critère.

Dans notre application, "TASK_EDIT" est créé dans le TaskVoter, qui implémente l'interface VoterInterface .

"Task_edit" vérifiera si l'utilisateur est soit le propriétaire de la tâche, soit si l'utilisateur est ROLE_ADMIN et que la tâche appartient à 'anonyme'.

l'exemple ci-dessus a utilisé 'TASK_EDIT' pour la route nommé 'task_edit', ce qui signifie que l'utilisateur devra répondre aux critères de TASK_EDIT pour y'accéder.

Logout

Afin de se déconnecter, il est nécessaire de l'avoir configuré dans le security.yaml (voir ici) et de créer une route correspondant au nom qui est configuré sur security.yaml (‘app_logout’ dans notre cas)

src > Controller > SecurityController.php > ...

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class SecurityController extends AbstractController
{
    #[Route('/logout', name: 'app_logout', methods:'GET')]
    public function logout(): void
    {

    }
```

Le document explique principalement l'authentification de Symfony qui est liée à notre application. Pour en savoir plus:

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