Documentation technique

Réalisé par Simon Charbonnier Mai 2024

SOMMAIRE

1- Introduction

a. Application

L'application de Todo & Co permet de gérer ses tâches quotidiennes. Afin d'améliorer la qualité ainsi que la sécurité du site, il a été décidé de faire la migration technique de Symfony 3.1 vers Symfony 5.4.

b. Authentification

L'authentification est un pilier fondamental de la sécurité, elle permet de garantir que seuls les utilisateurs autorisés peuvent accéder aux ressources et fonctionnalités protégées de l'application. Un exemple dans notre cas, on ne veut pas qu'un autre utilisateur puisse supprimer nos tâches quotidiennes...

c. Version Symfony

L'authentification a donc été mise en place via cette dernière version LTS (Long-Term Support) de Symfony. Symfony 5.4 propose une solution complète pour la gérer à travers son bundle "security-bundle".

2- Implémentation de l'authentification

a. Installation

Pour commencer, il faut installer le bundle avec "composer require symfony/security-bundle".

Ensuite il nous faut une entité qui possèdera les attributs nécessaires à l'implémentation de l'authentification. L'entité "User" doit implémenter l'interface UserInterface pour pouvoir fonctionner avec le bundle. "User" a donc un nom d'utilisateur et un mot de passe que nous utiliserons pour nous connecter.

```php
/**
 * @ORM\Table("user")
 * @ORM\Entity
 * @UniqueEntity("email")
 */
class User implements UserInterface, PasswordAuthenticatedUserInterface
{

    /** ...
    private $id;

    /** ...
    private $username;

    /** ...
    private $password;

    /** ...
    private $email;

    /** ...
    private $tasks;

    /** ...
    private $roles = [];
```

src/Entity/User.php

b. Configuration

Afin de configurer la sécurité du projet, il faut modifier le fichier *config/packages/security.yaml.*

```yaml
security:
    enable_authenticator_manager: true
    password_hashers:
        App\Entity\User: 'auto'
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    providers:
        database_users:
            entity: { class: App\Entity\User, property: username }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: database_users
            form_login:
                login_path: login
                check_path: login
            logout:
                path: logout
```

- “password_hashers” permet de hacher le mot de passe des utilisateurs et donc de sécuriser l’application.
- “providers” permet de définir une entité utilisateur (celle évoquée plus tôt) ainsi qu’un champ comme identifiant.
- “firewalls” permet de définir la manière dont l’authentification va s’effectuer.

c. Connexion

Afin de gérer la connexion des utilisateurs, nous avons un SecurityController avec une méthode loginAction qui va afficher le formulaire de connexion situé dans le fichier *templates/security/login.html.twig.*

```php
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="login")
     *
     * @return Response
     */
    public function loginAction(AuthenticationUtils $authenticationUtils): Response
    {
        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();

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

src/Controller/SecurityController.php

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

    <form action="{{ path('login') }}" 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 %}
```

templates/security/login.html.twig

Le composant LoginFormAuthenticator va récupérer les informations de l'utilisateur via le formulaire une fois soumis. S'il valide l'authentification, alors l'utilisateur se verra rediriger vers la page d'accueil en tant qu'utilisateur connecté. Sinon, une erreur sera affichée sur la page de connexion (erreur récupérée grâce au composant AuthenticationUtils de Symfony).

3- Stockage des utilisateurs

a. Inscription

Les utilisateurs représentés par l'entité User vue précédemment sont stockés dans une base de données relationnelle à l'aide de Doctrine ORM, comme indiqué dans *config/packages/security.yaml.*

```
providers:
    database_users:
        entity: { class: App\Entity\User, property: username }
```

Afin d'inscrire un nouvel utilisateur dans notre application, il faut accéder à la page d'inscription (en tant qu'administrateur) et soumettre le formulaire avec les informations nécessaires.

Symfony fournit un système pour gérer les formulaires, en voici un exemple. Le formulaire est créé à partir d'un type de formulaire prédéfini "UserType", ainsi que son builder.

```php
class UserType extends AbstractType
{
    /**
     * build form
     * @param FormBuilderInterface $builder builder
     * @param array $options options
     *
     * @return void
     */
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $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'],
            ])
            ->add('email', EmailType::class, ['label' => 'Adresse email'])
            ->add('roles', ChoiceType::class, [
                'choices'  => [
                    'Utilisateur' => 'ROLE_USER',
                    'Administrateur' => 'ROLE_ADMIN'
                ],
                'expanded' => true,
                'multiple' => true
            ])
        ;
    }
}
```

src/Form/UserType.php

b. Enregistrement en base de données

Dans le controller, il y a la vérification et validation du formulaire et de ses données. Si la validation passe, alors l'utilisateur va être créé en base de données (avec un mot de passe haché).

```php
/**
 * @Route("/users/create", name="user_create")
 */
public function createAction(Request $request, ManagerRegistry $doctrine, UserPasswordHasherInterface $userPasswordHasher)
{
    $user = new User();

    $this->denyAccessUnlessGranted(UserVoter::ADD, $user);

    $form = $this->createForm(UserType::class, $user);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $em = $doctrine->getManager();

        $user->setPassword(
            $userPasswordHasher->hashPassword(
                $user,
                $user->getPassword()
            )
        );

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

        $this->addFlash('success', "L'utilisateur a bien été ajouté.");

        return $this->redirectToRoute('user_list');
    }

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

src/Controller/UserController.php

Avec PHPMyAdmin nous pouvons voir les données des utilisateurs (id, nom d'utilisateur, mot de passe haché, mail et rôles.

id	username	password	email	roles (DC2Type:json)
6	anonyme	$2y$13$kTdQadjhyHBNZ5NP8fO5fu3hYZsU4G8OSGQDJXl8UOu...	anonyme@gmail.com	["ROLE_USER"]
7	simon	$2y$13$s3lzOinDxakbrNu2hvMZCuEB4.YpsIP8DMajFiUPnxd...	simoncharbonnier@gmail.com	["ROLE_USER","ROLE_ADMIN"]
8	zchauvet	$2y$13$r8t6NSq2C93ToyJRV91SJ.WOG2E8UHKad1.9xHaWzFi...	frederique04@example.net	["ROLE_USER"]
9	leclerc.pauline	$2y$13$D5BaSeVOwLMllhaVcPhaYujcYOWHcaiFwHXUrLBdeZd...	adele90@example.com	["ROLE_USER"]
10	marcelle.laine	$2y$13$rbnW9RpUE3lYw1Nkn5XN3.OERfVeqlk/0MGOTxONo6/...	marthe.georges@example.org	["ROLE_USER"]
11	dupre.robert	$2y$13$zOR5NReildStNjBQRU29R.z88F.6qKxP4RCBfED5l2q...	klein.valerie@example.orge	["ROLE_USER"]

table User

4- Personnalisation

a. Event Listener

Chaque application peut avoir des exigences spécifiques en matière d'authentification qui nécessitent une personnalisation du processus standard. Symfony offre une flexibilité suffisante pour personnaliser différents aspects de l'authentification.

Celui qui a été mis en place dans notre application est l'utilisation d'événements avec un Event Listener pour intercepter et modifier le comportement de l'autorisation.

Si un utilisateur n'a pas les droits pour accéder à une ressource ou une fonctionnalité, alors une instance de la classe AccessDeniedException est renvoyée.

Cette erreur est alors écoutée et prise en charge par l'Event Listener, plutôt que de renvoyer l'erreur à l'utilisateur, on le redirige et lui envoie une notification lui informant qu'il n'a pas les permissions.

```php
/**
 * On Kernel Exception
 * @param ExceptionEvent $event event exception
 *
 * @return void;
 */
public function onKernelException(ExceptionEvent $event): void
{
    $exception = $event->getThrowable();
    if (!$exception instanceof AccessDeniedException) {
        return;
    }

    if ($this->security->isGranted('ROLE_USER')) {
        $this->session->getFlashBag()->add('error', "Vous n'avez pas les permissions.");

        $event->setResponse(new RedirectResponse($this->urlGenerator->generate('homepage')));
    } else {
        $this->session->getFlashBag()->add('error', "Veuillez vous connecter.");

        $event->setResponse(new RedirectResponse($this->urlGenerator->generate('login')));
    }
}
```

src/EventListener/AccessDeniedListener.php

Pour que l'Event Listener soit pris en compte, il faut mettre à jour le fichier *config/services.yaml*.

```yaml
services:
    # default configuration for services in *this* file
    _defaults: ···

    # makes classes in src/ available to be used as services
    # this creates a service per class whose id is the fully-qualified class name
    App\: ···

    # controllers are imported separately to make sure services can be injected
    # as action arguments even if you don't extend any base controller class
    App\Controller\: ···

    # add more service definitions when explicit configuration is needed
    # please note that last definitions always *replace* previous ones
    App\EventListener\AccessDeniedListener:
        tags: [kernel.AccessDeniedException]
```

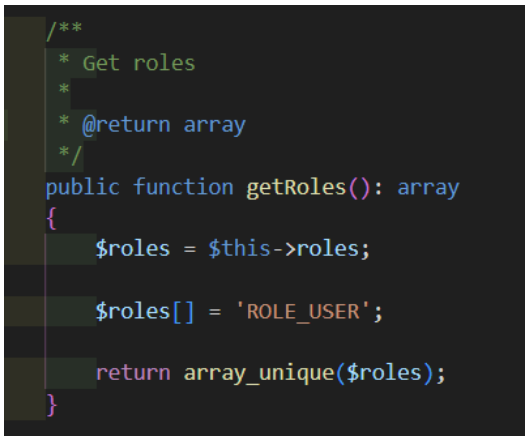
Cette personnalisation nous permet d'adapter l'authentification à nos besoins spécifiques tout en tirant parti des fonctionnalités puissantes offertes par Symfony.

5- Autorisation

a. Rôles

Outre la mise en place de l'authentification, il est essentiel de suivre les bonnes pratiques en matière de gestion des autorisations. Symfony propose une approche basée sur les "voters" pour gérer les autorisations, ce qui permet de différencier clairement la vérification de l'identité de l'utilisateur et les droits de l'utilisateur à accéder à une ressource ou une fonctionnalité. La personnalisation des droits est également très facile à mettre en place.

La classe User, représentant l'utilisateur implémente l'interface UserInterface qui contient la méthode getRoles(), chaque utilisateur a donc au moins un rôle.



```php
/**
 * Get roles
 *
 * @return array
 */
public function getRoles(): array
{
    $roles = $this->roles;

    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}
```

src/Entity/User.php

Les 2 rôles dans notre application sont :

- ROLE_USER : tous les utilisateurs possèdent ce rôle, il permet d'avoir accès aux fonctionnalités basiques de l'application et identifier un utilisateur connecté.
- ROLE_ADMIN : un administrateur a plus de droits, accès à la gestion des utilisateurs, de certaines tâches.

Sachant que les utilisateurs ont des rôles, on peut utiliser la fonction is_granted() dans les templates pour modifier les pages en fonction des droits de chaque utilisateur.

```twig
<div class="row">
    {% 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">Gestion des utilisateurs</a>
    {% endif %}

    {% if is_granted('ROLE_USER') %}
        <a href="{{ path('logout') }}" class="pull-right btn btn-danger">Se déconnecter</a>
    {% endif %}

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

templates/base.html.twig

Depuis les contrôleurs, on peut vérifier si l'utilisateur a les permissions de cette manière.

```php
/**
 * @Route("/tasks/{id}/edit", name="task_edit")
 */
public function editAction(Request $request, ManagerRegistry $doctrine, Task $task)
{
    $this->denyAccessUnlessGranted(TaskVoter::EDIT, $task);
```

src/Controller/TaskController.php

b. Voters

Cette dernière fonction va faire appel à TaskVoter, les voters en général nous permettent de personnaliser l'autorisation en fonction des fonctionnalités et des utilisateurs.

```php
/** ...
protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
{
    $user = $token->getUser();

    if (!$user instanceof User) {
        return false;
    }

    $task = $subject;

    return $attribute === self::ADD ? $this->canAdd($user) : $this->canEditOrDelete($task, $user);
}

/** ...
private function canAdd(User $user): bool
{
    return in_array('ROLE_USER', $user->getRoles());
}

/** ...
private function canEditOrDelete(Task $task, User $user): bool
{
    return $user === $task->getUser() || (in_array('ROLE_ADMIN', $user->getRoles()) && $task->getUser()->getUsername() === 'anonyme');
}
```

src/Security/TaskVoter.php

6- Ressources supplémentaires

a. Documentation Symfony

Pour approfondir notre compréhension de la sécurité dans Symfony 5.4, voici un lien vers la documentation officielle de Symfony sur la sécurité.

https://symfony.com/doc/5.x/security.html

b. Architecture de l'application

De plus, pour vous aider à naviguer dans le code de l'application, voici une brève explication de la structure des dossier Symfony :

- Dossier "bin" contient les exécutables disponibles dans le projet.
- Dossier "config" contient toute la configuration de l'application.
- Dossier "public" est le dossier accessible de l'extérieur, il contient le contrôleur frontal ainsi que les images et le style.
- Dossier "migrations" contient les migrations de la base de données.
- Dossier "src" contient le coeur de l'application (contrôleurs, entités, formulaires...).
- Dossier "tests" contient les tests (unitaires, fonctionnels).
- Dossier "templates" contient les vues, gabarits utilisés.
- Dossier "translations" contient la gestion des traductions.
- Dossier "var" contient les fichiers de cache et de log.
- Dossier "vendor" contient l'ensemble des dépendences du projet.