2023

OC-P8 To-do N CO

REALISATION DE L'AUTHENTIFICATION

DOCUENTATION TECHNIQUE

Jordan Deyme

Sommaire

Organisation des fichiers

L'entité User :

L'entité User contient des champs propres permettant d'identifier l'utilisateur, définir ses droits d'utilisateurs, de lui associer des tâches. Son entité possède donc ces champs :

- ID
- username
- roles
- password
- email
- $tasks

L'entité est stockée dans le dossiers /src/Entity.

NB : elle implémente le UserInterface ainsi que le PasswordAuthenticatedUserInterface permettant d'inclure les méthodes propres à la gestion d'utilisateur du composant security.

Security.yaml

Le fichier security.yaml permet de paramétrer le fonctionnement de l'authentification dans notre application. Ce fichier se situe dans le dossier config/packages/security.yaml

Providers

Le paramètre « providers » du fichier définit la classe et le champ à utiliser pour réaliser l'authentification.

Dans notre cas, il s'agit du champ « username » de la classe User de notre entité (il est possible d'utiliser le champ email, dans ce cas il faudra s'assurer de modifier les tests unitaires en conséquence).

```yaml
security:
    enable_authenticator_manager: true
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
```

Firewall

Le firewall permet de définir le processus d'authentification utilisé par l'application, c'est un élément important de la sécurité de l'application.

Le premier paramètre « dev » permet d'exclure du parefeu de symfony certaines sources en mode dev, c'est-à-dire, lorsque dans le fichier .ENV nous définissons le paramètre APP_ENV en test (ci-dessous, nous excluons la vérification des assets lors du mode dev) :

Le second paramètre « main » définit le fonctionnement pour l'ensemble de l'application.

Le paramètre lazy indique si le pare-feu doit être déclenché de manière "lente" ou "rapide". Si cette valeur est définie sur true, le pare-feu ne sera déclenché qu'une fois qu'un utilisateur tentera d'accéder à une zone protégée. Si la valeur est définie sur false, le pare-feu sera déclenché pour chaque requête.

Le paramètre provider spécifie le nom du "fournisseur d'utilisateur" à utiliser pour charger l'utilisateur actuellement connecté. Dans notre cas, le fournisseur d'utilisateur est défini par la clé app_user_provider.

Le paramètre custom_authenticator spécifie la classe à utiliser comme authentificateur personnalisé pour ce pare-feu. Cet authentificateur sera utilisé pour authentifier les utilisateurs qui tentent d'accéder à une zone protégée par ce pare-feu.

Le paramètre logout définit les options de déconnexion pour ce pare-feu. Dans ce cas, le champ path indique le chemin à utiliser pour déconnecter l'utilisateur

```yaml
firewalls:      You, 4 months ago • Add webapp packages
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider
        custom_authenticator: App\Security\AppAuthAuthenticator
        logout:
            path: app_logout
```

Détail des actions d'authentification

L'utilisateur accède à la page de connexion de l'application et remplit le formulaire de connexion (UserType) avec son nom d'utilisateur, son mot de passe (l'email n'est pas utilisé pour l'authentification, le formulaire contient ce champ car il peut être renseigné lors de la création d'un utilisateur par l'administrateur).

Le formulaire UserType se situe dans le dossier src/Form/UserType.php

```php
class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('username', TextType::class, [
                'label' => "Nom d'utilisateur",
                'attr' => [
                    'placeholder' => 'Votre pseudonyme',
                ],
            ])
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
                'invalid_message' => 'Les mots de passe doivent être identiques.',
                'options' => ['attr' => ['class' => 'password-field']],
                'required' => true,
                'first_options' => ['label' => 'Mot de passe'],
                'second_options' => ['label' => 'Confirmer le mot de passe'],
            ])
            ->add('email', EmailType::class);
    }

    public function configureOptions(OptionsResolver $resolver): void
    {
        $resolver->setDefaults([
            'data_class' => User::class,
        ]);
    }
}       You, 3 months ago • Forms Task and User
```

Le formulaire est soumis à l'action login du contrôleur SecurityController en utilisant une requête HTTP POST.

```php
class SecurityController extends AbstractController
{
    #[Route(path: '/login', name: 'app_login')]
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        if ($this->getUser()) {
            return $this->redirectToRoute('homepage');
        }

        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();      You, 3 months ago • relation User et mise à jour g
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

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

L'action récupère le nom d'utilisateur, le mot de passe envoyés dans le formulaire à l'aide de l'objet Request.

```php
class AppAuthAuthenticator extends AbstractLoginFormAuthenticator
{
    use TargetPathTrait;      You, 3 months ago • relation User et mise à jour gestion enregistreme…

    public const LOGIN_ROUTE = 'app_login';

    public function __construct(private UrlGeneratorInterface $urlGenerator)
    {
    }

    public function authenticate(Request $request): Passport
    {
        $username = $request->request->get('username', '');

        $request->getSession()->set(Security::LAST_USERNAME, $username);

        return new Passport(
            new UserBadge($username),
            new PasswordCredentials($request->request->get('password', '')),
            [
                new CsrfTokenBadge('authenticate', $request->request->get('_csrf_token')),
            ]
        );
    }

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
    {
        if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
            return new RedirectResponse($targetPath);
        }

        // For example:
        return new RedirectResponse($this->urlGenerator->generate('homepage'));
    }

    protected function getLoginUrl(Request $request): string
    {
        return $this->urlGenerator->generate(self::LOGIN_ROUTE);
    }
}
```

Elle appelle la méthode authenticate de l'authentificateur personnalisé AppAuthAuthenticator, (précédemment renseigné dans le security.yaml) en passant la requête en tant qu'argument.

```yaml
main:
    lazy: true
    provider: app_user_provider
    custom_authenticator: App\Security\AppAuthAuthenticator
    logout:
        path: app_logout
        # where to redirect after logout
```

L'authentificateur récupère le nom d'utilisateur et le mot de passe envoyés dans la requête, puis crée un "passeport" d'authentification contenant un "badge" utilisateur (UserBadge) avec le nom d'utilisateur, un "badge" de jeton CSRF (CsrfTokenBadge) avec le jeton CSRF envoyé dans le formulaire, et un objet de "credentials" de mot de passe (PasswordCredentials) avec le mot de passe envoyé dans la requête.

Le passeport d'authentification est envoyé au système d'authentification de Symfony qui vérifie les informations d'authentification. Si l'authentification réussie, le système crée un objet TokenInterface qui contient l'utilisateur connecté et le retourne à l'authentificateur.

L'authentificateur appelle la méthode onAuthenticationSuccess en passant la requête, le jeton d'authentification et le nom du pare-feu en tant qu'arguments. La méthode redirige l'utilisateur vers la page de destination précédemment enregistrée (si elle existe), sinon vers la page d'accueil.

Si l'authentification échoue, l'authentificateur retourne un échec d'authentification au système d'authentification de Symfony, qui retourne à son tour un échec d'authentification à l'action login. L'action affiche à nouveau le formulaire de connexion en affichant l'erreur d'authentification.

Stockage des données utilisateur

Processus

Lorsqu'un admin souhaite créer un nouvel utilisateur, la méthode createUser est appelée :

```php
#[Route('/user/create', name: 'user_create')]

public function createUser(Request $request, UserRepository $userRepository, UserPasswordHasherInterface $userPasswordHasher): Response
{

    if (!$this->isGranted('ROLE_ADMIN')) {
        $this->addFlash('danger', 'Vous devez être administrateur pour accéder à cette page.');

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


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

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $user->setRoles(['ROLE_USER']);

        $user->setPassword(
            $userPasswordHasher->hashPassword(
                $user,
                $form->get('password')->getData()
            )
        );

        $userRepository->add($user, true);
        $this->addFlash('success', 'L\'utilisateur a été bien été ajouté.');

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

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

Elle vérifie si l'utilisateur connecté a le rôle d'administrateur. Si ce n'est pas le cas, elle affiche un message d'erreur et redirige l'utilisateur vers la liste des tâches.

Elle crée une instance de l'entité User et un formulaire de type UserType associé à cette instance.

Elle traite la requête en appelant la méthode handleRequest du formulaire. Si le formulaire a été soumis et est valide, elle définit le rôle de l'utilisateur sur ROLE_USER, hache le mot de passe de l'utilisateur avec l'objet UserPasswordHasherInterface et enregistre l'utilisateur dans la base de

données avec la méthode add de UserEntity. L'utilisateur est enregistré dans la base, avec son mot de passe hashé pour éviter qu'il apparaisse en clair en cas d'accès à la base de données.

```php
public function add(User $entity, bool $flush = false): void
{
    $this->getEntityManager()->persist($entity);

    if ($flush) {
        $this->getEntityManager()->flush();
    }
}
```

Base de données

La base de donnée est précisée dans le fichier .env à la racine du projet :

```
DATABASE_URL="mysql://root:@127.0.0.1:3306/todonco?serverVersion=8.0.27"
```

Dans notre cas il s'agit d'une base locale, il est important de ne pas renseigner ce champ en cas de partage du code.