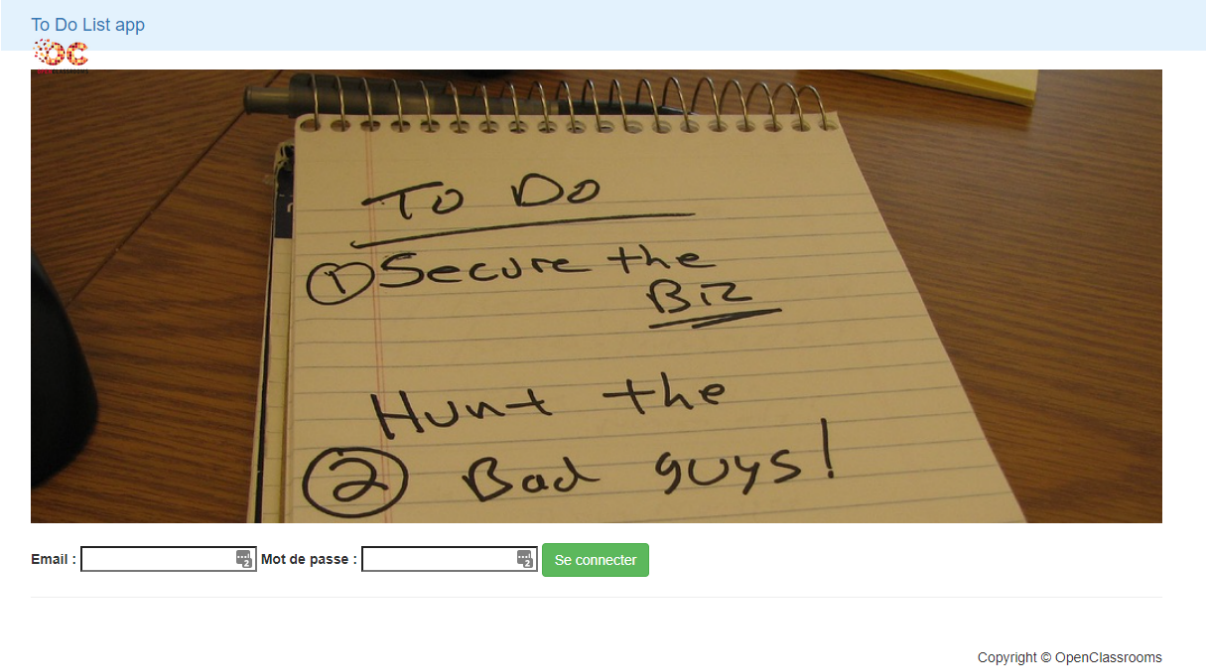


Documentation technique

ToDo & Co

Version : 0.1

Date de la dernière mise à jour : 16 février 2022

Sommaire

1. Authentification

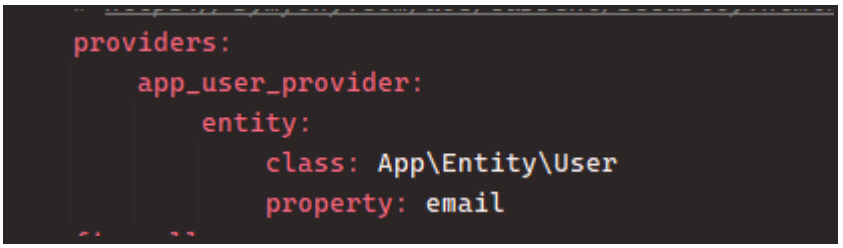
1.1. L'entité User

Les permissions dans Symfony sont toujours liées à un objet utilisateur. Si vous avez besoin de sécuriser (certaines parties de) votre application, vous devez créer une classe utilisateur. Il s'agit d'une classe qui implémente UserInterface ainsi que PasswordAuthenticatedUserInterface permettant la gestion des mots de passe hashés de l'application.

Vous avez la possibilité de créer une entité User avec le maker de Symfony en suivant la commande `make:user`

1.2. Le provider

Le provider permet (re)charger des utilisateurs à partir d'un stockage (par exemple une base de données) en se basant sur un "identifiant utilisateur" (par exemple l'adresse email ou le nom d'utilisateur de l'utilisateur). Dans le fichier de configuration security.yaml, la configuration ci-dessus utilise Doctrine pour charger l'entité User en utilisant la propriété email comme "identifiant utilisateur".



Suivant la configuration, vous pouvez identifier les utilisateurs de l'application avec leur username en changeant le paramètre property à "username".

1.3. L'encodeur

Comme expliqué au point 1.1 Entité User, l'entité User implémente aussi PasswordAuthenticatedUserInterface

De nombreuses applications exigent qu'un utilisateur se connecte avec un mot de passe. Pour ces applications, le SecurityBundle fournit une fonctionnalité de hashage et de vérification du mot de passe.
Un encodeur du nom “password_hashers” à été mis en place pour l’entité User, via la configuration suivante :

```
security:
    enable_authenticator_manager: true
    # https://symfony.com/doc/current/secu
    password_hashers:
        App\Entity\User:
            algorithm: 'auto'
```

Ici, dans le cas de l’entité User, le password sera encodé via cette encodeur.

1.4. Les Firewalls

Le firewall va permettre d’empêcher un utilisateur non authentifié d’accéder à certaines parties de l’application.
Le firewall “dev”est un faux pare-feu : il s'assure que vous ne bloquez pas accidentellement les outils de développement de Symfony, qui se trouvent sous des URL comme /_profiler et /_wdt. Vous n’avez donc pas à modifier cette partie

Le firewall “main” va permettre pour toute visite d’URL de l’application de vérifier suivant la sa configuration si l’utilisateur peut ou non accéder à cette URL
Dans la configuration de l’application, aucun pattern n’est renseigné, le firewall va donc vérifier toutes les routes possibles.
Nous lui indiquons que pour une authentification, la route (login_path) sera login. Cette route redirige vers un formulaire de connexion. De plus, nous activons la vérification csrf pour sécuriser d’autant plus le formulaire de connexion. Il nous reste à indiquer le route permettant la déconnexion, ici la route “logout”

```yaml
    main:
        lazy: true
        form_login:
            login_path: login
            check_path: login
            enable_csrf: true
        logout:
            path: logout
```

Pour permettre à un utilisateur non authentifié d'accéder au moins à cette route “login”, nous ajoutons des règles d’accès ou la route ‘login’ peut être ouverte à l' utilisateur non authentifié et que toutes autres routes ne sont accessibles qu’à un utilisateur authentifié.

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

L’utilisateur sera donc redirigé vers le formulaire de connexion géré par le controller “SecurityController” qui s’occupera de l’authentification et qui fournira la validation de l’accès à l’application ou au contraire un message d’erreur si l'authentification a échoué.

```php
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(AuthenticationUtils $authenticationUtils)
    {
        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();

        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

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

1.5. Gestion des accès et des rôles

L'accès à certaines parties de l'application peut aussi être fait directement en annotation depuis les controllers.

Pour respecter les souhaits des créateurs de l'application, les routes permettant la gestion des utilisateurs ne doivent être accessibles seulement pour les utilisateurs avec un ROLE_ADMIN :

Une annotation à donc été placée au plus haut du controller UserController permettant de verrouiller toutes les routes aux utilisateurs "admin"

```php
/**
 * @IsGranted("ROLE_ADMIN")
 */

class UserController extends AbstractController
{
    /**
     * @Route("/users", name="user_list")
     */
    public function listAction()
    {
        return $this->render( view: 'user/list.html.twig', ['users' => $this->getDoctrine()->getRepository( persistentObject: 'App:User')->findAll()]);
    }
```

De plus, certaines actions ne peuvent être réalisées seulement si l'utilisateur a role spécifique.

Une tâche ne peut par exemple être modifiée seulement pour son auteur.

Une mise en place d'un voter permet d'affiner ces restrictions.

TaskController :

```php
    /**
     * @Route("/tasks/{id}/edit", name="task_edit")
     */
    public function editAction(Task $task, Request $request)
    {
        $this->denyAccessUnlessGranted( attribute: 'task_edit', $task);
        $form = $this->createForm( type: TaskType::class, $task);

        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $this->getDoctrine()->getManager()->flush();

            $this->addFlash( type: 'success',  message: 'La tâche a bien été modifiée.');

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

TaskVoter :

```php
protected function voteOnAttribute(string $attribute, $task, TokenInterface $token): bool
{
    $user = $token->getUser();
    // if the user is anonymous, do not grant access
    if ($user instanceof UserInterface || null !== $task->getUser()) {
        switch ($attribute) {
            case self::TASK_EDIT:
                return $this->canEdit($task, $user);
                break;
            case self::TASK_DELETE:
                return $this->canDelete($task, $user);
                break;
        }
    }
    return false;
}

private function canEdit(Task $task, User $user)
{
    return $user === $task->getUser();
}
```

Si l'utilisateur n'est pas l'auteur, il ne pourra pas modifier la tâche et se verra refuser l'accès.

Résumé de l'opération d'authentification :

Lorsque l'utilisateur demande à se connecter, il va demander à Symfony le formulaire d'authentification en accédant à l'URL /login. Le système va générer une page HTML contenant ce formulaire, et le renvoyer à l'utilisateur.
Une fois les identifiants saisis et l'authentification demandée, Symfony va récupérer la requête envoyée par l'utilisateur, faire des vérifications de sécurité (assertions sur les entités, validité du token CSRF), et faire appel à l'UserRepository.
L'UserRepository est en contact avec la base de données, et va rechercher l'utilisateur demandé dans la base de données grâce à une requête construite avec le design pattern Factory. Si l'utilisateur est trouvé, l'entité User sera hydratée et envoyée au contrôleur, sinon, une erreur sera renvoyée avec le formulaire d'authentification initial.
De son côté, le firewall va permettre de restreindre l'accès à certaines pages. Pour cela, il va récupérer le rôle de l'utilisateur (ROLE_ADMIN ou ROLE_USER) courant, et vérifier s'il peut ou non accéder aux pages demandées.