**Authentification & Autorisation**

**Dans**



**Pour l'application**

**ToDo & Co**

Cette documentation détaille les différentes options de configuration et d'utilisation relatives à l'authentification sous Symfony que l'on peut retrouver dans la documentation officielle :

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



Notre application n'est utilisable que pour les utilisateurs authentifiés.

Symfony gère cette authentification avec son système de sécurité via son bundle ***symfony/security-bundle***.

\*\*\*

Camile Ghastine (février 2021)

**Sommaire**

1. **La classe User 3**
2. La classe user étendant UserInterface 3
3. Stockage des informations utilisateur 3
4. **Le fichier de configuration de l'authentification** **4**
5. L'authentification 4

- Encoders 5

- Providers 5

- Firewalls 6

2) L'autorisation 6

- Acces\_control 6

- Role\_hierachy 7

1. **Le SecurityController** **8**

1) Connexion et déconnexion 8

2) Validation du formulaire de connexion 9

1. **Les Voters 10**
2. Le rôle des voters 10
3. Les voters en pratique 10
4. **La classe User**
5. **La classe user étendant UserInterface**

Quelques soit le mode d'authentification, la création d'une classe "utilisateur" implémentant ***Symfony\Component\Security\Core\User\UserInterface*** est obligatoire :

Dans notre application la classe "utilisateur" est représentée par la classe **User** que l'on trouve dans le fichier ***src/Entity/User.php***

La class **User** étendant l'interface ***UserInterface***, elle doit implémenter les méthodes ci-dessous :

* getUsername() : retourne l'attribut servant à l'authentification (*username* dans notre application).
* getRoles() : renvoie un tableau de rôles attribués à un utilisateur.
* getPassword() : retourne le mot de passe d'authentification de l'utilisateur.
* getSalt() : retourne l’encodeur utilisé pour crypter le mot de passe.
* eraseCredentials() : supprime les informations d'identification stockés en texte brut.

<https://github.com/symfony/symfony/blob/5.2/src/Symfony/Component/Security/Core/User/UserInterface.php>

1. **Stockage des informations utilisateur**

Chaque utilisateur possède les attributs ci-dessous :

* *id* : identifiant auto-incrémenté
* *username* : nom d'utilisateur servant d'identifiant à l'authentification
* *email* : courriel
* *password* : mot de passe d'authentification
* *role* : ROLE\_USER ou ROLE\_ADMIN

Ces informations sont stockées par doctrine dans la table **user** de la base de données nommée **todoco**.

1. **Le fichier de configuration de l'authentification**

La configuration de l'authentification se trouve dans le fichier ***config/packages/securtiy.yaml***

security:  
 encoders:  
 App\Entity\User: bcrypt  
  
 providers:  
 doctrine:  
 entity:  
 class: App\Entity\User  
 property: username  
  
 firewalls:  
 dev:  
 pattern: ^/(\_(profiler|wdt)|css|images|js)/  
 security: false  
  
 main:  
 anonymous: true  
 lazy: true  
 logout:  
 path: app\_logout  
 # where to redirect after logout  
 # target: app\_any\_route  
 guard:  
 authenticators:  
 - App\Security\LoginFormAuthenticator  
 entry\_point: App\Security\LoginFormAuthenticator  
  
 access\_control:  
 - { path: ^/login, roles: IS\_AUTHENTICATED\_ANONYMOUSLY }  
  
 role\_hierarchy:  
 ROLE\_ADMIN: ROLE\_USER

Le fichier ***security.yaml*** se décompose en 5 parties :

* encoders
* providers
* firewalls
* acces\_control
* role\_hierachy

1. **L'authentification**

* **encoders**

<https://symfony.com/doc/current/security.html#c-encoding-passwords>

L'encoder permet de spécifier à Symfony quel algorithme utiliser pour encoder le mot de passe d'authentification de l'entité *User*.

encoders:  
 App\Entity\User: bcrypt

Dans notre application, c'est l'algorithme bcrypt qui est utilisé.

Avant d'enregistrer le mot de passe en base données, il OBLIGATOIRE de le hasher à l'aide du service : ***Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface***.

-> setPassword($this->passwordEncoder->encodePassword($user, 'mon\_mot\_de\_passe'));

<https://github.com/symfony/security-core/blob/5.x/Encoder/UserPasswordEncoderInterface.php>

* **providers**

<https://symfony.com/doc/current/security.html#b-the-user-provider>

Le "provider utilisateur" permet d'indiquer à Symfony où trouver les informations d'authentification.

providers:  
 doctrine:  
 entity:  
 class: App\Entity\User  
 property: username

Dans notre application, le provider nommé doctrine indique que c'est l'attribut username de la classe App\Entity\ User qui sera utiliser comme identifiant d'authentification.

* **firewalls**

<https://symfony.com/doc/current/security.html#a-authentication-firewalls>

Le pare-feu définit la méthode d'authentification pour un utilisateur.

firewalls:  
 dev:  
 pattern: ^/(\_(profiler|wdt)|css|images|js)/  
 security: false  
  
 main:  
 anonymous: true  
 lazy: true  
 logout:  
 path: app\_logout  
 # where to redirect after logout  
 # target: app\_any\_route  
 guard:  
 authenticators:  
 - App\Security\LoginFormAuthenticator

* Le pare-feu main n'ayant pas de pattern, c'est lui qui gère toutes les URL.
* Le mode anonymous à true permet d'attribuer le statut d'anonyme à tout utilisateur non-authentifiés.
* Le mode lazy à true empêche le démarrage de la session s'il n'y a pas besoin d'autorisation.
* logout indique dans path le nom de la route app\_logout pour opérer la déconnexion.
* On utilise le système d'authentification guard afin d'exercer un plus grand contrôle sur le processus d'authentification (authentification par token et éventuellement d'autres fonctionnalités futures).

Dans notre application, l'authentification authenticator se fait à travers un formulaire d'authentification App\Security\LoginFormAuthenticator.

<https://symfony.com/doc/current/security/guard_authentication.html>

1. **L'autorisation**

Maintenant que les utilisateurs peuvent s'authentifier grâce au formulaire de connexion, il est temps de gérer le problème des autorisations. Ceci, afin de permettre ou restreindre l'accès aux ressources voulues (principalement les URL) en fonction du rôle de l'utilisateur.

Pour l'instant, il n'existe que deux rôles : ROLE\_USER ou ROLE\_ADMIN.

Il n'est pas impossible qu'à l'avenir, les besoins de l'application nécessitent de créer des nouveaux rôles.

Vous lui attribuerez alors le rôle de votre choix en respectant le format de nommage : ROLE\_MON\_NOUVEAU\_ROLE.

* **acces\_control**

<https://symfony.com/doc/current/security.html#add-code-to-deny-access>

C'est dans access-control que l'on peut sécuriser simplement une partie de l'application, c’est-à-dire sécuriser un modèle d'URL.

access\_control:  
 - { path: ^/login, roles: IS\_AUTHENTICATED\_ANONYMOUSLY }

Ici, l'URL /login est accessible à tout le monde : IS\_AUTHENTICATED\_ANONYMOUSLY.

Avec la page de création d'un nouvel utilisateur, c'est la seule URL non sécurisée de l'application.

Pour accéder à toutes les autres URL, il faut être authentifié. Pour plus de flexibilité, nous avons fait le choix de gérer ces autorisations directement dans les contrôleurs.

<https://symfony.com/doc/current/security.html#securing-controllers-and-other-code>

Pour cela, l'AbstractController duquel étendent nos contrôleurs nous offre deux méthodes :

$this->denyAccessUnlessGranted('ROLE\_ADMIN');

If ($this->isGranted('ROLE\_ADMIN') {

// Do something

}

Si l'utilisateur n'est pas connecté denyAccessUnlessGranted redirigera l'utilisateur vers la page de connexion.

Si l'utilisateur est connecté, mais n'a pas le rôle ROLE\_ADMIN, une **réponse 403** sera renvoyée.

La deuxième méthode isGranted renvoye *true* si l'utilsateur à le rôle ROLE\_ADMIN, sinon il renvoie *false*.

Il est également possible d'utiliser is\_granted() directement dans une vue Twig afin contrôler l'affichage en fonction du rôle de l'utilisateur demandant la page.

{% if is\_granted('USER\_ADMIN') %}  
 <a href**="**#**"**>**Delete task**</a>  
{% endif %}

*Remarque 1 :* il est possible de vérifier si un utilisateur est connecté en utilisant un attribut spécial à la place d'un rôle : IS\_AUTHENTICATED\_FULLY.

Il en existe d'autres :

IS\_AUTHENTICATED\_ANONYMOUSLY, IS\_AUTHENTICATED\_REMEMBERED, etc.

<https://symfony.com/doc/current/security.html#checking-to-see-if-a-user-is-logged-in-is-authenticated-fully>

*Remarque 2 :* La gestion des autorisations peut également se faire dans les annotations.

<https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/security.html>

* **role\_hierachy**

<https://symfony.com/doc/current/security.html#hierarchical-roles>

Au lieu de donner de nombreux rôles à chaque utilisateur, on peut définir des règles d'héritage de rôle en créant une hiérarchie de rôles.

role\_hierarchy:  
 ROLE\_ADMIN: ROLE\_USER

Dans notre application, les utilisateurs avec le rôle ROLE\_ADMIN auront également le rôle ROLE\_USER.

1. **Le SecurityController**

Le contrôleur de sécurité se situe dans le fichier ***src/Controller/security.php*** et contient les méthodes :

* Login() appelée par l'URL : /login nommée app\_login.
* Logout() appelée par l'URL: /logout nommée app\_logout.

<?php  
  
namespace App\Controller;  
  
use LogicException;  
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 SecurityController extends AbstractController  
{  
 /\*\*  
 \* @Route("/login", name="app\_login")  
 \* @param AuthenticationUtils *$authenticationUtils* \* @return Response  
 \*/  
 public function login(AuthenticationUtils *$authenticationUtils*): Response  
 {  
 if ($this->isGranted('USER\_CONNECT')) {  
 return $this->redirectToRoute('homepage');  
 }  
  
 $error = *$authenticationUtils*->getLastAuthenticationError();  
 $lastUsername = *$authenticationUtils*->getLastUsername();  
  
 return $this->render(  
 'security/login.html.twig',  
 ['last\_username' => $lastUsername, 'error' => $error]  
 );  
 }  
  
 /\*\*  
 \* @Route("/logout", name="app\_logout")  
 \* @codeCoverageIgnore  
 \*/  
 public function logout()  
 {  
 throw new LogicException(  
 'This method can be blank - it will be intercepted by the logout key on your firewall.'  
 );  
 }  
}

1. **Connexion et déconnexion**

* **Login()**

Cette méthode permet d'afficher le formulaire de connexion.

Plusieurs choses sont à noter dans cette méthode :

* Grace à la méthode isGranted() une redirection est assurée vers la page d'accueil si un utilisateur déjà authentifié tente d'accéder à cette page.

if ($this->isGranted('USER\_CONNECT')) {  
 return $this->redirectToRoute('homepage');  
}

* Si une première connexion a été infructueuse, le formulaire est affiché avec le *username* préalablement saisie lors de cette connexion, ainsi que les messages d'erreurs expliquant l'échec de la connexion.
* **Logout()**

Cette méthode est vide, car elle est interceptée par le firewall lorsqu'elle est appelée.

Reportez-vous à la partie II-1-firwalls de cette documentation.

<https://symfony.com/doc/current/security.html#logging-out>

1. **Validation du formulaire de connexion**

Dans le *SecurityController*, il n'existe ni méthode, ni logique dans la méthode login() pour vérifier le formulaire de connexion et authentifier l'utilisateur.

Symfony gère cela en interne. En effet, lorsque le formulaire de connexion est soumis, il est récupéré par un **listener** qui va traiter le formulaire.

<https://symfony.com/doc/current/components/security/authentication.html>

En cas d'échec, la méthode login() du *SecurityController* est appelée et le formulaire déconnexion est affiché à nouveau avec les messages d'erreur.

En cas de réussite, la méthode onAuthenticationSuccess() du *LoginFormAuthenticator* est appelée.

/\*\*  
 \* @codeCoverageIgnore  
 \* @param Request *$request* \* @param TokenInterface *$token* \* @param string *$providerKey* \* @return RedirectResponse  
 \*/  
 public function onAuthenticationSuccess(  
 Request *$request*,  
 TokenInterface *$token*,  
 string *$providerKey* ): RedirectResponse {  
 if ($targetPath = $this->getTargetPath(*$request*->getSession(), *$providerKey*)) {  
 return new RedirectResponse($targetPath);  
 }  
  
 return new RedirectResponse($this->urlGenerator->generate('homepage'));  
// throw new \Exception('***TODO: provide a valid redirect inside '.\_\_FILE\_\_);*** }

Celle-ci redirige l'utilisateur vers la page d'accueil (homepage).

return new RedirectResponse($this->urlGenerator->generate('homepage'));

Si l'on souhaite qu'une connexion réussie soit redirigée vers une autre route, c'est ici qu'il faudra opérer le changement.

1. **Les Voters**
2. **Le rôle des voters**

Dans notre application, des **voters** ont été créés. Ils permettent de gérer les autorisations en centralisant toutes les logiques d'autorisation hors des contrôleurs.

Ces voters pourront être interrogé à loisir avec :

* isGranted() pour autoriser une action
* denyAccessUnlessGranted() pour organiser une redirection, afin qu'un code non autorisé soit lu.

<https://symfony.com/doc/current/security/voters.html>

1. **Les voters en pratique**

Abordons la question des **Voters** à travers une utilisation qui en est faite dans notre application.

Une tache peut être supprimée :

1. Par son créateur, si elle liée à un utilisateur.
2. Par un administrateur, si elle n'est liée à aucun utilisateur.

La condition 1 étant simple, elle ne nécessite pas forcément l'utilisation d'un voter. Le code ci-dessous serait suffisant pour satisfaire cette condition.

public function deleteTaskAction(…){  
 if ($this->getUser() !== *$task*->getUser()) {  
 return $this->redirectToRoute('task\_list');  
 }  
 // Delete task  
}

Mais la condition 2 qui s'ajoute justifie pleinement l'utilisation d'un voter.

public function deleteTaskAction(...) {  
 if (!$this->isGranted('TASK\_DELETE', $task)) {  
 return $this->redirectToRoute('app\_login');  
 }  
 //Delete task  
}

Ici, l'utilisation de isGranted('TASK\_DELETE', $task) entraine l'appel de certaines méthodes présentes dans les **Voters**.

<?php  
  
namespace App\Security\Voter;  
  
use App\Entity\Task;  
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;  
use Symfony\Component\Security\Core\Authorization\Voter\Voter;  
use Symfony\Component\Security\Core\User\UserInterface;  
  
class TaskVoter extends Voter  
{  
 protected function supports(*$attribute*, *$task*): bool  
 {  
 return *in\_array*(*$attribute*, ['TASK\_DELETE', 'TASK\_EDIT'])  
 && *$task* instanceof Task;  
 }  
  
 protected function voteOnAttribute(*$attribute*, *$task*, TokenInterface *$token*): bool  
 {  
 $user = *$token*->getUser();  
 if (!$user instanceof UserInterface) {  
 return false;  
 }  
  
 switch (*$attribute*) {  
 case 'TASK\_DELETE':  
 return $this->canDelete(*$task*, $user);  
 case 'TASK\_EDIT':  
 return true;  
 }  
  
 return false;  
 }  
  
 private function canDelete(*$task*, *$user*): bool  
 {  
 if (!*$task*->getUser() && *$user*->getRole() === 'ROLE\_ADMIN') {  
 return true ;  
 }  
  
 if (!*$task*->getUser()) {  
 return false;  
 }  
  
 if (*$user*->getId() === *$task*->getUser()->getId()) {  
 return true;  
 }  
  
 return false;  
 }  
}

isGranted('TASK\_DELETE', $task) entraine l'appel de la méthode supports($attribute, $task) de chaque Voter de notre application l'un après l'autre.

TASK\_DELETE correspond à un des attributs défini dans **TasKVoter**, la méthode voteOnAttribute() de ce Voter est donc appelée.

C'est dans celle-ci que la logique permettant de vérifier les conditions 1 et 2 sera écrite. Elle renvera true ou false pour autoriser ou pas l'action demandée.