Documentation technique

ToDo & Co

Juillet 2021

Sabouret Maxime

Sommaire

1) Pourquoi ce guide ?

Ce guide est à l'intention de tous les développeurs qui sont susceptibles de travailler sur le projet ToDoList de ToDo & Co.

Dans un premier temps, il détaille l'authentification de l'application afin de pouvoir en modifier facilement le fonctionnement. Il explicite où sont stockés les utilisateurs, quels fichiers sont utilisés, comment et pourquoi. Le guide détaille également toute la partie autorisation une fois que les utilisateurs sont connectés. La question principale à laquelle nous allons répondre est donc : comment gérer ce que les utilisateurs ont chacun le droit de faire sur notre application ?

2) Quelles sont les différences entre authentification et autorisation ?

Pour bien comprendre la différence entre authentification et autorisation, un schéma pour nous aider :

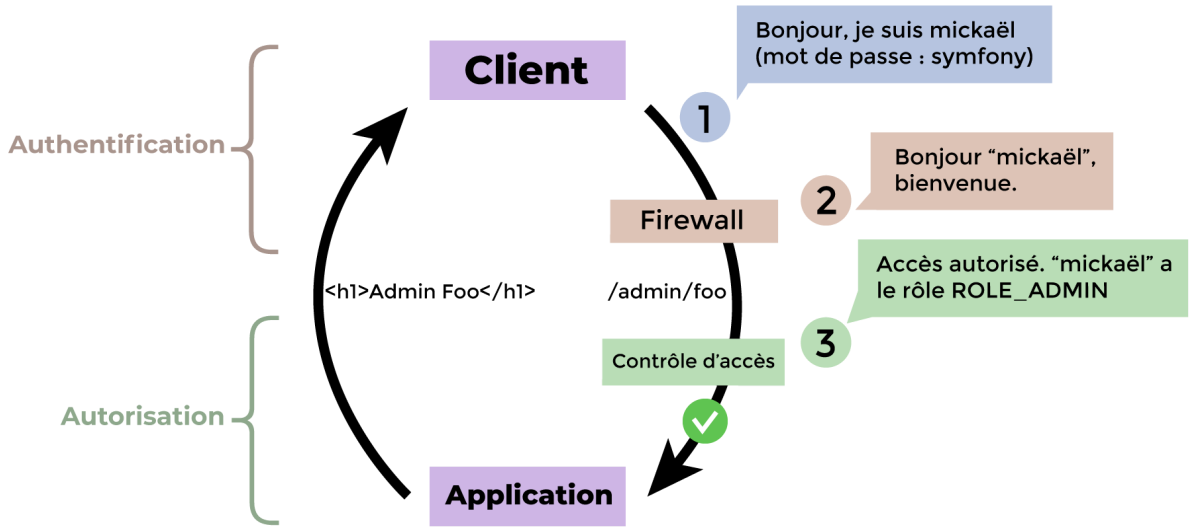


Schéma authentification/autorisation

<u>Première étape : authentification</u>
Correspond à la partie firewall de notre application. Certaines pages demandent à ce qu'un utilisateur ait un compte avec identifiant et mot de passe. Si les pages requièrent une authentification, il y a en général une page login permettant de se connecter. D'autres pages sont accessibles à des utilisateurs non connectés et dans ce cas le firewall laisse toujours passer les utilisateurs.

Deuxième étape : autorisation
Intervient après le firewall. Une fois que les utilisateurs sont connectés, il est possible de tester si l'utilisateur connecté a un rôle particulier. En fonction de ce rôle (admin par exemple), il pourra ou non accéder à certaines pages.

Les deux fonctionnalités permettent de laisser passer - ou non - un utilisateur pour accéder à une page mais l'autorisation arrive après l'authentification et la complète.

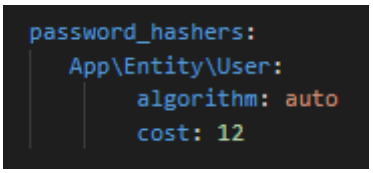
3) Comment fonctionne l'authentification ?

Dans Symfony, la configuration de la sécurité de l'application se trouve au niveau du fichier security.yaml. Avec Symfony Flex, ce fichier est créé lors de l'installation du bundle security (composer require symfony/security-bundle).

Ce fichier comporte, dans notre cas, 5 parties principales :
- les password_hashers
- les providers
- les firewalls
- l'access_control
- le role_hierarchy.

Les trois premières concernent plutôt l'authentification et les autres, l'autorisation.

Tout d'abord le password_hasher :



Configuration des password_hashers dans security.yaml

Cette partie concerne la complexification du mot de passe. Il est en effet nécessaire de ne pas garder les mots de passe en clair dans notre application pour qu'en cas de hacking, les personnes mal intentionnées ne puissent pas utiliser les comptes de nos utilisateurs.

Le hachage, c'est transformer une chaîne de caractères avec un algorithme complexe. La seule façon de vérifier que le mot de passe est le bon est de repasser par cet algorithme lors de la connexion par exemple. Il n'est pas possible, à partir d'une valeur hachée, de retrouver le mot de passe de base mais il est rapide de recréer le même type de hachage et de vérifier sa concordance lorsqu'on a le bon mot de passe.

Dans notre cas, nous laissons Symfony choisir le type d'algorithme (auto) le plus sécurisé de façon standard (bcrypt à partir de Symfony 5.3). Nous appliquons ce hachage sur notre

entité User et nous déterminons un coût de 12. Le coût est la valeur de complexité du hachage. Plus le coût est important, plus le hachage est sécurisé mais plus il mettra de temps à se faire. Un coût de 12 est un bon compromis entre sécurité et rapidité.
Pour utiliser ce système de hachage, Symfony nous met à disposition la UserPasswordHasherInterface. Nous l'avons notamment utilisé dans les fixtures du User avant d'enregistrer nos Users en base de données et dans l'ajout et la modification des utilisateurs dans nos controllers (/users/create et /users/{id}/edit).

Exemple dans nos fixtures :

```php
public function __construct(UserPasswordHasherInterface $passwordHasher)
{
    $this->passwordHasher = $passwordHasher;
}
```

Injection du passwordHasher dans le constructeur

```php
$user->setPassword($this->passwordHasher->hashPassword($user, "1234Jean%1234"));
```

Hachage du password

Résultat en base de donnée :

id	username	password	email	roles
2	admin	$2y$12$ymoHO.wxqOTFRVVoCaskf.wbQW2gKUE160Us9caqLhq...	admin@hotmail.fr	["ROLE_ADMIN"]

Table users

Nous avons vu l'enregistrement et le hachage de notre utilisateur en base de données mais nous n'avons pas encore parlé de l'entité User et de ses attributs en profondeur. Le security bundle utilise le App\Entity\User mais pour que cela soit fonctionnel il faut que le User implémente certaines classes : UserInterface et PasswordAuthenticatedUserInterface.

```php
class User implements UserInterface, PasswordAuthenticatedUserInterface
```

Class User qui implémente les différentes interfaces

La UserInterface oblige la classe User a avoir des méthodes indispensables pour le security bundle (https://symfony.com/doc/4.0/security/entity_provider.html#what-s-this-userinterface) :

- getRoles() : retourne les rôles de notre utilisateur
- getPassword() : retourne le password
- getSalt() : retourne le sel (utile pour certains type d'encodage) => pas utile dans notre cas mais la fonction reste obligatoire
- getUserIdentifier() (anciennement getUsername()) : retourne l'identifiant de l'utilisateur utilisé dans le provider
- eraseCredentials() : permet d'effacer les datas enregistrées si jamais on laisse le mot de passe en clair à un moment => pas utile dans notre cas mais la fonction reste obligatoire

La PasswordAuthenticatedUserInterface est liée à Symfony 5.3 et au niveau système de hachage, elle oblige à implémenter la méthode getPassword(). Il est bon d'avoir en tête que dans Symfony 5.3, UserInterface contient toujours les méthodes getPassword() et getSalt() mais elles seront supprimées dans Symfony 6.0.

Autrement, notre entité User contient les attributs id, password, username, email et roles. Mais quel est l'identifiant pour la sécurité ? Pour le savoir, il suffit de regarder **la partie provider** de notre security.yaml. C'est elle qui est en charge de fournir la classe utilisateur et de spécifier l'identifiant à utiliser (ici username) :

```yaml
providers:
    users:
        entity:
            class: App\Entity\User
            property: username
```

Configuration des providers dans security.yaml

C'est pourquoi nous renvoyons $this->username dans notre méthode getUserIdentifier() de notre entité User.

Maintenant que nous avons vu l'ensemble des prérequis pour l'authentification, passons maintenant à la connexion, au **firewall** et à son fonctionnement. Comme nous l'avons dit un peu plus haut, le firewall (pare-feu) va permettre de donner au non l'accès à certaines pages de notre application. On peut créer autant de firewalls que l'on veut en fonction des configurations que nous désirons.

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false

    main:
        pattern: ^/
        form_login:
            login_path: login
            check_path: login
        logout:
            path: /logout
            target: /login
```

Configuration des firewalls dans security.yaml

Dans notre cas, nous avons une configuration dev et une configuration main. Il n'y a pas grand chose à dire sur la configuration dev hormis qu'elle autorise le chargement des assets, des images, du profiler et des templates de notre site, c'est un peu un faux firewall.
En revanche, la configuration main est un peu plus complexe car c'est elle qui définit le fonctionnement de la connexion et de la déconnexion.

Le pattern est la clé qui va être utilisée pour savoir si le firewall s'applique. Dans main, nous voulons que le firewall s'applique sur toutes les pages (^/).
Ensuite vient la partie form_login (https://symfony.com/doc/current/security/form_login.html). Cette partie permet de dire à notre système d'authentification que nous allons utiliser un formulaire de login. La partie "login" après les ":" est le nom de la route que nous utilisons.
Désormais, lorsque le système de sécurité lance le processus d'authentification, il redirige l'utilisateur vers le formulaire de connexion /login.

Nous avons donc une route login :

```php
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(AuthenticationUtils $authenticationUtils): Response
    {
        if ($this->getUser()) {
            return $this->redirectToRoute('homepage');
        }

        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();

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

Route login

Et un template lié à cette route (templates/security/login.html.twig) :

```twig
{% extends 'base.html.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 %}
```

Template login

Chaque fois que l'utilisateur rentre ses informations dans le formulaire, les données sont vérifiées par le composant security dans la requête. Dans le cas d'éventuelles erreurs, le

composant authenticationUtils intercepte celles-ci ainsi que le nom du dernier utilisateur écrit et renvoie ces informations dans le template de login. Si l'utilisateur est authentifié, il est renvoyé vers la page d'accueil (homepage).

Dans le cas où nous voudrions avoir un contrôle plus en profondeur sur notre authentification, nous pourrions mettre en place un guard authenticator (https://symfony.com/doc/current/security.html#guard-authenticators et https://symfony.com/doc/current/security/guard_authentication.html). Le principe est d'intercepter les différentes actions de l'authentification à ces différentes étapes :

Pour ce qui est de la déconnexion,elle est activée par le paramètre logout qui conduit à une route sans action particulière et renvoie vers la page login. Pour paramétrer simplement cette route logout, nous l'avons ajouté dans le fichier config/routes.yaml plutôt que dans le controller.

```yaml
logout:
  path: /logout
```

Configuration de la route logout dans security.yaml

Pour plus d'informations sur le composant security, n'hésitez pas à consulter la documentation officielle : https://symfony.com/doc/current/security.html.

4) Comment fonctionne l'autorisation ?

Une fois les différents utilisateurs authentifiés, vient l'étape de l'autorisation. Certes les utilisateurs sont connectés mais quels droits ont-ils sur le site ? Peuvent-ils tout supprimer ? Peuvent-ils uniquement afficher certaines pages de l'application ? Le but est alors de limiter l'accès à certaines pages ou à certaines fonctionnalités de notre application.

Comment Symfony nous aide pour faire cela ?
Il existe plusieurs façons de gérer l'autorisation dans Symfony. La façon la plus simple et la plus générale concerne le **access control** de notre fichier security.yaml :

```yaml
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/users, roles: ROLE_ADMIN }
    - { path: ^/, roles: ROLE_USER }
```

Configuration du acces_control dans security.yaml

Ce fichier permet de donner l'accès à certaines pages sous conditions. Dans notre configuration :

- la page login est accessible pour tout le monde

- la page users et toutes les routes commençant par /users sont bloqués aux utilisateurs ayant le rôle admin,
- toutes les pages (hormis login) sont accessibles uniquement si on a le rôle user

Si jamais on ne respecte pas ces conditions, le access control renvoie directement vers la page login d'authentification.

Couplé à cela, nous avons le **role_hierarchy** qui permet de définir la hiérarchie des rôles dans notre application :

```
role_hierarchy:
    ROLE_ADMIN:       ROLE_USER
```

Configuration du role_hierarchy dans security.yaml

Ici nous disons que le rôle admin a les mêmes droit au minimum qu'un rôle user, la réciproque étant fausse.

Cependant, nous désirons souvent avoir un contrôle plus précis de l'autorisation dans notre application. Nous voulions par exemple autoriser la suppression de tâche à certains utilisateurs en fonction de leurs rôles (si la tâche n'avait pas d'auteur un admin pouvait la supprimer et si la tâche avait un auteur, seulement l'auteur de la tâche pouvait supprimer la tâche). Il est compliqué voire impossible de mettre une telle logique dans le security.yaml. C'est pourquoi Symfony nous permet d'utiliser la fonction isGranted dans les controllers et les templates. La fonction isGranted permet simplement de tester le rôle d'un utilisateur. Elle prend en compte la hiérarchie des rôles définie dans le security.yaml. On peut utiliser cette méthode dans les annotations (@IsGranted("ROLE_ADMIN")), directement dans les méthodes de nos controllers ($hasAccess = $this->isGranted('ROLE_ADMIN')) ou même dans les templates twig ({{ is_granted(role, object = null, field = null) }}).

Mais ce n'est pas la solution choisie car nous préférons externaliser la logique dans des services hors des controllers afin que ces derniers n'aient qu'un rôle principal : renvoyer une réponse en fonction d'une requête. C'est pourquoi nous avons utilisé les voters qui sont la façon la plus efficace et la plus maintenable de gérer les autorisations dans Symfony (https://symfony.com/doc/current/security/voters.html). L'autorisation fonctionne alors en 2 temps :

```
$this->denyAccessUnlessGranted('task_delete', $task);
```

Méthode denyAccessUnlessGranted

D'abord nous demandons à appeler l'ensemble des voters dans notre controller grâce à la fonction denyAccessUnlessGranted. Ensuite, nous créons un voter qui vérifie les permissions :

```php
class TaskVoter extends Voter
{
    const TASK_DELETE = "task_delete";

    protected function supports(string $attribute, $task): bool
    {
        return in_array($attribute, [self::TASK_DELETE])
            && $task instanceof Task;
    }

    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) {
            return false;
        }

        // No author => permission if admin
        if ($task->getAuthor() === null) {
            if (in_array("ROLE_ADMIN", $user->getRoles())) {
                return true;
            } else {
                return false;
            }
        }

        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case self::TASK_DELETE:
                return $this->canDelete($task, $user);
                break;
        }

        return false;
    }

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

TaskVoter

Ainsi, le voter renvoie true lorsque la permission est acceptée et false si l'accès refusé.

Pour plus d'informations sur le composant security, n'hésitez pas à consulter la documentation officielle : https://symfony.com/doc/current/security.html.