ToDo & Co

Documentation technique

L'authentification et la sécurité

Patrick Raspino

Kimealabs

MAJ 17/10/2002

Préambule

Une application comprenant une ou des parties « privées » et non accessibles au public nécessite un système d'authentification. A savoir qu'il existe plusieurs méthodes d'authentifications possibles. Pour ce projet nous avons opté pour un système d'authentification très classique, un formulaire de connexion nécessitant un couple adresse email / mot de passe, lequel sera comparé aux données résidant dans la base de données.

A l'origine, ce projet a été développé sous le framework Symfony 3.1 puis migrer jusqu'à la version 5.4 (version LTS : Long Term Support). Cette documentation technique vous permettre de connaître la logique ainsi que les classes utilisées au sein du framework Symfony afin de permettre la mise en place de cette authentification.

A noter qu'une fois le « visiteur » authentifié, celui-ci le restera jusqu'à sa déconnexion, la fermeture de navigateur ou expiration de sa session serveur. Le rôle du « connecté » sera présent tout au long de ses actions et navigations, rôle qui lui permettra d'accéder ou non à certaines actions.

En termes de sécurité, le framework Symfony utilise deux principes que sont l'authentification et l'autorisation.

Note :
Pour consulter la documentation officielle de la sécurité sous Symfony :
https://symfony.com/doc/5.4/security.html

1. Class User

L'Authentification permet à l'utilisateur de s'identifier.

Les utilisateurs sont, dans Symfony, représentés ici par la class User, laquelle reflète les utilisateurs en base de données grâce à l'ORM Doctrine.
En effet, à travers le système d'entités, la classe User (App\Entity\User) pourra créer un objet (POO) contenant les données d'un utilisateur.

Afin de réaliser une jonction entre la classe User et le système de sécurité de Symfony, celle-ci doit implémenter 2 interfaces.

class User implements UserInterface, PasswordAuthenticatedUserInterface

Concrètement, la création de ce type de classe est facilement réalisable depuis les migrations vers la version 5.4 de Symfony grâce à l'instruction (CLI Symfony) :

```
> symfony console make:user
```

2. Security.yaml

Le fichier centralisant toute la configuration du composant Security de Symfony se nomme **security.yaml**. Vous pourrez le trouver dans le dossier « config/packages ».
Ce fichier peut se décomposer en 5 parties, les 4 premières faisant partie de **l'authentification**, la dernière de **l'autorisation** :

A) Authenticator Manager

Afin de pouvoir utiliser le système d'authentification qui existe depuis la version 5.1 de Symfony, nous devons l'activer de cette manière.

```yaml
security:
    enable_authenticator_manager: true
```

Ce système nous permet d'accéder à un système d'authentification simplifié. Désormais, tout est relié à un seul concept et une seule interface (*AuthenticatorInterface*). Si vous voulez en savoir plus sur le sujet, vous pouvez consulter un très bon article :
https://blog.yousign.io/posts/les-nouveautes-dans-le-composant-securite-de-symfony

B) Les providers

Un provider permet d'indiquer quelle classe on doit utiliser et aussi sur quelle propriété se baser (outre le mot de passe) afin de contrôler l'authentification. Il va s'en dire de cette propriété doit comporter une contrainte d'unicité, par ex : 2 utilisateurs ne doivent pas avoir la même adresse email (si cette propriété est choisie dans le provider).

```
# config/packages/security.yaml
security:
    # ...
    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
```

La configuration du provider (il pourrait y en avoir plusieurs) de l'application est donc basée sur le nom d'utilisateur, aussi nous retrouverons ce champ dans le formulaire de connexion.

C) Le Password hashers

Etant donné que le mot de passe est stocké dans la base de données, celui-ci doit être crypté. En effet, imaginez une fuite/hack de données, une personne mal intentionnée pourrait alors se connecter avec le profil de n'importe qui et accéder à toutes les actions possibles.

Aussi, cette configuration permet à Symfony de choisir l'algorithme d'encodage des mots de passe de façon optimale !

```
# config/packages/security.yaml
security:
    # ...
    password_hashers:
        # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

D) Firewalls

.Avec Symfony, l'authentification est gérée par le **firewall**. Le firewall définit quelles parties de votre application sera sécurisés et comment les utilisateurs pourront s'authentifier.

```
# config/packages/security.yaml
security:
    # ...
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
             form_login:
                        login_path: login
                        check_path: login
                        enable_crsf: true
             logout:
                        path: logout
                        target: homepage
                        invalidate_session: false
```

Explications de nos 2 firewalls :

dev : En développement nous désirons avoir accès à des fichiers de Symfony. Aussi la propriété « **security** » doit rester à « **false** ». Ce firewall n'est actif qu'en environnement « dev » (voir fichier .env à la racine du projet) et nous permet d'accéder aux composants de Symfony et utiliser le **profiler**.

main : Ce firewall est utilisé par l'application dans tous les profils d'environnement.

- lazy : Permet d'améliorer les temps de chargement des pages publiques en utilisant du cache. Indique à Symfony de charger (session, entre autres) l'utilisateur seulement si l'application a besoin d'accéder à l'objet user.
- provider : définit quel provider choisir.
- form_login :

- login_path : Détermine la route du formulaire de connexion
- check_path : Détermine la route pour examiner la requête
- enable_crsf : Active le contrôle CRSF en corroborant avec un champ « hidden » dans le formulaire de connexion, celui-ci représente une sorte de mot de passe aléatoire généré à chaque affichage du formulaire.

- logout :

- path : Route du contrôleur
- target : Route de la redirection à effectuer
- invalidate_session : Permet de ne pas effacer la session afin de pouvoir utiliser les messages Flash et d'afficher le message de déconnexion.

Il existe bien d'autres fonctionnalités d'authentification que le « form_login » (ex : JSON login, http login, custom authenticator, ..), néanmoins, nous avons utilisé cette fonctionnalité par sa simplicité de mise en œuvre. Effet, une configuration du security.yaml et un contrôleur comprenant un formulaire de connexion suffise pour obtenir un système d'authentification.

E) Access Control

Contrairement aux 3 dernières parties, **l'access control** se s'occupe pas de l'authentification, mais de(s) l'autorisation(s). Chaque ligne permettant de déterminer selon un rôle, lequel fait partie de la class **User**, les autorisations d'accès à des dossiers.

```
# config/packages/security.yaml
security:
    # ...

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER

   access_control:
          -{ path : ^/users, roles: ROLE_ADMIN }
          -{ path : ^/tasks, roles: ROLE_USER }
          -{ path : ^/task, roles: ROLE_USER }
```

Par rapport au cahier des charges nous avons déterminés 2 rôles :
- ROLE_USER
- ROLE_ADMIN

Le role_hierarchy indique qu'un utilisateur de type ROLE_ADMIN est aussi un utilisateur de type ROLE_USER.

Les 3 lignes suivantes déterminent les autorisations de ces routes :

- Toutes les routes commençant par **/tasks** ou **/task** sont autorisées pour le ROLE_USER.
- Toutes les routes commençant par **/users** sont autorisées que pour le ROLE_ADMIN.
- Toutes les autres routes (formulaire de connexion, inscription, page d'accueil) ne sont pas contrôlées, par conséquent, tout utilisateur connecté ou non, en aura accès.

Notez que vous pouvez « affiner » les autorisations directement dans les contrôleurs au moyen de la fonction **IsGranted(ROLE)** ou **is_granted(ROLE)** au sein des templates *Twig*.

Un utilisateur non connecté essayant de visiter une route qui lui est interdite sera automatiquement redirigé vers la page de connexion (**login**).
Un utilisateur connecté essayant de visiter une route qui lui est interdite sera automatiquement dirigé vers l'accueil (**homepage**).

3) Le controller security

Notre formulaire de connexion se trouve dans le contrôleur :
src/controller/securityController.php
Ce contrôleur comprend 2 méthodes.

La première, « **login** », qui permet l'affichage (au travers du bundle Twig et de la template /security/login.html.twig) de notre formulaire de connexion. Et surtout, qui permet l'envoi du formulaire à **l'Authenticator** de notre fonctionnalité Symfony « **form_login** », laquelle gérera toute seule les systèmes d'authentifications.

Cette méthode utilise aussi le service « ***AuthenticationUtils*** », lequel nous permet de récupérer le nom du dernier utilisateur connecté et éventuellement de pouvoir consulter les erreurs de connexions.

La deuxième méthode, bien que présente ne sert à rien. Il s'agit de la méthode « **logout** », elle ne sera en effet jamais exécutée. Le composant **Security** a seulement besoin d'une *Route* pour effectuer la déconnexion.

4) EventSubscriber

L'utilisation du « **form_login** » **Authenticator** de Symfony est le plus simple à mettre en œuvre. Néanmoins, afin de pouvoir gérer les évènements de connexion, nous avons utilisé un **EventSubscriber**, lequel est capable de « surveiller » la plupart des évènements de Symfony.

Les 3 Méthodes suivantes, lesquelles sont dans notre `SecurityEventSubscriber` nous permettent de pouvoir créer des messages flash à destination de l'utilisateur qui sera alors averti du déroulement de sa demande de connexion.

```php
    public function logoutMessage(LogoutEvent $event)
    {
        $session = $event->getRequest()->getSession();
        $session->getFlashBag()->add('success', 'Vous êtes déconnecté !');
    }

    public function loginSuccesMessage(LoginSuccessEvent $event)
    {
        $session = $event->getRequest()->getSession();
        $session->getFlashBag()->add('success', 'Vous vous êtes connecté !');
    }

    public function loginFailureMessage(LoginFailureEvent $event)
    {
        $session = $event->getRequest()->getSession();
        $session->getFlashBag()->add('danger', 'Nom utilisateur ou mot de passe incorrect !');
    }
```

A noter que si vous voulez gérer tous les cas de figures, vous pouvez vous-même créer votre propre **Authenticator**. Il suffira de suivre l'implémentation de l'**AuthenticatorInterface**.
https://symfony.com/doc/5.4/security/custom_authenticator.html

5) Récapitulatif

A savoir que la fonctionnalité « **form_login** » utilise l'interface **AuthenticatorInterface**, laquelle utilise un système un peu plus « profond » du **security Bundle** de Symfony, à savoir les **Passport**, **userBadge**. Mais pour utiliser ce « **form_login** », et pour l'instant, vous n'avez pas besoin de vous compliquer encore plus la compréhension du système d'authentification, vous en avez déjà un bon aperçu.

Si vous voulez « creuser » plus profond ces classes et si vous désirez créer un **Custom Authentificator** :

Créez un Custom authentificator avec la CLI > symfony console **make: auth**

https://symfony.com/doc/current/security/custom_authenticator.html#security-passports