Authentification et Autorisation

En matière de sécurité, le framework symfony comporte deux mécanismes:

- L'authentification
- L'autorisation

1. L'authentification : Le fichier security.yml

Le fichier `app\config\security.yml` permet de répondre aux questions:

- Comment authenfifier les utilisateurs?
- Comment charger les utilisateurs?
- Comment protèger les mots de passe?

La partie la plus importante du fichier `security.yml` est la clé `firewalls`. On peut configurer autant de firewall que désiré.

1.1. Configurer l'authentification des utilisateurs

L'application ToDo List utilise le firewalls `main`.

Détails des clés utilisées :

- **anonymous** : Défini si l'on peut-être connecté comme utilisateur anonyme sur l'application.
- **pattern** : Une regex définissant les URL filtrées. Ici toutes les URL sont filtrés.
- **user_checker** : Une classe qui vérifie l'utilisateur avant l'authentification au moment du login.
- **form_login** :
    - **login_path** : Le nom de la route utilisée pour se connecter.
    - **check_path** : Le nom de la route utilisée pour vérifier le couple utilisateur/mot de passe.
    - **always_use_default_target_path** : Si à `true`, les utilisateurs sont toujours redirigés vers le chemin cible par défaut, quelle que soit l'URL précédente qui a été stockée dans la session.
    - **default_target_path** : L'URL par défaut pour la redirection, si aucune route n'est définis dans la session.
- **logout** : Autorise la déconnexion.
- **logout_on_user_change** : Si cette option est cochée, Symfony déclenche une déconnexion lorsque l'utilisateur change. Ne pas le faire est obsolète, donc cette option doit être définie sur true pour éviter d'obtenir des messages d'obsolescence.

```
        main:
            anonymous: true
            pattern: ^/
            user_checker: AppBundle\Security\UserChecker
            form_login:
```

```
                login_path: login
                check_path: login_check
                always_use_default_target_path:  true
                default_target_path:  /
            logout: ~
            logout_on_user_change: true
```

1.2. Configurer le chargement des utilisateurs

Ici c'est la clé `providers` qui nous intéresse.

Dans l'application ToDo List, les utilisateurs sont stokés en base de données (BDD), c'est pourquoi nous utiliserons Doctrine pour les récupérer. Nos utilistateurs sont représentés par l'entité `USER`, et nous retrouverons en BDD grâce au champ `username`.

```
    providers:
        doctrine:
            entity:
                class: AppBundle:User
                property: username
```

Plusieurs `providers` peuvent-être configurés.

1.3. Encoder le mot de passe utilisateur

La clé `encoders` est utilisé pour définir l'encoder utilisé. Ici l'encoder `bcrypt` est utilisé pour l'entité `User`. Les bibliothèques `bcrypt` ou `argon2i` sont recommendés. `Argon2i` est plus sécurisé, mais il requière PHP 7.2 ou l'extension Sodium.

```
    encoders:
        AppBundle\Entity\User: bcrypt
```

Différents encoders peuvent-être utilisé sur différentes classes.

2. Autorisation, accés et rôle utilisateur

L'`autorisation`, son travail consiste à décider si un utilisateur peut accéder à une ressource (une URL, un objet modèle, un appel de méthode, ...).

Le processus d'autorisation nécessite deux étapes différentes :

1. L'utilisateur reçoit un ensemble spécifique de rôles lors de la connexion (par exemple ROLE_ADMIN).
2. Vous ajoutez du code pour qu'une ressource (par exemple une URL, un contrôleur) nécessite un "attribut" spécifique (le plus souvent un rôle comme ROLE_ADMIN) afin d'être accessible.

2.1. Rôles

Lorsqu'un utilisateur se connecte, Symfony appelle la méthode `getRoles()` sur l'entité utilisateur pour en déterminer ses rôles. Dans notre classe `User`, les rôles sont un tableau qui est stocké en base de données, et chaque utilisateur se voit toujours attribuer au moins un rôle : `ROLE_USER`.

Dans l'application ToDo List, seul deux rôles sont définis :

- ROLE_USER
- ROLE_ADMIN

Le rôle admin possède les droits du rôle User. La hiérarchisation des rôle est défini dans le fichier `security.yml` avec la clé `role_hierarchy`.

```
    role_hierarchy:
        # Admin inherits user rights
        ROLE_ADMIN:       [ROLE_USER]
```

2.2. Ajout de code pour restreindre un accés

Il y a deux façons de refuser l'accès à une ressource :

1. La clé `access_control` dans security.yml vous permet de protéger les modèles d'URL (par exemple /admin/*). C'est facile, mais moins flexible.
2. Dans nos contrôleur (ou autre code).

2.2.1 Sécuriser les modèle d'URL (access_control)

La façon la plus simple de sécuriser une partie de l'application est de sécuriser un modèle d'URL entier dans `security.yml`.

Nous autorisons les utilisateurs anonyme (non connectés) sur la page de login.

En revanche, toutes les autres URL, nécessitent au minimun le `ROLE_USER`.

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

2.2.2 Sécuriser les controllers

L'application ToDo List utilise le bundle `SensioFrameworkExtraBundle`, et nous utilisons les annotations (comme recommandé par Symfony) pour sécuriser nos controllers.

```
// AppBundle\Controller\UserController.php

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

    /**
```

```
     * @Route("/users",
     *      name="user_list",
     *      methods={"GET"}
     * )
     *
     * @Security("has_role('ROLE_ADMIN')")
     */
    public function listAction(ObjectManager $entityManager)
    {
        return $this->render('user/list.html.twig', [
            'users' => $entityManager->getRepository(User::class)->findAll(),
        ]);
    }
```

2.3. Contrôle d'accés dans les templates Twig

Pour vérifier si l'utilisateur actuel a un rôle dans un modèle, utilisez la fonction `is_granted()` :

```
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

Le mot de la fin

Retrouvez plus d'information dans la documentation Symfony:

- Le composant security
- La configuration de référence