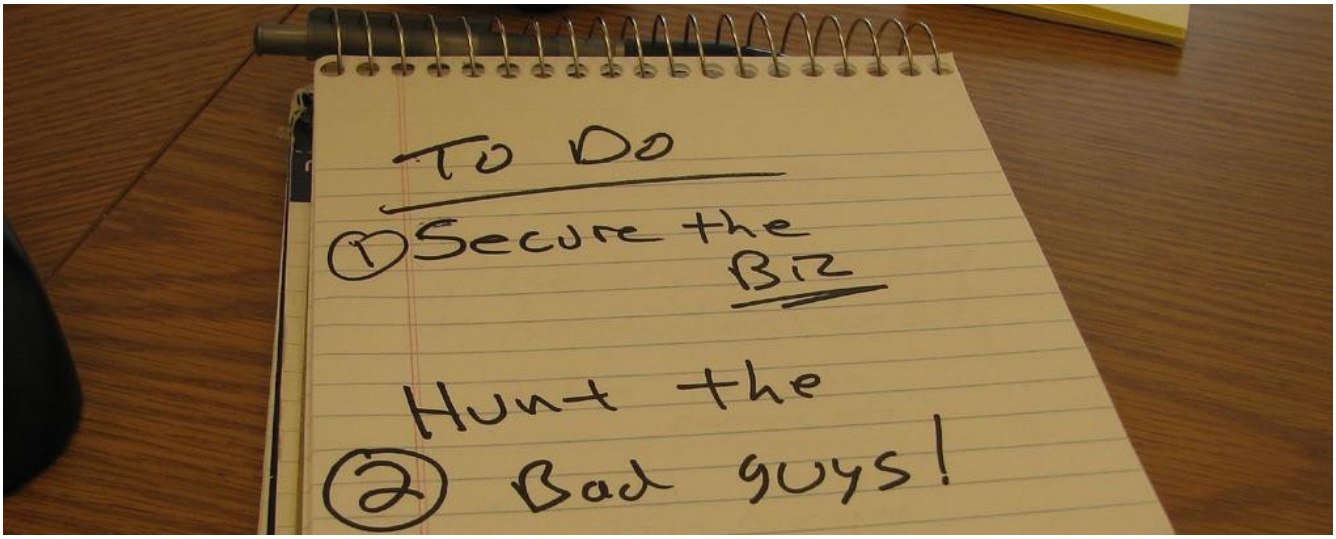
ToDoList

L'authentification



Sommaire

1) Le système de sécurité Symfony

En termes de sécurité, Symfony a séparé deux mécanismes distincts: l'authentification et l'autorisation.

1.1/ L'Authentification :

L'authentification est le processus qui va définir qui est l'utilisateur. Soit il n'est pas identifié sur le site et il est anonyme, soit il est identifié (via le formulaire d'identification) et il est un membre du site. C'est ce que la procédure d'authentification va déterminer. Ce qui gère l'authentification dans Symfony s'appelle un firewall.

Ainsi vous pourrez sécuriser des parties de l'application en forçant le visiteur à être un membre authentifié. Si le visiteur l'est, le firewall va le laisser passer, sinon il le redirigera sur la page d'identification. Cela se fera donc dans les paramètres du firewall.

1.2/ L'Autorisation :

L'autorisation est le processus qui va déterminer si l'utilisateur a le droit d'accéder à la ressource (page) demandée. Il agit donc après le firewall. Ce qui gère l'autorisation dans Symfony s'appelle l'***access control***.

Par exemple, dans notre application, seules les personnes ayant le rôle **ROLE_ADMIN** peuvent accéder aux pages relatives à la gestion des utilisateurs, c'est ce que vérifiera l'***access control***.

1.3/ Processus général :

Lorsqu'un utilisateur tente d'accéder à une ressource protégée, le processus est le suivant:

- Un utilisateur souhaite accéder à une ressource protégée
- Le pare-feu redirige l'utilisateur vers le formulaire de connexion
- L'utilisateur soumet ses identifiants
- Le pare-feu authentifie l'utilisateur
- L'utilisateur authentifié renvoie la requête initiale
- Le contrôle d'accès vérifie les droits de l'utilisateur et autorise ou non l'accès à la ressource protégée

2) Configuration de la sécurité Symfony

2.1/ Les utilisateurs :

Les utilisateurs sont stockés dans la base de données.

Dans l'application, un utilisateur est représenté par l'entité **USER** qui se trouve dans "***src/Entity/User.php***". Cette entité étends la classe **UserInterface** de Symfony qui définit les fonctions :

● **getUsername() :** Renvoie le nom d'utilisateur utilisé pour authentifier l'utilisateur.

● **getPassword() :** Renvoie le mot de passe utilisé pour authentifier l'utilisateur. Lors de l'authentification, un mot de passe en texte brut sera salé, codé, puis comparé à cette valeur.

● **getRoles() :** Renvoie les rôles accordés à l'utilisateur.

● **getSalt() :** Renvoie le sel qui a été utilisé pour coder le mot de passe. Cela peut retourner null si le mot de passe n'a pas été encodé en utilisant un sel.

● **eraseCredentials() :** Supprime les données sensibles de l'utilisateur. Ceci est important si, à un moment donné, des informations sensibles comme le mot de passe en texte brut sont stockées sur cet objet.

Les attributs ***username*** et ***tasks*** ont été rajouter à l'entité **USER** pour avoir la possibilité de récupérer le pseudo et les tâches d'un utilisateur.

Une contrainte d'unicité est appliquée à l'attribut username et email afin de ne pas avoir de doublon.

```
# src/Entity/User.php

/**
 * @ORM\Table("user")
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 * @UniqueEntity(
 *  fields = {"username"},
 *  message = "Ce pseudo est déjà utilisé"
 * )
 * @UniqueEntity(
 *  fields = {"email"},
 *  message = "Cet email est déjà utilisé"
 * )
 */
class User implements UserInterface
{
```

2.2/ CONFIGURATION DU FICHIER SECURITY.YAML :

Le fichier de configuration de la sécurité **security.yaml**, qui se situe dans ***"config/packages/"*** décrit les règles d'authentification et d'autorisation de l'application. Les informations de configuration du fichier **security.yaml** sont utilisées par la classe **USER**.

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

security:

    encoders:
        App\Entity\User:
            algorithm: auto
            cost: 12

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

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            lazy: true
            provider: app_user_provider
            pattern: ^/
            form_login:
                login_path: login
                check_path: login_check
                always_use_default_target_path:  true
                default_target_path:  /
            logout: ~

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

    role_hierarchy:
        ROLE_ADMIN: [ROLE_USER]
```

2.2.1) Encoders :

La partie ***encoders***, permet de déterminer l'algorithme que l'on souhaite utiliser pour l'encodage de données.

```
# config/packages/security.yaml

encoders:
    App\Entity\User:
        algorithm: auto
        cost: 12
```

Ici, on indique que l'on souhaite, pour l'entité **USER** (concernant le mot de passe), sélectionner automatiquement le meilleur algorithme de hachage possible (***auto)***, donc elle ne fait pas référence à un algorithme spécifique et elle changera à l'avenir et on indique un ***cost*** de 12. C'est à dire, que l'on détermine la durée pendant laquelle un mot de passe sera codé. Chaque incrément de ***cost*** double le temps nécessaire pour encoder un mot de passe. Le ***cost*** peut être dans la plage de 4-31.

2.2.2) Providers :

La partie **providers**, permet d'indiquer où se situe les informations que l'on souhaite utiliser pour l'authentification d'un utilisateur. C'est un «fournisseur d'utilisateurs».

```
# config/packages/security.yaml

providers:
    app_user_provider:
        entity:
            class: App\Entity\User
            property: username
```

Ici, on définit un seul provider **app_user_provider**. Ce provider utilise la classe **USER** que nous avons défini et l'attribut **username** va servir d'identifiant à l'utilisateur pour se connecter. Nous pouvons changer cette valeur en un attribut que nous aurons défini dans la classe **USER**. On peut indiquer la classe **USER**, puisqu'elle implémente **UserInterface**.

2.2.3) Firewalls :

La partie **firewalls**, permet de vérifier si l'utilisateur authentifié est autorisé a accéder à certaines parties/actions de l'application.

```
# config/packages/security.yaml

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

    main:
        anonymous: true
        lazy: true
        provider: app_user_provider
        pattern: ^/
        form_login:
            login_path: login
            check_path: login_check
            always_use_default_target_path:  true
            default_target_path:  /
        logout: ~
```

Ici, nous avons deux firewalls :

Le firewall **dev** n'est pas important, il s'assure simplement que les outils de développement de Symfony qui se trouvent sous les URL ***"/_profileret"*** et ***"/_wdt"*** ne sont pas bloqués par notre sécurité.

Le firewall **main** gère toutes les autres URLs :

● **anonymous: true** - L'accès en tant qu'anonyme, c'est à dire sans être authentifié, est autorisé. Les ressources sont protégées à travers des rôles.

● **provider: app_user_provider** - On indique le provider qui sera utilisé

● **pattern:** ^/ - C'est un masque d'URL. Cela signifie que toutes les URL commençant par "/" sont protégées par ce pare-feu.

● **form_login** - C'est la méthode d'authentification utilisée pour ce pare-feu. Il correspond à un formulaire de connexion.

```
# templates/security/login.html.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 class="login-form" action="{{ path('login_check') }}" method="post">
        <div class="input-login-form">
            <div class="input-username">
                <label for="username">Nom d'utilisateur :</label>
                <input type="text" id="username" name="_username" value="{{ last_username
}}" />
            </div>
            <div class="input-password">
                <label for="password">Mot de passe :</label>
                <input type="password" id="password" name="_password" />
            </div>
        </div>
        <button class="btn btn-success" type="submit">Se connecter</button>
    </form>
{% endblock %}
```

Ils possèdent les options :

● **login_path: login** - C'est la route correspondant au formulaire de connexion. Elle utilise la méthode **loginAction()** de **SecurityController**.

```
# src/Controller/SecurityController.php

class SecurityController extends AbstractController // Permet d'utiliser la méthode render
{
    /**
     * @Route("/login", name="login")
     * @param AuthenticationUtils $authenticationUtils
     * @return Response
     */
    public function loginAction(AuthenticationUtils $authenticationUtils)
    {
        // Si l'utilisateur est déjà connecté
        if ($this->getUser() != null) {
            // Redirection vers la page d'accueil
            return $this->redirectToRoute('homepage');
        }
        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();
        return $this->render('security/login.html.twig', array(
            'last_username' => $lastUsername,
            'error'         => $error,
        ));
```

● **check_path: login_check -** C'est la route de validation du formulaire de connexion. Les identifiants saisis par l'utilisateur seront vérifiés.

● **always_use_default_target_path: true -** Ignore l'URL demandée précédemment et redirige automatiquement vers la page par défaut: default_target_path.

● **default_target_path: / -** Une fois l'utilisateur authentifié, redirige automatiquement vers la page d'accueil.

Pour faire fonctionner la déconnexion **logout**, la route a été créé dans le fichier ***"config/routes.yaml"***.

```yaml
# config/routes.yaml

logout:
  path: /logout
  controller: App\Controller\LoginController::logoutCheck
```

2.2.4) Access_control :

La partie **access_control**, permet de limiter l'accès à certains espaces de l'application selon le rôle de l'utilisateur.

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

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

L'URL **"/login"** est accessible sans être authentifié.
L'URL **"/users"** n'est accessible qu'aux utilisateurs authentifiés possédant le rôle **ROLE_ADMIN**.
Tous le reste de l'application est accessible uniquement aux utilisateurs authentifiés possédant le rôle **ROLE_USER**.

2.2.5) Role_hierarchy :

La partie **role_hierarchy**, permet de définir des règles d'héritage de rôle en créant une hiérarchie de rôles.

```
# config/packages/security.yaml

role_hierarchy:
    ROLE_ADMIN: [ROLE_USER]
```

Ici, un utilisateur possédant le rôle **ROLE_ADMIN** possédera automatiquement le rôle **ROLE_USER**.

Cette méthode est très pratique pour afficher différents contenus en fonction des rôles des utilisateurs. Pour cela, Twig a la fonction **is_granted ()**. Ce qui permet d'avoir dans le menu de navigation l'accès à la gestion des utilisateurs uniquement pour les utilisateurs ayant le **ROLE_ADMIN**.

```
# templates/base.html.twig

{% if is_granted('ROLE_ADMIN') %}
    <a href="{{ path('user_create') }}" class="btn btn-primary">Créer un utilisateur</a>
    {% if 'user_list' != app.request.attributes.get('_route') %}
        <a href="{{ path('user_list') }}" class="btn btn-primary">Liste des
utilisateurs</a>
    {% endif %}
{% endif %}
```

3) FONCTIONNEMENT

Lorsque le firewall lance le processus d'authentification, il redirige l'utilisateur vers le formulaire de connexion.

Ce formulaire, une fois remplis, va transmettre les identifiants (username et password) à l'itinéraire **login_check**. C'est le système de sécurité Symfony qui se chargera de la gestion de ce formulaire. C'est à dire que nous ne définissons pas de fonction à exécuter pour la route **login_check**, Symfony interceptera la demande du visiteur sur cette route et gérera l'authentification lui-même.

En cas de succès, le visiteur sera authentifié. En cas d'échec, Symfony le renverra sur le formulaire de connexion.

4) ERREURS COURANTES

Ne pas oublier la définition des routes :

Il est impératif de créer les routes **“login”**, **“login_check”** et **“logout”**. Ce sont des routes obligatoires, si elles sont oubliées, il risque d'y avoir des erreurs 404 au milieu du processus d'authentification.

Les firewalls ne partagent pas :

Dans le cas de l'utilisation de plusieurs firewalls, sachez qu'ils ne partagent rien entre eux. Ainsi, si vous êtes authentifiés sur l'un, vous ne le serez pas forcément sur l'autre, et inversement. Cela permet d’accroître la sécurité lors d'un paramétrage complexe.

Bien mettre “/login_check” derrière le pare-feu :

Il faut s'assurer que l'URL du **check_path** (/login_check) est bien derrière le firewall que vous utilisez pour le formulaire de connexion. En effet, c'est la route qui permet l'authentification au niveau du firewall. Mais comme les firewalls ne partagent rien, si cette route n'appartient pas au firewall que l'on souhaite, vous aurez droit à une erreur.

Ne pas sécuriser le formulaire de connexion :

Si le formulaire est sécurisé, comment les nouveaux arrivants vont-ils pouvoir s'authentifier ? En l'occurrence, il faut faire attention que la page /login ne requière aucun rôle.