Documentation Technique - Authentification

v1.0 du 15/12/2021

1. Sommaire

2. Mise en place de l'Authentification

2.1. Les bundles à avoir

Voici la liste des bundles qu'il faut avoir dans `composer.json` :

- symfony/maker-bundle : composer require –dev symfony/security-bundle
- symfony/security-bundle : composer require symfony/security-bundle

Vérifiez aussi leur présence dans `config/bundles.php` :

```
Symfony\Bundle\MakerBundle\MakerBundle::class => ['dev' => true],
Symfony\Bundle\SecurityBundle\SecurityBundle::class => ['all' => true],
```

2.2. L'entité User

Le moyen le plus simple de générer une classe User prête à l'emploi, est de passer par le Maker Bundle via cette commande : php bin/console make:user .

Cette entité devra implémenter le *Symfony\Component\Security\Core\User\UserInterface* ainsi que *Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface*.

Nos utilisateurs représentés par cette classe User seront par la suite stockés en base de données sous une table du même nom que cette classe.

Pour plus d'information suivre ce lien.

2.3. Le contrôleur Login

La méthode de construction utilisée pour notre contrôleur viens de la documentation Symfony : php bin/console make:controller Login

L'autre commande, non appliquée ici mais utile à connaître pour créer son contrôleur liée à l'authentification PLUS les vues html : php bin/console make:auth .

3. Paramétrage de l'Authentification

Tout ce qui suit se passe dans le fichier `config/packages/security.yaml`.

3.1. Le Password Hasher

```yaml
# config/packages/security.yaml
security:
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: "auto"
```

Introduit depuis Symfony 5.3, et remplaçant l'*encoder*, c'est **la partie qui gère le choix de l'algorithme d'encodage** des mots de passes utilisateurs.
Le mode auto permet au hasher de sélectionner l'algorithme le plus sûr actuellement pour l'application (Bcrypt / 255 caractères).

 # Pour en savoir plus voir ce lien.

3.2. Le Provider

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

Le provider **sert à désigner à quel endroit le composant Sécurité de Symfony doit rechercher les informations d'un utilisateur**, en lui indiquant plusieurs éléments:

- un **nom** qui servira de référence pour la configuration du firewall, ici c'est app_user_provider,
- une **zone de recherche** : soit en base de donnée avec entity, soit directement dans le fichier yaml avec memory,
- la **classe ainsi que la propriété** qui servira d'identifiant de connexion avec le mot de passe (dans le cas d'une recherche en BDD seulement, pour memory suivre ce lien).

 # Pour en savoir plus voir ce lien.

3.3. Le Firewall

```yaml
# 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_csrf: true
        logout:
                    path: logout
```

Le firewall **définit les parties de notre application qui sont sécurisées et la façon dont nos utilisateurs pourront s'authentifier** (par exemple, formulaire de connexion, jeton API, etc).

Ici dev et main représentent le nom des sections par défaut du firewall, mais il est possible d'en créer d'autres (en savoir plus).

Principe de fonctionnement : le firewall va rechercher, en fonction de l'ordre de déclaration des sections, la correspondance avec un pattern ou un host si définie dans les section, ou s'arrêtera sur la section qui n'en définit pas (*pour cette raison pensez à toujours mettre la section main en dernière*).
 # pattern et host sont les 2 clés servant de filtre au firewall (en savoir plus).

Le but de la section dev, via le pattern est de s'assurer que le firewall n'est pas actif sur les outils de développement de Symfony, il n'y a donc rien à modifier sur cette partie.

Concernant la section main, elle représente ici notre application, en voici les détails :

- lazy: true indique à Symfony qu'il n'est pas nécessaire de charger l'utilisateur si les URLs/actions n'ont pas besoin d'un accès spécifique (type ROLE_USER), permettant une possible mise en cache et donc une amélioration des performances de l'application.(source)
- provider indique à Symfony d'utiliser un provider (ici notre app_user_provider) comme critère de contrôle.
- form_login indique à Symfony que l'authentification se base sur une authentification via login.
    - login_path indique vers quel URL ou le nom de route redirigé pour authentification.

- check_path indique l'URL ou le nom de la route qui se charge de l'authentification
- enable_csrf: true permet d'activer la prise en charge de la protection contre les attaques de type CSRF (Cross-site request forgery).
  Si activé il faut vérifier la présence du champ caché dans le template :

```
# templates/security/login.html.twig
{# ... #}
<form action="{{ path('login') }}" method="post">
    {# ... the login fields #}
    <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate')
}}">

</form>
```

- logout indique à Symfony l'URL le nom de la route qui se charge de la déconnexion.

 # Pour en savoir plus voir ce lien.

4. Gestion des Rôles

- Tout compte créé sera par la suite reconnu par Symfony comme ayant le ROLE_USER.
- Pour créer un nouveau rôle il suffit de respecter la convention de nommage suivante : ROLE_NOM_EN_MAJUSCULE .
- Actuellement deux Rôles existent : ROLE_USER et ROLE_ADMIN .

4.1. Hiérarchisation des Rôles

Il est possible d'attribuer une logique hiérarchique à nos Rôles afin d'éviter toutes répétitions des accès d'un Rôle à un autre.
Tout se passe dans le fichier `config/packages/security.yaml`.

```
# config/packages/security.yaml
security:
        # …
        role_hierarchy:
                ROLE_ADMIN: ROLE_USER
```

Dans notre cas le ROLE_ADMIN hérite des accès du ROLE_USER .

4.2. Restriction d'accès des URL

Il est possible de limiter l'accès au URL de notre application en fonction des Rôles.
Tout se passe dans le fichier `config/packages/security.yaml`.

```
# config/packages/security.yaml
security:
        # …
        access_control:
                - { path: ^/users/create, roles: PUBLIC_ACCESS }
                - { path: ^/users, roles: ROLE_USER }
                - { path: ^/tasks, roles: ROLE_USER }
```

- path indique l'URL à sécuriser et accepte les expressions régulières.
- roles indique quels Rôles peuvent accéder à l'URL (*si plusieurs Rôles soumis, les placer dans un array : [ROLE_USER, ROLE_ADMIN]*)

 # Pour en savoir plus voir ce lien.

4.3. Restriction d'accès des Routes et Contrôleurs

L'AbstractController hérité dans nos contrôleurs permet si nécessaire de restreindre les accès à une méthode en contrôlant les Rôles de l'utilisateur via la fonction : `$this->isGranted()`.

Il est aussi possible de contrôler les accès à une méthode directement via l'annotation *`@IsGranted()`* permis par l'utilisation du bundle sensio/framework-extra-bundle : composer require sensio/framework-extra-bundle .

De plus, il est possible de créer des autorisations spéciales via les Voters que vous pouvez retrouvez dans *src/Security/Voter* .
 # Pour en savoir plus voir ce lien.

4.4. Restriction d'accès dans les templates TWIG

Côté template TWIG, il y a la fonction `is_granted()` qui permet de contrôler les autorisations des utilisateurs afin d'afficher sous conditions certains éléments du template.
 # Pour en savoir plus voir ce lien.