L'authentification

Parmi tous les outils que Symfony met à notre disposition, il existe un bundle permettant de simplifier l'authentification.

Nous allons voir comment utiliser le composant Security de Symfony pour faire notre authentification. Pour rester simple, nous verrons le cas de 2 rôles (utilisateur et administrateur), ainsi qu'un affichage selon le type de rôle.

Installation du Bundle Security

Rien de bien compliquer ici, entrons la commande suivante en console :

```
composer require symfony/security-bundle
```

Si vous avez initialisé votre projet 'skeleton-website', cette étape ne devrait pas être nécessaire. Vérifiez en regardant si vous avez un fichier `security.yaml` dans le dossier `config`.

Configuration du composant Security

Pour configurer la manière dont l'authentification doit se faire, nous allons nous rendre dans le fichier `config/packages/security.yaml`.

Voici le contenu du fichier :

```
security:
    # https://symfony.com/doc/current/security.html#where-do-users-come-fro
m-user-providers
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: users_in_memory

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#firewalls-authe
ntication

            # https://symfony.com/doc/current/security/impersonating_user.h
tml
            # switch_user: true

    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
```

Nous avons une première partie (`providers`) où on va définir comment les utilisateurs vont être récupérés (mémoire, base de données, ...) :

```
providers:
        users_in_memory: { memory: null }
```

La seconde (`firewalls`) permet de définir les composants qui vont permettre d'authentifier les utilisateurs :

```
firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: users_in_memory

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#firewalls-authe
ntication

            # https://symfony.com/doc/current/security/impersonating_user.h
tml
            # switch_user: true
```

Quant à la dernière (`access-control`) permet de définir les niveaux d'accès à l'application :

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

Il nous faut maintenant créer notre table `user` dans notre base de données.

Création de la table `user`

Pour que l'authentification corresponde à nos besoins, il va nous falloir stocker les différents utilisateurs en base de données. On pourra stocker leurs informations (leur rôle notamment).

Créez une table `user` contenant les informations suivantes :

- un email (varchar 255, non null)
- un mot de passe (varchar 255, non null)
- un role (varchar 255, non null)

Rendons-nous ensuite dans l'entité `User` pour y implémenter la `UserInterface`

```
class User implements UserInterface
{

}
```

Selon l'IDE sur lequel vous travailler, vous observerez que 4 méthodes sont automatiquement créées :

- `public function getRoles()` : méthode qui retourne un tableau des roles des différents utilisateurs
- `public function getSalt()` : méthode utiliser pour certaines méthodes de chiffrement
- `public function getUsername()` : méthode qui retourne l'identifiant utiliser pour l'authentification

- `public function eraseCredentials()` : méthode qui permet d'effacer des informations sensibles (mot de passe par exemple) qui aurait pu être stocké dans l'entité.

Création d'un utilisateur

Créez un crud sur la table `user`.

Mettez en forme le formulaire d'ajout et sécurisez-le, faîtes en sorte que le champ `role` ne soit présent à l'affichage, le role par défaut lors de l'ajout d'un utilisateur sera 'client', par exemple.

Définissez donc le role par défaut d'un utilisateur sur 'client'.

Ajouter une méthode de cryptage pour le mot de passe (pour rappel, aucun mot de passe ne doit-être stocké en clair dans la base de données).

Vérifiez ensuite le bon fonctionnement de votre code en insérant un utilisateur dans la base de données.

Connexion

Commençons par créer un contrôleur que l'on nommera `SecurityController.php`, ainsi qu'une vue pour le formulaire de connexion (`security/login.html.twig`).

Concernant la vue, nous aurons le code suivant :

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

{% block title %}Connexion{% endblock %}

{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-sm-6 offset-sm-3">
                <form action="{{ path('login') }}" method="post">
                    <label for="email">Adresse Mail</label>
                    <input type="text" id="email" name="_username" class="f
orm-control" value="{{ last_username }}">
                    <label for="mdp">Mot de passe</label>
                    <input type="password" id="mdp" name="_password" class=
"form-control">
                    <input type="submit" class="btn btn-primary mt-3" value
="Se connecter">
                </form>
            </div>
        </div>
    </div>
{% endblock %}
```

Vous remarquerez que sur les `input` 'email' et 'password', les `name` ne correspondent pas aux champs de notre table. Par défaut, Symfony reconnait pour l'authentification les propriétés `username` et `password`. Pour correspondre, les `name` du formulaire doivent contenir ces 2 propriétés. Le _ devant fait partie de la nomenclature de Symfony.

Pour le contrôleur :

- Commençons par créer notre méthode, avec sa route :

```
 /**
     * @Route("/login", name="login")
     * @return Response
     */
    public function login(): Response
    {

    }
```

- Générons la vue qui affichera notre formulaire :

```
return $this->render('security/login.html.twig');
```

- Il pourrait être intéressant de récupérer les erreurs de connexion. Pour cela nous utiliserons le composant `AuthenticationUtils` :

```
/**
     * @Route("/login", name="login")
     * @param AuthenticationUtils $authenticationUtils
     * @return Response
     */
    public function login(AuthenticationUtils $authenticationUtils): Respon
se
    {
        // récupération des erreurs de connexion
        $error = $authenticationUtils->getLastAuthenticationError([
            'translation_domain' => 'security'
        ]);

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

- Enfin, on peut également récupérer le `username` du dernier utilisateur qui s'est connecté :

```
/**
* @Route("/login", name="login")
* @param AuthenticationUtils $authenticationUtils
* @return Response
*/
public function login(AuthenticationUtils $authenticationUtils): Response
{
   // récupération des erreurs de connexion
   $error = $authenticationUtils->getLastAuthenticationError([
            'translation_domain' => 'security'
           ]);
   // récupération du username du dernier utilisateur connecté
   $lastUsername = $authenticationUtils->getLastUsername();

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

Configuration de `UserInterface`

Pour configurer les getters ajoutés par la `UserInterface`, rendons-nous dans l'entité `User.php`.

getRoles()

Par défaut, le composant Security de Symfony propose 2 roles : `[ROLE_ADMIN]` et `[ROLE_USER]`.

Le role de l'utilisateur permettra de définir les actions possibles qui lui seront attribuées.

Pour faire simple, nous utiliserons également 2 roles, client et administrateur, directement renseigner dans la table `user`.

Il nous faut donc indiquer que si dans la table `user` nous avons 'client' alors l'utilisateur se voit attribuer `[ROLE_USER]`.

Si dans la table `user` nous avons 'administrateur' alors l'utilisateur se voit attribuer `[ROLE_ADMIN]`.

```
    public function getRoles()
    {
        if ($this->role == "administrateur")
            return ["ROLE_ADMIN"];
        if ($this->role == "client")
            return ["ROLE_USER"];
        return [];
    }
```

getSalt()

Ici, nous ne définissons pas méthode spécifique de chiffrement, donc la méthode retournera une chaine de caractère vide :

```
    public function getSalt()
    {
        return "";
    }
```

getUsername()

Puisque nous utilisons une propriété différente de `username` pour notre identification, nous devons indiquer à quelle propriété correspond `username`. Dans notre cas, l'adresse mail servira d'identifiant à notre utilisateur. Nous aurons donc:

```
  public function getUsername()
    {
        return $this->getEmail();
    }
```

eraseCredentials()

Dans notre cas, nous ne toucherons pas à cette méthode.

Configuration du fichier `security.yaml`

Nous allons dans premier temps indiquer que l'utilisateur se trouve dans la base de données. Nous devons aussi indiquer que l'utilisateur devra se connecter avec son adresse mail. Donc dans la clé `providers`, nous allons entrer le code suivant (attention à l'indentation) :

```
from_database:
      entity:
      class : App\Entity\USer
      property: email
```

Ensuite nous devons rappeler ce `provider` dans la clé `firewalls` :

```
firewalls
    main:
        provider: from_database
```

Il nous faut également indiquer par quelle méthode l'utilisateur pourra se connecter. Dans notre cas, il se connectera avec un formulaire que nous avons créé :

```
firewalls:
    main:
        anonymous: true
        form_login:
            check_path: /login
        provider: from_database
```

Nous allons définir le chemin permettant la déconnexion, ainsi que la redirection suite à cette déconnexion de la même manière :

```
firewalls:
    main:
        anonymous: true
        logout:
            path: /logout
            target: /
        form_login:
            check_path: /login
        provider: from_database
```

Si vous voulez restreindre l'accès à certaines routes selon le rôle, vous pouvez décommenter (ajouter / modifier) les lignes suivantes dans la clé `acces_control` :

```
- { path: ^/admin, roles: ROLE_ADMIN }
- { path: ^/profile, roles: ROLE_USER }
```

Enfin il nous faut renseigner l'encoder utilisé pour le hash du mot de passe :

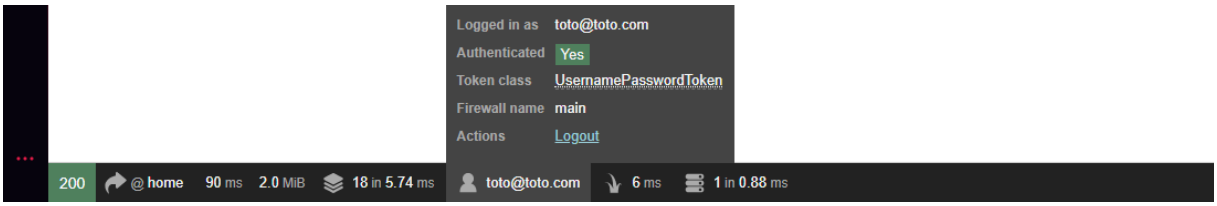
```
    encoders:
        App\Entity\USer:
            algorithm: bcrypt
            cost: 12
```

Nous renseignons où se trouve le hash, ainsi que la méthode de hashage.

Il nous faut définir une route pour la déconnexion. Pour cela, nous allons dans le fichier `config/route.yaml`, et ajoutons les lignes suivantes :

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

Testez la connexion. Si la connexion se fait, vous verrez l'identifiant de connexion dans la barre de débogage de Symfony.



Personnalisation de la vue

Une fois que la connexion est effective :

- Ajoutez sur la vue un bouton de connexion et de déconnexion, qui s'affichent selon s'il y a connexion ou non de l'utilisateur.
- Faire un affichage différent de la liste des produits selon si c'est un client qui se connecte, ou si c'est un administrateur.
- Codez une interface permettant à l'utilisateur de modifier ses informations personnelles.

Pour vous aider, consulter la documentation Symfony ici et là.