Gestion de connection avec Symfony 7

Dans un projet Symfony, il faut faire les étapes suivantes :

1) **Créer l'entité User** :

```
symfony console make:user
```

```
 The name of the security user class (e.g. User) [User]:
 > User

 Do you want to store user data in the database (via Doctrine)? (yes/no)
[yes]:
 > yes

 Enter a property name that will be the unique "display" name for the
user (e.g. email, username, uuid) [email]:
 > email

 Will this app need to hash/check user passwords? Choose No if passwords
are not needed or will be checked/hashed by some other system (e.g. a
single sign-on server).

 Does this app need to hash/check user passwords? (yes/no) [yes]:
 > yes
```

2) **Configurer l'entité User** :

Assurez-vous que votre entité `User` implémente l'interface `UserInterface` et a les annotations nécessaires pour les propriétés telles que `@ORM\Entity` et `@ORM\Column`.

3) **Faire la migration** :

Si votre utilisateur est une entité Doctrine, n'oubliez pas de créer les tables en créant et en lançant une migration :

```
symfony console make:migration
symfony console doctrine:migrations:migrate
```

4) **Créer le contrôleur User** :

```
symfony console make:controller
```

5) **Créer le formulaire de connexion** :

```
symfony console make:form
```

Nom : Connection

Entité : User

```
public function buildForm(FormBuilderInterface $builder, array $options): void
  {
    $builder
      ->add('email')
      //->add('roles')
      ->add('password')
    ;
  }
```

Modifier la fonction buildForm pour mettre des paramètres :

```php
public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('email', EmailType::class,
                ['label'=>'E-mail',
                'attr'=> ['placeholder'=> 'Saisir votre e-mail.']
                ])


            //->add('roles')
            ->add('password', PasswordType::class,
            ['label'=>'Mot de passe',
            'attr'=> ['placeholder'=> 'Saisir votre mot de passe.']
            ])
        ;
    }
```

N'oublier pas d'ajouter :

```php
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
```

6) Créer la fonction ajouter un user dans le contrôleur User :

```
#[Route('/user/add', name: 'app_user_add')]
   // public function addUser(UserPasswordEncoderInterface
$passwordEncoder, EntityManagerInterface $entityManager)
   public function addUser(EntityManagerInterface $entityManager)

   {
        // Créer une instance de l'entité User
        $user = new User();
        $form = $this->createForm(ConnectionType::class, $user);

        return $this->render('user/addUser.html.twig', [
            'form' => $form->createView()
            ]);
   }
```

Le twig `addUser.html.twig' :`

```
<h1>S'inscrire</h1>
{{ form_start(form) }}
    {{ form_widget(form) }}


    {{ form_end(form) }}
```

Ajouter la validation du formulaire :

```
          ->add('submit', SubmitType::class,
            ['label'=>'Valider'
            ])
```

Modifier l'action dans le contrôleur :

```
public function addUser(UserPasswordHasherInterface
$passwordHasher,EntityManagerInterface $entityManager, Request
$request):Response

   {
        // Créer une instance de l'entité User
        $user = new User();
        $form = $this->createForm(ConnectionType::class, $user);
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid())
        {
```

```
            $user = $form->getData();
            $textPassword = $user->getPassword();
            $hashedPassword = $passwordHasher->hashPassword($user,
$textPassword);
            $user->setPassword($hashedPassword);
            $user->setRoles(['ROLE_USER']);
            $entityManager->persist($user);
            $entityManager->flush();
        }
        return $this->render('user/addUser.html.twig', [
            'form' => $form->createView()
            ]);

    }
```

Vérifier la présence de ***password_hashers*** dans security.yaml :

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

C'est fait quand on a fait ***make :user***

7) **Gérer la connexion :**
   - Création d'un contrôleur de connexion (***Login***) :

     symfony console make:controller

     ➔ Login

```
class LoginController extends AbstractController
{
    #[Route('/login', name: 'app_login')]
    public function index(): Response
    {
        return $this->render('login/index.html.twig', [
            'controller_name' => 'LoginController',
        ]);
    }
}
```

Vérifier la présence de la route vers Login dans security.yaml

```
main:
            lazy: true
            provider: app_user_provider
            form_login:
                login_path: app_login
                check_path: app_login
```

- Tester la route
- Modifier la méthode login :

```
public function login(AuthenticationUtils $authenticationUtils): Response
    {
        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();

        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

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

Ajouter un twig login :

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

{% block title %}Log in!{% endblock %}

{% block body %}
    <form method="post">
        {% if error %}
            <div class="alert alert-danger">{{
error.messageKey|trans(error.messageData, 'security') }}</div>
        {% endif %}

        {% if app.user %}
            <div class="mb-3">
                You are logged in as {{ app.user.userIdentifier }}, <a
href="{{ path('app_logout') }}">Logout</a>
            </div>
        {% endif %}

        <h1 class="h3 mb-3 font-weight-normal">Please sign in</h1>
```

```
        <label for="username">Email</label>
        <input type="email" value="{{ last_username }}" name="_username"
id="username" class="form-control" autocomplete="email" required autofocus>
        <label for="password">Password</label>
        <input type="password" name="_password" id="password" class="form-
control" autocomplete="current-password" required>

        <input type="hidden" name="_target_path"
               value="/livre/"
        >


        <button class="btn btn-lg btn-primary" type="submit">
            Sign in
        </button>
    </form>
{% endblock %}
```

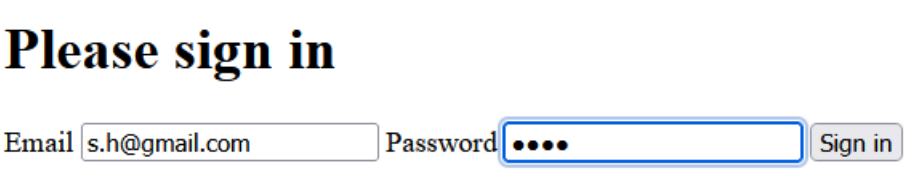
L'élément ***<input type="hidden" name="_target_path" value="/livre/">*** est généralement utilisé dans les formulaires de connexion, en particulier lors de l'utilisation du composant de sécurité et du système d'authentification.

Le champ ***_target_path*** est utilisé pour spécifier la redirection vers laquelle l'utilisateur doit être redirigé après s'être connecté avec succès. Dans cet exemple, après une connexion réussie, l'utilisateur serait redirigé vers l'URL ***"/livre/".***

Lorsqu'un utilisateur tente d'accéder à une page sécurisée sans être connecté, Symfony peut le rediriger vers la page de connexion. Cependant, Symfony conserve l'URL d'origine demandée dans le paramètre ***_target_path***. Après la connexion réussie, Symfony utilise cette valeur pour rediriger l'utilisateur vers l'URL demandée initialement.

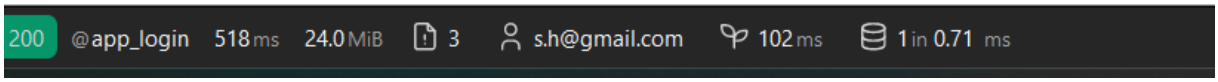
Assurez-vous que le chemin spécifié dans la valeur de l'attribut `value` est conforme à votre configuration de routage dans Symfony et correspond à une route valide de votre application.

Lancer le login :



Valider

On peut voir en bas les informations de la connexion :



8) Gérer les rôles :

Dans Symfony, le rôle ***ROLE_USER*** n'est pas automatiquement inclus dans le rôle ***ROLE_ADMIN***. Ces deux rôles sont généralement indépendants l'un de l'autre, et leur inclusion dans un utilisateur est déterminée par la configuration de sécurité spécifique de votre application.

Lorsque vous configurez les rôles dans Symfony, vous définissez explicitement quels rôles sont nécessaires pour accéder à certaines parties de votre application. Si un utilisateur a le rôle ***ROLE_ADMIN***, cela ne signifie pas nécessairement qu'il a automatiquement le rôle ***ROLE_USER*** à moins que vous ne le spécifiiez dans votre système d'autorisation.

Par exemple, dans un fichier de configuration de sécurité (***security.yaml***), vous pourriez avoir quelque chose comme ceci :

```yaml
yaml
security:
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
        - { path: ^/user, roles: ROLE_USER }
```

Cela signifie que pour accéder aux URL commençant par ***/admin***, l'utilisateur doit avoir le rôle ***ROLE_ADMIN***. De même, pour accéder aux URL commençant par ***/user***, l'utilisateur doit avoir le rôle ***ROLE_USER***.

En résumé, la hiérarchie entre les rôles dépend de la manière dont vous configurez votre système d'autorisation dans Symfony. Par défaut, Symfony ne suppose pas que ***ROLE_ADMIN*** inclut automatiquement ***ROLE_USER***. Vous devez définir ces relations explicitement dans votre configuration.

Dans Symfony, vous pouvez inclure le rôle ***ROLE_USER*** dans le rôle ***ROLE_ADMIN*** en utilisant une hiérarchie de rôles. Cela se fait généralement dans votre configuration de sécurité, dans le fichier ***security.yaml***.

Voici un exemple de configuration qui déclare une hiérarchie entre les rôles ***ROLE_ADMIN*** et ***ROLE_USER*** :

```yaml
yaml
security:
    role_hierarchy:
         ROLE_ADMIN: [ROLE_USER]

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

Dans cet exemple, la section ***role_hierarchy*** définit une hiérarchie des rôles. Elle indique que le rôle ***ROLE_ADMIN*** inclut automatiquement le rôle ***ROLE_USER***. Ainsi, un utilisateur qui possède le rôle ***ROLE_ADMIN*** aura également implicitement le rôle ***ROLE_USER***.

N'oubliez pas que cette configuration dépend de votre modèle d'authentification et d'autorisation spécifique. Assurez-vous d'adapter ces configurations en fonction de la logique de votre application et de vos besoins particuliers. Après avoir modifié votre configuration, assurez-vous également de vider le cache de Symfony pour que les changements prennent effet. Vous pouvez le faire en exécutant la commande suivante dans votre terminal :

```
bash
symfony console cache:clear
```

Par ailleurs, si vous avez besoin de conditions plus complexes pour autoriser ou non l'accès à une action en fonction du type d'utilisateur, vous pouvez utiliser des expressions plus élaborées dans le fichier de configuration. Par exemple :

```
yaml
# security.yaml

security:
    # ...

    access_control:
         - { path: ^/admin, roles: ROLE_ADMIN and
is_granted('ROLE_SUPER_ADMIN') }
         - { path: ^/user, roles: ROLE_USER and
is_granted('USER_TYPE_A') }
```

Dans cet exemple, l'accès à l'URL commençant par ***/admin*** est autorisé seulement si l'utilisateur a le rôle ***ROLE_ADMIN*** et possède également le rôle ***ROLE_SUPER_ADMIN***. De même, l'accès à l'URL commençant par ***/user*** est autorisé uniquement si l'utilisateur a le rôle ***ROLE_USER*** et détient le rôle ***USER_TYPE_A***.

Si vous souhaitez protéger une partie spécifique du code (par exemple, une méthode dans un contrôleur) en fonction du type d'utilisateur, vous pouvez utiliser les annotations de sécurité directement dans le code source.

Voici un exemple d'utilisation des annotations de sécurité dans un contrôleur Symfony :

```
php
// src/Controller/YourController.php

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

class YourController extends AbstractController
{

#[Route("/your-secured-route", name="your_secured_route")]
```

```
#[Security("is_granted('ROLE_USER')")]


    public function yourSecuredAction(): Response
    {
        // Votre code ici

        return $this->render('your_template.html.twig');
    }
}
```

Dans cet exemple, la méthode *yourSecuredAction* est annotée avec *#[Security("is_granted('ROLE_USER')")],* ce qui signifie que l'accès à cette action est autorisé uniquement aux utilisateurs ayant le rôle *ROLE_USER*.

Vous pouvez personnaliser cette annotation en fonction de vos besoins, en utilisant des expressions plus complexes pour définir les conditions d'accès. Assurez-vous d'adapter ces annotations en fonction de votre modèle d'authentification et d'autorisation spécifique.

Notez que l'utilisation des annotations de sécurité nécessite l'installation du bundle `sensio/framework-extra-bundle`. Vous pouvez l'ajouter à votre projet via Composer si ce n'est pas déjà fait :

bash
```
composer require sensio/framework-extra-bundle
```

Si vous souhaitez protéger un ensemble de code à l'intérieur d'une méthode en fonction du type d'utilisateur, vous pouvez utiliser la classe `Security` du composant Security de Symfony. Voici comment vous pouvez faire cela dans une méthode de votre contrôleur :

php
```
use Symfony\Component\Security\Core\Security;

class YourController extends AbstractController
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    public function yourMethod(): Response
    {
        // Votre code ici

        // Vérifiez si l'utilisateur a le rôle nécessaire
        if ($this->security->isGranted('ROLE_USER')) {
            // Le code que seuls les utilisateurs avec le rôle ROLE_USER
peuvent exécuter
            // ...

            return $this->render('your_template.html.twig');
        } else {
            // Gérez le cas où l'utilisateur n'a pas les autorisations
nécessaires
            throw $this->createAccessDeniedException('Accès interdit');
```

```
        }
    }
}
```

Dans cet exemple, la méthode ***yourMethod*** vérifie si l'utilisateur a le rôle nécessaire (dans cet exemple, le rôle ***ROLE_USER***). Si c'est le cas, le code à l'intérieur du bloc conditionnel sera exécuté. Sinon, une exception ***AccessDeniedException*** sera lancée.

Si vous souhaitez appliquer des contrôles d'accès plus fins à l'intérieur d'une méthode, vous pouvez également utiliser l'annotation ***Security*** à l'intérieur de la méthode pour définir des règles d'accès spécifiques. Par exemple :

```
php
use Symfony\Component\Security\Core\Security;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security as
SensioSecurity;

class YourController extends AbstractController
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    /**
     * @Route("/your-route", name="your_route")
     * @SensioSecurity("is_granted('ROLE_USER')")
     */

#[Route("/your-route", name="your_route")]
#[SensioSecurity("is_granted('ROLE_USER')")]

    public function yourMethod(): Response
    {
        // Votre code ici

        return $this->render('your_template.html.twig');
    }
}
```

Dans cet exemple, la méthode ***yourMethod*** est annotée avec ***#[SensioSecurity("is_granted('ROLE_USER')")],*** ce qui signifie que l'accès à cette méthode est autorisé uniquement aux utilisateurs ayant le rôle ***ROLE_USER***.