TP6 : Authentification / Login

L'objectif principal est de créer un sytème d'authentification, via formulaire, pour permettre notamment de gérer l'accès ou non à certaines pages.
Ainsi, nous devons pouvoir nous connecter en tant qu'utilisateur « simple », ou en tant qu'administrateur pour pouvoir gérer certaines fonctionnalités ; nous devons de ce fait pouvoir créer un compte et le modifier si besoin.

1. Création de l'entité sécurisée « User »

- Documentation liée : « https://symfony.com/doc/current/security.html »

A/ Création de l'entité et de ses attributs
On repart sur nos classiques via le Maker :

php bin/console make:user

Nous voulons une classe « User », stockée en BDD, avec des mots de passe sécurisés en bcrypt.

- Notre entité User implémente une interface : allez la voir, et si vous êtes en MySql et Maria dB il faut modifier '$role' en array.
  -> Qu'est-ce signifie le « unique=true » dans l'annotation liée à l'ORM sur l'attribut « email »?

- Ajouter les champs « FirstName » et « LastName » dans l'entité User, pensez aux getters / setters. (Et au form, et au twig :)).
Puis mettez à jour votre BDD.

=> Comment apparait le password en BDD ? Pourquoi ? Dans quel fichier l'encodage est-il réalisé ?

B/ Provider & encoder

- Dans la configuration `security.yaml` le make:user a rajouté un provider et un encoder

```
encoders:
    App\Entity\User:
        algorithm: auto
providers:
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email
```

=> - Provider : c'est une classe Php liée à la sécurité de Symfony qui permet à chaque rêquete de renvoyer le user courant depuis la session. C'est le fournisseur qui va dire au pare-feu de Symfony qui est le fournisseur d'entités. Il précise également quel attribut sera utilisé comme identifiant.
- Qu'est-ce qu'un encoder ?

C/ Role

- Un role correspond à un ensemble de permissions qu'on veut donner à un ou plusieurs utilisateurs. Une hiérarchie est définie, avec un système d'héritage :
  - un role utilisateur aura certains droits de CRUD sur certaines entités,
  - un role administrateur aura le role utilisateur + d'autres droits qu'on lui aura défini.
  - un role super administrateur aura le role utilisateur + le role administrateur + d'autres droits qu'on lui aura défini.

=> Il va nous falloir modifier notre entité « User » pour qu'elle puisse avoir par défaut le rôle « ROLE_USER ».

Nous allons modifier notre entité `User` pour qu'il puisse avoir par defaut le role `ROLE_USER`

Nous allons rajouter en haut de l'entité deux constantes :

```
class User implements UserInterface
{
    public const ROLE_USER = 'ROLE_USER';
    public const ROLE_ADMIN = 'ROLE_ADMIN';
```

Examinez la méthode getRoles() : qu'a-t-elle de spécial ? Pourquoi?

2. Partie Authentification qui va nous permettre de se connecter sur le site

A/ Login

Nous allons créer le formulaire de login et le guard pour gérer notre authentification : définition GUARD, toujours via le maker :

```
php bin/console make:auth
```

Je veux une authentification par formulaire ; je vous laisse allez inspecter les fichiers ainsi crées.

- Il faut maintenant indiquer la redirection dans la méthode « onAuthenticationSuccess » de votre AppCustomAuthenticator dans le dossier Security.

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
    {
        if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
            return new RedirectResponse($targetPath);
        }

        // For example : return new RedirectResponse($this->router->generate('some_route'));
        throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
    }
```

=> Le AppCustomAuthenticator n'étend pas l'AbstractController, nous n'avons donc pas accès au raccourci et à la méthode « $this->redirectToRoute('nomdelaroute') ».
En revanche, nous pouvons faire appel au constructeur « RedirectResponse » comme indiqué ci-dessus:

```
return new RedirectResponse($this->router->generate('some_route'));
```

Indiquer la route correspondante à la page d'accueil.

- Dans le fichier de configuration de security.yaml, le lien se fait par `guard`

```
guard:
    authenticators:
```

```
            - App\Security\AppAuthenticator
```

B/ Logout

- Maintenant que nous pouvons nous connectez, il va falloir aussi nous déconnecter.

Nous allons créer une méthode dans « SecurityController » qui va nous permettre de déclarer une route pour ce déconnecter

```
/**
 * @Route("/logout", name="app_logout")
 */
public function logout()
{
}
```

Cette méthode est vide elle nous permet seulement de déclarer la route pour permettre de la mettre dans le fichier de configuration de security « security.yml ». Juste en dessous de guard nous allons rajouter le lien de logout :

```
        guard:
            authenticators:
                - App\Security\AppCustomAuthenticator

        logout:
          path:   app_logout
```

Ce lien permettra à Symfony de gérer la destruction de la session et des cookies associés.

C/ Créer un nouvel onglet « Login » dans la navBar

A vous de jouer !

3. Créer des utilisateurs

- Créer un formulaire pour créer nos utilisateurs :

php bin/console make:registration-form

=> Concernant le « send an email to verify the user » : c'est très bien, mais dans un environnement local avec des adresses en gmail, vous n'avez aucune sécurité sur votre environnement, et gmail va vous hurler dessus que NON vous ne passerez pas.
Pour ceux que ça intéresse, vous pouvez l'activer : ça nécessite l'installation de dépendances supplémentaires, de mettre à jour la BDD et la configuration dans le .env, et de prendre le risque d'avoir des erreurs à l'affichage, mais qui ne bloqueront pas la création de compte, puisque le mail est envoyé après la création du compte en BDD.

- Allons regarder le controller de plus près :

```
/**
 * @Route("/register", name="app_register")
 */
public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, AppAuthenticator $authenticator): Response
    {
        $user = new User();
        $form = $this->createForm(RegistrationFormType::class, $user);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            // encode the plain password
            $user->setPassword(
                $passwordEncoder->encodePassword(
                    $user,
                    $form->get('plainPassword')->getData()
                )
            );

            $entityManager = $this->getDoctrine()->getManager();
            $entityManager->persist($user);
            $entityManager->flush();

            // do anything else you need here, like send an email

            return $guardHandler->authenticateUserAndHandleSuccess(
                $user,
                $request,
                $authenticator,
                'main' // firewall name in security.yaml
            );
        }
```

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

=> Il y a pas mal de nouvelles choses dans ce formulaire, on peut voir que nous utilisons « UserPasswordEncoderInterface » pour encoder le password donner dans le formulaire.

Nous passons par le « guardHandler » en injection de dépendance pour nous permettre de récupérer la route de la méthode « authenticateUserAndHandleSuccess() »

Grace à ces trois commandes, Symfony nous a permis d'initialiser un système d'authentification et de l'utiliser presque clé en main, à nous d'analyser ce dernier pour voir ce qu'il y a dedans.

Attention : Le fait d'utiliser des outils qui nous facilitent la vie, n'exclut pas de comprendre ces derniers !
- Créer un nouvel onglet « Créer un compte » : je vous laisse le choix du graphisme, on peut imaginer un bouton dépliant « Login » qui contient l'option « Créer un compte » dans la navbar par exemple.

4. Restreindre les accès selon les utilisateurs connectés

- Nous voulons restreindre les accès aux méthodes « new », « edit », « delete » des concerts aux seuls administrateurs à l'aide de l'annotation « @isGranted ».

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
[…]

    /**
     * @Route("/new", name=« concert_new", methods={"GET","POST"})
     * @isGranted("ROLE_ADMIN")
     */
```

Si jamais nous ne sommes pas connecter avec une utilisateur qui à dans son tableau de role `ROLE_ADMIN`, j'aurais automatiquement un « accès denied » ou une redirection vers le formulaire de login si je ne suis pas connecté.

- Testez le après avoir crée un utilisateur au « ROLE_USER ».

- Créez un utilisateur au role « ROLE_ADMIN » en modifiant votre entité « User » et testez si vous avez accès ou non à la page en question.

- Configurez ainsi l'ensemble des routes concernées.

5. Créer une page de modification de son compte

Let's go ! Vous avez toutes les compétences nécessaires pour le réaliser.
- Une page qui affiche les informations liées à son compte, accessible depuis la navbar.
- Un bouton qui permet de les modifier.