SYMFONY : 9-Inscription-authentification

Comment protéger une application ? Il va falloir protéger la sécurité des données, on ne garde jamais les mots de passe en clair, nous devons les hachés, les encoder comme vous le savez (cf. cours procédural). Nous allons faire appel au fichier de configuration security de Symfony.

Nous allons faire appel aux « FireWalls » c'est-à-dire quels sont les points d'entrées de notre applications que nous allons protéger.

Il peut y avoir des parties qui seront protégés (profil, BackOffice etc..) et des parties non protégés (accès à la boutique, panier etc….), qui serait protégé par un login ou par un service de jetons, de token (clé de sécurité).

Nous allons faire appel aux « providers » c'est-à-dire à savoir où sont les données de l'utilisateur (annuaire LAPD, BDD, fichiers…) Comment reconnaitre l'utilisateur ?

Comment sont sécurisées les données ? Symfony nous propose d'utiliser les « encoders » comment créer des hash ? Des algorithmes ? Possibilités d'encodeurs différents en fonction des entités.

Rendons nous dans le dossier « config » puis dans le dossier « packages » et ouvrir le fichier « security.yaml » Nous observons les FireWalls, providers !

```
dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
```

Cela permet d'accéder à la barre de développement d'administration en bas de la page web Symfony. Nous souhaitons laisser l'accès et qu'il n'y est pas de sécurité.

En fait tout le reste de l'application est sous le FireWalls 'main'. Nous pourrions créer autant de FireWalls que l'on souhaite qui matche avec les patterns qui nous donnent l'accès.

```
main:
            anonymous: lazy
```

On voit que le FireWalls 'main' gère tout le reste de l'application et que et que n'importe qui peut y accéder (anonymous : lazy)

Observer la barre d'administration et on voit que c'est comme nous étions connectés. Maintenant nous allons voir comment faire pour authentifier les utilisateurs. Nous allons donc créer une entité 'utilisateur' (une table SQL) pour stocker les données des utilisateurs.

Taper dans la console :

```
php bin/console make:entity
```

Taper ensuite 'User' pour le nom de la table SQL :

```
Class name of the entity to create or update (e.g. VictoriousGnome):
> User
```

```
 created: src/Entity/User.php
 created: src/Repository/UserRepository.php
```

Nous allons maintenant créer les différents champs de notre table.

Email :

```
New property name (press <return> to stop adding fields):
 > email

 Field type (enter ? to see all types) [string]:
 >

 Field length [255]:
 >

 Can this field be null in the database (nullable) (yes/no) [no]:
 >
```

username :

```
 Add another property? Enter the property name (or press <return> to stop addi
ng fields):
 > username

 Field type (enter ? to see all types) [string]:
 >

 Field length [255]:
 >

 Can this field be null in the database (nullable) (yes/no) [no]:
 >
```

Password :

```
Add another property? Enter the property name (or press <return> to stop addin
g fields):
 > password

 Field type (enter ? to see all types) [string]:
 >

 Field length [255]:
 >

 Can this field be null in the database (nullable) (yes/no) [no]:
 >
```

Nous avons terminé notre ‘entité’ (table SQL). Taper maintenant ‘enter’ pour valider

Nous allons maintenant faire la migration, taper :

```
php bin/console make:migration
```

La CLI (command ligne interface) va nous créer un fichier de migration, c’est ce fichier qui va nous créer la table des utilisateurs tel que nous l’avons demandé.

Pour réellement créer la table SQL dans phpmyadmin, lancer la commande suivante :

```
php bin/console doctrine:migrations:migrate
```

Taper ensuite ‘y’ pour valider la demande de migration. Voila ! Notre table SQL est maintenant crée. Allez voir dans phpmyadmin la table ‘user’

Nous allons maintenant créer un formulaire d’inscription.

Rendez-vous dans la console et taper la commande suivante pour créer le formulaire :

```
php bin/console make:form RegistrationType
```

Et la Symfony nous demande à quelle entité (table SQL) notre formulaire va enregistrer les données ? Et bien à l’entité ‘User’

```
 The name of Entity or fully qualified model class name that the new form will
 be bound to (empty for none):
 > User
```

Allez dans le fichier crée par Symfony src/Form/RegistrationType.php et ajouter le champ pour confirmer le mot de passe :

```
$builder
            ->add('email')
            ->add('username')
            ->add('password')
            ->add('confirm_password')
        ;
```

Attention pour que le champ ‘confirm_password’ fonctionne, il faut se rendre dans le fichier Entity/User.php et ajouter l’entité ‘confirm_password’

```
/**
     * @ORM\Column(type="string", length=255)
     */
    private $password;

    public $confirm_password;
```

Pas besoin de lui ajouter un 'ORM' puisque c'est un champ qui ne sera pas enregistré dans la BDD

Nous verrons plus tard comment faire en sorte d'indiquer à l'internaute qu'il faut que le mot de passe soit le même dans les 2 champs. Pour être sûr que la personne à taper le bon mot de passe.

Nous allons maintenant créer une méthode qui permettra d'afficher le formulaire.

Se rendre dans la console te taper la commande suivante pour créer le Controller :

```
php bin/console make:controller
```

Nous allons maintenant nommer notre Controller qui contiendra l'ensemble de l'authentification, c'est-à-dire la connexion mais aussi l'inscription

```
Choose a name for your controller class (e.g. BraveJellybeanController):
 > SecurityController
```

Rendez-vous dans le fichier Src/Controller/SecurityController.php, supprimer la route 'index'

```
/**
     * @Route("/security", name="security")
     */
    public function index()
    {
        return $this->render('security/index.html.twig', [
            'controller_name' => 'SecurityController',
        ]);
    }
```

Créer la méthode registration() dans le fichier Src/Controller/SecurityController.php

```
/**
     * @Route("/inscription", name="security_registration")
     */
    public function registration()
    {
        $user  = new User(); // on précise à quelle entité va être relié notre
 formulaire

        $form = $this-
>createForm(registrationType::class); // on appel la classe qui permet de cons
truire le formulaire
    }
```

Attention ! Nous faisons appel à la classe User et RegistrationType, nous devons donc les appeler :

```
use App\Entity\User;
use App\Form\RegistrationType;
```

Ajouter le 2ème paramètres à la méthode createForm () pour relier notre formulaire à l'entité 'user'

```
$form = $this->createForm(registrationType::class, $user);
```

Ajouter maintenant la ligne permettant d'envoyer le formulaire sur un fichier Template :

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

Ajouter le paramètre permettant de récupérer le formulaire à envoyer à TWIG :

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

Nous pouvons maintenant créer notre Template dans le dossier templates/security/registration.html.twig

Ecrire seulement pour le moment, pour voir si la route est bonne :

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

{% form_theme formArticle 'bootstrap_4_layout.html.twig' %}

{% block body %}
    <h1>Bonjour à tous !!</h1>
{% endblock %}
```

Ajouter la création du formulaire :

```
{% block body %}
    <h1>Inscription sur le site</h1>

    {{ form_start(form) }}

    {{ form_widget(form) }}

    <button type="submit" class="btn btn-primary">Inscription</button>

    {{ form_end(form) }}

{% endblock %}
```

Mais nous aurions souhaité des attributs 'placeholder' nous allons donc supprimer form_widget () afin de créer les champs 1 par 1

Ajouter les placeholder et remplacer les labels

```
{{ form_start(form) }}
```

```
    {{ form_row(form.username, {'label': 'Nom d\'utilisateur', 'attr': {'place
holder': 'Nom d\'utilisateur...'}}) }}
    {{ form_row(form.email, {'attr': {'placeholder': 'Adresse email...'}}) }}
    {{ form_row(form.password, {'label': 'Mot de passe', 'attr': {'placeholder
': 'Mot de passe...'}}) }}
    {{ form_row(form.confirm_password, {'label': 'Confirmation du mot de passe
', 'attr': {'placeholder': 'Répétez votre mot de passe...'}}) }}

    <button type="submit" class="btn btn-primary">Inscription</button>

    {{ form_end(form) }}
```

Attention les champs ‘password’ et ‘confirm_password’ sont en clair, il va falloir les passer en champ type ‘password’

Rendons nous dans le fichier src/Form/RegistrationType.php pour modifier ces 2 champs

Pour importer directement les classes avec Symfony, ajouter l’extension **PHP Namespace Resolver** et sélectionner la classe et taper « **Ctrl + Alt + i** » pour importer la classe.

```
$builder
            ->add('email')
            ->add('username')
            ->add('password', PasswordType::class)
            ->add('confirm_password', PasswordType::class)
        ;
```

Ne pas oublier d’importer la bonne classe :

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

Maintenant nous aimerions pouvoir nous insérer dans la table ‘user’, pour cela rendons nous dans le fichier SecurityController.php, c’est-à-dire dans le Controller.

Il faut appeler les classes suivantes :

```
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\HttpFoundation\Request;
```

```
public function registration(Request $request, EntityManagerInterface $manager
) // on appel en argument la class Request pour executer la requete et insérer
 dans la BDD
    {
        $user  = new User(); // on précise à quelle entité va être relié notre
 formulaire

        $form = $this-
>createForm(registrationType::class, $user); // on appel la classe qui permet
de construire le formulaire relié à l'entité user
```

```
        $form->handleRequest($request);

        if($form->isSubmitted() && $form->isValid())
        {
            $manager-
>persist($user); // on fait persisiter dans le temps l'utilisateur, prépare to
i à la sauvegarder
            $manager->flush(); // on lance la requete d'insertion
        }

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

Maintenant nous allons faire en sorte d'informer l'internaute si les mots de passe ne correspondent pas. Pour cela rendons nous dans le fichier src/Entity/User.php

Ajouter la classe suivante pour effectuer les contrôles :

```
use Symfony\Component\Validator\Constraints as Assert;
```

Cela permettra de dire que tel paramètre est égal à tel paramètre.

```
/**
     * @ORM\Column(type="string", length=255)
     * @Assert\Length(min="8", minMessage="Votre mot de passe doit faire minim
um 8 caractères")
     * @Assert\EqualTo(propertyPath="confirm_password", message="Les mots de p
asse ne correspondent pas")
     */
    private $password;

    /**
     * @Assert\EqualTo(propertyPath="password",message="Les mots de passe ne c
orrespondent pas")
     */
```

Nous pouvons retirer l'Assert pour le champ 'password' puisque nous l'avons sur 'confirm_password'

```
/**
     * @ORM\Column(type="string", length=255)
     * @Assert\Length(min="8", minMessage="Votre mot de passe doit faire minim
um 8 caractères")
     */
    private $password;

    /**
```

```
     * @Assert\EqualTo(propertyPath="password",message="Les mots de passe ne c
orrespondent pas")
     */
    public $confirm_password;
```

Le problème maintenant est que le mot de passe est en clair dans la BDD, donc si quelqu'un arrive à accès à notre BDD, il pourra donc se connecter avec n'importe quel compte utilisateur. Rendons nous pour cela dans le fichier packages/security.yaml

```
security:
    encoders:
        App\Entity\User:
            algorithm: bcrypt
```

Nous demandons à Symfony d'utiliser l'algorithme brcypt pour notre entité 'User'.

Rendons maintenant le fichier du Controller SecurityController.php. Ajouter un paramètre à la fonction registration() pour encoder le mot de passe.

```
 public function registration(Request $request, EntityManagerInterface $manage
r, UserPasswordEncoderInterface $encoder)
```

Ne pas oublier l'appel de la classe :

```
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
```

Modifier la méthode registration()

```
if($form->isSubmitted() && $form->isValid())
        {
            $hash = $encoder->encodePassword($user, $user-
>getPassword()); // on lui demande d'encoder le mot de passe et lui envoi un a
rgument de type $user puisque c'est au moment de l'insertion d'un utilisateur
que l'on veut crypter le mot de passe et en 2ème argument on lui envoi le cham
p 'password'

            $user-
>setPassword($hash); // on appel le setteur du mot pde passe et on lui demande
 de le hacher

            $manager-
>persist($user); // on fait persisiter dans le temps l'utilisateur, prépare to
i à la sauvegarder
            $manager->flush(); // on lance la requete d'insertion
        }
```

Et là nous avons une grosse erreur ! Pourquoi ? Parce que notre objet user a besoin de certaines méthodes, c'est-à-dire que nous devons implémenter une interface pour y appeler certaines méthodes. Pour cela rendons nous dans notre entité 'src/Entity/User.php'

Il faut préciser à Symfony que le class User n'est pas une simple table mais la table des utilisateurs de l'application. Donc il faut bien préciser à Symfony qu'il doit implémenter certaines fonctions.

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 */
class User implements UserInterface
```

Ne pas oublier l'appel de la classe :

```
use Symfony\Component\Security\Core\User\UserInterface;
```

Il faut absolument implémenter les fonctions getRoles(), getPassword(), getSalt(), getUsername() et eraseCredentials(). Nous avons déjà implémenter les méthodes getPassword() et getUsername() puisque nous avons déjà ces 2 champs.(https://symfony.com/doc/3.3/security/entity_provider.html)

```
// Nous décalrons ces méthodes à vide puisque nous n'avons rien à faire de par
ticulier
    //La eraseCredentials()méthode est uniquement destinée à nettoyer les mots
 de passe en texte brut éventuellement stockés
    public function eraseCredentials() {}

    // Renvoie la chaine de caractères non encodé que l'utilisateur a saisi, q
ui a été utilisé à l'origine pour coder le mot de passe.
    public function getSalt() {}

    // cette fonction doit renvoyer un tableau de chaine de caractères
    //Renvoie les rôles accordés à l'utilisateur.
    public function getRoles() {
        return ['ROLE_USER']; // utilisateur classique
    }
```

Actualiser la page web, cela fonctionne !! Le mot de passe est maintenant encodé dans la BDD !

Nous allons maintenant faire en sorte que le nom d'utilisateur et l'adresse mail soit unique. Donc supprimer tous les utilisateurs de la BDD. Pour cela, rendez-vous dans le fichier src/Entity/User.php

Ajouter ensuite les contraintes pour les propriétés 'email' et 'username'

```
/**
     * @ORM\Column(type="string", length=255)
     * @Assert\Email()
     */
    private $email;
```

Et pour que l'email soit unique, nous allons ajouter une contrainte directement sur la classe '**User'** elle-même '**UniqueEntity'**. Faire appel à la classe suivante :

```
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 * @UniqueEntity(
 *  fields = {"email"},
 *  message = "Un compte est déja existant à cette adresse Email!"
 * )
 */
class User implements UserInterface
```

Nous allons maintenant créer un formulaire pour se connecter, c'est-à-dire qu'après l'inscription, cela renverra directement sur la page de connexion. Rendons nous dans le fichier src/Controller/SecurityController.php

Créer la fonction suivante :

```
/**
     * @Route("/connexion", name="security_login")
     */
    public function login()
    {
        return $this->render('security/login.html.twig');
    }
```

Et dans la fonction registration (), afin de renvoyer vers la page connexion après inscription, ajouter la ligne suivante :

```
$manager-
>persist($user); // on fait persisiter dans le temps l'utilisateur, prépare to
i à la sauvegarder
            $manager->flush(); // on lance la requete d'insertion

            return $this-
>redirectToRoute('security_login'); // on redirige vers la page login aprés in
scription
```

Nous allons maintenant préciser à Symfony que c'est ce formulaire (de connexion que nous allons créer) qui nous permettra de nous connecter ! Il va falloir dire à Symfony où se trouve les utilisateurs, c'est-à-dire dans quelle entité (table SQL). Comment vérifier si les mots de passe correspondent à ceux enregistré dans la BDD. Nous allons donc utiliser les « **providers** » nous pouvons en avoir autant que l'on souhaite !

Rendons nous dans le fichier packages/security.yaml et créons un providers :

```
in_database:
            entity:
                class: App\Entity\User
                property: email
```

Nous créons un « **providers** » appelé 'in_database' pour bien préciser à Symfony que les données à contrôler sont en BDD, ensuite de quelle entité (table SQL) les données proviennent ? Et bien de l'entité 'User' qui représente finalement la table SQL 'User' et enfin quel champ allons-nous contrôler ? Et bien nous allons comparer si l'email que l'utilisateur a saisi dans le formulaire de connexion correspond bien à un email dans la BDD.

Ajouter maintenant un « FireWalls » :

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

            provider: in_database

            form_login:
                login_path: security_login
                check_path: security_login
```

Là on précise à Symfony de protéger les données de notre « **providers** » et nous allons aussi préciser que c'est pour une connexion donc nous utilisons l'option « **form**_login » et on précise la route du formulaire « **security_login** ». On précise enfin à Symfony l'endroit où les informations doivent être contrôlées, donc la même route que le formulaire de connexion.

Maintenant occupons-nous de créer le formulaire de connexion, nous n'avons pas besoin de créer un formulaire comme nous avons fait précédemment. Nous allons donc créer un formulaire à la main.

```
{% block body %}

    <h1 class="display-4 text-center mt-2">Connexion</h1>

    <form action="{{ path('security_login') }}" method="post" class="col-md-
5 mx-auto">

        <div class="form-group">
            <label for="_username">Email</label>
            <input type="text" class="form-
control" name="_username" placeholder="Adresse email..." required>
        </div>

        <div class="form-group">
            <label for="_password">Mot de passe</label>
```

```
            <input type="password" class="form-
control" name="_password" placeholder="Mot de passe..." required>
        </div>

        <input type="submit" value="Connexion" class="btn btn-primary">
    </form>

{% endblock %}
```

Les attributs ‘name’ pour un formulaire de connexion doivent être impérativement ‘_username’ et ‘_password’ puisque Symfony va faire le travail pour nous !!

L’attribut ‘action’ contient le path avec le nom de la route ramenant vers la connexion, en fait c’est normal puisque nous avons précisé à Symfony que pour contrôler les données du formulaire, la route était ‘security_login’, la route vers le formulaire. Faites le test, ça fonctionne !! Et ça nous emmène directement vers la page ‘home’

Nous allons maintenant mettre en place une déconnexion. Dans le fichier src/Controller/SecurityController.php nous allons créer la fonction suivante :

```
/**
     * @Route("\deconnexion", name="security_logout")
     */
    public function logout(){
        // cette fonction ne retourne rien, il nous suffit d'avoir une route p
our la deconnexion, une fois créer, modifier le providers form_login
    }
```

Cette méthode ne retourne rien, nous avons seulement besoin d’une route pour la déconnexion, Symfony se charge du reste, nous allons maintenant modifier le « **providers** », donc rendez-vous dans le fichier packages/security.yaml

```
form_login:
                login_path: security_login
                check_path: security_login

            logout:
                path: security_logout
                target: blog
```

Nous avons créé un « providers » pour se déconnecter, nous lui indiquons la route « security_logout » et la destination après déconnexion, on renvoie finalement vers la page ‘blog.html.twig’ que nous avons créé précédemment. Dans la barre d’administration, nous observons que nous sommes bien connecté et qu’il y a aussi un lien « logout », cliquez sur le lien et cela renvoi bien vers la page ‘blog’.

Nous allons maintenant mettre en place un bouton connexion et déconnexion dans la barre de navigation. Donc rendons-nous dans le fichier ‘templates/base.html.twig’ et ajouter les 2 liens suivant

```
<li class="nav-item">
                    <a class="nav-
link" href="{{ path('security_login') }}">Connexion</a>
                </li>
                <li class="nav-item">
                    <a class="nav-
link" href="{{ path('security_logout') }}">Deconnexion</a>
                </li>
```

Maintenant comment Symfony sait si nous sommes connectés ou pas, et bien il existe un variable « app » dans TWIG qui contient beaucoup de variables d'environnement qui va nous aider.

Modifier la navigation dans le fichier templates/base.html.twig' :

```
{# Si il n'y a rien dans app.user, en gros si on est connecté, on entre dans l
e IF #}
                {% if not app.user %}
                    <li class="nav-item">
                        <a class="nav-
link" href="{{ path('security_login') }}">Connexion</a>
                    </li>
                {% else %}
                    <li class="nav-item">
                        <a class="nav-
link" href="{{ path('security_logout') }}">Deconnexion</a>
                    </li>
                {% endif %}
```

Faites la même chose pour faire en sorte qu'un utilisateur soit connecté pour créer un article

```
{% if app.user %}
                <li class="nav-item">
                    <a class="nav-
link" href="{{ path('blog_create') }}">Créer un article</a>
                </li>
                {% endif %}
```

Nous allons maintenant faire en sorte de pouvoir insérer des commentaires que si l'utilisateur est connecté. Rendons nous dans la console pour créer le formulaire d'ajout d'articles.

```
php bin/console make:form
```

Nous créons ensuite le nom de la classe

```
 The name of the form class (e.g. GrumpyChefType):
 > CommentType
```

De l'entité 'Comment'

```
The name of Entity or fully qualified model class name that the new form will
be bound to (empty for none):
> Comment
```

Rendez-vous dans le fichier ‘src/Form/CommentType’, on aperçoit que Symfony nous a créé un formulaire qui correspond à la table ‘Comment’ des commentaires. Dans ce fichier ne laissé comme champ que :

```
public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('author')
            ->add('Content')
        ;
    }
```

Nous n’allons pas demandé à l’utilisateur de s’occuper de relier le commentaire à un article et de saisir une date de création de l’article. Rendons-nous dans le fichier ‘src/Controller/BlogController’

Dans la fonction show() qui permet d’afficher un article, nous allons envoyer le formulaire de commentaires que nous venons de créer. Ajouter les lignes suivantes :

```
public function show(Article $article)
    {
        //$repo = $this->getDoctrine()->getRepository(Article::class);

        //$article = $repo->find($id);
        $comment= new Comment();
        $form = $this->createForm(CommentType::class, $comment);

        return $this->render('blog/show.html.twig', [
            'article' => $article,
            'commentForm' => $form->createView()

        ]);
    }
```

Et ne pas oublier l’appel de la classe ‘Comment’

```
use App\Entity\Comment;
use App\Form\CommentType;
```

Rendons-nous maintenant dans le fichier ‘templates/blog/show.html.twig’ pour afficher le formulaire.

Modifier le titre pour afficher le nombre de commentaires :

```
<h2 class="text-center mx-auto m-
3">{{ article.comments | length }}Commentaires</h2>
```

Ajouter ensuite le formulaire à la fin du fichier (après la boucle for)

```
{{ form_start(commentForm) }}

            {{ form_row(commentForm.author, {'label': 'Auteur', 'attr': {'plac
eholder': "Votre nom..."}}) }}

            {{ form_row(commentForm.Content, {'label': 'Votre message', 'attr'
: {'placeholder': "Saisir votre message..."}}) }}

            <button type="submit" class="btn btn-primary mb-
3">Poster votre commentaire</button>

             {{ form_end(commentForm) }}
```

Retourner dans le fichier 'src/Controller/BlogController' afin de préparer et d'exécuter l'insertion.

Modifier la fonction show() qui doit recevoir en paramètre la classe Request pour l'insertion SQL et la classe EntityManagerInterface pour réellement insérer dans la BDD

```
 public function show(Article $article, Request $request, EntityManagerInterfa
ce $manager)
```

Modifier ensuite la méthode show() pour préparer et exécuter la requête d'insertion :

```
 $form->handleRequest($request);

        if($form->isSubmitted() && $form->isValid())
        {
            $comment-
>setCreatedAt(new \DateTime()) // on génère la date pour l'insertion
                    -
>setArticle($article); // on relie l'article au commentaire
            $manager->persist($comment); // on prépare l'insertion
            $manager->flush(); // on execute l'insertion

            return $this->redirectToRoute('blog_show', [
                'id' => $article->getId()
            ]);
        }
```

Pour afficher un message d'erreur en cas de mauvais 'email' ou mot de passe, rendez-vous dans le fichier 'src/Controller/SecurityController.php' et ajouter les lignes suivantes à la fonction login(), ajouter la classe **AuthenticationUtils** et la classe **Response**.

```php
public function login(AuthenticationUtils $authenticationUtils): Response
    {
        // affiche le message d'erreur
        $error = $authenticationUtils->getLastAuthenticationError();
        // recupère le dernier username saisi par l'internaute
        $lastUsername = $authenticationUtils->getLastUsername();

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

Ne pas oublier d'appeler les classes suivantes :

```php
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;
use Symfony\Component\HttpFoundation\Response;
```

Se rendre ensuite dans le fichier 'templates/security/login.html.twig' et ajouter les lignes suivantes :

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

Pour conserver le dernier username saisie ajouter l'attribut 'value' :

```
 <input type="text" class="form-
control" value="{{ last_username }}" name="_username" placeholder="Adresse ema
il..." required>
```

Tester avec des identifiants incorrect, ça fonctionne !!! Pour modifier le message d'erreur, créer un fichier **security.en.php** dans le dossier **translations** et ajouter le code suivant :

```php
<?php
// translations/security.en.php
return [
    'Invalid credentials.' => 'Email ou mot de passe incorrect',
];
```