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:
            lazy: true
```

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

Observer la barre d'administration et on voit que c'est comme si 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.

Nous allons tout d'abord créer notre classe d'utilisateurs, l'entité qui nous permettra d'enregistrer et de stocker en BDD tous les membres inscrit sur le Blog.

Pour cela, ouvrir un terminal et lancer la commande suivante :

php bin/console make:user

Suivez maintenant les instructions dans le terminal.

```
> php bin/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

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

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

- Entrer le nom de la classe de votre entité **: User**
- Voulez-vous stocker les données utilisateur dans la base de données **: YES**
- Entrez un nom de propriété qui sera le nom « d'affichage » unique de l'utilisateur (par ex. email, nom, d'utilisateur, uuid : **email**
- Cette application doit-elle hacher/vérifier les mots de passe des utilisateurs ? **YES**

Magnifique !

La commande pose plusieurs questions afin qu'elle puisse générer exactement ce dont vous avez besoin. Le plus important est le User.php fichier lui-même. La seule règle concernant votre Userclasse est qu'elle doit implémenter Symfony\Component\Security\Core\User\UserInterface. N'hésitez pas à ajouter tout autre champ ou logique dont vous avez besoin. Si votre Userclasse est une entité (comme dans cet exemple), vous pouvez utiliser la commande make:entity pour ajouter plus de champs.

Lancer la commande suivante :

php bin/console make :entity User

Ajoutez les propriétés suivantes :

prenom → string → 255 → NOT NULL

nom → string → 255 → NOT NULL

adresse → string → 255 → NOT NULL

ville → string → 255 → NOT NULL

codePostal → INT → NOT NULL

Assurez-vous également d'effectuer et d'exécuter une migration pour la nouvelle entité :

```
> php bin/console make:migration
> php bin/console doctrine:migrations:migrate
```

The “User Provider”

Heureusement, la make:user commande en a déjà configuré un pour vous dans votre security.yaml fichier sous la providers clé :

```yaml
# config/packages/security.yaml
security:
    # ...

    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
```

Le provider permet d’indiquer à Symfony de quelle entité provienne les données des utilisateurs inscrit sur le Blog et quelle propriété sera contrôlée au moment de la connexion (email).

Hachage des mots de passe

Toutes les applications n'ont pas des « utilisateurs » qui ont besoin de mots de passe. Si vos utilisateurs ont des mots de passe, vous pouvez contrôler la façon dont ces mots de passe sont hachés dans security.yaml. La make:user commande va pré-configurer ceci pour vous :

```yaml
# config/packages/security.yaml
security:
    # ...

    password_hashers:
        # use your user class name here
        App\Entity\User:
            # Use native password hasher, which auto-selects the best
            # possible hashing algorithm (starting from Symfony 5.3 this is "bcrypt")
            algorithm: auto
```

Maintenant nous allons créer un formulaire d'inscription en se basant par rapport à l'entité User afin de pouvoir enregistrer les utilisateurs du blog dans la base de données.

Afin de gagner du temps, exécutez la commande dans un terminal :

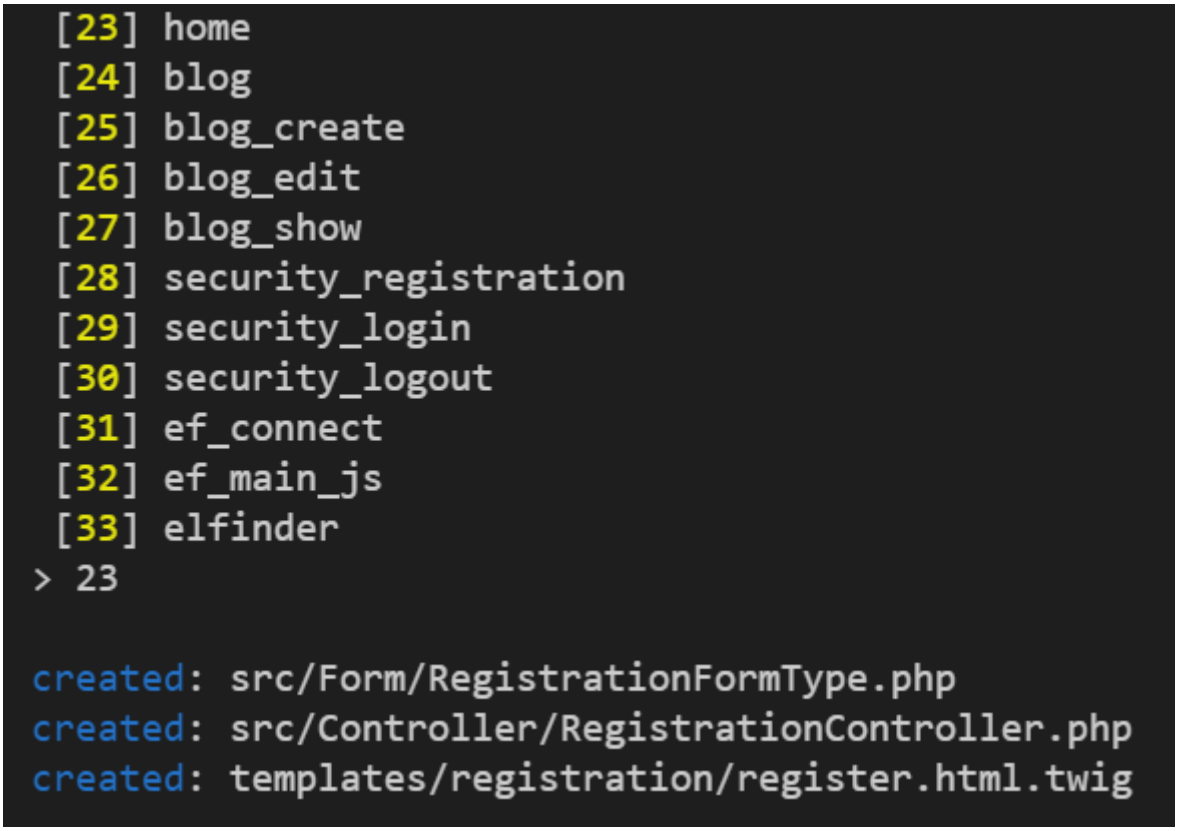
php bin/console make :registration-form

Cette commande doit connaître plusieurs choses - comme votre **Userclasse** et des informations sur les propriétés de cette classe. Les questions varieront en fonction de votre configuration, car la commande devinera autant que possible.

Lorsque la commande est terminée, félicitations ! Vous disposez d'un système de formulaire d'inscription fonctionnel que vous pouvez personnaliser.

```
C:\xampp\htdocs\SYMFONY5\demoBlog>php bin/console make:registration-form

 Enter the User class that you want to create during registration (e.g. App\Entity\User) [App\Entity\User]:
 >

 Creating a registration form for App\Entity\User

 Which field on your App\Entity\User class will people enter when logging in? [username]:
  [0] id
  [1] nom
  [2] prenom
  [3] email
  [4] username
  [5] password
  [6] confirm_password
  [7] roles
 > 3

 Do you want to send an email to verify the user's email address after registration? (yes/no) [yes]:
 > no

 Do you want to automatically authenticate the user after registration? (yes/no) [yes]:
 > no
```

```
What route should the user be redirected to after registration?:
```



- Entrez la classe d'utilisateurs que vous souhaitez créer lors de l'inscription : **App\Entity\User**
- Dans champ de votre classe App\Entity\User sera contrôlé lors de la connexion ? **3 – Email**
- Voulez-vous envoyer un e-mail pour vérifier l'adresse e-mail de l'utilisateur après l'enregistrement ? **YES**
- Voulez-vous authentifier automatiquement l'utilisateur après l'enregistrement ? **NO**
- Vers quel itinéraire l'utilisateur doit-il être redirigé après l'enregistrement ? **23 – home**

Magnifique !

Symfony est magique puisqu'il a créé automatiquement tous les fichiers dont nous avons besoin pour rendre fonctionnel l'inscription des membres sur le Blog :

RegistrationFromType

Symfony crée donc une classe permettant de générer le formulaire d'inscription, pensez à déclarer le reste des champs (prenom, nom, adresse, ville et code postal) comme indiqué ci-dessous. Vous pouvez également mettre en place différentes contraintes de validation afin d'insérer de valeurs correct dans les différentes colonnes de la table SQL de la base de données.

```php
class RegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('email')
            ->add('prenom')
            ->add('nom')
            ->add('adresse')
            ->add('ville')
            ->add('codePostal')
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
                'invalid_message' => 'Les mots de passe ne correspondent pas.',
                'options' => ['attr' => ['class' => 'password-field']],
                'required' => false,
                'first_options'  => ['label' => 'Mot de passe'],
                'second_options' => ['label' => 'Confirmer votre mot de passe'],
                'constraints'    => [
                    new NotBlank([
                        'message' => 'Veuillez renseigner un mot de passe svp !',
                    ]),
                    new Length([
                        'min' => 8,
                        'minMessage'  => 'Votre mot de passe doit contenir au moins 8 caractères !',
                        'max' => 4096,
                    ]),
                    new Regex(array(
                        'pattern'    => '/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!#%*?&])[A-Za-z\d@$!%*?&]{8,}$/',
                        'match'      => true,
                        'message'    => 'Votre mot de passe doit contenir au moins un chiffre, un caractère spécial (@$!#%*?&), une lettre minuscule et une lettre majuscule !'
                    )),
                ]
            ]);
    }
```

Symfony a créé dans le même temps un contrôleur spécialement dédié à l'inscription et l'enregistrement des utilisateurs dans la base de données. Il est important de bien dissocier les différentes parties et traitement de notre blog en créant différent contrôleurs pour chaque partie.

```php
class RegistrationController extends AbstractController
{
    /**
     * @Route("/register", name="app_register")
     */
    public function register(Request $request, UserPasswordHasherInterface $encoder): Response
    {
        // UserPasswordHasherInterface : interface permettant d'encoder le mot de passe

        $user = new User();

        $form = $this->createForm(RegistrationFormType::class, $user, [
            'userFront' => true
        ]);

        $form->handleRequest($request); // $user->setPrenom($_POST['prenom'])

        dump($request); // $request permet de stocker toute les informations des superglobales ($_GET, $_POST, $_FILES etc..)

        dump($user);

        if($form->isSubmitted() && $form->isValid())
        {
            // On fait appel à l'objet $encoder afin de hacher le mot de passe
            // hashPassword() : méthode issue de UserPasswordHasherInterface permettant de créer une clé de hachage pour le mot de passe
            $hash = $encoder->hashPassword($user, $user->getPassword());

            dump($hash); // on observe le mot de passe haché, encodé

            // On affecte à l'entité le mot de passe haché qui sera inséré en BDD
            $user->setPassword($hash);

            $entityManager = $this->getDoctrine()->getManager();
            // $pdo->prepare("INSERT INTO user VALUES ('$user->getPrenom()', '$user->getEmail()')")
            $entityManager->persist($user);

            // $pdo->execute();
            $entityManager->flush();

            return $this->redirectToRoute('home');
        }

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

Pour encoder le mot de passe (jamais en clair dans la BDD), nous avons besoin d'une interface **UserPasswordHasherInterace**, qui contient des méthodes permettant de créer une clé de hachage enregistrée dans la base de données. Afin de pouvoir l'utiliser, il faut que la classe entité dans laquelle nous allons encoder un paramètre implémente 2 autres interfaces : **UserInterace** et **PasswordAuthenticatorUserInterface**. Ces interfaces contiennent des méthodes abstraites que nous devons absolument déclarer dans notre entité User :

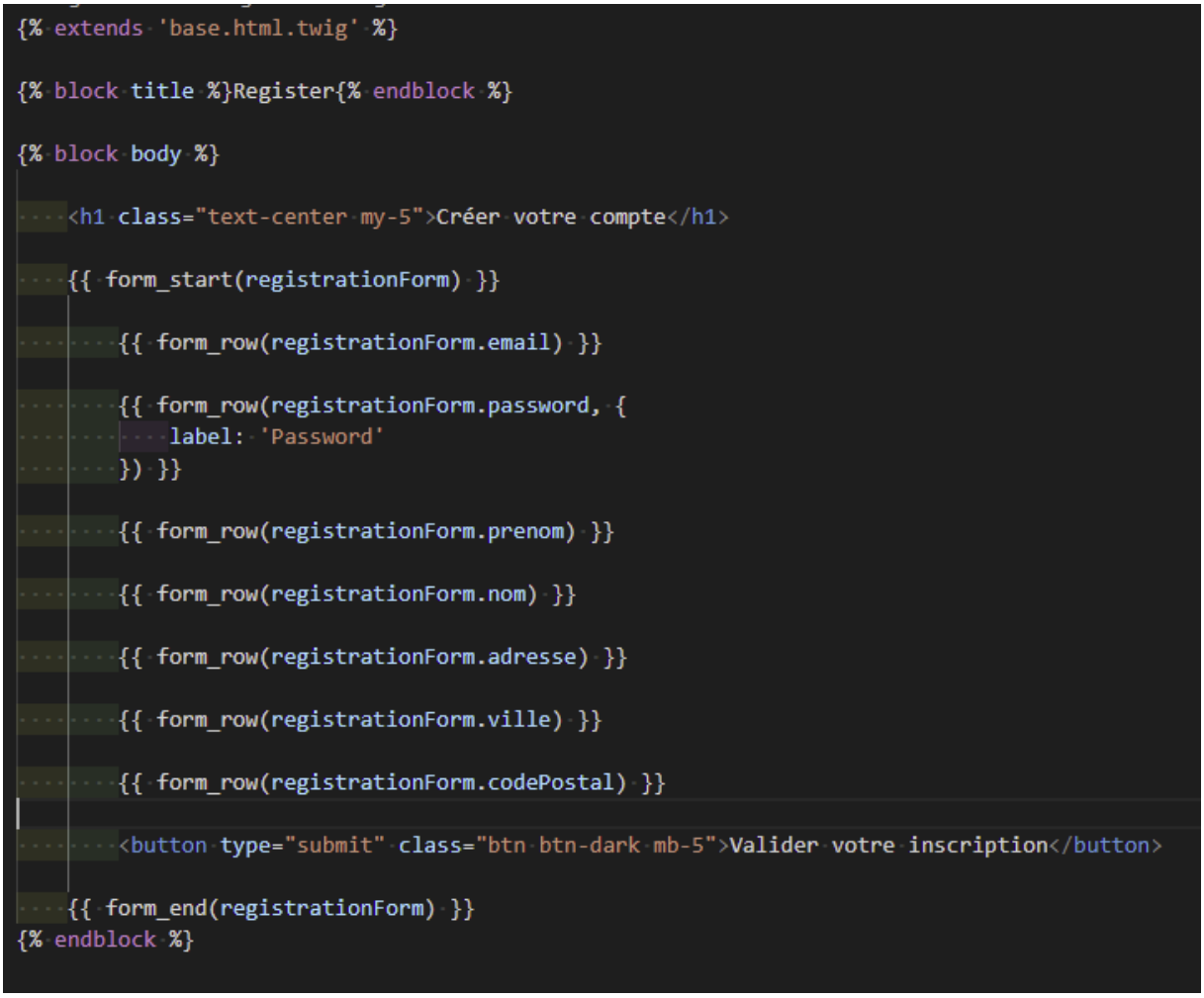
- getUserIdentifier() : renvoi le paramètre avec lequel nous nous identifions (email)
- getRoles() : renvoi les rôles accordés à l'utilisateur
- getPassword() : renvoi le mot de passe de l'utilisateur
- getSalt() : renvoi le mot de passe en brut initialement saisi par l'utilisateur lors de l'inscription
- eraseCrediantials() : destinée uniquement à nettoyer les mots de passe en texte brut éventuellement stockés
- getUserName() : renvoi l'éventuel nom d'utilisateur stocké ne BDD

La méthode hashPassword() issue l'interface **UserPasswordHasherInterace** permet de générer une clé de hachage, il faut lui fournir en argument l'objet entité dans laquelle nous allons encoder un paramètre (**$user**) et lui fournir en 2ème arguement le mot de passe saisi initialement dans le formulaire grâce au getteur de l'entité (**$user->getPassword()**).

Il faut ensuite renvoyer au setteur du mot de passe la clé de hachage afin de l'enregistrer en base de données : **$user->setPassword($hash).**

register.html.twig

Symfony a créée dans le même temps un template Twig permettant l'affichage du formulaire d'inscription :



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

{% block title %}Register{% endblock %}

{% block body %}

    <h1 class="text-center my-5">Créer votre compte</h1>

    {{ form_start(registrationForm) }}

        {{ form_row(registrationForm.email) }}

        {{ form_row(registrationForm.password, {
            label: 'Password'
        }) }}

        {{ form_row(registrationForm.prenom) }}

        {{ form_row(registrationForm.nom) }}

        {{ form_row(registrationForm.adresse) }}

        {{ form_row(registrationForm.ville) }}

        {{ form_row(registrationForm.codePostal) }}

        <button type="submit" class="btn btn-dark mb-5">Valider votre inscription</button>

    {{ form_end(registrationForm) }}
{% endblock %}
```

Authentification

Nous allons maintenant mettre en place et créer un formulaire d'authentification, Symfony met à notre disposition des outils qui vont nous permettre assez simplement de générer tout ce dont nous avons besoin pour authentifier l'utilisateur.

Dans un terminal, exécuter la commande suivante :

php bin/console make:auth

Suivant maintenant les instructions demandées par Symfony :

```
> php bin/console make:auth

What style of authentication do you want? [Empty authenticator]:
 [0] Empty authenticator
 [1] Login form authenticator
> 1

The class name of the authenticator to create (e.g. AppCustomAuthenticator):
> LoginFormAuthenticator

Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
> SecurityController

Do you want to generate a '/logout' URL? (yes/no) [yes]:
> yes

 created: src/Security/LoginFormAuthenticator.php
 updated: config/packages/security.yaml
 created: src/Controller/SecurityController.php
 created: templates/security/login.html.twig
```

- Quel style d'authentification souhaitez-vous ?
- [0] Authentificateur vide
- [1] Authentificateur de formulaire de connexion
- > **1**
- Le nom de la classe de l'authentificateur à créer : **LoginFormAuthenticator**
- Choisissez un nom pour la classe de contrôleur : **SecurityController**
- Voulez générer une URL '/logout' pour la déconnexion : **YES**

Magnifique !

Cela génère les éléments suivants :

1. Routes et contrôleurs de connexion/déconnexion (SecurityController.php).
2. Un modèle qui affiche le formulaire de connexion (templates/security/login.html.twig).
3. Une classe d'authentificateur Guard qui traite la soumission de connexion (LoginFormAuthenticator).
4. Met à jour le fichier de configuration de sécurité principal (security.yaml).

Voici le contenu du contrôleur SecurityController, Symfony a générer 2 méthodes, l'une permettant à l'utilisateur de s'authentifier et l'autre de pouvoir se déconnecter.

```php
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        // permet de stocker un message d'erreur dans le cas d'erreur de connexion, si l'internaute a saisi le mauvais email ou mdp
        $error = $authenticationUtils->getLastAuthenticationError();

        // permet de stocker dans la variable $lastUsername le dernier email qui a été saisi dans le formulaire de connexion
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render('security/login.html.twig', [
            'last_username' => $lastUsername,
            'error' => $error // on transmet le message d'erreur au template afin de pouvoir l'afficher
        ]);
    }

    /**
     * @Route("/logout", name="app_logout")
     */
    public function logout()
    {
        throw new \LogicException('This method can be blank - it will be intercepted by the logout key on your firewall.');
    }
}
```

La classe **LoginFormAuthenticator** permet à Symfony de contrôler les données au moment de la connexion de l'utilisateur (email et mot de passe) :

```php
class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
{
    use TargetPathTrait;

    public const LOGIN_ROUTE = 'app_login';

    private UrlGeneratorInterface $urlGenerator;

    public function __construct(UrlGeneratorInterface $urlGenerator)
    {
        $this->urlGenerator = $urlGenerator;
    }

    public function authenticate(Request $request): PassportInterface
    {
        $email = $request->request->get('email', '');

        $request->getSession()->set(Security::LAST_USERNAME, $email);

        return new Passport(
            new UserBadge($email),
            new PasswordCredentials($request->request->get('password', '')),
            [
                new CsrfTokenBadge('authenticate', $request->get('_csrf_token')),
            ]
        );
    }
```

Cette classe contient également une méthode permettant de rediriger l'utilisateur après s'être authentifié :

```php
public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
        return new RedirectResponse($targetPath);
    }

    // For example:
    // On définit la route de destination après que l'utilisateur se soit authentifié, connecté sur le site
    return new RedirectResponse($this->urlGenerator->generate('blog'));
    // throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
}
```

Et enfin, Symfony a mis à jour le fichier de configuration de sécurité principal (security.yaml). Ce fichier permet de préciser à Symfony où sont stockés les données des utilisateurs (provider), quelle partie du site va être protégée (firewalls) et par quel moyen. Nous pouvons aussi indiqué à Symfony la route permettant de se déconnecter (**logout : path : app_logout**) et la route de destination après s'être déconnecté (**target : logout**).

```yaml
security:
    # https://symfony.com/doc/current/security/experimental_authenticators.html
    enable_authenticator_manager: true

    password_hashers:
        App\Entity\User:
            algorithm: auto

    # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            # On indique à Symfony la classe permettant de s'authentifié (LoginFormAuthenticator),
            l'email et le mot de passe
            custom_authenticator: App\Security\LoginFormAuthenticator
            # on indique à Symfony la route permettant de se déconnecter
            logout:
                path: app_logout
                # On définit la route de destination une fois déconnecté
                target: blog
```

N'hésitez pas à faire différents test mais vous pouvez dès à présent vous inscrire et vous authentifié sur le Blog !