# L’authentification avec Symfony

Le processus d’authentification nous permet de définir l’identité et les accès d’un utilisateur.

La gestion de la sécurité se fait via le composant **Security** de symfony.
Pour en savoir plus sur son fonctionnement, vous pouvez vous référer à la documentation officielle: https://symfony.com/doc/current/security.html

Dans le cas de notre application, nous avons 3 type d’accès pour nos utilisateurs:

- Non connecté
- Utilisateur connecté (ROLE_USER)
- Administrateur (ROLE_ADMIN)

Voyons maintenant comment se fait cette gestion sur notre application.

## 1. Le fichier security

Situé dans le dossier **/config/packages**, le fichier **security.yaml** contient l’essentiel des configurations liées à la sécurité et se décompose en plusieurs parties.

```yaml
security:
    enable_authenticator_manager: true
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
        App\Entity\User:
          algorithm: auto
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider

    providers:
        users_in_memory:
            entity:
                class: App\Entity\User
                property: username
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory
            form_login:
                # "login" is the name of the route created previously
                login_path: login
                check_path: login
            logout:
                path: logout
                # where to redirect after logout
                target: homepage

    # 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: ^/login, roles: PUBLIC_ACCESS }
        - { path: ^/users, roles: ROLE_ADMIN }
        - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
```

- **Le password hasher:**
  https://symfony.com/doc/5.4/security/passwords.html

```yaml
password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    App\Entity\User:
      algorithm: auto
# https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
```

  Système de cryptage de mot de passe du composant sécurité, ici défini en ‘auto’, il utilise actuellement **Bcrypt Hasher** mais pourra changer lors de mise à jour de symfony.

- **Les Providers:**

  https://symfony.com/doc/5.4/security.html#loading-the-user-the-user-provider

```yaml
providers:
    users_in_memory:
        entity:
            class: App\Entity\User
            property: username
```

  Ou fournisseurs d’utilisateurs, il permet ici de définir ici le chemin de fichier permettant d'accéder à notre classe User et de définir la propriété servant d’identifiant.
  le firewall de l’application récupère les utilisateurs en communiquant avec les providers

- **Les firewalls**

  https://symfony.com/doc/5.4/security.html#the-firewall

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: users_in_memory
        form_login:
            # "login" is the name of the route created previously
            login_path: login
            check_path: login
        logout:
            path: logout
            # where to redirect after logout
            target: homepage
```

  Il s’agit ici de notre système d’authentification, il définit quelle partie est sécurisée ou non et comment l’utilisateur va pouvoir s’authentifier.

**login_path** contient le nom de la route permettant d'accéder à la fonction d'authentification située dans le fichier Controller ,
logout: contient le nom de la route de la fonction de déconnexion et la target ici, sert de chemin de redirection.

- ### L'access control:

https://symfony.com/doc/5.4/security.html#access-control-authorization

```yaml
access_control:
    - { path: ^/login, roles: PUBLIC_ACCESS }
    - { path: ^/users, roles: ROLE_ADMIN }
    - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
```

Nous définissons ici les rôles qui ont accès aux différentes parties de l'application.
Nous avons ici défini que:

- Tous les utilisateurs ont accès à notre page de connexion.
- Seuls les utilisateurs ayant le 'ROLE_ADMIN' peuvent accéder aux pages dont le chemin commence par /users.
- Seuls les utilisateurs connectés peuvent accéder aux autres pages de l'application

## 2. L'entité User

Située dans le dossier **/src/entity**, la classe **User** implémente la classe UserInterface qui contient les fonctions nécessaires à notre système d'authentification.

Nous définissons dans la classe User les propriétés et fonctions qui seront utilisées par notre fichier de configuration, par exemple, les rôles seront utilisés par l'access_control.
Les entités étant des objets que l'ORM va manipuler pour gérer la base de données, nous allons également y ajouter les autres propriétés de nos utilisateurs, telles que les tâches qui lui sont associées.

## 3. Le TaskVoter

https://symfony.com/doc/5.4/security/voters.html
Les **Voters** permettent de définir des règles de sécurité plus précises que dans notre fichier de configuration ou les annotations @isGranted().

Nous avons créé, dans le dossier **/src/Security/Voter** une classe **TaskVoter**

```php
class TaskVoter extends Voter
{
    public const EDIT = 'TASK_EDIT';
    public const DELETE = 'TASK_DELETE';

    protected function supports(string $attribute, $subject): bool
    {
        return in_array($attribute, [self::EDIT, self::DELETE])
            && $subject instanceof \App\Entity\Task;
    }

    protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
    {
        $user = $token->getUser();
        // if the user is anonymous, do not grant access
        if (!$user instanceof UserInterface) {
            return false;
        }

        // ... (check conditions and return true to grant permission) ...

        return $subject->getAuthor() === $user || ($subject->getAuthor()->getUsername() === 'anonyme' && $user->hasRole( role: 'ROLE_ADMIN'));

    }
}
```

Ici, cette classe de permet aux utilisateurs ayant le 'ROLE_ADMIN' de modifier et supprimer toutes les tâches dont l'utilisateur associé contient le username 'anonyme'.

## 4. Le Controller

Le controller est généré et géré automatiquement par le composant sécurité de symfony.
Il est géré à l'aide du gestionnaire d'évènement symfony.
Si les informations de connexions sont correctes, l'utilisateur sera authentifié, sinon la page sera rechargée et contiendra un message d'erreur.

```php
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(AuthenticationUtils $authenticationUtils)
    {
        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

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

Il vaut mieux éviter de le modifier.

## 5. La vue

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

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

    <form method="post">
        <label for="username">Nom d'utilisateur :</label>
        <input type="text" id="username" name="_username" value="{{ last_username }}" />

        <label for="password">Mot de passe :</label>
        <input type="password" id="password" name="_password" />

        <button class="btn btn-success" type="submit">Se connecter</button>
    </form>
{% endblock %}
```

Le fichier twig associé au formulaire de connexion se trouve dans **/templates/security/login.html.twig**

## 6. Le fonction create

La création de l'utilisateur se faire via le UserController

```php
    /**
     * @Route("/users/create", name="user_create")
     */
    public function createAction(Request $request, EntityManagerInterface $em)
    {
        $user = new User();
        $form = $this->createForm( type: UserType::class, $user);

        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $em->persist($user);
            $em->flush();

            $this->addFlash( type: 'success', message: "L'utilisateur a bien été ajouté.");

            return $this->redirectToRoute( route: 'user_list');
        }

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

Nous pouvons noter que le mot de passe n'a été hashé à aucun moment.

Cela se fait via un écouteur d'événements que l'on a nommé **UserSubscriber** situé dans le dossier dans le dossier **/src/EventListener.**

```php
class UserSubscriber implements EventSubscriberInterface
{
    public function __construct(private UserPasswordHasherInterface $passwordHasher)
    {
    }

    public function getSubscribedEvents(): array
    {
        return [
            Events::prePersist,
            Events::preUpdate,
        ];
    }

    public function prePersist(LifecycleEventArgs $eventArgs)
    {
        $this->hashSetPassword($eventArgs);
    }

    public function preUpdate(LifecycleEventArgs $eventArgs)
    {
        $this->hashSetPassword($eventArgs);
    }

    /**
     * Check event arg and hash password
     *
     * @param LifecycleEventArgs $eventArgs
     * @return void
     */
    public function hashSetPassword(LifecycleEventArgs $eventArgs): void
    {
        $entity = $eventArgs->getEntity();

        if ($entity instanceof User) {
            $entity->setPassword($this->passwordHasher->hashPassword($entity, $entity->getPassword()));
        }
    }
}
```

Les fonctions **prePersist** et **preUpdate** se lancent avant la persistance des données de l'entité dans la base de données.
Elles appellent toutes deux la fonction **hashSetPassword** qui fera finalement le hashage.