# Créer un utilisateur

On va voir maintenant comment créer des utilisateurs. Nous pourrions penser qu'il nous faut créer une entité, mais non. Symfony gère un maker spécifique à la création d'utilisateur.

## Entité

```
symfony console make:user
```

- Il nous demande le nom de notre classe, par défaut "**User**" ce qui nous convient très bien.
- Puis si on veut stocker les informations de l'utilisateur en BDD, là aussi la valeur par défaut convient "**Oui**".
- Puis quelle sera le nom de la propriété qui devra être unique à chaque utilisateur, en valeur par défaut "**email**" qui nous convient.
- Puis est ce que l'application a besoin de hacher le mot de passe, encore une fois la valeur par défaut est parfaite puisqu'elle dit "**oui**"

Il a maintenant créé deux fichiers et modifiers deux fichiers.

- Il a créé une entité et son repository.
- Il a mit à jour l'entité.
- Enfin modifié la configuration du fichier **security.yaml**

Si on regarde ce dernier, on verra qu'il a indiqué quel est l'entité utilisateur et quelle est la propriété d'authentification (unique).

Maintenant regardons notre nouvelle entité utilisateur.
Par rapport à une entité classique, elle implémente deux interfaces.

Elle a aussi 4 propriétés au lieu d'une par défaut:

- Id évidemment,
- Email car on a indiqué que c'était la propriété unique,
- Roles pour gérer les rôles des utilisateurs.
- Password pour le mot de passe.

Ensuite voyons les méthodes maintenant, elles sont au nombre de 9.

- 6 setter et getters classiques pour les propriétés id, email, password et le setter de role.
- "**getUserIdentifier**" qui permet de récupérer le champ qui sert à identifier l'utilisateur. (ici email)
- "**getRoles**" nous retourne un tableau qui contiendra au moins "**ROLE_USER**" garantissant que tout utilisateur a au moins ce rôle.
- "**eraseCredentials**" qui permet de supprimer des données possiblement sensible (comme le mot de passe en clair.)

Si nous souhaitons ajouter des champs, User est maintenant une entité classique, en utilisant "**make:entity**" on pourra lui ajouter n'importe quelle propriété.

Pensez à faire une migration pour ajouter nos utilisateurs en BDD.

## Connexion Déconnexion

Maintenant qu'on peut avoir des utilisateurs, il nous une connexion et une déconnexion, pour cela pas de magie, ou plutôt si puisque voilà :

```
symfony console make:auth
```

- Il nous propose alors deux types d'authenticator, utilisons le login form avec "**1**"
- Puis il nous demande le nom à lui donner, appelons le "**LoginAuthenticator**"
- Puis il nous demande un nom de controller, "**SecurityController**" qui est la valeur par défaut est bien.
- Enfin il nous demande si on veut une route logout, là aussi acceptons.

Il nous a alors créé 3 fichiers et mis à jour 1:

- "**src/Security/LoginAuthenticator.php**" qui gère la connexion/déconnexion.
- "**src/Controller/SecurityController.php**" pour nos routes.
- "**templates/security/login.html.twig**" pour notre formulaire de connexion.
- "**security.yaml**" a été mis à jour.

Enfin il nous indique de modifier à notre guise le nouvelle authenticator,
De finir la redirection dans la méthode "**onAuthenticationSuccess**", Et d'adapter le template de login à nos besoins.

Si nous regardons ensemble ce LoginAuthenticator que trouverons nous :

- Une classe avec de nombreux use.
- **TargetPathTrait** qui permet de garder en mémoire le dernier lien visité avant le login.
- Une constante qui contient la route vers la page de login.
- une méthode qui gère l'authentification. qui retourne un passeport contenant toute les informations qui doivent être réuni pour permettre la connexion de l'utilisateur.
- une autre qui gère ce qui doit se passer en cas d'authentification réussite.
- Une dernière qui retourne l'url de login.

Penchons nous sur l'authentification en cas de succès pour commenter l'exception et décommenter la redirection en lui donnant le nom du chemin que l'on souhaite voir afficher après une connexion réussite.

On ira aussi dans le fichier "**security.yaml**" pour indiquer vers quelle page rediriger après une déconnexion.

### Hachage de mot de passe et fixtures

Créons de nouvelles fixtures pour nos utilisateurs "**UserFixtures**";

Ensuite ajoutons les lignes suivantes à nos utilisateurs :

```
class UserFixtures extends Fixture implements FixtureGroupInterface
{
```

```
    public function __construct(private UserPasswordHasherInterface $hasher)
    {}
    public function load(ObjectManager $manager): void
    {
        for ($i=0; $i < 5; $i++) {
            $u = new User();
            $u->setEmail("user$i@gmail.com")
                ->setPassword($this->hasher->hashPassword($u, "User$i"));
            if($i===0)$u->setRoles(["ROLE_ADMIN"]);
            $manager->persist($u);
        }

        $manager->flush();
    }
    public static function getGroups(): array
    {
        return ["user"];
    }
}
```

Quelques nouveautés présentes ici. Tout d'abord ma classe implémente une nouvelle interface et une méthode qui va avec. Elles permettent de créer des groupes et d'indiquer à Symfony de ne charger que les fixtures de ce groupe.

Ensuite dans le construct on récupère l'outil de hachage de mot de passe de Symfony.

Puis je crée 5 utilisateurs dont un administrateur. Lançon donc mon load :

```
symfony console doctrine:fixtures:load --group=user --append
```

## Remember me

Actuellement si on ferme notre navigateur, nos utilisateurs sont déconnecté. Si on souhaite que nos utilisateurs puissent rester connecté plus longtemps, cela va se passer en 3 étapes :

Dans le twig de connexion on va décommenter les lignes concernant la checkbox remember me. (Vous verrez aussi un lien vers la documentation.)

Ensuite dans le "**security.yaml**" on va ajouter les lignes suivantes dans **main**:

```
remember_me:
  secret:   '%kernel.secret%'
  lifetime: 604800
```

Il récupère la clef secrète du .env et indique une durée de vie (ici 1 semaine).

Enfin nous allons ajouter la ligne suivante à notre loginAuthenticator dans la liste des badges :

```
new RememberMeBadge()
```

Il est aussi possible de garder l'utilisateur connecté sans lui demander son avis.

## Récupérer les informations de l'utilisateur connecté

Il n'est pas rare d'avoir besoin d'utiliser les données de l'utilisateur actuellement connecté. Rien de bien compliqué mais cela varie selon où on se trouve.

- Dans un contrôleur :

```
$user = $this->getUser();
```

- Dans un service :

```
use Symfony\Component\Security\Core\Security;
// dans la classe :
public function __construct(private Security $security){}
// dans les méthodes :
$user = $this->security->getUser();
```

- Dans le twig :

```
{{ app.user.propriété }}
```

## Formulaire d'inscription

Nos utilisateurs peuvent se connecter et déconnecter, mais pas s'inscrire.
Vous ne serez pas surpris si je dis qu'il y a une commande spécifique au formulaire d'inscription n'est ce pas ?

```
symfony console make:registration-form
```

- Sa première question est de savoir si on veut un champ "@UniqueEntity" pour ne pas avoir de doublons. Évidement que oui.
- Puis voulons nous envoyer un email pour vérifier l'adresse email de l'utilisateur. encore une fois oui. Il nous précise qu'on va devoir installer ce bundle :

  ```
  composer require symfonycasts/verify-email-bundle
  ```

- Ensuite on répondra non à la question, voulons nous que l'utilisateur puisse vérifier son email sans être connecté.
- Puis quel est l'email qui sera utilisé pour envoyer les mails, mettons en un faux "noreply@cours.fr"
- Cela continue avec le nom qui sera associé à l'email, prenons par exemple "Cours Symfony Mail Bot"
- Et enfin voulons nous que l'utilisateur soit automatiquement connecté après inscription.

Une fois notre formulaire créé, installons le bundle dont on a parlé précédemment. Puis allons dans le "**registrationController**" dans la méthode de vérification d'email pour comme il nous l'indique changer la redirection en cas de succès et changer le message flash.

On peut maintenant modifier le formulaire d'inscription si besoin. Puis on finira par faire les migrations.

## Gestion des rôles

On a vu dans notre entité **User** que chaque utilisateur a au moins le rôle User.
Dans notre fixture, on a créé un utilisateur avec le rôle Admin.

Il faut savoir que ce ne sont pas des rôles clairement défini, vous pouvez créer tous les rôles que vous voulez, les seules règles sont qu'ils doivent commencer par "**ROLE_**" et être en majuscule.

De plus on a vu que les rôles étaient stocké en tableau, ce qui implique qu'on puisse donner plusieurs rôles à un utilisateur. Si on se souvient dans notre entity user, il donne par défaut le "**ROLE_USER**" à nos utilisateurs, donc pas besoin de donner ce rôle de base.

Maintenant rendons nous dans "**config/package/security.yaml**" dans la partie "**access_control**".

Cette partie va nous permettre de limiter l'accès à certaines pages à des rôles particuliers. Attention, seulement le premier access_control qui correspond sera sélectionné.

Si on regarde les exemples commentés, que signifie cela :

- les routes qui commencent par "/admin" sont réservé aux ROLE_ADMIN.
- les routes qui commencent par "/profil" sont réservé aux ROLE_USER.

Amusons nous un peu et écrivons les lignes suivantes :

```
- {path: ^/login, roles: PUBLIC_ACCESS}
- {path: ^/register, roles: PUBLIC_ACCESS}
- {path: ^/, roles: ROLE_USER}
```

La troisième ligne indique que absolument toute les routes sont réservé aux utilisateurs. Mais comme il s'arrête à la première correspondance, avant je lui ai indiqué que la route de connexion et celle d'inscription sont en "**PUBLIC_ACCESS**" ce qui permet que tout le monde y ai accès.

Cela est bien pour des règle un peu général mais parfois on pourrait vouloir bloquer que certaines fonctionnalités ou certaines pages en particulier. Pour cela on peut directement bloquer l'accès à une route en de la façon suivante :

```
#[
  Route('/new', name: 'app_departement_new', methods: ['GET', 'POST']),
  IsGranted("ROLE_ADMIN")
]
```

On voit ici que je limite l'accès à la route de création de nouveau département aux administrateurs.

On pourrait aussi vouloir bloquer pas la route en entière mais seulement certaines fonctionnalités. Rendons nous par exemple dans notre route de création de nouvelle **ville**. Et ajoutons au début de notre condition "**if($photo)**"

```
$this->denyAccessUnlessGranted("ROLE_ADMIN");
```

Maintenant, seul un administrateur peut téléverser une photo.

Pareillement si vous voulez bloquer un fonctionnalité d'un service :

```
use Symfony\Component\Security\Core\Security;
// dans la classe :
public function __construct(private Security $security){}
// dans les méthodes :
$user = $this->security->isGranted("ROLE_ADMIN");
```

On remarquera que c'est le même outil que pour récupérer un utilisateur.

On a bloqué l'accès à l'ajout de photo pour nos villes aux administrateurs, alors pourquoi afficher l'input file aux simple utilisateurs ?

Allons maintenant dans notre twig de création et remplaçons "**{{form_rest(villeForm)}}**" par :

```
{% if is_granted("ROLE_ADMIN") %}
    {{form_row(villeForm.photoFile)}}
{% endif %}
{{form_row(villeForm.departement)}}
{{form_row(villeForm.Envoyer)}}
```

En vérité cela aurait été plus simple de retirer le champ si ce n'était pas un administrateur, mais c'était pour l'exemple, maintenant voyons un autre exemple dans notre "**header**" :

```
<nav class="navbar navbar-expand">
  <ul class="navbar-nav">
    {% if is_granted("IS_AUTHENTICATED_REMEMBERED") %}
      <li class="nav-item"><a class="nav-link" href="
{{path("app_logout")}}">Déconnexion</a></li>
```

```
    {% else %}
      <li class="nav-item"><a class="nav-link" href="
{{path("app_login")}}">Connexion</a></li>
      <li class="nav-item"><a class="nav-link" href="
{{path("app_register")}}">Inscription</a></li>
    {% endif %}
  </ul>
</nav>
```

- "**IS_AUTHENTICATED_REMEMBERED**" vérifie si l'utilisateur est connecté, peu importe son rôle.
- "**IS_AUTHENTICATED_FULLY**" fait de même mais ignore les utilisateurs qui ont été connecté par un cookie avec l'option "Remember me".

Cela peut être utile si vous voulez forcer un utilisateur à se reconnecter pour acceder à des données sensible.

### Hiérarchie des rôles ###

On a vu que l'on pouvait créer tous les rôles que l'on veut, et jusque là on a utilisé des rôles ADMIN et USER mais imaginons que nous ayons une application avec plein de rôles différents, certes on peut donner plusieurs roles à un utilisateur, mais on peut aussi indiquer qu'un rôle donné possède aussi les droits d'autres rôles.

Pour cela allons dans "**config/package/security.yaml**" et dans "**security**" ajoutons :

```
role_hierarchy:
  ROLE_ADMIN: [ROLE_MODO, ROLE_PARTNER]
  ROLE_MODO: ROLE_WRITER
  ROLE_WRITER: ROLE_USER
  ROLE_PARTNER: ROLE_USER
```

Dans cet exemple on indique que :

- les gens qui ont les rôles "writer" et "partner" ont aussi le rôle "User" (pas très utile vu que tous les utilisateurs ont le rôle user par défaut mais c'est pour l'exemple).
- Ceux qui ont le rôle "modo" ont automatiquement aussi le rôle "writer" et donc le rôle "user".
- Enfin les "admin" ont aussi les rôles "modo" et "partner" donc aussi les rôles "writer" et "user".

Après tout cela, il est temps pour *l'exercice 2 !*