ToDo List - Documentation Technique

1. Implémentation de l'authentification

Afin de mettre en place un système d'authentification, on utilise le Bundle "Security" de Symfony. Ce bundle s'installe à l'aide de la commande :

> composer require symfony/security-bundle.

L'installation de ce bundle va automatiquement créer un fichier de configuration : "config/packages/security.yaml".

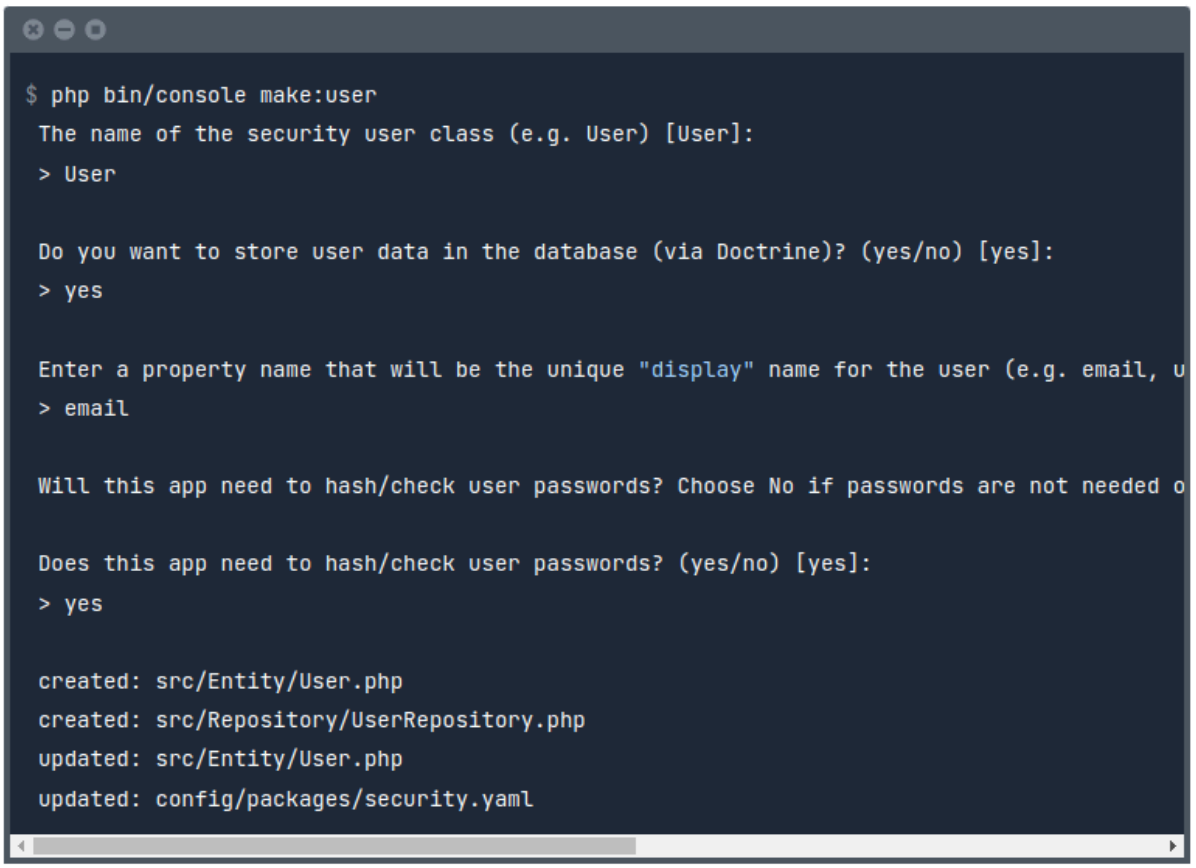
Ce fichier est à **modifier** selon nos attentes : en effet, il nous permet de définir le pare-feu, les systèmes d'encodage de mot de passe, les chemins de connexions ainsi que la sécurité d'accès à certaines pages de notre application si l'on devait ajouter des restrictions.

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

            form_login:
                login_path: login
                check_path: login

            logout:
                path: logout
            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#the-firewall

            # https://symfony.com/doc/current/security/impersonating_user.html
            # switch_user: true

    # 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: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
```

Ensuite, il va être nécessaire de créer l'entité User. Grâce au bundle "Security" de Symfony on peut utiliser la commande :

> php bin/console make:user

La console va ensuite nous poser une série de question pour nous aider à créer l'entité selon nos paramètres prédéfinis :



A la suite de cette série de questions, un nouveau fichier src/Entity/User.php est automatiquement créé.

```php
<?php

namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;

#[ORM\Entity(repositoryClass: UserRepository::class)]
#[ORM\UniqueConstraint(name: 'UNIQ_IDENTIFIER_EMAIL', fields: ['email'])]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 180)]
    private ?string $email = null;

    /**
     * @var list<string> The user roles
     */
    #[ORM\Column]
    private array $roles = [];

    /**
     * @var string The hashed password
     */
    #[ORM\Column]
    private ?string $password = null;

    #[ORM\Column(length: 25)]
    private ?string $username = null;

        public function getId(): ?int
    {
        return $this->id;
    }

    public function getEmail(): ?string
    {
        return $this->email;
    }
```

```php
public function setEmail(string $email): static
{
    $this->email = $email;

    return $this;
}

/**
 * A visual identifier that represents this user.
 *
 * @see UserInterface
 */
public function getUserIdentifier(): string
{
    return (string) $this->email;
}

/**
 * @see UserInterface
 *
 * @return list<string>
 */
public function getRoles(): array
{
    $roles = $this->roles;
    // guarantee every user at least has ROLE_USER
    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}

/**
 * @param list<string> $roles
 */
public function setRoles(array $roles): static
{
    $this->roles = $roles;

    return $this;
}
```

```php
    /**
     * @see PasswordAuthenticatedUserInterface
     */
    public function getPassword(): string
    {
        return $this->password;
    }

    public function setPassword(string $password): static
    {
        $this->password = $password;

        return $this;
    }

    /**
     * @see UserInterface
     */
    public function eraseCredentials(): void
    {
        // If you store any temporary, sensitive data on the user, clear it her
        // $this->plainPassword = null;
    }

    public function getUsername(): ?string
    {
        return $this->username;
    }

    public function setUsername(string $username): static
    {
        $this->username = $username;

        return $this;
    }
}
```

Avec la création de l'entité User, le paramètre

```yaml
security:
    # ...

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

s'est ajouté dans notre fichier config/packages/security.yaml. Ces quelques lignes nous permettent de **définir comment s'opère l'authentification.** En effet, ici, on confirme qu'on souhaite que l'authentification de l'utilisateur se fasse à partir de son email (on aurait pu choisir son username à la place si on l'avait souhaité).

L'entité User fait partie de Doctrine dans notre cas. Nous allons donc **stocker nos utilisateurs dans un tableau "users"**, faisant partie de la base de données MySQL de notre application.

On va pouvoir créer la table users avec la commande
> php bin/console make:migration
> php bin/console doctrine:migrations:migrate

Remarques :

Attention à bien vérifier que notre entité User implémente UserInterface et PasswordAuthenticatedUserInterface.
Dans notre contrôleur, il faudra utiliser le UserPasswordHasherInterface dans notre contrôleur afin d'hasher le mot de passe avant de le sauvegarder dans la base de données.

L'authentification se fait grâce à un formulaire de connexion géré par le bundle "Security", grâce à la commande

> php bin/console make:security:form-login

et par le contrôleur de login

> php bin/console make:controller Security

Ce contrôleur va permettre de gérer les actions de connexion et déconnexion.

2. Travail en collaboration

Pour maintenir un travail de haute qualité en collaboration, il est important d'utiliser un outil de versioning comme Git. L'idée va être de cloner le répertoire de travail, et de travailler chaque fonctionnalité sur une branche parallèle afin de permettre un développement efficace et maîtrisé de l'application. Chaque modification devra être accompagnée d'un commit qui détaille les changements effectués et leur but. Chacune des

branches sera mergée (après une demande de pull request) par un administrateur qui validera ou non le travail effectué.

Il faut également s'assurer avant de faire les merges requests que le code est couvert par des tests unitaires et fonctionnels validés.

Il pourra être intéressant de mettre en place un outil d'intégration continue, ce qui impliquerait de déclencher automatiquement les tests à chaque intégration afin de vérifier si tout se déroule comme prévu.

3. Bonnes pratiques de code

Il est préférable de suivre les PSR-12 https://www.php-fig.org/psr/psr-12/ de PHP lorsque l'on développe une application. Les noms de variables, fonctions, classes doivent être cohérents et explicites à la première lecture. Le code doit être documenté avec des commentaires clairs.
Il est important de typer les variables que l'on utilise dans les fonctions, ainsi que préciser le type de retour attendu par chaque méthode.
Chacune de ces règles permet d'avoir un code de qualité.