TODO-LIST

DOCUMENTATION TECHNIQUE
GUIDE D'AUTHENTIFICATION

I. Le système d'authentification sous Symfony

Le composant Sécurité de Symfony est un système de complet qui permet de gérer plus ou moins facilement la sécurité des applications web. Par sécurité et il faut comprendre la gestion de l'authentification mais aussi la gestion des accès aux ressources selon le profil de chaque utilisateur.

II. L'implémentation de l'authentification

L'application est accessible par tous. Mais il faut s'enrégistrer et s'authentifier pour en bénéficier. Nous allons vous décrire le processus depuis la creation de l'entité utilisateur jusqu'à la gestion de la sécurité dans le fichier "security.yml".

A. L'entité User

L'utilisateur est représenté par la classe App\Entity\User. Une contrainte d'unicité est appliquée sur les attributs "email" et "username'' afin d'éviter un doublon.
Cette classe User implément la UserInterface. Elle hérite donc des functions de cette classe qui sont essentielles à la gestion des utilisateurs.

```
/**
 * @ORM\Table("user")
 * @ORM\Entity
 * @UniqueEntity(fields={"email", "username"}, message="Ce compte existe déjà.")
 */
class User implements UserInterface
{
```

B. Le security.yml

C'est dans ce fichier que la sécurité est entièrement configurée. Il comprend:

- encoders : encodage pour le mot de passe
- provider : le comment, la classe et la propriété qui sera utilisée l'authentification.
- firewalls : définis les différents authentifications possibles.
- access_control : le gestionnaire des accès/routes.
- role_hierarchy : mise en place d'une hierarchie des roles.

L'encoder

Par défaut Symfony utilise l'algorithme de cryptage Bcrypt. C'est cet encoder qu'on récupèrera plus tard via la classe `UserPasswordEncoderInterface`

```
security:
    encoders:
```

```
        App\Entity\User:
            algorithm: auto
```

Pour plus d'information:

- https://symfony.com/doc/current/security/named_encoders.html
- https://symfony.com/blog/new-in-symfony-4-3-native-password-encoder

Le Providers

Il indique où se situe les informations que l'on souhaite utiliser pour authentifier l'utilisateur.
Il faut que la classe donné ici implémente la UserInterface.

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

Les firewalls

```
firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: app_user_provider
            guard:
                authenticators:
                    - App\Security\AppAuthenticator
            logout:
                path: app_logout
                # where to redirect after logout
                # target: app_any_route

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#firewalls-
authentication

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

L'access_control

Cette partie permet de gérer la limitation d'accès à certaines parties du site à partir des routes et et des roles des utlisateurs.
Dans le cas d'espèce:

- Il faut avoir le role admin pour accéder aux routes commençants par /users;
- Pas besoin d'être connecté pour accéder à la route /login;

- Pour toutes les autres routes, le role user est requis.

```
access_control:
        - { path: ^/users, roles: ROLE_ADMIN }
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/, roles: ROLE_USER }
```

Lorsque l'une des conditions n'est pas respectée, Symfony fait appel au guard du firewall pour gérer les redirections.

```
            guard:
                authenticators:
                    - App\Security\AppAuthenticator
```

C. Le fichier Security/AppAuthenticator.php

Ce fichier comprend les fonctions suivantes:

- Vérification de la route et la méthode de connexion

```
  public function supports(Request $request)
    {
        return self::LOGIN_ROUTE === $request->attributes->get('_route')
            && $request->isMethod('POST');
    }
```

- Vérification des credentials et du token.

```
public function getCredentials(Request $request)
    {
        $credentials = [
            'username' => $request->request->get('username'),
            'password' => $request->request->get('password'),
            'csrf_token' => $request->request->get('_csrf_token'),
        ];
        $request->getSession()->set(
            Security::LAST_USERNAME,
            $credentials['username']
        );

        return $credentials;
    }
```

- Recherche et recupération de l'utilisateur grâce à son username ou gestion d'erreur en cas d'utilisateur non existant dans la base de donnnées.

```
public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $token = new CsrfToken('authenticate', $credentials['csrf_token']);
        if (!$this->csrfTokenManager->isTokenValid($token)) {
            throw new InvalidCsrfTokenException();
```

```
        }

        $user = $this->entityManager->getRepository(User::class)-
>findOneBy(['username' => $credentials['username']]);

        if (!$user) {
            // fail authentication with a custom error
            throw new CustomUserMessageAuthenticationException('Utilisateur non
trouvé dans la base de données.');
        }

        return $user;
    }
```

- Vérification des credentials

```
public function checkCredentials($credentials, UserInterface $user)
    {
        // Check the user's password or other credentials and return true or false
        // If there are no credentials to check, you can just return true
        return $this->passwordEncoder->isPasswordValid($user,
$credentials['password']);
        throw new \Exception('TODO: check the credentials inside '.__FILE__);
    }
```

- Configuration de la route à retourner en cas de succès d'authentification

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token,
$providerKey)
    {
        if ($targetPath = $this->getTargetPath($request->getSession(),
$providerKey)) {
            return new RedirectResponse($targetPath);
        }

        return new RedirectResponse($this->urlGenerator->generate('homepage'));
        //throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
    }
```

- Récupération de la route envoyé par le client.

```
protected function getLoginUrl()
    {
        return $this->urlGenerator->generate(self::LOGIN_ROUTE);
    }
```

III. La gestion de la sécurité dans le Src

A. @IsGranted

Permet de limiter l'accès d'une route à un role donnée.

```
/**
 * @Security("is_granted('ROLE_USER')", statusCode=404)
 */
class TaskController extends AbstractController
{
```

B. Voters

- Dans le controller
  Ici, nous avons appelé un voter vérifier les accès de l'utilisateur, selon que le Task est anonyme ou pas.

```
public function deleteTaskAction(Task $task)
    {
        if ($task->getAuthor() === 'anonyme')
        {
            $this->denyAccessUnlessGranted('TASK_DELETE_ANONYME', $task,
$message='Vous devez être admin.');
        } else {
                $this->denyAccessUnlessGranted('TASK_DELETE', $task, $message='Vous
ne disposez pas des droits de suppression.');
        }

        $em = $this->getDoctrine()->getManager();
        $em->remove($task);
        $em->flush();

        $this->addFlash('success', 'La tâche a bien été supprimée.');

        return $this->redirectToRoute('task_list');
    }
```

Dans le security/voter

Ici, nous avons définis deux voters sur le task,

- Le premier, 'TASK_DELETE', vérifie que l'utilisateur est bien propriétaire du task
- Le second, 'TASK_DELETE_ANONYME', vérifie que l'utilisateur un administrateur et que le Task n'a pas d'auteur ("anonyme").

```
class TaskVoter extends Voter
{
    private const DELETE = 'TASK_DELETE';
    private const DELETE_ANONYME = 'TASK_DELETE_ANONYME';
```

```
    protected function supports($attribute, $subject)
    {
        return in_array($attribute, [Self::DELETE, Self::DELETE_ANONYME])
            && $subject instanceof Task;
    }

    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
        $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) ...
        switch ($attribute) {
            case Self::DELETE:
                return $user === $subject->getAuthor();
                break;
            case Self::DELETE_ANONYME:
                //l'auteur est vide et le user est un admin
                //dd($subject->getAuthor(),$user->getRoles());
                if ($subject->getAuthor() === 'anonyme' && in_array("ROLE_ADMIN",
$user->getRoles())) {
                    return true;
                } else { return false; }
                break;
        }

        return false;
    }
}
```