TODOLIST

Authentification et utilisateurs :

L'implémentation de l'authentification a été réalisée selon les standards Symfony 5,4. Pour la résumer brièvement avant de rentrer dans le détail, l'entité User de l'application implémente une UserInterface et une PasswordAuthenticatedUserInterface, mises à disposition nativement par Symfony. Ces interfaces permettent l'accès à différentes méthodes de gestion des Users, ainsi que le stockage en session de différentes informations de base auxquelles on peut ajouter ce qui nous est utile.

1. Fichiers concernés et configuration

▪ AppBundle/Entity/User.php

Entité user qui implémente l'interface Symfony « UserInterface » ligne 18. C'est par cette implémentation que tout le système d'authentification devient actif, et Symfony vient utiliser l'entité User que nous avons crée pour lui donner certaines méthodes obligatoires dont il peut se servir :

- ➢ getRoles()/SetRoles() qui servent à déterminer le rôle de l'utilisateur
- ➢ getSalt() qui sert au chiffrement/déchiffrement des mots de passe
- ➢ getPassword() qui récupère le mot de passe utilisateur
- ➢ eraseCredentials() qui efface toute trace d'accès stockés en clair si on en a besoin à un moment donné du fonctionnement de l'application
- ➢ getUsername() qui sert à reconnaître un utilisateur dans le process d'authentification (on peut utiliser n'importe quel attribut de l'objet User comme un email ou un uuid, il faut simplement retourner une string présente en base de donnée).

Ces méthodes ne sont pas obligatoires, elles peuvent être laissées vide selon la configuration de l'application. En l'occurrence, getSalt() et etraseCredentials() ne servent pas.

- ➢ **Update Symfony 5.4 :** Une méthode getUserIdentifier() est désormais requise pour la future upgrade en symfony 6.0. Elle renvoie l'ancien getUsername() récupéré par l'interface.

```
/**
 * @ORM\Entity(repositoryClass=UserRepository::class)
 * @UniqueEntity("username")
 */
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;
```

- **app/config/security.yml**

Ce fichier sert à la configuration du système d'authentification et à la sécurité (rôles, routes protégées, firewall, password encoder). Ce qui nous intéresse se trouve ligne 5 : le provider. Dans le système Symfony, ce provider, qui est un objet réalisant certaines taches à notre place, va nous fournir une entité user à utiliser pour créer des sessions PHP et remplir ces sessions avec les informations que Symfony ira chercher en base de données via son entité User (et Doctrine) qui implémente sa UserInterface. Voilà donc ce qu'on déclare dans la partie providers :

providers : (on déclare un provider)

app_user_provider : (on déclare à symfony d'utiliser cette entité pour l'auth)

entity : (on veut récupérer une entité)

class : App\Entity\User (on veut utiliser l'entité User avec l'interface Symfony)

property : (On veut reconnaître un User par son attribut récupéré par getUsername() (plus tard getUserIdentifier()), quel qu'il soit.)

Allant de pair avec le provider, l'access_control détermine les routes protégées ou non par une authentification. On y déclare des paths (un par ligne), ainsi que le rôle nécessaire pour accéder à la ressource. Par rôle, il faut surtout entendre « droit », car ça comprend aussi bien les roles utilisateurs que les états de connexion. Dans notre cas, seule la route /login est accessible sans authentification :

- { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }

Il ne manque qu'une partie et la configuration sera complète : le firewall. C'est lui qui va décider de rediriger vers l'authentification, et quelle route intercepter à la submission du form (voir le SecurityController pour en savoir plus sur l'interception du form).

Comme on va utiliser un LoginFormAuthenticator, on lui passe la configuration suivante :

main :

custom_authenticator : App\Security/LoginFormAuthenticator

Avec cette configuration, Symfony sait maintenant quel entité aller chercher pour remplir son système d'authentification, et comment les reconnaître les unes des autres.

```yaml
security:
    enable_authenticator_manager: true
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-p
    password_hashers:
        App\Entity\User: 'auto'
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
            algorithm: 'auto'
            cost: 15
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-prov
    providers:
#        users_in_memory: { memory: null }
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            custom_authenticator: App\Security\LoginFormAuthenticator
            logout: true
            # 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: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/, roles: ROLE_USER }
        - { path: ^/users, roles: ROLE_ADMIN }
```

- **src/Security/LoginFormAuthenticator**

Update Symfony 5.4 : En 5.4 il est recommandé de passer par LoginFormAuthenticator custom afin d'avoir la main sur toute la partie sécurité et login de son application. On peut y faire autant de check que nécessaire et adapter à tous les besoins de l'application. Cette classe se trouve dans src/Security et se décompose en deux méthodes principales :

- start() qui va chercher la route à appeler pour mettre en place le process d'authentification
- supports() qui check si la route et la méthode sont bonnes
- authenticate qui récupère le login form et le valide via un objet Passport natif requérant un UserBadge (façon de dire que l'utilisateur est bien présent en base et en faire un objet implémentant des méthodes utiles pour l'auth Symfony.), Cette méthode check également le csrf token du login form afin que vérifier sa provenance, ainsi que la validé du mot de passe

selon l'algorithme choisi dans security.yaml.

```php
namespace App\Security;

use ...

class LoginFormAuthenticator extends AbstractAuthenticator implements AuthenticationEntryPointInterface
{

    private RouterInterface $router;
    private UserRepository $userRepository;

    public function __construct(UserRepository $userRepository, RouterInterface $router)
    {
        $this->router = $router;
        $this->userRepository = $userRepository;
    }

    /**
     * @inheritDoc
     */
    public function supports(Request $request): ?bool
    {
        return ($request->getPathInfo() === '/login' && $request->isMethod('POST'));
    }

    /**
     * @inheritDoc
     */
    public function authenticate(Request $request): Passport
    {
        $username = $request->request->get('username');
        $password = $request->request->get('password');

        return new Passport(
            new UserBadge($username, function($userIdentifier) {
                // optionally pass a callback to load the User manually
                $user = $this->userRepository->findOneBy(['username' => $userIdentifier]);

                if (!$user) {
                    throw new UserNotFoundException();
                }

                return $user;
            }),
            new PasswordCredentials($password),
            [
                new CsrfTokenBadge(
                    'authenticate',
                    $request->request->get('_csrf_token')
                ),
                (new RememberMeBadge())->enable(),
            ]
        );
    }
```

- Si authenticate réussi, on appelle onAuthenticateSuccess et on redirige sur /
- Sinon on appelle onAuthenticateFailure, on remonte une erreur et on redirige sur le form

```php
/**
 * @inheritDoc
 */
public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?RedirectRes
{
    return new RedirectResponse(
        $this->router->generate('homepage')
    );
}

/**
 * @inheritDoc
 */
public function onAuthenticationFailure(Request $request, AuthenticationException $exception): ?Response
{
    $request->getSession()->set(Security::AUTHENTICATION_ERROR, $exception);

    return new RedirectResponse(
        $this->router->generate('app_login')
    );
}

public function start(Request $request, AuthenticationException $authException = null): RedirectResponse
{
    return new RedirectResponse(
        $this->router->generate('app_login')
    );
}
```

- **App/Controller/SecurityController**

Ce controller sert spécifiquement à l'authentification, et est appelé depuis la route /login de l'application. C'est la seule route accessible sans être authentifié, comme précisé dans le security.yml

Dans le système Symfony, l'authentification est largement prise en charge par le framework. Dans notre cas, nous avons poussé la prise en main un peu plus loin par l'implémentation du LoginFormAuthenticator, qui, comme nous l'avons dit plus haut, intercepte la route /login de ce Controller.

On notera que l'erreur de onAuthenticateFailure est ici récupérée par Symfony et transmise au template via la variable twig error. authenticateUtils accède aux erreurs d'authentifications remontées nativement, et on peut les afficher pour indiquer ce qu'il s'est passé.

A noter que dans ce cas, le FormType n'est pas obligatoire, même s'il est préférable de toujours en passer par lui pour harmoniser tous les forms de l'application, notamment en terme de style automatique si c'est en place (via bootstrap ou autre). Dans notre cas, le form est simplement fait en HTML, et envoyé sur la route du controller qui a généré le template, en l'occurence /login.

La methode logout peut intérroger, car il y a bien un boutton de logout sur le site. Néanmoins, la route est interceptée par Symfony pour détruire la session sans que l'on ait besoin de le faire manuellement.

```php
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        return $this->render('security/login.html.twig', [
            'error' => $authenticationUtils->getLastAuthenticationError()
        ]);
    }

    /**
     * @codeCoverageIgnore
     * @Route("/logout", name="logout")
     * @throws Exception
     */
    public function logout()
    {
        throw new Exception( message: 'logout() should never be reached');
    }
}
```