Authentification

Sommaire

A) Processus d'authentification de symfony

1) EntryPoint

Lorsqu'un utilisateur n'est pas authentifié et qu'il essaie d'accéder à une ressource protégée le système de point d'entrée va déterminer quoi faire. Pour cela on va utiliser une classe qui implémente l'interface « AuthenticationEntryPointInterface » et qui définit l'action à effectuer via la méthode start(Request $request, AuthenticationException $authException = null). Par exemple, dans le cas d'un formulaire de connexion on redirige l'utilisateur vers le formulaire de connexion.

```php
public function start(Request $request, AuthenticationException $authException = null): Response
{
    $url = $this->getLoginUrl($request);
    return new RedirectResponse($url);
}
```

2) Authenticator

L'authenticator permet à l'utilisateur de démarrer le processus d'authentification. La classe en charge de cette authentification devra implémenter l'interface « AuthenticatorInterface ». Une première méthode supports() permettra de déterminer si la requête courante est une requête d'authentification.

```php
public function supports(Request $request): ?bool;
```

Si le retour de cette fonction est vrai alors l'authenticateur va déclencher le processus d'authentification via la méthode authenticate qui devra renvoyer un objet implémentant la « PassportInterface ».

```php
public function authenticate(Request $request): PassportInterface
{
    $password = $request->request->get('password');
    $username = $request->request->get('username');
    $csrfToken = $request->request->get('csrf_token');

    return new Passport(
        new UserBadge($username),
        new PasswordCredentials($password),
        [new CsrfTokenBadge('login', $csrfToken)]
    );
}
```

Ce système de passeport a été ajouté avec Symfony 5.1 et va à terme remplacer l'ancien système d'authentification. Un passeport est un simple objet qui est accompagné de badges qui vont ensuite être utilisés par les écouteurs d'évènements qui se déclenchent lors du processus d'authentification.

Le UserBadge, recevra un loader, et sera capable de récupérer l'utilisateur pour la suite du processus.

Le PasswordCredentials, sera utilisé par le CheckCredientialsListener et validera les informations d'identification.

Les autres badges sont utilisés par d'autres listeners en fonction de la situation.

3) UserProvider

La prochaine étape est donc de récupérer l'utilisateur à partir de l'identifiant du UserBadge. Pour cela Symfony va utiliser un objet qui implémente l'interface UserProviderInterface (par défaut l'implémentation utilisée est EntityUserProvider). Cette méthode aura 2 méthodes principales.

loadUserByIdentifier, qui permet de récupérer l'utilisateur à partir de son identifiant (par défaut l'email).

refreshUser, qui permet de recharger l'utilisateur depuis le token.

4) Token

Une fois l'authentification terminée un token représentant les informations de connexion va être renvoyé par l'Authenticator via la méthode createAuthenticatedToken(PassportInterface $passport, string $firewallName): TokenInterface (et sera ensuite sauvegardée en session dans le cas d'une connexion statefull). En général, le comportement offert par défaut est suffisant et on peut étendre de la class AbstractAuthenticator.

5) Navigation

Maintenant que vous êtes connecté, vous pouvez naviguer entre les pages et la session sera persistée. Pour détecter la session, Symfony utilise le ContextListener qui va détecter la présence du token en session et qui pourra déclencher le refraichissement de l'utilisateur grâce au provider (si vous avez l'attribut lazy ce processus n'est déclenché que lors de l'accès à l'utilisateur).

B) Implémentation de l'authentification sur le projet

La sécurité concernant l'authentification est configurée dans le fichier config/packages/security.yaml
Vous trouverez plus d'informations concernant ce fichier et ses différentes parties dans la documentation officielle de Symfony.

Voici tous les éléments à ajouter ou à modifier :

1) L'entité User

Avant toute de chose, il est nécessaire d'avoir défini une entité qui représentera l'utilisateur connecté. Cette classe doit implémenter l'interface « UserInterface » et « PasswordAuthenticatedUserInterface » puis implémenter les différentes méthodes définis.

Implémentation des interfaces :

src/Entity/User.php

```php
class User implements UserInterface, PasswordAuthenticatedUserInterface
```

Implémentation des différentes méthodes définis par les interfaces :

src/Entity/User.php

```php
/**
 * Returning a salt is only needed, if you are not using a modern
 * hashing algorithm (e.g. bcrypt or sodium) in your security.yaml.
 *
 * @see UserInterface
 */
public function getSalt()
{
    return null;
}

public function eraseCredentials()
{
}

public function getUserIdentifier(): string
{
    return $this->username;
}
```

2) Les Providers

Un provider va nous permettre d'indiquer où se situe les informations que l'on souhaite utiliser pour authentifier l'utilisateur, dans ce cas-ci, on indique qu'on récupérera les utilisateurs via Doctrine grâce à l'entité User dont la propriété username sera utilisé pour s'authentifier sur le site. Attention, on peut indiquer ici la classe User car celle-ci implémente l'interface UserInterface !

config/packages/security.yaml

```yaml
providers:
    in_memory: { memory: ~ }
    app_user_provider:
        entity:
            class: App\Entity\User
            property: username
```

3) Le password_hashers

Le password_hashers va simplement nous permettre de déterminer quel est l'algorithme que l'on souhaite utiliser lors du « hashage » d'une certaine information dans une certaine entité. Dans ce cas-ci on laissera symfony gérer automatiquement le « hashage » lorsque que quelque chose doit être « hasher » dans l'entité App\Entity\User via « PasswordAuthenticatedUserInterface » , dans ce cas-ci cela concerne le mot de passe.

config/packages/security.yaml

```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'
```

4) Les Firewalls et Authenticator

Un firewall va définir comment nos utilisateurs vont être authentifiés sur certaines parties du site. Le firewall dev ne concerne que le développement ainsi que le profiler et ne devra à priori pas être modifié. Le firewall main englobe l'entièreté du site à partir de la racine défini via pattern: ^/, l'accès y est autorisé en anonyme c-à-d sans être authentifié, on y indique que c'est le provider « app_user_provider » qui sera utilisé. Afin de s'authentifier, on crée « AppCustomAuthenticator.php » grâce au maker de symfony. Celui-ci contient la logique pour s'authentifier, elle ne se situe pas dans le controller.

Création de « AppCustomAuthenticator.php » :

Tapez dans le terminal : php bin/console make :auth.

```
Jysa@HUAWEI-DEV MINGW64 /d/Sites/Lab/ToDo-Co (controller)
$ php bin/console make:auth
```

Voici à quoi ressemble la classe :

src/Security/AppCustomAuthenticator.php

```php
class AppCustomAuthenticator extends AbstractLoginFormAuthenticator
{
    use TargetPathTrait;

    public const LOGIN_ROUTE = 'app_login';

    private UrlGeneratorInterface $urlGenerator;

    public function __construct(UrlGeneratorInterface $urlGenerator)
    {
        $this->urlGenerator = $urlGenerator;
    }

    public function authenticate(Request $request): Passport{...}

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
    {
        if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
            return new RedirectResponse($targetPath);
        }
        // For example:
        return new RedirectResponse($this->urlGenerator->generate( name: '/'));
    }

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

- Public function authenticate : Cela permet d'intercepter la requête et de définir ou de créer un passeport.
- Public function onAhtenticationSuccess : Redirection en cas de validation d'authentification. Dans notre cas, on redirige vers la page « home »
- Protected function getLoginUrl : Redirection en cas d'échec d'authentification. Dans notre cas, on redirige vers la page « login »

Configuration du fichier security.yaml :

config/packages/security.yaml

```yaml
main:
    lazy: true
    provider: app_user_provider
    custom_authenticator: App\Security\AppCustomAuthenticator
    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#the-firewall

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

5) Les Access_Control

Un access_control va définir les limitations d'accès à certaines parties du site. Dans ce cas-ci, on indique que :

- Toutes les urls qui débutent par « /admin » n'est accessible qu'en étant authentifié avec un utilisateur ayant le rôle "ROLE_ADMIN".

- Toutes les urls qui débutent par « /tasks » n'est accessible qu'en étant authentifié avec un utilisateur ayant le rôle "ROLE_USER".

config/packages/security.yaml

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

6) Les Role_Hierarchy

Un role_hierarchy permet de s'assurer qu'un utilisateur ayant un certain rôle aura automatiquement d'autres rôles. Dans ce cas-ci, un utilisateur possédant le rôle "ROLE_ADMIN" aura automatiquement le rôle "ROLE_USER".

config/packages/security.yaml

```yaml
role_hierarchy:
    ROLE_ADMIN: ROLE_USER
```