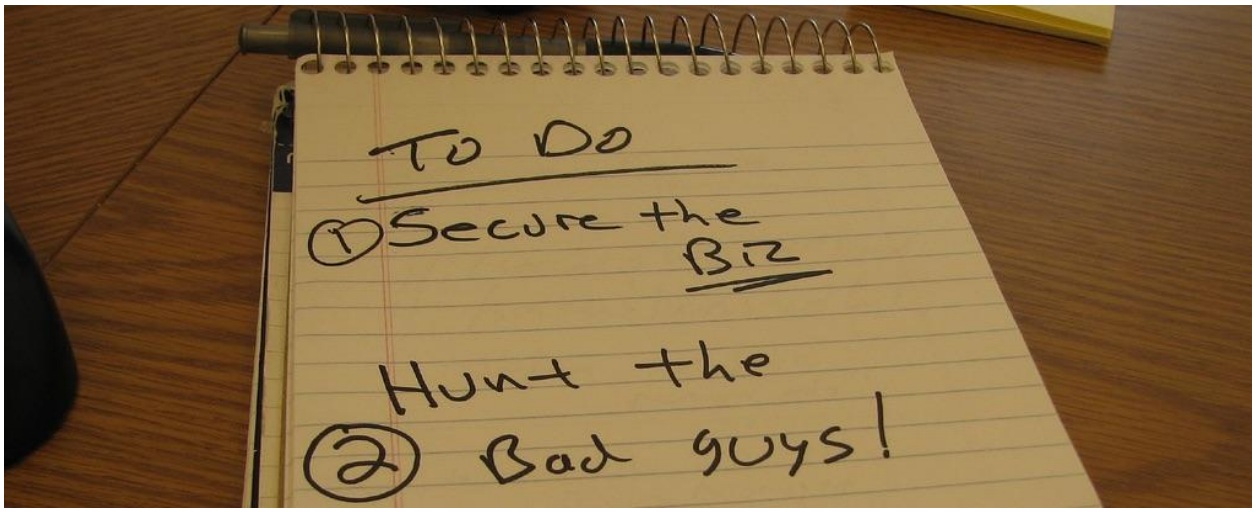
Documentation technique

Implémentation de l'authentification



Introduction

La grande majorité de la mise en place de l'authentification va se dérouler dans le fichier de configuration **config/packages/security.yaml**, vous pouvez retrouver la suite des explications dans la documentation officielle de Symfony.

Les exemples utilisés dans cette documentation sont toutes, sans exception, reprise du code source de l'application ToDo & Co.

L'entité User

Dans un premier temps, avant de pouvoir passer à la mise en place et à la configuration de l'authentification, il va falloir créer une entité représentant l'utilisateur qui implémente l'interface **UserInterface**.

Cette interface va apporter plusieurs méthodes dont :

- getRoles: retourne les rôles accordés à l'utilisateur
- getPassword: retourne le hash du mot passe
- getSalt: retourne le salage utilisé par l'encodeur
- getUsername: retourne le nom d'utilisateur permettant de l'authentifier
- eraseCredentials: efface les données sensibles de l'utilisateur

Composant de sécurité

Provider

Un provider à deux grands rôles:

- Recharger l'utilisateur depuis la session. A chaque requête (à moins qu'elle ne soit stateless), Symfony va recharger l'objet User à partir de la session. Afin de s'assurer qu'il ne soit pas obsolète, le provider va le "rafraîchir". Pour le user provider Doctrine, par exemple, celui-ci va envoyer une requête à la base de données pour récupérer les informations. Symfony va alors vérifier que l'utilisateur n'a pas changé, dans le cas contraire, ce dernier se voit déconnecté.

- Ajouter diverses fonctionnalités. Le provider peut apporter quelques fonctionnalités telles que l'"impersonation" pour changer d'utilisateur,

“remember me” pour rester connecter et bien d’autres, selon le provider utilisé.

Symfony vient avec plusieurs providers intégrés permettant ainsi de charger des utilisateurs depuis la base de données, un serveur LDAP, un fichier de configuration, mais également la possibilité de combiner plusieurs providers.

Ces providers intégrés couvrent la majeure partie des besoins, si cela n’est pas le cas, il vous est toujours possible de créer un provider personnalisé en suivant cette documentation.

Ci-dessous, l’implémentation d’un provider intégré par Symfony afin de recharger l’utilisateur avec le nom d’utilisateur depuis la base de données.

```yaml
# config/packages/security.yaml
security:

    providers:
        doctrine:
            entity:
                class: App\Entity\User
                property: username

```

Encoder

L’encoder va permettre de contrôler quel algorithme à utiliser ainsi que certains paramètres tels que le salage, le coût en mémoire ou encore le nombre d’itérations afin d'encoder les mots de passe.

```yaml
# config/packages/security.yaml
security:

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

La valeur “auto” sélectionne automatiquement le meilleur encodeur. Actuellement, celle-ci essaye d’utiliser Sodium, dans le cas contraire, la fonction de hachage bcrypt est

utilisée.

Firewall

Le firewall est la partie centrale du système d'authentification et va définir la manière dont l'utilisateur va pouvoir s'identifier.

Seulement un firewall peut être actif pour chaque requête, Symfony utilise le pattern fournit dans la configuration (le premier correspondant aux critères) afin de déterminer celui à utiliser.

```yaml
# config/packages/security.yaml
security:

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: true
            pattern: ^/
            guard:
                authenticators:
                    - App\Security\LoginFormAuthenticator
            logout:
                path: logout
```

Dans cette configuration, le firewall dev permet de s'assurer qu'aucunes ressources des outils de développement de Symfony ne soient bloquées par accident et donc ceux faisant partie des urls /_profiler, /_wdt etc.

Un second firewall a été configuré et est attaché à toute l'application définie avec le pattern ^/ et l'accès est autorisé pour les utilisateurs anonymes (ceux n'étant pas authentifié). Un guard personnalisé a été spécialement créé pour ce firewall permettant la connexion des utilisateurs via un formulaire de login, mais Symfony inclut également de nombreux authentificateurs que vous pouvez retrouver dans la documentation.

Enfin, une route de logout a été spécifiée, permettant aux utilisateurs se rendant sur l'url associée, de pouvoir se déconnecter.

Access Control

Cette configuration va permettre de définir les limitations d'accès à certaines parties du site, permettant ainsi de restreindre des utilisateurs, selon un rôle, une IP ou encore avec des conditions personnalisées.

Pour chaque requête, Symfony va vérifier dans la partie Access control et essayer de trouver la première url, défini via le pattern, qui correspond à la requête.

```yaml
# config/packages/security.yaml
security:

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

Dans cette configuration, trois access control ont été définis. Tout d'abord pour la route de connexion afin qu'elles puissent être accédé par tous les utilisateurs (incluant les utilisateurs anonymes), ensuite pour les routes liées à la gestion des utilisateurs, celles-ci seront accessibles qu'aux utilisateurs possédant des privilèges d'administrateur et enfin la dernière permettant aux utilisateurs connectés d'accéder à toutes les autres pages.

Role hierarchy

La hiérarchie des rôles va permettre de faire hériter certains rôles et ainsi, de donner les mêmes privilèges que certains autres rôles.

```yaml
# config/packages/security.yaml
security:

    role_hierarchy:
        ROLE_ADMIN: ROLE_USER
```

Dans cet exemple, le rôle d'administrateur va hériter des privilèges de ceux de l'utilisateur.

LoginFormAuthenticator

Dans le cadre de l'application ToDo & Co, un guard personnalisé a été créé pour permettre aux utilisateurs de se connecter via un formulaire de login.

Un authenticator similaire est déjà intégré par défaut dans Symfony, mais ce choix à l'avantage de nous permettre de garder le contrôle de chaque partie de l'authentification et de pouvoir les modifier au besoin.

Tout d'abord, pour la mise en place de celle-ci, une nouvelle classe LoginFormAuthenticator a été créée héritant de la classe abstraite AbstractGuardAuthenticator. Celle-ci nous apporte de nombreuses méthodes essentielles pour l'authentification des utilisateurs.

Start

Cette méthode est appelée lorsqu'un client essaye d'accéder à une ressource qui nécessite une authentification mais qu'aucun détail n'a été donné pour s'authentifier.

```php
# src/Security/LoginFormAuthenticator
<?php

class LoginFormAuthenticator extends AbstractGuardAuthenticator
{
    public function start(Request $request, AuthenticationException $authException = null): Response
    {
        return new RedirectResponse($this->urlGenerator->generate('login'));
    }
}

```

Pour notre authenticator, une simple redirection vers le formulaire de connexion est retourné.

Supports

Appelée sur chacune des requêtes, elle permet de définir si l'authentification doit être utilisée pour cette requête (si elle renvoie vrai) ou doit être passée (si elle renvoie faux).

```php
# src/Security/LoginFormAuthenticator
<?php

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

```

Dans le cadre de notre formulaire de connexion, nous vérifions que la route correspond bien à celle attendue et que la méthode de la requête soit POST.

GetCredentials

Cette méthode lit simplement les informations envoyées via la requête et les retourne à la méthode suivante getUser.

```php
# src/Security/LoginFormAuthenticator
<?php

class LoginFormAuthenticator extends AbstractGuardAuthenticator
{
    public function getCredentials(Request $request): array
    {
        $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;
    }
}

```

Ici, la méthode va récupérer les informations de l'utilisateur ainsi que le token CSRF contenues dans la requête et les retourner.

Elle va également inscrire dans la session avec une clé définie par le module de sécurité, le nom de l'utilisateur, ce qui permettra, en cas d'erreur sur le formulaire par exemple, d'avoir le champ du nom d'utilisateur pré rempli.

GetUser

Cette méthode récupère les identifiants précédemment retournés afin de lui permettre de récupérer et retourner un utilisateur.

```php
# src/Security/LoginFormAuthenticator
<?php

class LoginFormAuthenticator extends AbstractGuardAuthenticator
{
    public function getUser($credentials, UserProviderInterface $userProvider): UserInterface
    {
        $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) {
            throw new CustomUserMessageAuthenticationException('Identifiants invalides');
        }

        return $user;
    }
}

```

Plusieurs vérifications s'opèrent dans notre cas, nous avons tout d'abord la vérification du token CSRF permettant de s'assurer que ça ne soit pas une redirection malveillante et ensuite nous essayons de récupérer l'utilisateur via le nom d'utilisateur qui a été fourni, si celui-ci existe nous le renvoyons, dans le cas contraire, nous levons une exception.

CheckCredentials

Lorsqu'un utilisateur est retourné, cette méthode est appelée. Elle a pour but de vérifier que les identifiants fournis sont valides.

```php
# src/Security/LoginFormAuthenticator
<?php

class LoginFormAuthenticator extends AbstractGuardAuthenticator
{
    public function checkCredentials($credentials, UserInterface $user): bool
    {
        return $this->passwordEncoder->isPasswordValid($user, $credentials['password']);
    }
}

```

Dans notre formulaire de login, l'utilisateur doit se connecter en fournissant un mot de

passe, c'est pour cette raison que dans la méthode checkCredentials, le hash du mot de passe fourni est comparé avec celui de la base de données.

OnAuthenticationSuccess

Cette méthode est appelée lorsque l'authentification s'est déroulée avec succès.

```php
# src/Security/LoginFormAuthenticator
<?php

class LoginFormAuthenticator extends AbstractGuardAuthenticator
{
    public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey):
Response
    {
        if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
            return new RedirectResponse($targetPath);
        }

        return new RedirectResponse($this->urlGenerator->generate('homepage'));
    }
}

```

Lorsque l'authentification a été un succès, deux issues sont possibles, soit l'utilisateur souhaite se rendre sur une route protégée, dans ce cas, nous le redirigeons vers cette dernière à l'aide du TargetPathTrait, sinon il sera redirigé vers la page d'accueil.

OnAuthenticationFailure

Cette méthode est appelée lorsqu'une erreur s'est produite lors de l'authentification.

```php
# src/Security/LoginFormAuthenticator
<?php

class LoginFormAuthenticator extends AbstractGuardAuthenticator
{
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception):
Response
    {
        $request->getSession()->set(Security::AUTHENTICATION_ERROR, $exception);

        return new RedirectResponse($this->urlGenerator->generate('login'));
    }
}

```

Si l'authentification échoue, l'erreur qui décrit ce qui s'est produit est sauvegardée dans la session, selon la clé définie par le module de sécurité afin d'être récupérée et affichée sur le formulaire et enfin l'utilisateur est renvoyé sur le formulaire de login.

SupportsRememberMe

Permet de définir si l'authenticator intègre la fonctionnalité permettant de garder en mémoire les utilisateurs sur un plus long terme que les sessions.

```php
# src/Security/LoginFormAuthenticator
<?php

class LoginFormAuthenticator extends AbstractGuardAuthenticator
{
    public function supportsRememberMe(): bool
    {
        return false;
    }
}

```

Dans le cadre du formulaire de login, celui-ci n'intègre pas cette fonctionnalité et renvoie donc faux afin de la désactiver.