® Todo & Co
Auteur: Pierre Gaimard
Le 19 mars 2021

Documentation technique

Système d’authentification

Version 1.0.0

Table des matières

Introduction

L’application a été développée avec le framework Symfony, en version 4.4.20.

Dans une première partie, cette documentation expliquera les concepts de base de la sécurité dans Symfony, puis une deuxième partie sera consacrée au détail du système d'authentification mis en place dans cette application.

Partie 1: La sécurité avec Symfony

Introduction

Les composants de Symfony en général

Cette version de Symfony est structurée sous forme de composants que l'on peut installer indépendamment les uns des autres en fonction du besoin. Le composant principal étant **symfony/framework-bundle**.

Chaque composant possède (ou non) des fichiers de configuration permettant de personnaliser le comportement en fonction du besoin.

Le composant security

Dans le framework Symfony, l'ensemble de la sécurité est gérée par le composant **symfony/security-bundle**.

Les concepts clés

Dans Symfony la sécurité s'organise autour de trois concepts clés :

Les utilisateurs

L'entité utilisateur et son interface, les **user providers**, et l'encodage des mots de passe.

L'authentification

Les différents systèmes d'authentification (firewalls) et leur processus (guard authenticators).

L'accès et les rôles

Accès aux différentes ressources de l'application à travers des rôles attribués aux utilisateurs et les mécanismes de vérification d'autorisation.

Les utilisateurs

Quelle que soit la méthode d'authentification (formulaire, JWT, oauth, LDAP, …), et quel que soit l'endroit ou les utilisateurs sont stockés, Symfony utilise une classe représentant l'utilisateur pour gérer la sécurité.

User Interface

Pour pouvoir fonctionner avec le composant **security-bundle** de Symfony, la classe **utilisateur** (qui peut être nommée comme on le souhaite) doit impérativement implémenter l'interface **Symfony\Component\Security\Core\User\UserInterface**.

User Provider

Le **UserProvider** a deux fonctions :

Recharger l'utilisateur depuis la session

A la fin de chaque requête, (A moins que le pare-feu ne soit stateless), l'objet **utilisateur** est sérialisé dans la session. Au début de la requête suivante, il est déserialisé puis passé au **UserProvider** pour qu'il soit actualisé.

Afin d'être sûre que l'utilisateur soit toujours à jour, le **UserProvider** recharge ensuite l'utilisateur depuis l'endroit qui a été défini dans le fichier de configuration (la base de donnée par exemple) puis les deux objets sont comparés pour vérifier qu'ils soient identiques.

> Par défaut le **UserProvider** compare les valeurs retournées par les méthodes getPassword(), getSalt() et getUsername(). Si l'une de ces méthodes renvoie une valeur différente, l'utilisateur est déconnecté par mesure de sécurité.

Charger l'utilisateur pour les fonctionnalités de l'application

Beaucoup de fonctionnalités de Symfony utilisent le **UserProvider** pour récupérer l'utilisateur, comme la fonction "remember me" ou "user impersonation" ou la plupart des UserProvider fournis par Symfony comme le DoctrineUserProvider.

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

    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
```

Exemple de configuration

Password Encoder

Lorsque l'application utilise une authentification par mot de passe, cette fonctionnalité permet de gérer la manière dont les mots de passe sont encodés à travers un fichier de configuration.

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

    encoders:
        # use your user class name here
        App\Entity\User:
            # Use native password encoder
            # This value auto-selects the best possible hashing algorithm
            # (i.e. Sodium when available).
            algorithm: auto
```

Exemple de configuration

Une fois le paramètre réglé dans le fichier de configuration, on peut utiliser le service **UserPasswordEncoderInterface** pour encoder ou vérifier le mot de passe de l'utilisateur.

Authentification

Les firewalls

Le système de sécurité de Symfony est configuré dans le fichier **config/packages/security.yaml**.

```yaml
# config/packages/security.yaml
security:
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy
```

Exemple de configuration

Un **Firewall** est un système d'authentification. Il définit la manière dont les utilisateurs pourront s'authentifier.

Il est possible de configurer plusieurs firewalls pour une même application (par exemple lorsqu'on développe une application avec d'un côté une interface utilisateurs et de l'autre une API.). Par contre il n'y a qu'un seul firewall actif à chaque requête.

Symfony utilise le paramètre **pattern** pour déterminer quel firewall doit être utilisé pour la requête. Si aucun attribut **pattern** n'est déclaré dans la configuration, le firewall sera utilisé pour toutes les urls.

Si l'utilisateur n'est pas authentifié, le mode anonyme est activé pour la requête. De cette façon, il est possible de protéger uniquement certaines parties de l'application et d'en laisser d'autres accessibles à tous les internautes. (comme la page de login par exemple qui doit être accessible à tout le monde pour que l'utilisateur puisse s'authentifier)

*Note: le mode **lazy** du paramètre **anonymous** permet de ne pas démarrer la session si l'utilisateur n'est pas authentifié et donc de mettre la réponse en cache.*

Authentication Provider

Lorsqu'on envoie une requête au serveur, Symfony appelle un **Authentication Provider**. Ce code est appelé avant l'exécution du contrôleur.

Guard Authenticators

Pour gérer l'authentification, il est recommandé d'utiliser un **Guard Authenticator**. Cette classe permet d'avoir un contrôle complet sur le processus d'authentification.

Un **Guard Authenticator** est une classe qui implémente l'interface **Symfony\Component\Security\Guard\AuthenticatorInterface** ou étend la classe **Symfony\Component\Security\Guard\AbstractGuardAuthenticator**.

Voici les différentes méthodes qui doivent être implémentées:

supports(Request $request)

Cette méthode est appelée à chaque requête. Elle permet de décider si cet **authenticator** doit être appelé ou non pour la requête. Elle doit retourner **true** si oui et **false** si non.

getCredentials(Request $request)

Cette méthode est la deuxième appelée. Son travail est de retourner les informations d'authentification qui se trouvent dans l'objet Request. Ces informations seront passées à la méthode suivante **getUser()**.

getUser($credentials, UserProviderInterface $userProvider)

$credentials contient les informations retournées par la méthode **getUser()**. Le travail de cette méthode est de retourner un objet qui implémente l'interface **UserInterface**. Si c'est le cas, la méthode **checkCredentials()** sera appelée. Si la méthode retourne **nul**, ou si on lève une exception (**AuthenticationException**), l'authentification échouera.

checkCredentials($credentials, UserInterface $user)

Cette méthode est appelée si la méthode **getUser()** retourne un objet de type **UserInterface**. Son travail est de vérifier les identifiants de l'utilisateur. Si cette méthode retourne **true**, la méthode **onAuthenticationSuccess()** sera appelée, si elle retourne false, ou si une exception (**AuthenticationException**) est levée, l'authentification échouera.

onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)

Cette méthode est appelée si l'authentification est réussie. Son travail est soit de retourner un objet de type **Response**, ou null si l'on souhaite laisser la requête continuer pour laisser le contrôleur s'exécuter.

onAuthenticationFailure(Request $request, AuthenticationException $exception)

Cette méthode est appelée si l'authentification échoue. Son job est de retourner un objet de type **Symfony\Component\HttpFoundation\Response** qui devrait être retournée au client. La variable **$exception** contient les informations sur les raisons de l'échec.

start(Request $request, AuthenticationException $authException = null)

Cette méthode est appelée si le client tente d'accéder à une ressource qui requiert une authentification mais qu'aucune information d'authentification n'a été fournie. Son travail est de retourner un objet de type **Symfony\Component\HttpFoundation\Response** afin d'aider l'utilisateur à s'authentifier. (Ex rediriger l'utilisateur vers la page de login …)

supportsRememberMe()

Cette méthode doit retourner **true** si l'on souhaite supporter la fonctionnalité "remember me". Attention, si l'on souhaite utiliser cette fonctionnalité, il faut également ajouter la configuration dans le firewall.

createAuthenticatedToken(UserInterface $user, string $providerKey)

Si l'authenticator implémente l'interface **Symfony\Component\Security\Guard\AuthenticatorInterface** plutôt que d'étendre la classe **Symfony\Component\Security\Guard\AbstractGuardAuthenticator**, cette méthode doit être implémentée. Cette méthode sera appelée après une authentification réussie et devra retourner un **token** qui implémente l'interface **Symfony\Component\Security\Guard\Token\GuardTokenInterface**.

Pour plus d'informations sur les **guards authenticators**, référez vous à la documentation officielle.

Ci-après, un schéma complet du système d'authentification de Symfony:

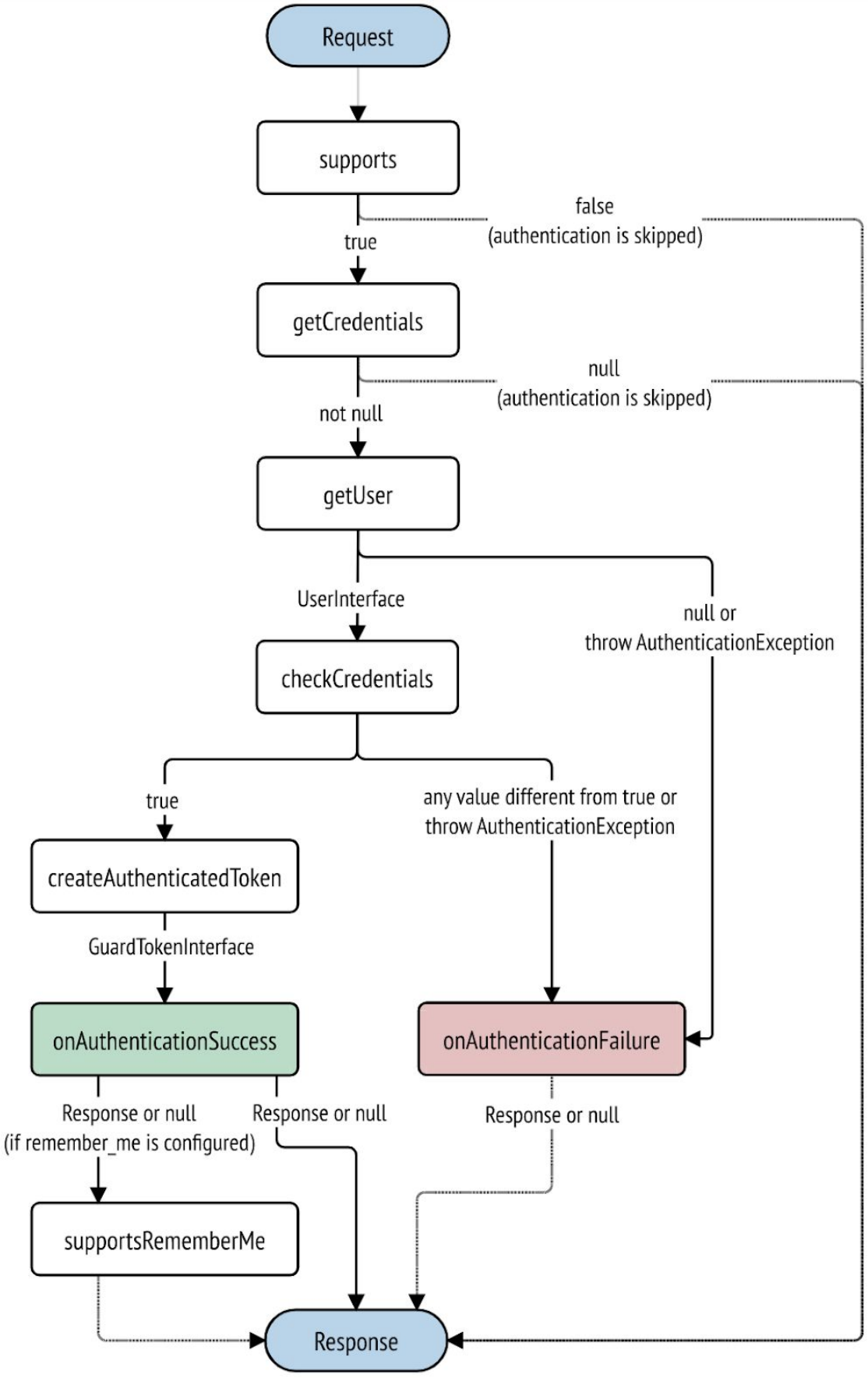

schéma complet du système d'authentification de Symfony

Accès & Rôles

Le système d'autorisation d'accès aux ressources a deux aspects. Le premier indique quel rôle on doit avoir pour accéder à une ressource, et l'autre indique quels rôles l'utilisateur authentifié possède et donc à quelles ressources il a accès.

Les rôles

La classe qui représente l'utilisateur (**User** par exemple) implémente l'interface **UserInterface**.
Cette interface contient la méthode **getRoles()** qui retourne un tableau contenant les différents rôles de l'utilisateur.

```php
// src/Entity/User.php

// ...
class User
{
    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    // ...
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';

        return array_unique($roles);
    }
}
```

Exemple d'entité User.

Chaque rôle doit contenir le préfix **ROLE_**. Exemple **ROLE_USER** ou **ROLE_ADMIN**.

Il est possible de paramétrer une hiérarchie des rôles dans le fichier de configuration

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

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

Exemple de configuration

Sécuriser l’accès de manière globale avec access_control

Il est possible de contrôler l’accès aux ressources de manière globale en ajoutant une configuration au niveau du fichier de configuration **config/packages/security.yaml**.

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

    firewalls:
        # ...
        main:
            # ...

    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: '^/admin', roles: ROLE_ADMIN }

        # or require ROLE_ADMIN or IS_AUTHENTICATED_FULLY for /admin*
        - { path: '^/admin', roles: [IS_AUTHENTICATED_FULLY, ROLE_ADMIN] }

        # the 'path' value can be any valid regular expression
        # (this one will match URLs like /api/post/7298 and /api/comment/528491)
        - { path: ^/api/(post|comment)/\d+$, roles: ROLE_USER }
```

Exemple de configuration

Sécuriser l’accès simplement au niveau des ressources avec denyAccessUnlessGranted() ou @isGranted()

denyAccessUnlessGranted($role)

Une autre façon de contrôler l’accès à une ressource est d’utiliser la méthode **denyAccessUnlessGranted($role)**. Celle-ci permet de restreindre l’accès à une ressource depuis un contrôleur.

```php
// ...

public function adminDashboard()
{
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');

    // ...
}
```

Exemple d'utilisation de denyAccessUnlessGranted

@isGranted($role)

Cette annotation est disponible si on installe le composant de symfony **sensio/framework-extra-bundle**.

Elle permet d'ajouter une annotation au niveau des contrôleurs qui permet de sécuriser l'accès à une ressource de façon simple.

```
    // src/Controller/AdminController.php
    // ...

+ use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

+ /**
+  * Require ROLE_ADMIN for *every* controller method in this class.
+  *
+  * @IsGranted("ROLE_ADMIN")
+  */
  class AdminController extends AbstractController
  {
+     /**
+      * Require ROLE_ADMIN for only this controller method.
+      *
+      * @IsGranted("ROLE_ADMIN")
+      */
      public function adminDashboard()
      {
          // ...
      }
  }
```

Exemple d'annotation @isGranted

Les contrôles complexes avec les Voters

Les **Voters** sont des classes qui étendent la classe **Voter**. Elles permettent d'avoir un contrôle très précis sur l'accès à une ressource.

Le sujet de cette documentation n'étant pas le contrôle d'accès, pour plus d'information se référer à la documentation officielle de Symfony sur le sujet.

Résumé de la première partie

Dans cette première partie, vous avez pris connaissance de façon rapide des différentes briques qui composent la sécurité dans Symfony. Ce résumé me semblait important pour présenter le système

d'authentification implémenté dans cette application, sujet qui sera abordé dans la deuxième partie de cette documentation.

Pour plus d'informations il est impératif de se référer à la documentation officielle. Celle-ci est très complète et comporte de nombreux exemples.

Partie 2: Configuration de l'authentification dans l'application

Les fichiers

Ci dessous la liste des fichiers utilisés pour l'authentification de l'utilisateur. Leur utilisation sera détaillée dans la suite de cette documentation.

Configuration

Toute la configuration du système de sécurité et donc d'authentification se trouve dans un seul fichier: **config/packages/security.yaml**.

Celui-ci comporte plusieurs sections :

- **encoders**: paramètre d'encodage des mot de passe
- **providers**: ici sont configurés les **User Providers**
- **role_hierarchy**: configuration de la hiérarchie des rôles
- **firewalls**: configuration des différents systèmes d'authentification
- **access_control**: sécurisation globale des différentes parties de l'application.

Utilisateur

L'utilisateur est représenté par la classe **App\Entity\User**.

Controleur

La gestion de la route **/login** de l'application est gérée par la classe Controller **App\Controller\SecurityController**.

Template

Le template qui correspond au formulaire HTML de login se trouve dans le fichier **templates/security/login.html.twig**

Guard authenticator

Le guard authenticator utilisé pour l'application est la classe **App\Security\AppUserAuthenticator**.

Processus d'authentification

Scénario

Regardons d'abord le processus d'authentification d'un point de vue fonctionnel:

L'utilisateur se rend sur la page d'accueil de l'application. Si il n'est pas authentifié, il est automatiquement redirigé vers la page de login comportant un formulaire de login. L'utilisateur entre alors ses identifiants (**nom d'utilisateur** et **mot de passe**). S'ils sont valides, l'utilisateur est ensuite

redirigé automatiquement vers la page d'accueil de l'application, sinon, il est redirigé vers la page de login et un message affiche le problème rencontré (utilisateur inexistant, mot de passe incorrect, …).

Processus technique

Derrière ce scénario, voici le processus qui s'opère:

1 - "L'utilisateur se rend sur la page d'accueil de l'application. Si il n'est pas authentifié, il est automatiquement redirigé vers la page de login comportant un formulaire de login"

Le paramétrage de **access_control** du fichier de configuration indique que pour accéder à l'URI "/", l'utilisateur doit posséder le rôle **ROLE_USER**. L'utilisateur n'étant pas encore authentifié, Il est automatiquement redirigé vers la route **login** (URI "/login") qui autorise les utilisateurs anonymes (**IS_AUTHENTICATED_ANONYMOUSLY**). Ce comportement est défini dans la méthode **start** du guard authenticator.

*Note: Le guard authenticator de l'application **App\Security\AppUserAuthenticator** étend la classe **Symfony\Component\Security\Guard\Authenticator\AbstractFormLoginAuthenticator** fourni par Symfony. C'est dans cette classe que la méthode **start()** est définie.*

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

*Configuration de la partie **access_control***
*dans le fichier de configuration **config/packages/security.yaml***

2 - "L'utilisateur entre alors ses identifiants (**nom d'utilisateur** et **mot de passe**)"

Lorsque l'utilisateur est redirigé vers l'URI "/login", Symfony exécute alors la méthode **login** du contrôleur responsable de cette route : **App\Controller\SecurityController** qui appelle le template **templates/security/login.html.twig**

Celui-ci fait plusieurs choses:

Tout d'abord, il vérifie si l'utilisateur est déjà authentifié, auquel cas, il le redirige vers la route **homepage** correspondant à l'URI "/".

Puis, si l'utilisateur a déjà essayé une première fois de s'authentifier, il récupère les informations correspondant à cette précédente authentification.
Deux paramètres sont récupérés: **$error** qui contient le message renvoyé par une éventuelle exception levée dans le **Guard Authenticator**, et **$lastUsername** qui contient le nom de l'utilisateur saisi dans la précédente tentative d'authentification.

*Note: ici $**lastUsername** représente l'attribut déclaré dans le fichier de configuration pour récupérer l'utilisateur. Il peut s'agir de l'attribut **username**, mais cela pourrait aussi être un attribut **email**, ou tout autre attribut unique de l'objet User qui permettrait d'identifier l'utilisateur.*

Enfin il renvoie un objet **Symfony\Component\HttpFoundation\Response** contenant la vue correspondant à la page login de l'application.

Formulaire d'authentification

Ce formulaire HTML contient les champs suivants:

Username

Correspond à l'attribut **username** de l'entité **App\Entity\User**. Cette information sera récupérée par le **guard authenticator** dans la méthode **getCredentials()**, puis passée à la méthode **getUser()** qui permettra de récupérer l'utilisateur.

Password

Correspond à l'attribut **password** de l'utilisateur. Celui-ci sera également récupéré par le **guard authenticator** dans la méthode **getCredentials()** puis passée à la méthode **checkCredentials()** pour vérifier le mot de passe saisi par l'utilisateur.

Se souvenir de moi

Correspond à la fonctionnalité "remember me" de symfony qui permettra d'envoyer un cookie au navigateur permettant à l'utilisateur de ne pas se ré-authentifier à chaque fois. (voir la section **remember_me** du fichier de configuration config/packages/security.yaml)

CSRF Token

Ce paramètre sera vérifié lors de l'exécution du **guard authenticator** afin de prévenir le formulaire des attaques CSRF.

3 - "S'ils sont valides, l'utilisateur est ensuite redirigé automatiquement vers la page d'accueil de l'application, sinon, il est redirigé vers la page de login et un message affiche le problème rencontré (utilisateur inexistant, mot de passe incorrect, …)"

Une fois le formulaire d'authentification soumis par l'utilisateur, cette dernière étape correspond à l'exécution du **guard authenticator**.

Dans l'application il s'agit de la classe **App\Security\AppUserAuthenticator**.

Voici étape par étape le détail de son exécution:

Supports

Dans l'app la méthode supports vérifie que la route de la requête correspond bien à la route **/login** et que la méthode de cette requête est bien **POST** (soumission du formulaire).

getCredentials

Ici on extrait les trois attributs **username, password** et **_csrf_token** de l'objet **Request**, puis on initialise la variable de session **Security::LAST_USERNAME** avec la valeur **username** récupérée. (de cette façon, si l'authentification échoue, on pourra récupérer le dernier nom d'utilisateur saisi dans le formulaire de login et l'injecter dans la variable **$lastUsername** vue précédemment).
Enfin on retourne ces trois attributs dans un tableau qui sera passé à la méthode **getUser()**.

getUser

Ici on commence par vérifier le **csrf token** pour vérifier qu'il correspond bien à celui utilisé dans le formulaire. Si ce n'est pas le cas, une exception de type **InvalidCsrfTokenException** est levée.

Comme on le verra plus en détail dans la section **Utilisateur** de cette documentation, le choix a été fait de persister les utilisateurs en base de données. L'étape suivante consiste donc à envoyer une requête en base de données pour tenter de récupérer l'utilisateur à partir du paramètre **username**. Si l'utilisateur n'est pas trouvé, une exception de type **CustomUserMessageAuthenticationException** est levée (ce message sera donc renvoyé à la variable **$error** de la méthode **login** du **SecurityController**.
Enfin, si l'utilisateur est trouvé, celui-ci sera renvoyé à la méthode **checkCredentials()**.

checkCredentials

On fait ici appel au service **UserPasswordEncoderInterface** dont on a parlé dans la première partie de cette documentation. Celui-ci va nous permettre de vérifier que le mot de passe saisi par l'utilisateur correspond bien à celui de l'utilisateur récupéré par la méthode **getUser()**.

onAuthenticationSuccess

Une fois l'utilisateur récupéré et son mot de passe validé, cette méthode est exécutée.

> **Important !**
> Le **guard authenticator** utilisé dans cette application étend un **guard authenticator** fourni par Symfony. Il s'agit du **AbstractFormLoginAuthenticator**.
> Celui-ci fait plusieurs choses. Tout d'abord il définit le comportement de la méthode **onAuthenticationFailure**: Dans celle-ci il récupères l'exception relevée précédemment et la passe dans la variable de session **Security::AUTHENTICATION_ERROR** puis elle redirige l'utilisateur vers la page de login.
> Elle définit également la méthode **start**: Si l'utilisateur tente d'accéder à une ressource nécessitant une authentification mais qu'aucune information n'est fournie dans la requête, elle redirige celui-ci vers la page de login.
> Enfin elle définit la méthode **supportsRememberMe()**: elle renvoie toujours **true**, ce qui permet de supporter par défaut la fonctionnalité "remember me"

Comme vu ci-dessus, si l'utilisateur tente d'accéder à une ressource protégée mais qu'il n'est pas authentifié, il est redirigé vers la page de login. Cette méthode commence donc par vérifier si l'utilisateur a été redirigé de cette façon, et si oui, le redirige vers la route précédemment demandée automatiquement !

Si ce n'est pas le cas, l'utilisateur est redirigé vers la page d'accueil de l'application.

Note

Maintenant que le processus d'authentification a été vu en détail, nous allons nous intéresser à l'entité utilisateur et au fichier de configuration de la sécurité.

L’entité User

App\Entity\User

Comme démontré dans la partie précédente, une application symfony nécessitant un espace sécurisé doit avoir une classe qui implémente l’interface **Symfony\Component\Security\Core\User\UserInterface**.

L’utilisateur dans l’application est donc représenté par la classe **App\Entity\User.**

Cette classe possède les attributs suivants:

- **username**: attribut unique permettant d’authentifier l’utilisateur
- **password**: mot de passe encrypté de l’utilisateur
- **plainPassword**: attribut utilisé lors de la création ou la modification d’un utilisateur pour stocker le mot de passe avant son cryptage.
- **email**: attribut unique (ne sert pas à l’authentification de l’utilisateur)

Stockage des utilisateurs

Les utilisateurs sont persistés en base de données via l’ORM Doctrine.

La gestion de la base de données n’est pas le sujet de cette documentation, pour plus d’informations, se référer à la documentation officielle.

Configuration

La configuration du système d'authentification se trouve dans le fichier **config/packages/security.yaml**.

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

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

    role_hierarchy:
        ROLE_ADMIN: ROLE_USER

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: app_user_provider
            guard:
                authenticators:
                    - App\Security\AppUserAuthenticator
            remember_me:
                secret: '%kernel.secret%'
                lifetime: 604800 # 1 week in seconds
                path: /
                secure: true
                httponly: true

            logout:
                path: logout
                target: homepage

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

```

*Fichier de configuration **config/packages/security.yaml***

Détail de la configuration :

encoders

Dans cette section on définit le paramètre **algorithm** pour l'entité **App\Entity\User**. Celui-ci est défini à **auto** ce qui signifie que Symfony tentera d'utiliser le meilleur algorithme disponible sur le serveur ou l'application est installée.

providers

C'est dans cette section que l'on définit le **user provider** utilisé pour l'application. Dans notre cas il s'agit de l'**EntityUserProvider** fourni par Doctrine. Celui-ci est défini par la clé **entity** et nécessite deux paramètres. Le paramètre **class** qui indique le nom de la classe correspondant à notre entité utilisateur (dans notre cas l'entité **App\Entity\User**) et le paramètre **property** qui correspond à la propriété qui identifie notre utilisateur de façon unique et qui est utilisée par le provider pour récupérer l'utilisateur. Dans notre cas l'attribut **username**.

role_hierarchy

Dans cette section, on définit la hiérarchie des rôles utilisés dans l'application. Dans notre cas le rôle **ROLE_ADMIN** hérite des droits du rôle **ROLE_USER**.

firewalls

Cette section est très importante. C'est ici que l'on définit le système de sécurité de l'application.
On peut voir dans notre configuration que deux **firewalls** ont été définis.

dev

Ce firewall est configuré par défaut lors de l'installation du composant de sécurité **symfony/security-bundle** de Symfony. Il permet de ne pas accidentellement bloquer les routes correspondantes au système de débogage de de Symfony.

main

Ce firewall définit la configuration complète du système de sécurité mis en place dans l'application.

anonymous

Ce paramètre permet d'être connecté en tant qu'anonyme dans le cas ou l'utilisateur n'est pas authentifié. La valeur **lazy** permet de ne pas démarrer la session dans ce cas de configuration. Cela permet de rendre les requêtes "cachables".

provider

Ce paramètre indique quel user provider doit être utilisé pour ce firewall. Dans notre cas, le paramètre vaut **app_user_provider**. (celui déclaré dans la section **providers** du fichier de configuration)

guard > authenticators

Ce paramètre définit quel authenticator doit être utilisé lors du processus d'authentification de l'utilisateur. Dans notre cas nous utilisons la classe **App\Security\AppUserAuthenticator**.

remember_me

Dans cette section on définit les différents paramètres du cookie qui sera envoyé vers le client :

- **secret**: la clé secrète utilisée pour le cookie. Dans notre cas **'%kernel.secret%'** qui fait référence à la variable d'environnement **APP_SECRET**.
- **lifetime**: le temps de validité du cookie.
- **path:** cette valeur indique à quelle partie de l'application le cookie s'applique. Dans notre cas la valeur est "**/"** ce qui indique que ce cookie est valable pour toutes les routes de l'application.
- **secure**: si ce paramètre est à **true**, cela indique que le cookie doit être envoyé au client via **https**.
- **httponly**: si défini à **true**, indique que le cookie n'est accessible que par le protocole http. Cela évite que celui-ci ne soit accessible via des langages de script comme javascript.

logout

Dans cette section on définit les arguments suivants:

- **path:** correspond au nom de la route qui correspond au logout. (la méthode de contrôleur ne sera jamais exécutée. Elle sera interceptée par le système, cependant elle doit être présente dans l'application.
- **target**: indique le nom de la route vers laquelle l'utilisateur doit être redirigé après la déconnexion.

access_control

Dans cette section sont définis les paramètres de sécurité globaux de l'application.

Ils sont déclarés sous forme de liste. Cette liste est exécutée dans l'ordre à chaque requête, et le premier item trouvé et qui correspond à la route sera utilisé. Les autres seront ignorés.

Trois paramètres ont été définis:

- { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
Celui-ci indique que le path **/login** est accessible aux utilisateurs anonymes.

- { path: ^/users, roles: ROLE_ADMIN }
Celui-ci indique que les routes commençant par le préfix **/users** ne sont accessibles qu'aux utilisateurs ayant le rôle **ROLE_ADMIN**.

- { path: ^/, roles: ROLE_USER }
Celui-ci indique que l'ensemble des routes de l'application sont accessibles uniquement aux utilisateurs ayant le rôle **ROLE_USER**.

Note: Cette règle s’applique aussi aux utilisateurs ayant le rôle **ROLE_ADMIN** car dans la section **roles_hierarchy**, il a été défini que les utilisateurs ayant le rôle **ROLE_ADMIN** héritent automatiquement du rôle **ROLE_USER**.

Conclusion

Cette documentation explique rapidement le système d'authentification mis en place dans l'application. Cependant, il est vivement conseillé de se référer régulièrement à la documentation officielle de Symfony. Celle-ci est très documentée et comporte de nombreux exemples expliquant en détail chaque point abordés.