Explication du système d'authentification

Symfony 5.3.7

Version : 0.1
Date de la dernière mise à jour : 23/09/2021

Sommaire

Table des matières

1. Configuration de base

1.1 Création de l'utilisateur et du système d'authentification avec le Maker Bundle

Grâce au Maker Bundle, Symfony permet de créer un système fonctionnel d'authentification. Pour cela, il suffit de lancer les commandes :
`php bin/console make:user` et :
`php bin/console make:auth`

Il est également possible de créer ces fichiers à la main pour bien comprendre comment tout fonctionne :

- Dans notre cas, pour la création de l'utilisateur, il suffit de créer une entité User qui comporte au moins les attributs :
  - (int) Id
  - (string) userIdentifier (par exemple email ou username) : sert d'identifiant unique pour chaque utilisateur, doit donc être taggé comme unique = true.
  - (string) password
  - (array) roles : permet de gérer les accès en fonction du rôle de l'utilisateur.

  Les getters et setters correspondants doivent également être créés.
  Attention, pour l'identifiant unique (par exemple l'username), il faudra en plus créer un getter nommé getUserIdentifier() car le getUsername() est déprécié depuis Symfony 5.3
- Le UserRepository doit comporter une méthode upgradePassword() qui s'occupe de hasher à nouveau le mot de passe de l'utilisateur passé en paramètre.
- Pour la création du système d'authentification, voir la partie 2. Aller plus loin : fonctionnement de l'authentification

Une fois l'User créé, ne pas oublier de créer le fichier de migration et le migrer en base de données.

1.2 Le fichier security.yaml

Le fichier config/packages/security.yaml contient la configuration de notre système. Il a du être actualisé lors du make:auth. Définir la valeur de la clef 'enable_authenticator_manager' comme true permet d'utiliser le nouveau système d'authentification de Symfony, que nous utilisons dans ce projet 😊

1.2.1 Hash du mot de passe

La clef 'password_hashers' permet de déterminer l'algorythme de hashage à utiliser pour l'entitée donnée. Il faut donc lui préciser sous forme de clef l'entité concernée, dans notre

cas ‘App\Entity\User’ , à laquelle nous précisons la valeur de l’algorythme avec la clef ‘algorithm‘. Dans notre cas, nous la définirons sur ‘auto’. En effet, ceci permet de modifier la technologie utilisée dans le cas où celle actuelle devient obsolète. Les hashs seront automatiquement mis à jour en base de données avec la technologie la plus pertinente.

1.2.2 Providers

Un provider permet à Symfony de retrouver l’User lié. Par exemple, à chaque requête, l’utilisateur enregistré en session peut avoir été modifié. C’est le provider qui se charge d’effectuer la requête pour le retrouver (c’est pour cela que dans l’audit de performance, on voit à chaque fois qu’une requête au moins apparaît lorsque l’on est connecté).
Pour en définir un, dans la partie ‘providers’, on nomme un ‘app_user_provider’ : Il s’agit d’une entité présente en base de données (correspond à la clef ‘class’) et à un champ de la table (clef ‘property’). Dans notre cas, nous avons l’entité ‘App\Entity\User’ et le champ ‘username’.
C’est la classe EntityUserProvider dans le namespace Symfony\Bridge\Doctrine\Security\User qui se charge d’utiliser dans le repository correspondant à la valeur de la clef ‘class’ la méthode findOneBy() avec comme paramètre la valeur qui a été donnée dans la clef ‘property’.

1.2.3 Firewalls

Un firewall permet d’utiliser un élément de sécurité spécifique en fonction de la requête, comme l’URL ou la méthode. L’ordre de déclaration est important puisque le 1er firewall qui matchera avec la requête sera utilisé.
Pour en créer un, il suffit de lui donner un nom, puis d’indiquer dans quel cas il doit être déclenché. Par exemple, la clef ‘pattern’ permet de définir les chemins qui déclencheront le firewall ; sa valeur doit être une expression régulière. Dans notre application, nous voyons que nous avons un firewall ‘dev’ : Il permet de désactiver la sécurité lorsque l’URL commence par /_profiler, /_wdt, /css, /images ou /js. Cela est utile pour s’assurer un accès permanent à la debug bar et aux fichiers publics.

Voici comment configurer un firewall :

- Pour modifier le point d’entrée (page vers laquelle est redirigé l’utilisateur lorsqu’il souhaite accéder à une page dont il n’a pas l’accès), il faut rajouter une clef ‘entry_point’ dans le firewall en question et lui donner le service qui correspond à la gestion du point d’entrée. Ce service est configuré au même niveau grâce à la clef du même nom. On peut alors lui spécifier un ‘login_path’ : le chemin vers la page souhaitée (dans notre cas, nous redirigeons vers la page de login), ainsi qu’un ‘default_target_path’ qui correspond au chemin vers lequel est redirigé l’utilisateur lorsqu’il se connecte mais qu’il n’a pas demandé de page spécifique en amont (par exemple s’il se rend directement sur la page de login). Dans notre cas, nous redirigeons vers la page d’accueil.

- Pour définir la page vers laquelle est redirigée l'utilisateur lors de la déconnexion, il faut modifier la valeur de la clef 'path' dans 'logout' dans le firewall en question. Dans notre cas, nous redirigeons vers la page d'accueil.

- La clef 'custom_authenticators' correspond à la classe Authenticator que nous souhaitons utiliser. Dans notre cas, nous utilisons celle générée par la commande make:auth.

1.2.4 Contrôle des accès

Cette partie permet de modifier les règles d'accès en fonction des URL. C'est ici par exemple que nous autorisons un utilisateur non connecté à accéder à la page de connexion, mais que nous lui refusons l'accès aux pages des tâches.

Voici les règles à connaître :

- L'ordre est important : le premier contrôle d'accès qui match avec le chemin demandé sera utilisé
- La valeur de la clef 'path' correspond au chemin pour lequel l'utilisateur doit correspondre aux valeurs suivantes, comme par exemple la valeur de la clef 'roles'.
- Le ^ permet d'indiquer que seuls les chemins qui commencent par l'expression qui suit sont concernés.

Dans notre cas, nous voyons que tout le monde peut accéder à la page de connexion et de création d'un utilisateur, qu'il faut être administrateur pour accéder à la gestion des membres, et qu'il faut être connecté pour accéder à la page d'accueil et à toute autre url.

Cette approche peut être suffisante. Toutefois, ce système a ses limites lorsque nous voulons avoir des règles d'accès plus précises ; nous allons voir comment les mettre en place dans la partie suivante.

1.3 Autres méthodes de gestion des accès et des droits

1.3.1 Contrôle du rôle dans les contrôleurs et les templates

La classe AbstractController contient une méthode denyAccessUnlessGranted() qu'il est possible d'appeler pour refuser l'accès à la fonction dans un contrôleur en fonction du rôle passé en paramètre. Il est aussi possible de l'utiliser sous forme d'attribut en important Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted.
Avec Twig, la même fonction peut être appelé en utilisant is_granted() qui retournera un booléen.

1.3.2 Utilisation des voters

Les voters permettent de préciser les droits d'accès en fonction de différentes situations. Un voter doit étendre la classe Voter dans le namespace Symfony\Component\Security\Core\Authorization\Voter, qui implémente l'interface VoterInterface dans le même namespace.

Elle comporte 2 méthodes :

- supports(), qui permet de savoir si l'on est bien dans la situation.
- voteOnAttribute(), qui permet d'appeler la bonne méthode en fonction du paramètre envoyé.
- canAction() est une méthode privée appelée dans voteOnAttribute(), qui ne fait donc pas partie de l'interface. Elle contient la logique qui permet de déterminer si l'autorisation est donnée ou non.

Par exemple, dans le cas de Todo & Co, un TaskVoter a été créé dans App\Security\Voter.
Il a pour rôle de n'autoriser l'accès à la suppression d'une tâche que si l'utilisateur en est l'auteur, ou si la tâche est liée à un utilisateur anonyme et que l'utilisateur est un administrateur.
Pour appeler un voter, il faut utiliser la même fonction que pour le contrôle de rôle, mais avec en 1er paramètre l'action à vérifier et en 2nd paramètre le sujet qui doit être vérifié.

2. Aller plus loin : fonctionnement de l'authentification

2.1 Détail de Make:auth

2.1.1 Vision d'ensemble

Comme nous l'avons vu, une fois que nous avons notre User, il suffit de lancer la commande make:auth pour avoir un système fonctionnel. Mais que fait cette commande ? 🤔
Elle met à jour le fichier security.yaml (que nous avons déjà passé en revue ici) et créer 3 autres fichiers :

- src/Security/LoginFormAuthenticator.php
- src/Controller/SecurityController.php
- templates/security/login.html.twig

Le contrôleur SecurityController contient les routes login et logout, mais la logique ne se trouve pas à l'intérieur. La fonction login permet éventuellement de récupérer le dernier nom d'utilisateur et les erreurs.
La logique se trouve effectivement dans la classe LoginFormAuthenticator, dont la classe parente implémente l'interface AuthenticatorInterface dans le namespace Symfony\Component\Security\Http\Authenticator. Les méthodes à déclarer sont au nombre de 5 :

2.1.2 Utiliser l'interface AuthenticatorInterface

- supports() : Permet d'identifier si l'authentification est demandée (en analysant la requête : s'il s'agit d'une requête d'authentification, elle retourne true)
  Dans notre cas, elle renvoie true si la requête est de méthode POST et correspond à la route 'app_login'.

- authenticate() : Elle doit renvoyer un Passeport de type PasseportInterface.
  Dans notre cas, elle récupère les informations de connexion et le token CSRF depuis la requête, elle sauvegarde le nom d'utilisateur en session (pour remettre l'email dans le champ en cas d'erreur d'authentification), puis elle génère un Passerport.

  Ce système de passeport fait partie des changements dans le nouveau système d'authentification (le traitement ne se déroule plus directement dans la fonction authenticate()).

  L'objet Passeport prend trois paramètres :

- Le 1er est un Badge qui correspond à l'UserIdentifier (dans notre cas, le nom d'utilisateur).
- Le 2ème est un Badge qui contient le mot de passe.
- Le 3ème est un tableau de badges additionnels : dans notre cas, nous souhaitons également valider le badge token CSRF.

Une fois que le passeport est créé, la logique se poursuit dans des listeners qui vont se charger de vérifier que les informations sont correctes. Par exemple, le UserCheckerListener dans Symfony\Component\Security\Http\EventListener va vérifier si le passeport implémente une certaine interface et envoyer au UserChecker (qui implémente la UserCheckerInterface) l'utilisateur envoyé depuis le badge. La logique est semblable pour la vérification du mot de passe et du token CSRF.
Il est donc possible de personnaliser le système d'authentification en créant de nouveaux badges et listeners qui seront construits de la même manière.
Pour mieux comprendre, il est possible de faire l'analogie avec un aéroport : le voyageur présente son passeport, les vérifications se font ensuite en fonction des badges présents dessus.

× createAuthenticatedToken() : Créer un token qui correspond à l'User authentifié, sauvegardé dans la session. Dans notre cas, nous gardons la fonction générée par défaut dans l'AbstractAuthenticator.

× onAuthenticationSuccess() : Est exécutée lorsque l'authentification est correcte. Dans notre cas, nous redirigeons vers la page d'accueil.

× onAuthenticationFailure() : Est exécutée lorsque l'authentification est incorrecte. Dans notre cas, nous redirigeons vers la route 'invalid-credentials' qui se charge d'ajouter un message flash d'erreur de connexion puis de rediriger vers la page de connexion.

Pour rappel, cet Authenticator est enregistré dans le fichier config/packages/security.yaml, dans le pare-feu principal, nommé avec la clef 'custom_authenticators'.

2.2 Etendre le système d'authentification

2.2.1 Comment créer un un Authenticator personnalisé

Pour certains besoins, il peut être nécessaire de créer un autre système d'authentification. Pour cela, il est possible de créer un nouvel Authenticator ; voici les étapes à suivre :

× Créer une nouvelle classe dans App\Security qui étend le AbstractAuthenticator (dans le cas où nous ne souhaitons pas personnaliser createAuthenticatedToken, ce qui est très souvent le cas).

× Ajouter les méthodes supports(), authenticate(), onAuthenticationSuccess() et onAuthenticationFailure().

Suivre la logique expliquée plus haut pour remplir ces fonctions.

Pour la méthode authenticate(), dans le cas où il n'y a pas besoin de vérifier un mot de passe (par exemple lors de l'utilisation d'un token d'API), il convient d'utiliser un nouveau Symfony\Component\Security\Http\Authenticator\Passport\SelfValidatingPassport au lieu d'un nouveau Passeport.

× Se rendre dans le fichier security.yaml et ajouter un nouveau custom_authenticators qui correspond au namespace de la classe créée.

2.2.2 Personnaliser la réponse en fonction du firewall

Pour personnaliser la réponse donnée en cas d'accès refusé selon le firewall déclenché : Créer une nouvelle classe dans App\Security qui implémente AccessDeniedHandlerInterface et ajouter la fonction handle qui prend en paramètre la requête et une exception de type AccessDeniedException. Retourner dans cette fonction le code voulu. Dans security.yaml, rajouter une clef 'access_denied_handler' dans le firewall en question et y indiquer comme valeur la classe créée.

Conclusion

L'authentification est un composant complexe, mais qu'il est nécessaire d'appréhender si l'on souhaiter personnaliser les choses et créer une authentification plus complexe et précise que celle offerte par défaut dans Symfony.
Nous avons vu que l'Authenticator est le cœur de notre firewall, il s'agit d'une classe qui a pour objectif d'intercepter les requêtes et d'authentifier l'utilisateur avec un système de passeport. Des listeners interviennent ensuite pour vérifier les badges du passeport, puis l'Authenticator reprend la main en créant le token de l'User identifié et indiquant l'action à effectuer en fonction de la réponse des listeners.