Sécurité et authentification avec Symfony

Table des matières

Introduction

La sécurité des applications est essentielle pour s'assurer de la confidentialité des données, qui peuvent être sensibles, notamment des informations à caractère personnel ou encore des données financières, qui doivent être protégées.

Cette protection peut être mise en place au moyen d'un système d'authentification qui permet de limiter les accès des utilisateurs en fonction de privilèges qu'on souhaite ou non leur accorder.

Symfony est doté d'un composant nommé 'Security' destiné à l'implémentation de l'authentification.

Authentification ou autorisation ?

Tout d'abord, il faut différencier authentification et autorisation.

L'authentification est le fait d'identifier un utilisateur, c'est-à-dire de le reconnaître et l'associer à une personne, ou même à une entité, clairement définie.

L'autorisation est le fait d'accorder ou non l'accès à une partie de l'application, en fonction de critères spécifiques, tels que le fait que l'utilisateur soit authentifié ou non, ou le rôle qui peut être accordé à un utilisateur authentifié.

Les deux notions fonctionnement souvent ensemble, les autorisations étant accordées à des utilisateurs authentifiés.

Les 'briques' du composant Security

Le composant Security de Symfony est basé sur trois axes :

- Le provider : il a pour rôle de stocker et fournir les utilisateurs à l'application
- Le firewall : c'est lui qui vérifie à chaque demande de l'utilisateur si une authentification est nécessaire pour accéder à la fonctionnalité
- L'Access Control : il contrôle les autorisations pour les routes où une permission est requise

Le provider

Le provider a pour rôle de fournir les utilisateurs à l'application, c'est-à-dire l'ensemble des informations utiles les concernant. La source de ces utilisateurs peut être une autre application, un fichier… Cependant, dans nombreux cas les utilisateurs sont stockés dans une base de données. C'est la solution qui a été retenue pour l'application « ToDo List ».

Lorsque les utilisateurs sont stockés dans la base de données, il faut une Entité « User » pour les représenter.

Créer l'entité User

La commande « *php bin/console make:user* » permet de créer l'entité User, ou par la suite de la modifier. Il est recommandé de l'utiliser, car elle permet de ne rien oublier et créé ou modifie automatiquement les éléments requis, parmi lesquels les fichiers de configuration.

L'entité User doit implémenter les interfaces *UserInterface* et *PasswordAuthenticatedUserInterface*, la seconde permettant le hashage des mots de passe.
Il est important de définir un champ en tant que « *user identifier* », autrement dit un identifiant qui permettra à l'utilisateur de s'authentifier.

Configurer le provider

La configuration du provider, c'est-à-dire de quelle source charger les utilisateurs, requiert de modifier le fichier « config/packages/security.yaml ».
Rappel : dans notre cas, il s'agira de la base de données, les utilisateurs seront représentés par l'entité Doctrine « User ».

Si la commande « *make:user* » a été utilisée, la section 'providers' du fichier a été mise à jour automatiquement :

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

    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
```

Fonctionnement du provider

Au début de chaque requête, l'utilisateur est chargé à partir de la session (sauf si le pare-feu est sans état, ce qui peut être le cas par exemple pour les API REST). S'il est authentifié, le provider « rafraîchit » les données (il interroge la base de données pour récupérer les informations relatives à l'utilisateur) pour s'assurer que toutes les informations sont à jour (et si nécessaire, si quelque chose a changé, l'utilisateur est déconnecté et l'authentification supprimée de la session).

Enregistrer de nouveaux utilisateurs

Pour permettre l'ajout de nouveaux utilisateurs, il faut créer un formulaire d'inscription. Cela peut être facilité via le commande « make:registration-form » : elle aide à configurer le contrôleur d'inscription et à ajouter des fonctionnalités telles que la vérification de l'adresse e-mail en utilisant le SymfonyCastsVerifyEmailBundle.

Cette extension doit être installée au préalable :

```
composer require symfonycasts/verify-email-bundle
php bin/console make:registration-form
```

Le hashage des mots de passe

Le hashage est l'action qui consiste à transformer une chaîne de caractères en une autre, alors appelée « hash », ou « empreinte ». La transformation est non réversible, c'est-à-dire que la chaîne d'origine ne peut pas être retrouvée.
Cette méthode permet notamment de stocker des mots de passe dans une base de données de façon sécurisée. Ainsi quelqu'un qui accéderait aux informations de l'utilisateur ne pourrait pas connaître son mot de passe et ne pourrait rien faire à partir du hash.

Il existe plusieurs algorithmes permettant de réaliser le hash d'une chaîne de caractères. Symfony utilise « bcrypt ».

Configuration

Pour permettre le hashage des mots de passe, la classe User doit implémenter l'interface *PasswordAuthenticatedUserInterface*.

Il faut spécifier dans le fichier de configuration « config/packages/security.yaml » la classe à utiliser pour le hashage. La commande *make:user* devrait l'avoir déjà modifié, si elle a bien été utilisée pour créer l'entité User :

```
# config/packages/security.yaml
security:
    # ...
    password_hashers:
        # Use native password hasher, which auto-selects and migrates the best
        # possible hashing algorithm (starting from Symfony 5.3 this is "bcrypt")
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

Cela permet ensuite d'utiliser le service *UserPasswordHasherInterface* pour hasher les mots de passe avant de les enregistrer.

Voici un exemple d'utilisation (pensez à inclure l'interface avec « use ») :

```
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

$hashedPassword = $passwordHasher->hashPassword(
            $user,
            $plaintextPassword
        );
```

$user est un objet User, représentant l'utilisateur dont il faut hasher le mot de passe.
$plaintextPassword est la chaîne de caractères contenant le mot de passe non hashé.
$hashedPassword est la chaîne de caractères obtenue après hashage de $plaintextPassword.

Le firewall

Le firewall, autrement dit « pare-feu », est le cœur du système d'authentification : il définit les parties de l'application qui sont sécurisées et la façon dont les utilisateurs pourront s'authentifier (par exemple, formulaire de connexion, jeton API, etc).

Authentifier les utilisateurs

Pour que les utilisateurs puissent s'authentifier, créer d'abord un controller pour le formulaire de connexion. Il doit contenir une fonction permettant la connexion des utilisateurs (avec le name « app_login » dans l'exemple ci-dessous).
Activer ensuite l'*authenticator* pour permettre l'authentification via le formulaire en utilisant le paramètre form_login :

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

    firewalls:
        main:
            # ...
            form_login:
                # "app_login" is the name of the route created previously
                login_path: app_login
                check_path: app_login
```

Une fois activé, le système de sécurité redirige les visiteurs non authentifiés vers le *login_path* lorsqu'ils essaient d'accéder à un endroit sécurisé.

Pensez bien sûr à créer ou modifier le template correspondant au formulaire.

Exemple de fonction d'authentification :

```
// SecurityController.php

    /**
     * @Route("/login", name="app_login")
     */
    public function loginAction(AuthenticationUtils $authenticationUtils)
    {
        if (null !== $this->getUser()) {
            $this->addFlash('error', "Vous êtes déjà authentifié.");
            return $this->redirectToRoute('app_home');
        }

        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render('security/login.html.twig', array(
            'last_username' => $lastUsername,
```

```
            'error'         => $error,
        ));
    }
```

Retourner les erreurs de connexion

En cas d'erreur d'authentification, la variable d'erreur passée dans le modèle est une instance de *AuthenticationException*. Elle peut contenir des informations sensibles sur l'échec de l'authentification, c'est pourquoi il ne faut jamais utiliser *error.message* dans le template : on doit utiliser la propriété *messageKey*.

```
{% if error %}
    <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
{% endif %}
```

Protection CSRF

Activer la protection CSRF pour le formulaire de connexion permet de le protéger des attaques. Pour l'activer pour le formulaire de connexion, il faut de nouveau modifier le fichier de configuration « security.yaml ».

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

    firewalls:
        secured_area:
            # ...
            form_login:
                # ...
                enable_csrf: true
```

Ensuite, utilisez la fonction csrf_token() dans le modèle Twig pour générer un jeton CSRF et le stocker dans un champ caché du formulaire. Par défaut, le champ HTML doit être appelé _csrf_token et la chaîne utilisée pour générer la valeur doit être authenticate :

```
{# connection form #}

{# ... #}
<form action="{{ path('app_login') }}" method="post">
    {# ... the login fields #}

    <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">

    <button type="submit">login</button>
</form>
```

Déconnexion

Pour permettre la déconnexion des utilisateurs, c'est-à-dire la suppression de leur authentification dans la session, il faut activer la fonctionnalité dans le fichier de configuration.

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

    firewalls:
        main:
            # ...
            logout:
                path: app_logout

                # where to redirect after logout
                # target: app_any_route
```

Puis créer une route dans le Security Controller (dans cet exemple, on veillera à ce que le *name* de cette route soit « app_logout » pour correspondre à ce qui est défini ci-dessus) :

```
    /**
     * @Route("/logout", name="app_logout", methods={"GET"})
     */
    public function logout(): void
    {
        // controller can be blank: it will never be called!
        throw new \Exception('Don\'t forget to activate logout in security.yaml');
    }
```

Pendant la déconnexion, un événement *LogoutEvent* est envoyé.

Personnaliser la déconnexion

Pour personnaliser la déconnexion des utilisateurs, on peut se servir du *LogoutEvent* qui est généré au moment de la déconnexion d'un utilisateur. Pour cela, il faut enregistrer un écouteur d'événement :

```
// src/EventListener/LogoutSubscriber.php
namespace App\EventListener;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;
use Symfony\Component\Security\Http\Event\LogoutEvent;

class LogoutSubscriber implements EventSubscriberInterface
{
    public function __construct(
        private UrlGeneratorInterface $urlGenerator
    ) {
    }
```

```
    public static function getSubscribedEvents(): array
    {
        return [LogoutEvent::class => 'onLogout'];
    }

    public function onLogout(LogoutEvent $event): void
    {
        // get the security token of the session that is about to be logged out
        $token = $event->getToken();

        // get the current request
        $request = $event->getRequest();

        // get the current response, if it is already set by another listener
        $response = $event->getResponse();

        // configure a custom logout response to the homepage
        $response = new RedirectResponse(
            $this->urlGenerator->generate('homepage'),
            RedirectResponse::HTTP_SEE_OTHER
        );
        $event->setResponse($response);
    }
}
```

Accéder à l'objet utilisateur

Dans un controller

Après l'authentification, l'objet User représentant l'utilisateur actuel est accessible via le raccourci *getUser()* dans le contrôleur de base.

```
        // usually you'll want to make sure the user is authenticated first,
        // see "Authorization" below
        $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');


        // returns your User object, or null if the user is not authenticated
        // use inline documentation to tell your editor your exact User class
        /** @var \App\Entity\User $user */
        $user = $this->getUser();
```

Dans un service

Pour récupérer l'utilisateur dans un service, il faut utiliser le Security service.

```
use Symfony\Component\Security\Core\Security;

class ExampleService
{
    private $security;

    public function __construct(Security $security)
    {
        // Avoid calling getUser() in the constructor: auth may not
        // be complete yet. Instead, store the entire Security object.
        $this->security = $security;
    }

    public function someMethod()
    {
        // returns User object or null if not authenticated
        $user = $this->security->getUser();

        // ...
    }
}
```

Dans les templates Twig

Dans Twig, l'utilisateur est disponible dans la variable app.user

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <p>Email: {{ app.user.email }}</p>
{% endif %}
```

L'autorisation et les rôles

On peut faire un premier contrôle d'accès basé sur le fait que l'utilisateur soit authentifié ou non, mais dans la plupart des cas ce n'est pas suffisant, et on souhaite protéger des parties de l'application et en restreindre l'accès à certains utilisateurs particuliers. C'est le cas pour les pages destinées à l'administration d'un site, par exemple.

Les rôles

L'autorisation est basée sur les **rôles**, que l'utilisateur reçoit quand il se connecte.

Le rôle par défaut est ROLE_USER. On s'assure que les utilisateurs aient toujours au moins « ROLE_USER », via la fonction *getRoles()* dans la classe *User* :

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

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

Les rôles sont un tableau dans la base de données (encodé en JSON). Les utilisateurs peuvent en avoir plusieurs (le nombre n'est pas limité).

Rôles accordés automatiquement à l'authentification d'un utilisateur

En plus du rôle « ROLE_USER », lorsqu'un utilisateur se connecte, il obtient automatiquement des rôles qui ne sont pas vraiment des rôles mais peuvent être utilisés comme tel :

- IS_AUTHENTICATED_REMEMBERED : Tous les utilisateurs connectés ont cet attribut, même s'ils sont connectés grâce à un cookie « remember me ». Même si vous n'utilisez pas la fonctionnalité « remember me », vous pouvez utiliser cet attribut pour vérifier si l'utilisateur est connecté.
- IS_AUTHENTICATED_FULLY : similaire à *IS_AUTHENTICATED_REMEMBERED*, mais plus fort. Les utilisateurs qui ne sont connectés que grâce à un cookie « remember me » auront IS_AUTHENTICATED_REMEMBERED mais pas IS_AUTHENTICATED_FULLY.
- IS_REMEMBERED : Seuls les utilisateurs authentifiés à l'aide de la fonctionnalité remember-me (c'est-à-dire un cookie remember-me).
- IS_IMPERSONATOR : Lorsque l'utilisateur actuel se fait passer pour un autre utilisateur dans cette session, cet attribut correspondra.

Ils ne sont pas réellement des rôles mais agissent comme tel, et peuvent être utilisés en remplacement des rôles dans toutes les situations où les rôles peuvent être utilisés : comme *access_control* ou dans *Twig*.

Définir des rôles personnalisés

Les rôles doivent toujours commencer par « ROLE_ ».
Hors de cette contrainte, ils peuvent être déterminés librement.

Héritage des rôles

Il est possible de définir des règles d'héritage des rôles en créant une hiérarchie de rôles, comme dans l'exemple ci-dessous :

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

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

Ici : Les utilisateurs ayant le rôle ROLE_ADMIN auront également le rôle ROLE_USER.
Les utilisateurs ayant le rôle ROLE_SUPER_ADMIN auront automatiquement les rôles ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH et ROLE_USER (hérité de ROLE_ADMIN).

Pour que la hiérarchie des rôles fonctionne, il ne faut pas utiliser *$user->getRoles()* manuellement, mais utiliser à la place *$this->isGranted('ROLE_MY_ROLE')*. Par exemple, dans un contrôleur étendant le contrôleur de base :

```
// BAD - $user->getRoles() will not know about the role hierarchy
$hasAccess = in_array('ROLE_ADMIN', $user->getRoles());

// GOOD - use of the normal security methods
$hasAccess = $this->isGranted('ROLE_ADMIN');
$this->denyAccessUnlessGranted('ROLE_ADMIN');
```

Restreindre les accès

Le pare-feu permet de définir des routes, via un système de de patterns, autrement dit « clé de motif », et de leur associer des critères d'accès, notamment basés sur les rôles.

A chaque requête, Symfony vérifie si la route est protégée. Il s'arrête à la première correspondance trouvée, regarde si l'utilisateur répond aux conditions d'accès qui sont définies, et ignore toutes les autres correspondances qui pourraient exister.

Toutes les URLs réelles sont gérées par le pare-feu principal (pas de clé de motif signifie qu'il correspond à toutes les URLs).

Pour paramétrer les accès en fonction des routes, il faut paramétrer la section « access_control » du fichier de configuration « security.yaml ».

Voici un exemple permettant de limiter l'accès aux routes contenant « /admin* ».

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

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

Note : Le pare-feu de développement est en fait un faux pare-feu : il s'assure que vous ne bloquez pas accidentellement les outils de développement de Symfony, qui se trouvent sous des URL comme /_profiler et /_wdt.

L'exemple ci-dessus montre que les patterns des routes sont en fait des expressions régulières.

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

    access_control:
        # matches /admin/users/*
        - { path: '^/admin/users', roles: ROLE_SUPER_ADMIN }

        # matches /admin/* except for anything matching the above rule
        - { path: '^/admin', roles: ROLE_ADMIN }
```

Le fait de faire précéder le chemin d'accès de ^ signifie que seules les URL commençant par le motif sont prises en compte. Par exemple, un chemin d'accès de /admin (sans ^) correspondrait à /admin/foo mais aussi à des URL comme /foo/admin.

Chaque contrôle d'accès peut également correspondre à l'adresse IP, au nom d'hôte et aux méthodes HTTP. Il peut également être utilisé pour rediriger un utilisateur vers la version https d'un motif d'URL.

Rendre publique l'accès à certaines pages

Dans la configuration du contrôle d'accès, vous pouvez utiliser l'attribut de sécurité PUBLIC_ACCESS pour exclure certaines routes de l'accès non authentifié (par exemple, la page de connexion) :

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

    # ...
    access_control:
        # allow unauthenticated users to access the login form
        - { path: ^/admin/login, roles: PUBLIC_ACCESS }

        # but require authentication for all other admin routes
        - { path: ^/admin, roles: ROLE_ADMIN }
```

Permettre l'accès à tous les utilisateurs authentifiés

Si certaines pages doivent être accessibles à tous les utilisateurs authentifiés, sans restriction de rôle, il y a deux possibilités :

- Vérifier qu'il ait le rôle utilisateur
- Utiliser l'attribut spécifique : $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');

Les autres attributs spécifiques listés précédemment peuvent être utilisés de la même façon.

Sécurisation des contrôleurs et d'autres codes

A l'intérieur d'un contrôleur, l'accès peut être vérifié et protégé selon les rôles de l'utilisateur, grâce à la méthode *denyAccessUnlessGranted* :

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

public function adminDashboard(): Response
{
   $this->denyAccessUnlessGranted('ROLE_ADMIN');

   // or add an optional message - seen by developers
   $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'User tried to access a page without
having ROLE_ADMIN');
}
```

Si l'accès n'est pas accordé, une exception spéciale *AccessDeniedException* est levée et aucun autre code du contrôleur n'est exécuté. Ensuite, l'une des deux choses suivantes se produira :

- Si l'utilisateur n'est pas encore connecté, il lui sera demandé de s'authentifier (par exemple, il sera redirigé vers la page de connexion).
- Si l'utilisateur est connecté, mais n'a pas le rôle ROLE_ADMIN, il verra apparaître la page 403 *access denied*.

Grâce au *SensioFrameworkExtraBundle*, on peut également sécuriser votre contrôleur en utilisant des annotations :

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

use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

/**
 * Require ROLE_ADMIN for all the actions of this controller
 *
 * @IsGranted("ROLE_ADMIN")
 */
class AdminController extends AbstractController
{
   /**
    * Require ROLE_SUPER_ADMIN only for this action
    *
    * @IsGranted("ROLE_SUPER_ADMIN")
    */
   public function adminDashboard(): Response
   {
      // ...
   }
}
```

Sécuriser les services

Les conditions d'accès peuvent être vérifiées n'importe où dans le code en injectant le service 'Security' de Symfony.
Dans l'exemple ci-dessous, le rapport sur les ventes ne sera accessible que pour les utilisateurs ayant le rôle ROLE_SALES_ADMIN :

```
// src/SalesReport/SalesReportManager.php

// ...
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Core\Security;

class SalesReportManager
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    public function generateReport()
    {
        $salesData = [];

        if ($this->security->isGranted('ROLE_SALES_ADMIN')) {
            $salesData['top_secret_numbers'] = rand();
        }

        // ...
    }

    // ...
}
```