

Documentation technique

Système d'authentification **Symfony 6**

Sommaire

1. Introduction

2. Implémentation de l'authentification

2.1 Quel(s) fichier(s) modifier et pourquoi ?

- Security bundle
- L'utilisateur
- Le pare-feu
- Le contrôle d'accès

2.2 Comment s'opère l'authentification.

- Connexion au formulaire
- Protection Csrf
- Json
- Http de base
- Lien de connexion
- Déconnexion

2.3 Où sont stockés les utilisateurs.

- Récupération du user
- Contrôle d'accès
- Rôles
- Sécurisation des données

2.4 En apprendre encore plus.

3. Apporter des modifications au projet et processus de qualité à utiliser

1. Introduction :

Dans le cadre de mes études et en vue du diplôme de Développeur d'application - PHP / Symfony avec Openclassrooms,

Il m'était demandé de produire une documentation expliquant comment l'implémentation de l'authentification avait été faite. Cette documentation se destine aux prochains développeurs juniors qui rejoindront l'équipe dans quelques semaines. Dans cette documentation, il doit être possible pour un débutant avec le framework Symfony de :

- comprendre quel(s) fichier(s) il faut modifier et pourquoi ;
- comment s'opère l'authentification ;
- et où sont stockés les utilisateurs.

Par ailleurs, il m'était également demandé de produire un document expliquant comment devront procéder tous les développeurs souhaitant apporter des modifications au projet.

Ce document devra aussi détailler le processus de qualité à utiliser ainsi que les règles à respecter.

2. Implémentation de l'authentification :

Symfony fournit de nombreux outils pour sécuriser votre application. Certains outils de sécurité liés à HTTP, tels [que les cookies de session sécurisée](#) et [la protection CSRF](#), sont fournis par défaut.

2.1 Le SecurityBundle :

Le SecurityBundle fournit toutes les fonctionnalités d'authentification et d'autorisation nécessaires pour sécuriser votre application.

Installez le SecurityBundle :

```
composer require symfony/security-bundle
```

Si Symfony Flex est installé, cela crée également un [security.yaml](#) fichier de configuration.

```
# config/packages/security.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'
    # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#firewalls-authentication

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

    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
```

L'utilisateur (providers)

Toute section sécurisée de votre application nécessite un certain concept d'utilisateur. Le fournisseur d'utilisateurs charge les utilisateurs à partir de n'importe quel stockage (par exemple, la base de données) sur la base d'un "identifiant d'utilisateur" (par exemple, l'adresse e-mail de l'utilisateur) ;

Le pare -feu et l' authentification des utilisateurs (firewalls)

Le pare-feu est au cœur de la sécurisation de votre application. Chaque demande dans le pare-feu est vérifiée si elle nécessite un utilisateur authentifié. Le pare-feu se charge également d'authentifier cet utilisateur (par exemple à l'aide d'un formulaire de connexion) ;

Contrôle d'accès (autorisation) (access_control)

À l'aide du contrôle d'accès et du vérificateur d'autorisation, vous contrôlez les autorisations requises pour effectuer une action spécifique ou visiter une URL spécifique.

2.1.1 L'utilisateur :

Les autorisations dans Symfony sont toujours liées à un objet utilisateur. Si vous avez besoin de sécuriser (des parties de) votre application, vous devez créer une classe d'utilisateurs. Il s'agit d'une classe qui implémente [UserInterface](#) . Il s'agit souvent d'une entité Doctrine, mais vous pouvez également utiliser une classe d'utilisateurs dédiée à la sécurité.

Le moyen le plus simple de générer une classe d'utilisateurs consiste à utiliser la `make:user` commande du [MakerBundle](#) :

```
$ php bin/console make:user
The name of the security user class (e.g. User) [User]:
> User

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> yes

Enter a property name that will be the unique "display" name for the user (e.g. email, username) [username]:
> email

Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be handled manually [yes]:
Does this app need to hash/check user passwords? (yes/no) [yes]:
> yes

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

Pour plus d'info : [authenticating-users](#)

Si votre utilisateur est une entité Doctrine, comme dans l'exemple ci-dessus, n'oubliez pas de créer les tables en créant et exécutant une migration :

```
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

Chargement de l'utilisateur : The User Provider :

Outre la création de l'entité, la `make:usercommande` ajoute également la configuration d'un fournisseur d'utilisateurs dans votre configuration de sécurité :

```
YAML XML PHP
# config/packages/security.yaml
security:
    # ...

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

Ce fournisseur d'utilisateurs sait comment (re)charger des utilisateurs à partir d'un stockage (par exemple une base de données) sur la base d'un "identifiant d'utilisateur" (par exemple l'adresse e-mail ou le nom d'utilisateur de l'utilisateur). La configuration ci-dessus utilise Doctrine pour charger l'User entité en utilisant la email propriété comme "identifiant d'utilisateur".

Les fournisseurs d'utilisateurs sont utilisés à plusieurs endroits au cours du cycle de vie de la sécurité :

Charger l'utilisateur en fonction d'un identifiant

Lors de la connexion (ou de tout autre authentificateur), le fournisseur charge l'utilisateur en fonction de l'identifiant de l'utilisateur. Certaines autres fonctionnalités, telles que l'emprunt d'identité d'utilisateur et Se souvenir de moi , l'utilisent également.

Recharger l'utilisateur depuis la session

Au début de chaque requête, l'utilisateur est chargé depuis la session (sauf si votre pare-feu est stateless). Le fournisseur "actualise" l'utilisateur (par exemple, la base de données est interrogée à nouveau pour de nouvelles données) pour s'assurer que toutes les informations de l'utilisateur sont à jour (et si nécessaire, l'utilisateur est désauthentié/déconnecté si quelque chose a changé). Voir Sécurité pour plus d'informations sur ce processus.

Symfony est livré avec plusieurs fournisseurs d'utilisateurs intégrés :

[Fournisseur d'utilisateurs d'entité](#)

Charge les utilisateurs d'une base de données à l'aide de [Doctrine](#) ;

[Fournisseur d'utilisateurs LDAP](#)

Charge les utilisateurs à partir d'un serveur LDAP ;

[Fournisseur d'utilisateurs de mémoire](#)

Charge les utilisateurs à partir d'un fichier de configuration ;

[Fournisseur d'utilisateurs de la chaîne](#)

Fusionne deux ou plusieurs fournisseurs d'utilisateurs en un nouveau fournisseur d'utilisateurs.

Enregistrement de l'utilisateur : hachage des mots de passe :

SecurityBundle fournit des fonctionnalités de hachage et de vérification de mot de passe.

Assurez-vous que votre classe User implémente la [PasswordAuthenticatedUserInterface](#) :

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

// ...
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;

class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    // ...

    /**
     * @return string the hashed password for this user
     */
    public function getPassword(): string
    {
        return $this->password;
    }
}
```

Ensuite, configurez le hacheur de mot de passe à utiliser pour cette classe. Si votre security.yaml fichier n'était pas déjà préconfiguré, il make:user aurait dû le faire pour vous :

YAML XML PHP

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

Maintenant que Symfony sait *comment* vous voulez hacher les mots de passe, vous pouvez utiliser le `UserPasswordHasherInterface` service pour le faire avant d'enregistrer vos utilisateurs dans la base de données :

```
// src/Controller/RegistrationController.php
namespace App\Controller;

// ...
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class RegistrationController extends AbstractController
{
    public function index(UserPasswordHasherInterface $passwordHasher)
    {
        // ... e.g. get the user data from a registration form
        $user = new User(...);
        $plaintextPassword = ...;

        // hash the password (based on the security.yaml config for the $user class)
        $hashedPassword = $passwordHasher->hashPassword(
            $user,
            $plaintextPassword
        );
        $user->setPassword($hashedPassword);

        // ...
    }
}
```

 Copy

2.2.2 Le pare-feu et l'authentification des utilisateurs (firewalls) :

Le firewalls section de config/packages/security.yaml est la section la plus importante. Un « pare-feu » est votre système d'authentification : le pare-feu définit quelles parties de votre application sont sécurisées et comment vos utilisateurs pourront s'authentifier (par exemple, formulaire de connexion, jeton API, etc.).

YAML XML PHP

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

    # activate different ways to authenticate
    # https://symfony.com/doc/current/security.html#firewalls-authentication

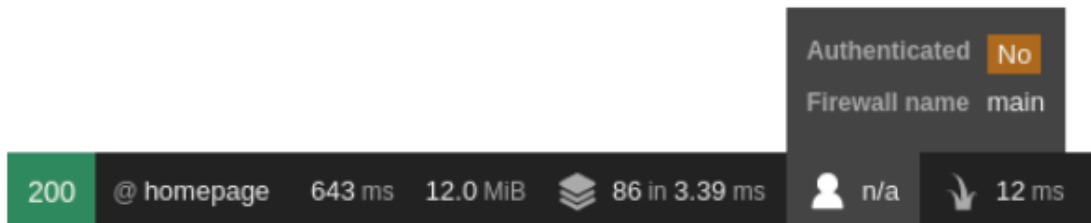
    # https://symfony.com/doc/current/security/impersonating_user.html
    # switch_user: true
```

Un seul pare-feu est actif sur chaque requête : Symfony utilise la pattern clé pour trouver la première correspondance (vous pouvez également faire [correspondre par hôte ou d'autres choses](#)).

Le dev pare-feu est vraiment un faux pare-feu : il s'assure que vous ne bloquez pas accidentellement les outils de développement de Symfony - qui vivent sous des URL comme `/_profiler` et `/_wdt`.

Toutes les URL *réelles* sont gérées par le main pare-feu (aucune pattern clé signifie qu'elle correspond à *toutes les* URL). Un pare-feu peut avoir plusieurs modes d'authentification, c'est-à-dire qu'il permet plusieurs façons de poser la question "Qui êtes-vous ?".

Souvent, l'utilisateur est inconnu (c'est-à-dire qu'il n'est pas connecté) lorsqu'il visite votre site Web pour la première fois. Si vous visitez votre page d'accueil en ce moment, vous y aurez accès et vous verrez que vous visitez une page derrière le pare-feu dans la barre d'utils :



Si vous ne voyez pas la barre d'outils, installez le [profileur](#) avec :

```
composer require --dev symfony/profiler-pack
```

2.2. Authentification des utilisateurs :

Lors de l'authentification, le système essaie de trouver un utilisateur correspondant au visiteur de la page Web. Traditionnellement, cela se faisait à l'aide d'un formulaire de connexion ou d'une boîte de dialogue de base HTTP dans le navigateur. Cependant, le SecurityBundle est livré avec de nombreux autres authenticateurs :

- [Connexion au formulaire](#)
- [Connexion JSON](#)
- [HTTP de base](#)
- [Lien de connexion](#)
- [Certificats clients X.509](#)
- [Utilisateurs distants](#)
- [Authenticateurs personnalisés](#)

- **Connexion au formulaire**

La plupart des sites Web ont un formulaire de connexion où les utilisateurs s'authentifient à l'aide d'un identifiant (par exemple, une adresse e-mail ou un nom d'utilisateur) et un mot de passe. Cette fonctionnalité est fournie par l'authentificateur de connexion par formulaire .

Tout d'abord, créez un contrôleur pour le formulaire de connexion :

```
$ php bin/console make:controller Login

created: src/Controller/LoginController.php
created: templates/login/index.html.twig
```

```
// src/Controller/LoginController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class LoginController extends AbstractController
{
    #[Route('/login', name: 'login')]
    public function index(): Response
    {
        return $this->render('login/index.html.twig', [
            'controller_name' => 'LoginController',
        ]);
    }
}
```

Ensuite, activez l'authentificateur de connexion par formulaire en utilisant le `form_login` paramètre :

```
YAML XML PHP
# config/packages/security.yaml
security:
    # ...

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



Les `login_path` et `check_path` prennent en charge les URL et les noms de route (mais ne peuvent pas avoir de caractères génériques obligatoires - par exemple, `/login/{foo}` où `foo` n'a pas de valeur par défaut).

Une fois activé, le système de sécurité redirige les visiteurs non authentifiés vers le `login_path` lorsqu'ils tentent d'accéder à un lieu sécurisé (ce comportement peut être personnalisé à l'aide de [points d'entrée d'authentification](#)).

Modifiez le contrôleur de connexion pour afficher le formulaire de connexion :

```
// ...
+ use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class LoginController extends AbstractController
{
    #[Route('/login', name: 'login')]
-    public function index(): Response
+    public function index(AuthenticationUtils $authenticationUtils): Response
    {
+        // get the login error if there is one
+        $error = $authenticationUtils->getLastAuthenticationError();
+
+        // last username entered by the user
+        $lastUsername = $authenticationUtils->getLastUsername();
+
        return $this->render('login/index.html.twig', [
-            'controller_name' => 'LoginController',
+            'last_username' => $lastUsername,
+            'error'          => $error,
        ]);
    }
}
```

Ne laissez pas ce contrôleur vous confondre. Son travail consiste uniquement à *rendre* le formulaire : l'`form_login` authentificateur gère automatiquement la *soumission du formulaire*. Si l'utilisateur soumet un e-mail ou un mot de passe invalide, cet authentificateur stockera l'erreur et la redirigera vers ce contrôleur, où nous lirons l'erreur (en utilisant `AuthenticationUtils`) afin qu'elle puisse être affichée à l'utilisateur.

Enfin, créez ou mettez à jour le modèle :

```
{# templates/login/index.html.twig #}
{% extends 'base.html.twig' %}

{# ... #}

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

    <form action="{{ path('login') }}" method="post">
        <label for="username">Email:</label>
        <input type="text" id="username" name="_username" value="{{ last_username }}" />

        <label for="password">Password:</label>
        <input type="password" id="password" name="_password" />

        {# If you want to control the URL the user is redirected to on success
        <input type="hidden" name="_target_path" value="/account" /> #}

        <button type="submit">login</button>
    </form>
{% endblock %}
```



La `error` variable transmise au modèle est une instance de [AuthenticationException](#). Il peut contenir des informations sensibles sur l'échec de l'authentification. *Ne jamais* utiliser `error.message` : utilisez la `messageKey` propriété à la place, comme indiqué dans l'exemple. Ce message peut toujours être affiché en toute sécurité.

Le formulaire peut ressembler à n'importe quoi, mais il suit généralement certaines conventions :

- L' `<form>` élément envoie une POST requête à la login route, puisque c'est ce que vous avez configuré `check_path` sous la `form_login` clé dans `security.yaml`;
- Le champ nom d'utilisateur (ou quel que soit "l'identifiant" de votre utilisateur, comme un e-mail) a le nom `_username` et le champ mot de passe a le nom `_password`.

En fait, tout cela peut être configuré sous la `form_login` . Voir [Référence de configuration de la sécurité \(SecurityBundle\)](#) pour plus de détails.

Ce formulaire de connexion n'est actuellement pas protégé contre les attaques CSRF. Lisez [Sécurité](#) pour savoir comment protéger votre formulaire de connexion.

Et c'est tout! Lorsque vous soumettez le formulaire, le système de sécurité lit automatiquement le paramètre `_username` et `_password` POST, charge l'utilisateur via le fournisseur d'utilisateurs, vérifie les informations d'identification de l'utilisateur et authentifie l'utilisateur ou le renvoie au formulaire de connexion où l'erreur peut être affichée.

Pour revoir l'ensemble du processus :

1. L'utilisateur tente d'accéder à une ressource protégée (par exemple `/admin`);
2. Le pare-feu initie le processus d'authentification en redirigeant l'utilisateur vers le formulaire de connexion (`/login`) ;
3. La `/login` page affiche le formulaire de connexion via la route et le contrôleur créés dans cet exemple ;
4. L'utilisateur soumet le formulaire de connexion à `/login`;
5. Le système de sécurité (c'est-à-dire l' `form_login` authentificateur) intercepte la demande, vérifie les informations d'identification soumises par l'utilisateur, authentifie l'utilisateur si elles sont correctes et renvoie l'utilisateur au formulaire de connexion si elles ne le sont pas.

- **Protection CSRF dans les formulaires de connexion**

Les attaques [CSRF de connexion](#) peuvent être évitées en utilisant la même technique consistant à ajouter des jetons CSRF cachés dans les formulaires de connexion. Le composant Sécurité fournit déjà une protection CSRF, mais vous devez configurer certaines options avant de l'utiliser.

Tout d'abord, vous devez activer CSRF sur le formulaire de connexion :

```
YAML  XML  PHP

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

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

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

```
{# templates/security/login.html.twig #}

{# ... #}

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

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

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

Après cela, vous avez protégé votre formulaire de connexion contre les attaques CSRF.



Vous pouvez modifier le nom du champ en définissant `csrf_parameter` et modifier l'ID de jeton en définissant `csrf_token_id` dans votre configuration. Voir [Référence de configuration de la sécurité \(SecurityBundle\)](#) pour plus de détails.

● Connexion JSON

Certaines applications fournissent une API sécurisée à l'aide de jetons. Ces applications peuvent utiliser un point de terminaison qui fournit ces jetons en fonction d'un nom d'utilisateur (ou d'une adresse e-mail) et d'un mot de passe. L'authentificateur de connexion JSON vous aide à créer cette fonctionnalité.

Activez l'authenticateur à l'aide du `json_login` paramètre :

```
YAML XML PHP
1 # config/packages/security.yaml
2 security:
3     # ...
4
5     firewalls:
6         main:
7             # ...
8             json_login:
9                 # api_login is a route we will create below
10                check_path: api_login
```



Le `check_path` prend en charge les URL et les noms de route (mais ne peut pas avoir de caractères génériques obligatoires - par exemple, `/login/{foo}` où `foo` n'a pas de valeur par défaut).

L'authenticateur s'exécute lorsqu'un client demande le `check_path`. Tout d'abord, créez un contrôleur pour ce chemin :

```
$ php bin/console make:controller --no-template ApiLogin

created: src/Controller/ApiLoginController.php
```

```
1 // src/Controller/ApiLoginController.php
2 namespace App\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5 use Symfony\Component\HttpFoundation\Response;
6 use Symfony\Component\Routing\Annotation\Route;
7
8 class ApiLoginController extends AbstractController
9 {
10     #[Route('/api/login', name: 'api_login')]
11     public function index(): Response
12     {
13         return $this->json([
14             'message' => 'Welcome to your new controller!',
15             'path' => 'src/Controller/ApiLoginController.php',
16         ]);
17     }
18 }
```


Ce contrôleur de connexion sera appelé une fois que l'authentificateur aura réussi à authentifier l'utilisateur. Vous pouvez obtenir l'utilisateur authentifié, générer un jeton (ou tout ce que vous devez renvoyer) et renvoyer la réponse JSON :

```
// ...
+ use App\Entity\User;
+ use Symfony\Component\Security\Http\Attribute\CurrentUser;

class ApiLoginController extends AbstractController
{
    #[Route('/api/login', name: 'api_login')]
- public function index(): Response
+ public function index(#[CurrentUser] ?User $user): Response
    {
+         if (null === $user) {
+             return $this->json([
+                 'message' => 'missing credentials',
+             ], Response::HTTP_UNAUTHORIZED);
+         }
+
+         $token = ...; // somehow create an API token for $user
+
+         return $this->json([
-             'message' => 'Welcome to your new controller!',
-             'path' => 'src/Controller/ApiLoginController.php',
+             'user' => $user->getUserIdentifier(),
+             'token' => $token,
+         ]);
    }
}
```

Plus d'info : [Connexion Json](#)

- **HTTP de base**

L'[authentification HTTP Basic](#) est une structure d'authentification HTTP standardisée. Il demande des informations d'identification (nom d'utilisateur et mot de passe) à l'aide d'une boîte de dialogue dans le navigateur et l'authentificateur HTTP de base de Symfony vérifiera ces informations d'identification.

Ajoutez la `http_basic` clé à votre pare-feu pour activer l'authentification HTTP Basic :

[En voir plus sur Symfony](#)

- **Lien de connexion**

Les liens de connexion sont un mécanisme d'authentification sans mot de passe. L'utilisateur recevra un lien de courte durée (par exemple par e-mail) qui l'authentifiera sur le site Web. Vous pouvez tout savoir sur cet authenticateur dans [Comment utiliser l'authentification par lien de connexion sans mot de passe](#) .

- **Déconnexion**

Pour activer la déconnexion, activez le `logout` paramètre de configuration sous votre pare-feu :

YAML XML PHP

```
1 # config/packages/security.yaml
2 security:
3     # ...
4
5     firewalls:
6         main:
7             # ...
8             logout:
9                 path: app_logout
10
11             # where to redirect after logout
12             # target: app_any_route
```

Ensuite, vous devez créer une route pour cette URL (mais pas un contrôleur) :

Remarques Les attributs YAML XML

```
1 # config/routes.yaml
2 app_logout:
3     path: /logout
4     methods: GET
```

C'est ça! En envoyant un utilisateur vers la `app_logout` route (c'est-à-dire vers `/logout`), Symfony désauthentifiera l'utilisateur actuel et le redirigera.

Personnalisation de la déconnexion

Dans certains cas, vous devez exécuter une logique supplémentaire lors de la déconnexion (par exemple, invalider certains jetons) ou souhaitez personnaliser ce qui se passe après une déconnexion. Lors de la déconnexion, un [LogoutEvent](#) est envoyé. Enregistrez un [écouteur d'événement](#) ou un [abonné](#) pour exécuter une logique personnalisée. Les informations suivantes sont disponibles dans la classe d'événement :

`getToken()`

Renvoie le jeton de sécurité de la session sur le point d'être déconnectée.

`getRequest()`

Renvoie la requête en cours.

`getResponse()`

Renvoie une réponse, si elle est déjà définie par un écouteur personnalisé. Utilisez `setResponse()` pour configurer une réponse de déconnexion personnalisée.

2.3 Où sont stockés les utilisateurs. :

Après authentification, l' `User` objet de l'utilisateur courant est accessible via le `getUser()` raccourci dans la [base controller](#) :

```
1 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
2
3 class ProfileController extends AbstractController
4 {
5     public function index(): Response
6     {
7         // usually you'll want to make sure the user is authenticated first,
8         // see "Authorization" below
9         $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');
10
11         // returns your User object, or null if the user is not authenticated
12         // use inline documentation to tell your editor your exact User class
13         /** @var \App\Entity\User $user */
14         $user = $this->getUser();
15
16         // Call whatever methods you've added to your User class
17         // For example, if you added a getFirstName() method, you can use that.
18         return new Response('Well hi there '.$user->getFirstName());
19     }
20 }
```

- **Récupérer l'utilisateur d'un service**

```
// src/Service/ExampleService.php
// ...

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();

        // ...
    }
}
```

- **Récupérer l'utilisateur dans un modèle**

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

- **Contrôle d'accès (autorisation)**

Les utilisateurs peuvent maintenant se connecter à votre application en utilisant votre formulaire de connexion. Génial! Maintenant, vous devez apprendre à refuser l'accès et à travailler avec l'objet Utilisateur. C'est ce qu'on appelle l' autorisation , et son travail consiste à décider si un utilisateur peut accéder à une ressource (une URL, un objet de modèle, un appel de méthode, ...).

Le processus d'autorisation comporte deux volets différents :

1. L'utilisateur reçoit un rôle spécifique lors de la connexion (par exemple ROLE_ADMIN).
2. Vous ajoutez du code pour qu'une ressource (par exemple, une URL, un contrôleur) nécessite un "attribut" spécifique (par exemple, un rôle comme ROLE_ADMIN) afin d'être accessible.

- **Les rôles**

Lorsqu'un utilisateur se connecte, Symfony appelle la getRoles() méthode sur votre User objet pour déterminer les rôles de cet utilisateur. Dans la User classe générée précédemment, les rôles sont un tableau stocké dans la base de données et chaque utilisateur se voit *toujours* attribuer au moins un rôle : ROLE_USER :

```
// 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);
    }
}
```

C'est une bonne valeur par défaut, mais vous pouvez faire ce *que* vous voulez pour déterminer les rôles qu'un utilisateur devrait avoir. Voici quelques lignes directrices :

- Chaque rôle doit commencer par ROLE_ (sinon, les choses ne fonctionneront pas comme prévu)
- En dehors de la règle ci-dessus, un rôle n'est qu'une chaîne et vous pouvez inventer ce dont vous avez besoin (par exemple ROLE_PRODUCT_ADMIN).

Vous utiliserez ensuite ces rôles pour accorder l'accès à des sections spécifiques de votre site.

• Rôles hiérarchiques

Au lieu d'attribuer plusieurs rôles à chaque utilisateur, vous pouvez définir des règles d'héritage des rôles en créant une hiérarchie des rôles :

```
YAML XML PHP
# config/packages/security.yaml
security:
    # ...

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

Les utilisateurs avec le ROLE_ADMIN rôle auront également le ROLE_USER rôle. Les utilisateurs avec ROLE_SUPER_ADMIN, auront automatiquement ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH et ROLE_USER (hérité de ROLE_ADMIN).

Pour que la hiérarchie des rôles fonctionne, ne l'utilisez pas `$user->getRoles()` manuellement. Par exemple, dans un contrôleur prolongeant 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');`

- **Sécurisation des modèles d'URL (access_control)**

Le moyen le plus simple de sécuriser une partie de votre application consiste à sécuriser un modèle d'URL complet au format security.yaml. Par exemple, pour exiger ROLE_ADMIN pour toutes les URL commençant par /admin, vous pouvez :

```
# 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 }
```

Vous pouvez définir autant de modèles d'URL que vous le souhaitez - chacun est une expression régulière. MAIS , un seul sera trouvé par requête : Symfony démarre en haut de la liste et s'arrête lorsqu'il trouve la première correspondance :

```
# 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 }
```

Préfixer le chemin avec ^ signifie que seules les URL *commençant* par le modèle sont mises en correspondance. Par exemple, un chemin de /admin(sans ^) correspondrait /admin/foo mais correspondrait également à des URL telles que /foo/admin.

Chacun `access_control` 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 https version d'un modèle d'URL.

Voir [Comment fonctionne la sécurité access_control ?](#).

- **Sécurisation des contrôleurs et autre code**

Vous pouvez refuser l'accès depuis l'intérieur d'un contrôleur :

```
// 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 ha
}
```

C'est ça! Si l'accès n'est pas accordé, une [AccessDeniedException](#) spéciale est lancée et plus aucun code de votre contrôleur n'est appelé. Ensuite, l'une des deux choses suivantes se produira :

1. Si l'utilisateur n'est pas encore connecté, il lui sera demandé de se connecter (par exemple redirigé vers la page de connexion).
2. Si l'utilisateur est connecté, mais n'a *pas* le `ROLE_ADMIN` rôle, il verra la page 403 accès refusé (que vous pouvez [personnaliser](#)).

Grâce au `SensioFrameworkExtraBundle`, vous pouvez également sécuriser votre manette grâce aux 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
    {
        // ...
    }
}
```

Pour plus d'informations, consultez la [documentation de FrameworkExtraBundle](#) .

- **Contrôle d'accès dans les modèles**

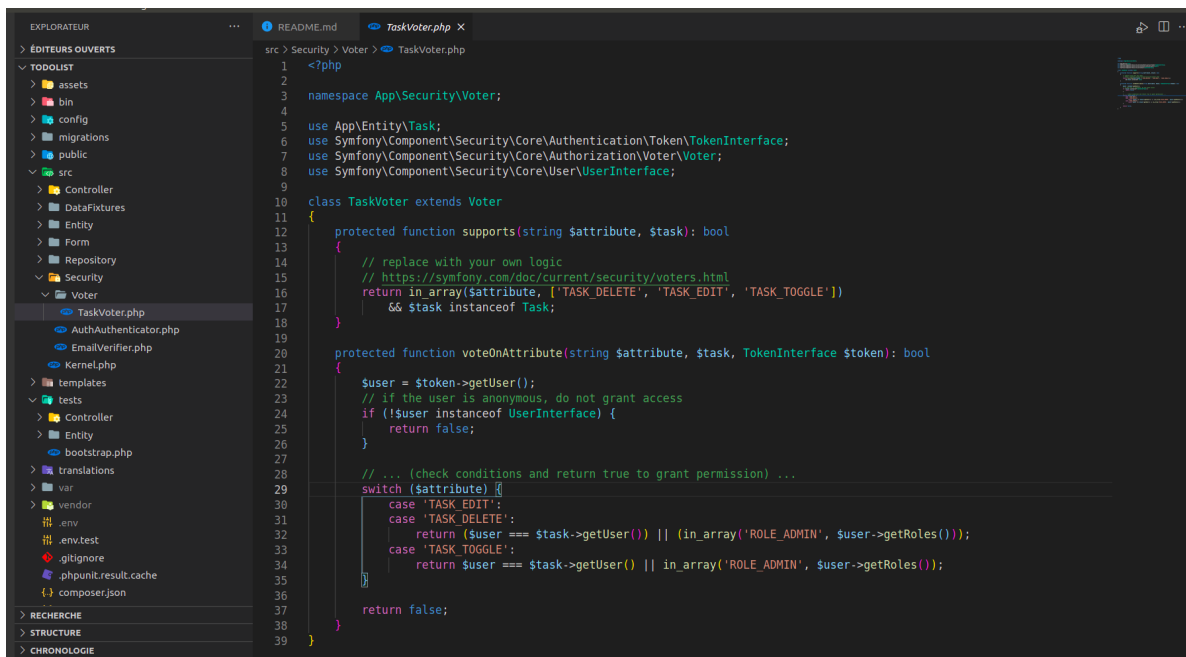
Si vous souhaitez vérifier si l'utilisateur actuel a un certain rôle, vous pouvez utiliser la `is_granted()` fonction d'assistance intégrée dans n'importe quel modèle Twig :

```
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

 Copy

- **Accorder aux utilisateurs anonymes l'accès à un Voter**

Les voteurs sont un moyen puissant de gérer les autorisations par Symfony. Cela permet de centraliser la logique afin de pouvoir les réutiliser à plusieurs endroits.



```
1 <?php
2
3 namespace App\Security\Voter;
4
5 use App\Entity\Task;
6 use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
7 use Symfony\Component\Security\Core\Authorization\Voter\Voter;
8 use Symfony\Component\Security\Core\User\UserInterface;
9
10 class TaskVoter extends Voter
11 {
12     protected function supports(string $attribute, $task): bool
13     {
14         // replace with your own logic
15         // https://symfony.com/doc/current/security/voters.html
16         return in_array($attribute, ['TASK_DELETE', 'TASK_EDIT', 'TASK_TOGGLE'])
17             && $task instanceof Task;
18     }
19
20     protected function voteOnAttribute(string $attribute, $task, TokenInterface $token): bool
21     {
22         $user = $token->getUser();
23         // if the user is anonymous, do not grant access
24         if (!$user instanceof UserInterface) {
25             return false;
26         }
27
28         // ... (check conditions and return true to grant permission) ...
29         switch ($attribute) {
30             case 'TASK_EDIT':
31             case 'TASK_DELETE':
32                 return ($user === $task->getUser()) || (in_array('ROLE_ADMIN', $user->getRoles()));
33             case 'TASK_TOGGLE':
34                 return $user === $task->getUser() || in_array('ROLE_ADMIN', $user->getRoles());
35             default:
36                 return false;
37         }
38     }
39 }
```

La commande à utiliser :

php/bin console make:voter TaskVoter.

voir ressource : <https://symfony.com/doc/current/security/voters.html>

Dans notre cas, nous avons besoin de vérifier que l'autorisation d'éditer ou de supprimer une tâche ne peut être faite que si le user l'a créé ou s'il a le rôle admin.

Aussi pour autoriser une tâche à être marquée comme faite dans le même cas de figure que précédemment. C'est à dire si le user est bien l'auteur de la tâche ou s'il a bien le rôle d'admin.

2.4 Apprendre encore plus

(lien cliquable)

Authentification (identification/connexion de l'utilisateur)

- Hachage et vérification du mot de passe
- Authentification sur un serveur LDAP
- Comment ajouter la fonctionnalité de connexion "Se souvenir de moi"
- Comment usurper l'identité d'un utilisateur
- Comment créer et activer des vérificateurs d'utilisateurs personnalisés
- Comment limiter les pare-feu à une requête
- Comment implémenter la protection CSRF
- Personnalisation des réponses de l'authentificateur de connexion par formulaire
- Comment écrire un authentificateur personnalisé
- Le point d'entrée : aider les utilisateurs à démarrer l'authentification

Autorisation (refus d'accès)

- Comment utiliser les électeurs pour vérifier les autorisations des utilisateurs
- Comment fonctionne la sécurité access_control ?
- Comment personnaliser les réponses d'accès refusé
- Comment forcer HTTPS ou HTTP pour différentes URL

3. Apporter des modifications au projet (Contribution) :

Utiliser le ReadMe.md

Soit :

- **Source**

1. Clone the GitHub repository :

```
git clone https://github.com/marue59/todoList.git
```

- **Installation**

Configure your environment variables :

Docker containers, at the root of the project :

```
cp .env.example .env
```

SMTP server and database :

```
cp app/html/vendor/.env app/html/vendor/.env.local app/html/vendor/.env.test
```

Create the docker network

```
docker network create project8
```

Launch the containers

```
docker-compose up -d
```

Enter the PHP container to launch the commands for the database

```
docker exec -ti [nom du container php] bash
```

Install php dependencies with composer

```
composer install
```

Install the database

```
php bin/console doctrine:migrations:migrate
```

Install the fixture (dummy data demo)

```
php bin/console doctrine:fixtures:load
```

Leave the container

```
exit
```

- **Contribuer au projet et processus de qualité attendu :**

Dans le but de proposer un code évolutif et compréhensible de tous, il vous sera demandé de respecter les standards de la programmation (soit les bonnes pratiques) :

- Le code doit respecter **les normes PSR (PSR-1, PSR-2, PSR-4 et PSR-12)**
-> ressources : <https://www.php-fig.org/psr/>
- Le code doit être bien **indenté et commenté** afin d'optimiser la compréhension de chacun.
-> utilisation de : php-cs-fixer
- La qualité de votre code doit obtenir au minimum un B.
-> codacy (ou Code Climate ou Symfony Insight)
- La performance soumettre votre code à BlackFire contrôleur de performance.

Il vous sera demandé pour chaque nouvelle fonctionnalité d'y apporter **les tests** correspondants afin de couvrir l'ensemble de votre code.

(tests unitaires et tests fonctionnels).

Vérifier que l'ensemble des test soient au vert

-> php bin/phpunit

Après chaque fonctionnalité créée et testée, vous devrez **versionner** votre code sur Git en commitant sur votre branche :

- git add .
- git commit -m "*description de la fonctionnalité*"
- git push origin *nomdevotrebranche*.

Réaliser une Pull Request :

- sur le repo du projet, cliquer sur New Pull Request.
- comparer : *main* et *nomdevotrebranche*.
- cliquer sur Merger la PR