

Sommaire :

<i>Les notions d'authentification et d'autorisation 2</i>
<i>L'authentification 2</i>
<i>L'autorisation 3</i>
<i>Processus général 3</i>
<i>Mettre en place le composant de sécurité de Symfony : 3</i>
<i>Création de l'authentification 4</i>
<i>Création du formulaire d'inscription 4</i>
<i>Le security.yaml 5</i>
Firewalls : 5
Encoders : 5
Providers : 6
Access control : 6
<i>L'inscription 7</i>
<i>Le UserType.php 7</i>
<i>Formulaire d'inscription : 7</i>
<i>Le RegistrationController.php 8</i>
L'entité USER : 8
<i>Créez un formulaire de connexion 8</i>
<i>Créez un contrôleur(securityController) 9</i>
<i>Le userFormAuthenticator : 9</i>

L'authentification



Les notions d'authentification et d'autorisation

L'authentification

L'authentification est le processus qui va définir qui vous êtes, en tant que visiteur. L'enjeu est vraiment très simple : soit vous ne vous êtes pas identifié sur le site et vous êtes un anonyme, soit vous vous êtes identifié (via le formulaire d'identification ou via un cookie « Se souvenir de moi ») et vous êtes un membre du site. C'est ce que la procédure d'authentification va déterminer. Ce qui gère l'authentification dans Symfony s'appelle un *firewall*.

Ainsi vous pourrez sécuriser des parties de votre site internet juste en forçant le visiteur à être un membre authentifié. Si le visiteur l'est, le firewall va le laisser passer, sinon il le redirigera sur la page d'identification. Cela se fera donc dans les paramètres du firewall.

L'autorisation

L'autorisation est le processus qui va déterminer si vous avez le droit d'accéder à la ressource (la page) demandée. Il agit donc après le firewall. Ce qui gère l'autorisation dans Symfony s'appelle l'*Access control*.

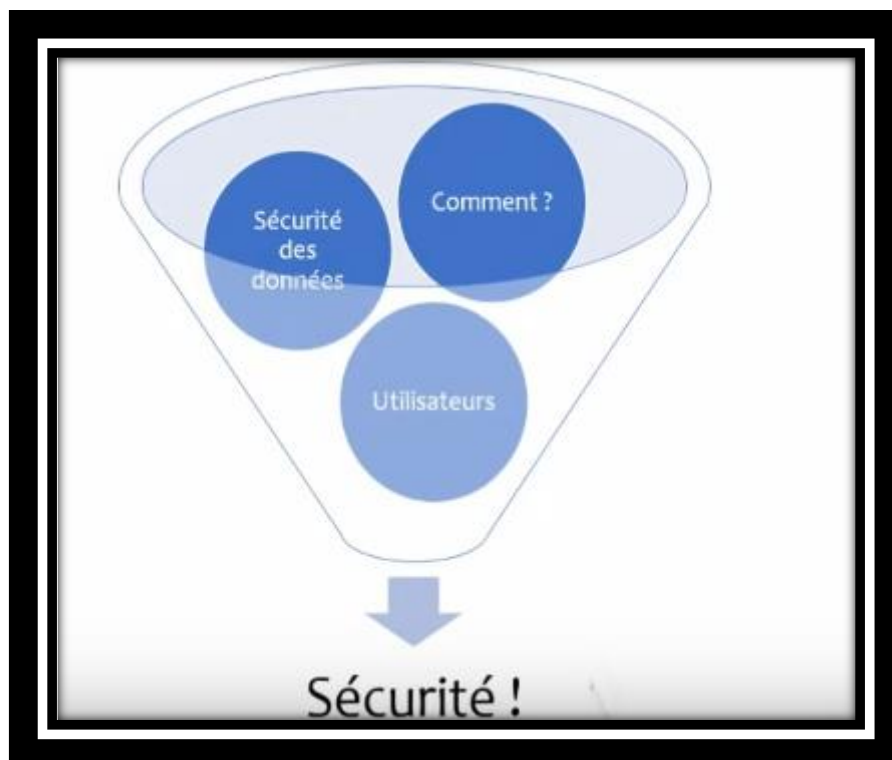
Par exemple, un membre identifié aura accès à la liste des tasks, mais ne peut pas accéder à la liste des utilisateurs. Seuls les membres disposant des droits d'administrateur le peuvent, ce que l'Access control va vérifier.

Processus général

Lorsqu'un utilisateur tente d'accéder à une ressource protégée, le processus est finalement toujours le même, le voici :


1. Un utilisateur veut accéder à une ressource protégée ;
2. Le firewall redirige l'utilisateur au formulaire de connexion ;
3. L'utilisateur soumet ses informations d'identification (par exemple login et mot de passe) ;
4. Le firewall authentifie l'utilisateur ;
5. L'utilisateur authentifié renvoie la requête initiale ;
6. Le contrôle d'accès vérifie les droits de l'utilisateur, et autorise ou non l'accès à la ressource protégée

Mettre en place le composant de sécurité de Symfony :



Création de l'authentification

Pour créer l'authentification, nous utiliserons la commande



```
symfony console make:auth
```

Cette commande va lancer un assistant qui va vous demander de renseigner les informations suivantes :


- Le type d'authentification (avec ou sans formulaire de connexion)
- Le nom de la classe contenant l'authentification (UserFormAuthenticator dans mon exemple)
- Le nom du contrôleur qui contiendra les routes de connexion et déconnexion (SecurityController)
- La création ou non d'une route de déconnexion (/logout)

Après l'exécution de la commande, les fichiers suivants auront été créés ou modifiés

- config/packages/security.yaml
- src/Controller/SecurityController.php
- src/Security/UserFormAuthenticator.php
- templates/security/login.html.twig

Création du formulaire d'inscription

Pour créer le formulaire d'inscription sur le site, nous utiliserons la commande



```
symfony console make:registration-form
```

Cette commande va lancer un assistant qui va vous demander de renseigner les informations suivantes :

- Veut-on ajouter une annotation “@UniqueEntity” dans notre classe Users pour les rendre uniques
- Veut-on envoyer un email aux utilisateurs pour activer leur compte
- Veut-on connecter automatiquement les utilisateurs après leur inscription

Après l'exécution de la commande, les fichiers suivants auront été créés ou modifiés

- config/packages/security.yaml
- src/Controller/RegistrationController.php

- src/Entity/User.php
- src/Form/UserType.php

Nous pourrions ensuite modifier les fichiers en fonction du contexte de notre site.

Le security.yaml

Firewalls :

Permet de définir comment protéger notre application, quelle sont les parties de l'application qu'on veut protéger et comment les protéger.

On peut utiliser plusieurs méthodes tel que le formulaire de login...

```
provider: app_user_provider
custom_authenticator: App\Security\LoginFormAuthenticator
logout:
  path: app_user_logout
```

Encoders :

On peut gérer la sécurité des données notamment les mots de passe Utilisant un encoder pour les hacher.

```
# https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
password_hashers:
  Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: "auto"
  App\Entity\User:
    algorithm: auto
```

Grâce à l'interface UserPasswordHasherInterface on peut encoder nos mot de passe

```
/**
 * @Route(path="/register", name="user_register")
 */
public function createAction(Request $request, UserPasswordHasherInterface $encoder)
{
    $user = new User();
    $form = $this->createForm(UserType::class, $user);
    $form->handleRequest($request);
    if ($form->isSubmitted()) {
        $password = $encoder->hashPassword($user, $user->getPassword());
        $user->setPassword($password);
        $this->manager->persist($user);
        $this->manager->flush();
        $this->addFlash('success', "Superbe ! votre inscription s'est déroulée avec succès.");
        return $this->redirectToRoute('login_check');
    }
    return $this->render('register/register.html.twig', ['form' => $form->createView()]);
}
```

Providers :

Permet de définir où se trouvent les utilisateurs :

- Une base de données
- Un annuaire d'entreprise
- Un fichier texte....

```
# https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
providers:
    # used to reload user from session & other features (e.g. switch_user)
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email
```

Access control :

Le contrôle d'accès vérifie que l'utilisateur a le(s) rôle(s) requis pour accéder au contenu demandé. Les contrôles d'accès peuvent être utilisés :

- À partir du fichier de configuration, comme c'est le cas ici. Pour cela, il faut appliquer règles sur des URL. Par exemple, on peut sécuriser toutes les URL commençant par /users

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

- Dans les contrôleurs directement

```
public function edit(Article $article)
{
    if ($this->getUser() !== $article->getAuthor() || !$this->isGranted('ROLE_ADMIN')) {
        throw $this->createAccessDeniedException();
    }
}
```

L'inscription

Le UserType.php

```
class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('username', TextType::class, ['label' => "Nom d'utilisateur"])
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
                'invalid_message' => 'Les deux mots de passe doivent correspondre.',
                'required' => true,
                'first_options' => ['label' => 'Mot de passe'],
                'second_options' => ['label' => 'Tapez le mot de passe à nouveau'],
            ])
            ->add('email', EmailType::class, ['label' => 'Adresse email'])
            ->add(
                'roles',
                ChoiceType::class,
                [
                    'label' => 'Roles',
                    'choices' => ['ROLE_ADMIN' => 'ROLE_ADMIN', 'ROLE_USER' => 'ROLE_USER'],
                    'required' => true,
                    'multiple' => true,
                    'expanded' => true,
                    'attr' => [
                        'class' => 'form-control'
                    ]
                ]
            );
    }
}
```

Formulaire d'inscription :

```
{% extends 'base.html.twig' %}

{% block header_title %}<h1>Créer un utilisateur</h1>{% endblock %}
{% block header_img %}{% endblock %}

{% block body %}
    <div class="row">
        {{ form_start(form, {'action' : path('user_register')}) }}
        {{ include('user/_form.html.twig') }}
        <br><br><br>
        <button type="submit" class="btn btn-success pull-right">Ajouter</button>
        {{ form_end(form, {'render_rest': false}) }}
    </div>
{% endblock %}
```

Le RegistrationController.php

```
/**
 * @Route(path="/register", name="user_register")
 */
public function createAction(Request $request, UserPasswordEncoderInterface $encoder)
{
    $user = new User();
    $form = $this->createForm(UserType::class, $user);
    $form->handleRequest($request);
    if ($form->isSubmitted()) {
        $password = $encoder->hashPassword($user, $user->getPassword());
        $user->setPassword($password);
        $this->manager->persist($user);
        $this->manager->flush();
        $this->addFlash('success', "Superbe ! votre inscription s'est déroulée avec succès.");
        return $this->redirectToRoute('login_check');
    }
    return $this->render('register/register.html.twig', ['form' => $form->createView()]);
}
```

L'entité USER :

On va créer une entité utilisateur qui va stocker nos utilisateurs en base de données.

Une seule contrainte pour la création d'un utilisateur est d'implémenter l'interface UserInterface du composant Security.

L'implémentation requière l'ajout des méthode suivantes :

- getRoles : Retourne les rôles de l'utilisateur.
- getPassword : Récupération du mot de passe encodé.
- getSalt : Retourne le sel pour l'encodage du mot de passe.
- getUsername : Récupère le nom de l'utilisateur.
- eraseCredentials : Efface les données sensibles comme le mot de passe.

Créez un formulaire de connexion

Pour pouvoir authentifier nos utilisateurs nous avons besoin de mettre en place un formulaire de connexion qui demande à l'utilisateur son adresse mail et un mot de passe.

```
{% extends 'base.html.twig' %}

{% block body %}
    {% if error %}
        <div class="alert alert-danger" role="alert">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
    {% endif %}

    <form action="{{ path('login_check') }}" method="post">
        <label for="username">Nom d'utilisateur :</label>
        <input type="email" value="{{ last_username }}" name="email" id="inputEmail" autocomplete="email" required autofocus>

        <label for="password">Mot de passe :</label>
        <input type="password" name="password" id="inputPassword" autocomplete="current-password" required>

        <button class="btn btn-success" type="submit">Se connecter</button>
        <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">
    </form>
{% endblock %}
```


Créez un contrôleur(securityController)

Contient une

Route : /login_check

Récupère les erreurs d'authentification

Récupère le dernier utilisateur connecté et l'injecter dans le formulaire

Au niveau du logout on a rien du tout .

```
class SecurityController extends AbstractController
{
    /**
     * @Route(path="/login_check", name="login_check", methods={"POST", "GET"})
     */
    public function loginAction(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('security/login.html.twig', ['last_username' => $lastUsername, 'error' => $error]);
    }

    /**
     * @Route(path="/logout", name="logout")
     */
    public function logout(): void
    {
        throw new \LogicException('This method can be blank - it will be intercepted by the logout key on your firewall.');
```

Le userFormAuthenticator :

Toute la logique d'authentification se situe au niveau de l'authenticator

Qui aura comme rôle d'initialiser l'authentification c'est à dire il vérifie si L'authenticator vérifie si la requête est une requête d'authentification ensuite il va commencer le début du processus

Créez de la méthode d'authentification

Pour créer une méthode d'authentification, nous allons utiliser une extension du composant **Security** appelée [Guard](#). Changement majeur par rapport aux anciennes versions de Symfony, il est maintenant très facile de créer sa méthode d'authentification ou encore "authenticator".

Pour cela, nous allons créer une classe qui étend `AbstractFormLoginAuthenticator` du composant **Guard**. Il faudra implémenter et compléter quelques fonctions :

- **supports()** : la fonction définit dans quelles conditions la classe sera appelée.
- **getCredentials()** : retourne les éléments d'information d'authentification.
- **getUser()** : retourne un utilisateur au sens Symfony (instance de **UserInterface** du composant Security).

- **checkCredentials()** : contrôle à la connexion que les informations d'authentification sont valides.
- **onAuthenticationSuccess()** : décide que faire dans le cas où l'utilisateur est bien authentifié, généralement une redirection vers une URL donnée.
- **getLoginUrl()** : définit l'URL du formulaire de connexion, dans notre cas `security_login`.

```
class UserFormAuthenticator extends AbstractLoginFormAuthenticator
{
    use TargetPathTrait;

    public const LOGIN_ROUTE = 'login_check';

    private UrlGeneratorInterface $urlGenerator;

    public function __construct(UrlGeneratorInterface $urlGenerator)
    {
        $this->urlGenerator = $urlGenerator;
    }

    public function authenticate(Request $request): Passport
    {
        $email = $request->request->get('email', '');

        $request->getSession()->set(Security::LAST_USERNAME, $email);

        return new Passport(
            new UserBadge($email),
            new PasswordCredentials($request->request->get('password', '')),
            [
                new CsrfTokenBadge('authenticate', $request->request->get('_csrf_token')),
            ]
        );
    }

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
    {
        if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
            return new RedirectResponse($targetPath);
        }

        // For example:
        return new RedirectResponse($this->urlGenerator->generate('homepage'));
    }

    protected function getLoginUrl(Request $request): string
    {
        return $this->urlGenerator->generate(self::LOGIN_ROUTE);
    }
}
```

La méthode `Supports` de l'abstract `LoginFormAuthenticator` : vérifier si la méthode est POST et l'url correspond à l'url de login.

En cas d'échec de connexion on redirige vers la page de connexion

```
protected function getLoginUrl(Request $request): string
{
    return $this->urlGenerator->generate(self::LOGIN_ROUTE);
}
```

En cas de succès on redirige vers la page d'accueil :

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
        return new RedirectResponse($targetPath);
    }

    // For example:
    return new RedirectResponse($this->urlGenerator->generate('homepage'));
}
```

La méthode authenticate renvoie un objet de type passport

L'authenticator récupère l'email depuis la requête

Il le sauvegarde dans la session s'il y a une erreur d'authentification ça permet de remettre l'email dans le champ et ensuite il génère un passport qui va être vérifié par différentes méthodes

Le badge va nous donner des informations sur l'utilisateur notamment son identifiant dans notre système on a choisi l'email, on aura également un autre badge qui contiendra le mot de passe et

Un badge supplémentaire pour vérifier le token CSRF

Les listeners vont venir écouter le processus d'authentification pour récupérer le passport et vérifier les badges

```
public function authenticate(Request $request): Passport
{
    $email = $request->request->get('email', '');

    $request->getSession()->set(Security::LAST_USERNAME, $email);

    return new Passport(
        new UserBadge($email),
        new PasswordCredentials($request->request->get('password', '')),
        [
            new CsrfTokenBadge('authenticate', $request->request->get('_csrf_token')),
        ]
    );
}
```