Dépôt Github : https://github.com/magali-thuaire/oc-todolist

DOCUMENTATION TECHNIQUE
Guide d'authentification

Version : 1
Date de la dernière mise à jour : 10/06/2022

SOMMAIRE

1. CONTEXTE

1.1 Présentation

La startup **ToDo & Co** a développé une application permettant de gérer les tâches quotidiennes (ToDoList).

L'application a initialement été développée avec le framework **Symfony version 3.1** et se trouve actuellement dans sa **version 6.1**.

Cette documentation a pour but d'expliquer comment l'implémentation de la sécurité est réalisée via l'**authentification** et l'**autorisation**.

1.2 Sécurité

Le contrôle de la sécurité de l'application est géré par le **bundle Security** de Symfony qui fournit toutes les fonctionnalités d'**authentification** et d'**autorisation** nécessaires pour sécuriser l'application.

Lien vers la documentation technique du bundle : https://Symfony.com/doc/current/security.html

Symfony distingue :

- L'authentification : qui permet la connexion à l'application
- L'autorisation : qui permet d'accorder les droits d'accès au contenu

2. GESTION DE L'AUTHENTIFICATION

2.1 Fichiers de configuration

Type	Fichier	Description
Configuration	config/packages/security.yaml	Configuration du process d'authentification et d'autorisation
Entité	src/Entity/User.php	Objet qui représente l'utilisateur
Contrôleur	src/Controller/SecurityController.php	Contrôleur de connexion et de déconnexion
Authenticateur	vendor/Symfony/security-http/Authenticator/FormLoginAuthenticator	Classe qui gère la connexion de l'utilisateur
Vue	templates/security/login.html.twig	Template du formulaire de connexion

Depuis la version 5.1, Symfony propose un nouveau système d'authentification et ce système change le comportement interne de la sécurité de Symfony, pour le rendre extensible et plus compréhensible.

Pour utiliser ce système, celui-ci est activé via l'option **enable_authenticator_manager** définie à **true** dans le fichier de configuration.

config/packages/security.yaml

```yaml
security:
    enable_authenticator_manager: true
```

2.2 L'utilisateur

Les autorisations dans Symfony sont toujours liées à un **objet utilisateur**.

Il s'agit d'une classe qui implémente l'interface **UserInterface**.

Dans l'application, il s'agit de l'entité Doctrine **App\Entity\User**.

2.2.1 Chargement de l'utilisateur : le provider

⇨ *Section **provider** du fichier de configuration config/packages/security.yaml*

Le fournisseur d'utilisateur **(provider)** permet de charger/recharger l'utilisateur à partir de la base de données sur la base d'un « identifiant d'utilisateur ».

config/packages/security.yaml

```yaml
providers:
    doctrine:
        entity:
            class: App\Entity\User
            property: username
```

La configuration actuelle permet de récupérer les utilisateurs via Doctrine grâce à l'entité **App\Entity\User** en utilisant la propriété **username** comme « identifiant d'utilisateur ».

Attention, on peut indiquer ici l'entité **App\Entity\User** car celle-ci implémente bien l'interface **UserInterface** !

Le fournisseur d'utilisateur est utilisé à plusieurs endroits au cours du cycle de vie de la sécurité :

- **Charger l'utilisateur en fonction d'un identifiant**

  Lors de la connexion, le fournisseur charge l'utilisateur en fonction de l'identifiant de l'utilisateur

- **Recharger l'utilisateur depuis la session**

  Au début de chaque requête, l'utilisateur est chargé depuis la session. Le fournisseur « actualise » l'utilisateur (la base de données est interrogée à nouveau) pour s'assurer que toutes les informations de l'utilisateur sont à jour (et si nécessaire, l'utilisateur est déconnecté si quelque chose a changé).

2.2.2 Enregistrement de l'utilisateur : hachage du mot de passe

⇨ *Section **password_hasher** du fichier de configuration config/packages/security.yaml*

L'application nécessite que l'utilisateur **se connecte avec un mot de passe**. Le bundle **Security** fournit des fonctionnalités de **hachage** et de **vérification** de mot de passe.

Pour cela, il est nécessaire que :

- **L'entité App\Entity\User** implémente l'interface **PasswordAuthenticatedUserInterface**

- Le **hacheur** soit défini dans le fichier de configuration pour cette entité, ce qui permet de définir l'algorithme utilisé lors du hachage du mot de passe

config/packages/security.yaml

```yaml
password_hashers:
    legacy_bcrypt:
        algorithm: bcrypt
    App\Entity\User:
        algorithm: 'auto'
        migrate_from:
            - legacy_bcrypt
```

Dans l'application l'algorithme utilisé est optimisé par Symfony : **auto**.

Dans la version initiale de l'application (Symfony 3.1) l'algorithme utilisé était le **bcrypt**.

Cela signifie que le mot de passe des précédents utilisateurs doit être *rehaché* à l'aide du nouvel algorithme et stocké. C'est ce que permet l'option **migrate_from**.

Le hachage "auto"

Il sélectionne automatiquement le meilleur « hacheur » disponible (actuellement ***bcrypt***). Si PHP ou Symfony ajoute de nouveaux hacheurs de mot de passe à l'avenir, il pourra sélectionner un autre hacheur.

Pour cette raison, la longueur des mots de passe hachés peut changer à l'avenir, il faut donc s'assurer d'allouer suffisamment d'espace pour qu'ils soient persistants (un varchar(255) est recommandé).

L'option migrate_from

Lorsqu'un meilleur algorithme de hachage devient disponible, il est nécessaire de conserver le(s) hacheur(s) existant(s), le(s) renommer, puis définir le nouveau.

Avec cette option :

- Le mot de passe des nouveaux utilisateurs est haché avec le nouvel algorithme
- Chaque fois qu'un utilisateur se connecte avec un mot de passe toujours stocké à l'aide de l'ancien algorithme, Symfony vérifie le mot de passe avec l'ancien algorithme, puis met à jour le mot de passe à l'aide du nouvel algorithme.

2.3 Le pare-feu

⇨ *Section **firewalls** du fichier de configuration config/packages/security.yaml*

Le pare-feu (**firewall**) est la porte d'entrée du **système d'authentification** : le pare-feu définit quelles parties de l'application sont sécurisées et comment les utilisateurs peuvent s'authentifier.

Le pare-feu permet de déterminer pour un **pattern d'url** (une requête), la méthode d'authentification à utiliser.

config/packages/security.yaml

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        form_login:
            login_path: login
            check_path: login
            enable_csrf: true
        logout:
            path: logout
            target: login
        remember_me:
            secret: '%kernel.secret%'
            always_remember_me: true
            lifetime: 604800
            signature_properties:
                - password
        login_throttling: true
```

Toutes les URL *réelles* sont gérées par le pare-feu **main** : la clé **pattern** définie à **^/** signifie qu'elle correspond à ***toutes les* URL**.

Le pare-feu **dev** est un **faux pare-feu** : il s'assure que l'application ne bloque pas accidentellement les outils de développement de Symfony - qui vivent sous des URL comme /_profiler et /_wdt et l'accès aux assets.

Visiter une URL sous un pare-feu ne nécessite pas nécessairement que l'utilisateur soit authentifié (par exemple, le formulaire de connexion doit être accessible). Cela est possible via les **autorisations**.

2.4 Authentification de l'utilisateur

L'application utilise un **formulaire de connexion** où les utilisateurs s'authentifient à l'aide d'un de leur nom d'utilisateur et d'un mot de passe.

Pour obtenir les paramètres actuels de l'authenticateur, il est possible de saisir la commande suivante dans le terminal :

```
Symfony console debug:config security firewalls.main.form_login
```

```
Current configuration for "security.firewalls.main.form_login"
==============================================================

login_path: login
check_path: login
enable_csrf: true
remember_me: true
use_forward: false
require_previous_session: false
username_parameter: _username
password_parameter: _password
csrf_parameter: _csrf_token
csrf_token_id: authenticate
post_only: true
form_only: false
always_use_default_target_path: false
default_target_path: /
target_path_parameter: _target_path
use_referer: false
failure_path: null
failure_forward: false
failure_path_parameter: _failure_path
```

2.4.1 Formulaire de connexion

⇨ *Section **form_login** du fichier de configuration config/packages/security.yaml*

L'authentificateur de connexion par formulaire est activé via le paramètre **form_login**.

config/packages/security.yaml

```yaml
24          form_login:
25              login_path: login
26              check_path: login
27              enable_csrf: true
```

Le système de sécurité redirige le visiteur non authentifié vers le **login_path** lorsqu'il tente d'accéder à un lieu sécurisé.

Le contrôleur de connexion

Le travail du contrôleur de connexion consiste uniquement à *rendre* le formulaire.

src/Controller/SecurityController.php

```php
#[Route(path: '/login', name: 'login', methods: ['GET', 'POST'])]
public function loginAction(AuthenticationUtils $authenticationUtils): Response
{
    $error = $authenticationUtils->getLastAuthenticationError();
    $lastUsername = $authenticationUtils->getLastUsername();

    return $this->render( view: 'security/login', [
        'last_username' => $lastUsername,
        'error' => $error,
    ]);
}
```

L'authenticateur **form_login** gère automatiquement la *soumission du formulaire.*

Si l'utilisateur soumet un **nom d'utilisateur** ou un **mot de passe** invalide, l'authentificateur stockera l'erreur et redirigera l'utilisateur vers ce contrôleur, qui affichera l'erreur (en utilisant **AuthenticationUtils**).

Ainsi, lorsque le formulaire est posté, le système de sécurité lit automatiquement les paramètres **_username** et **_password**, il charge l'utilisateur via le fournisseur d'utilisateurs, vérifie les informations d'identification et authentifie l'utilisateur ou le renvoie au formulaire de connexion où l'erreur peut être affichée.

Protection CSRF dans le formulaire de connexion

Les attaques CSRF de connexion sont évitées en utilisant la technique consistant à ajouter un **jeton CSRF** caché dans le formulaire.

Le composant **Security** fournit cette protection CSRF via le paramètre **enable_csrf** défini à **true**.

templates/security/login.html.twig

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

    <label for="password">Mot de passe :</label>
    <input type="password" id="password" name="_password" />

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

    <button class="btn btn-success" type="submit">Se connecter</button>
</form>
```

Le formulaire intègre la fonction **csrf_token()** 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**.

2.4.2 Limitation des tentatives de connexion

Symfony fournit une protection de base contre les attaques de connexion par **force brute**.

config/packages/security.yaml

```
37            login_throttling: true
```

Cette fonctionnalité est activée via le paramètre **login_throttling** défini à **true**.

Les paramètres par défaut sont activés : les tentatives de connexion sont limitées à 5 demandes échouées et réinitialisé toutes les minutes :

```
Symfony console debug:config security firewalls.main.login_throttling
```

```
Current configuration for "security.firewalls.main.login_throttling"
=====================================================================

max_attempts: 5
interval: '1 minute'
lock_factory: null
```

2.4.3 Se souvenir de moi

En règle générale, une fois que l'utilisateur est authentifié, ses informations d'identification sont stockées dans la session. Cela signifie qu'à la fin de la session, il est déconnecté et devra saisir à nouveau ses identifiants de connexion la prochaine fois qu'il souhaitera accéder à l'application.

Dans l'application, l'authentification est paramétrée pour que l'utilisateur reste connecté plus longtemps que la durée de la session via l'utilisation d'un cookie avec l'option **remember_me** et le paramètre **always_remember_me**.

config/packages/security.yaml

```
31            remember_me:
32                secret: '%kernel.secret%'
33                always_remember_me: true
34                lifetime: 604800
35                signature_properties:
36                    - password
```

Chaque authentification réussie produira **un cookie « se souvenir de moi »**.

L'option **lifetime** est le nombre de secondes après lesquelles le cookie expirera. Celui-ci définit le temps maximum entre deux visites pour que l'utilisateur reste authentifié. Ici il est paramétré sur 1 semaine.

Avec l'option **signature_properties**, le cookie « se souvenir de moi » ne sera plus considéré comme valide si le mot de passe (password) de l'utilisateur change.

L'option **secret** est utilisée pour signer le cookie se souvenir de moi. Il est courant d'utiliser le paramètre **kernel.secret**, qui est défini à l'aide de la variable d'environnement **APP_SECRET**.

2.5 Authenticateur

Voici le workflow d'authentification :

1. L'utilisateur tente d'accéder à une ressource protégée (par exemple : **/tasks**)
2. Le pare-feu initie le processus d'authentification en redirigeant l'utilisateur vers le formulaire de connexion : **/login**
3. La page **/login** affiche le formulaire de connexion via la route définie dans le **SecurityController** et qui n'est pas protégée
4. L'utilisateur soumet le formulaire de connexion à **/login**
5. Le système de sécurité (c'est-à-dire l'**authentificateur** **form_login**) 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.

2.5.1 Méthode support

Au début de chaque requête, la méthode **support()** de l'authenticateur est appelée.

vendor/Symfony/security-http/Authenticator/FormLoginAuthenticator.php

```
74    public function supports(Request $request): bool
75    {
76        return ($this->options['post_only'] ? $request->isMethod( method: 'POST') : true)
77            && $this->httpUtils->checkRequestPath($request, $this->options['check_path'])
78            && ($this->options['form_only'] ? 'form' === $request->getContentType() : true);
79    }
```

La méthode vérifie que la requête est de type **POST**, qu'elle appelle la route définie par **check_path** **(/login**) et que la requête contient bien un formulaire.

Si toutes ces conditions sont vérifiées, la méthode retourne **true**, la demande est interceptée et le processus d'authentification se poursuit par l'appel de la méthode **authenticate()**.

Dans le cas contraire, le processus d'authentification est stoppé, la requête se poursuit.

2.5.2 Méthode authenticate

vendor/Symfony/security-http/Authenticator/FormLoginAuthenticator.php

```php
public function authenticate(Request $request): Passport
{
    $credentials = $this->getCredentials($request);

    $passport = new Passport(
        new UserBadge($credentials['username'], $this->userProvider->loadUserByIdentifier(...)),
        new PasswordCredentials($credentials['password']),
        [new RememberMeBadge()]
    );
    if ($this->options['enable_csrf']) {
        $passport->addBadge(new CsrfTokenBadge($this->options['csrf_token_id'], $credentials['csrf_token']));
    }

    if ($this->userProvider instanceof PasswordUpgraderInterface) {
        $passport->addBadge(new PasswordUpgradeBadge($credentials['password'], $this->userProvider));
    }

    return $passport;
}
```

La méthode récupère les informations postées dans le formulaire via **getCredentials** et utilise des **badges** qui doivent tous retourner **true** pour que l'authentification réussisse :

- **UserBadge** va récupérer l'utilisateur via le fournisseur d'utilisateur (**provider**)

  Le fournisseur d'utilisateur paramétré précédemment permet de récupérer l'utilisateur en base de données via son nom d'utilisateur (**username**)
- **PasswordCredentials** va vérifier la validité du mot de passe
- **RememberMeBadge** va vérifier la présence d'un cookie remember_me
- **CrsfTokenBadge** va vérifier que le token CRSF est valide

La méthode retourne un objet **Passport** qui contient toutes les informations contenues dans les badges.

Si tous les badges sont valides, l'authentification se poursuit par l'appel de la méthode **createToken()** puis **onAuthentificationSuccess()**.

Dans le cas contraire, c'est la méthode **onAuthenticationFailure()** qui est appelée.

2.5.3 Méthodes createToken et onAuthenticationSuccess

vendor/Symfony/security-http/Authenticator/FormLoginAuthenticator.php

```php
public function createToken(Passport $passport, string $firewallName): TokenInterface
{
    return new UsernamePasswordToken($passport->getUser(), $firewallName, $passport->getUser()->getRoles());
}
```

La méthode **createToken()** permet de créer le **token d'authentification** et de le stocker en session.

vendor/Symfony/security-http/Authenticator/DefaultAuthenticationSuccessHandler.php

```php
public function onAuthenticationSuccess(Request $request, TokenInterface $token): Response
{
    return $this->httpUtils->createRedirectResponse($request, $this->determineTargetUrl($request));
}
```

La méthode **onAuthentificationSuccess()** **redirige l'utilisateur** vers la page demandée intialement ou à défaut vers la page d'accueil.

Il est possible de paramétrer la page et le comportement de défaut avec les paramètres du **form_login** : respectivement **default_target_path** et **always_use_default_target_path.**

2.5.4 Méthode onAuthenticationFailure

vendor/Symfony/security-http/Authenticator/DefaultAuthenticationFailureHandler.php

```php
    $request->getSession()->set(Security::AUTHENTICATION_ERROR, $exception);

    return $this->httpUtils->createRedirectResponse($request, $options['failure_path']);
```

La méthode **onAuthenticationFailure()** stocke les erreurs de connexion en session et redirige l'utilisateur vers la page de login pour les afficher.

2.6 Déconnexion

La fonctionnalité de déconnexion est activée via le paramètre **logout**.

config/packages/security.yaml

```yaml
28            logout:
29                path: logout
30                target: login
```

Lors de la demande de déconnexion, le système redirige l'utilisateur vers le paramètre **target**, c'est-à-dire ici vers la **page de connexion**.

Il est nécessaire que la route **logout** soit définie dans l'application mais l'action ne sera pas exécutée, elle sera interceptée par le système et l'utilisateur sera déconnecté.

src/Controller/SecurityController.php

```php
24      #[Route(path: '/logout', name: 'logout', methods: 'GET')]
25      public function logoutCheck(): void
26      {
27          // controller can be blank: it will never be called!
28          throw new Exception( message: 'Don\'t forget to activate logout in security.yaml');
29      }
```

3. GESTION DES AUTORISATIONS

Après la connexion de l'utilisateur (**authentification**), le système vérifie si celui-ci peut accéder à la ressource demandée : c'est l'**autorisation**.

Le processus d'autorisation comporte deux volets différents :

1. L'utilisateur reçoit un **rôle** spécifique lors de la connexion
2. L'accès à une ressource (par exemple, une URL, un contrôleur) peut nécessiter un rôle spécifique afin d'être accessible

3.1 Les rôles

Lorsqu'un utilisateur se connecte, Symfony appelle la méthode **getRoles()** sur l'entité **App\Entity\User** pour déterminer les rôles de cet utilisateur.

Dans la classe User, les rôles sont un tableau stocké dans la base de données et chaque utilisateur se voit *toujours* attribuer au moins un rôle : le ROLE_USER, auquel seront ajoutés les rôles stockés.

src/Entity/User.php

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

    return array_unique($roles);
}
```

Deux rôles sont paramétrés sur l'application :

- **ROLE_USER** (utilisateur) : pour l'accès à la gestion des tâches
- **ROLE_ADMIN** (administrateur) : pour l'accès à la gestion des utilisateurs

L'utilisateur qui dispose du rôle **ROLE_ADMIN** peut donc avoir accès à la gestion des tâches et des utilisateurs.

3.1.1 Rôles hiérarchiques

Au lieu d'attribuer plusieurs rôles à chaque utilisateur, il est possible de définir des règles d'héritage des rôles en créant une hiérarchie des rôles.

config/packages/security.yaml

```yaml
role_hierarchy:
    ROLE_ADMIN: [ ROLE_TASK_MANAGE ]
```

Dans l'application, les utilisateurs avec le rôle **ROLE_ADMIN** auront également le rôle **ROLE_TASK_MANAGE**.

3.2 Sécurisation des modèles d'url (access_control)

Le moyen le plus simple de sécuriser une partie de l'application consiste à sécuriser un modèle d'URL complet dans la configuration de l' **access_control**.

config/packages/security.yaml

```yaml
access_control:
    - { path: ^/login, roles: PUBLIC_ACCESS }
    - { path: ^/reset-password, roles: PUBLIC_ACCESS }
    - { path: ^/, roles: ROLE_USER }
```

L'application exige que l'utilisateur ait le rôle :

- **ROLE_USER** (c'est-à-dire qu'il soit authentifié) pour accéder aux différentes pages **(^/)**
- **ROLE_ADMIN** pour accéder aux pages d'administration des utilisateurs **(^/users)**

3.2.1 Autorisation de l'accès non sécurisé

Lorsqu'un visiteur n'est pas encore connecté à l'application, il est traité comme « non authentifié » (utilisateur anonyme) et n'a aucun rôle. Cela empêche la visite sur les pages de l'application.

Néanmoins, les pages de connexion **(^/login)** et de réinitialisation de mot de passe **(^/reset-password)** doivent disposer d'un accès non sécurisé (c'est-à-dire qu'elles doivent êtres accessibles par des utilisateurs anonymes).

Pour ces 2 routes, l'**access_control** est configuré avec l'attribut de sécurité **PUBLIC_ACCESS** qui permet de les exclure de l'accès authentifié qui est paramétré sur l'ensemble des pages de l'application

3.3 Contrôle d'accès dans les vues

Pour vérifier si l'utilisateur actuel a un certain rôle, il est possible d'utiliser la fonction d'assistance intégrée dans n'importe quel modèle Twig : **is_granted()**

templates/base.html.twig

```twig
{% if is_granted('ROLE_USER') %}
    <div class="collapse navbar-collapse navbar-right" id="bs-example-navbar-collapse-1">
        <ul class="nav navbar-nav">
            <li><a href="{{ path('/')}}">Accueil</a></li>
            <li class="dropdown">
                <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button">
                    Tâches
                    <span class="caret"></span>
                </a>
                <ul class="dropdown-menu">
                    <li><a href="{{ path('task_list') }}">Tâches à faire</a></li>
                    <li><a href="{{ path('/tasks/done') }}">Tâches terminées</a></li>
                    <li><a href="{{ path('/tasks/create') }}">Créer un nouvelle tâche</a></li>
                </ul>
            {% if is_granted('ROLE_ADMIN') %}
                <li><a href="{{ path('user_list') }}">Utilisateurs</a></li>
            {% endif %}
            <li><a class="btn btn-danger logout" href="{{ path('/logout') }}">Se déconnecter</a></li>
        </ul>
    </div>
{% endif %}
```

Dès lors que l'utilisateur est authentifié (**ROLE_USER**), il a accès au menu de navigation.

Si l'utilisateur dispose également du rôle **ROLE_ADMIN**, il a également accès au lien lui permettant de se rendre sur la page d'administration des utilisateurs.

3.4 Sécurisation des accès dans les contrôleurs et méthodes

L'application nécessite que l'utilisateur ait le rôle **ROLE_ADMIN** pour la gestion des utilisateurs.

src/Controller/UserController.php

```
15    #[Route('/users')]
16    #[IsGranted('ROLE_ADMIN')]
17    class UserController extends BaseController{...}
101
```

Si l'accès n'est pas accordé, une **AccessDeniedException** est lancée et plus aucun code du contrôleur n'est appelé. Ensuite, l'une des deux choses suivantes se produira :

- L'utilisateur n'est pas encore authentifié : il lui sera demandé de se connecter (il sera redirigé vers la page de connexion)
- L'utilisateur est authentifié, mais n'a pas le rôle **ROLE_ADMIN** : il lui sera affiché la page 403 accès refusé

3.5 Autorisations utilisateur individuelles

L'application nécessite des règles d'accès plus spécifiques.

Pour la **suppression d'une tâche** :

- Un utilisateur ne doit pas pouvoir supprimer une tâche qui ne lui appartient pas
- Un administrateur doit pouvoir supprimer toutes les tâches dont les tâches non rattachées à un utilisateur (tâche anonyme)

Pour la **modification d'une tâche** :

- Un utilisateur ne doit pas pouvoir modifier une tâche terminée

Pour la **suppression d'un utilisateur** :

- Un administrateur ne doit pas pouvoir supprimer son propre compte

Pour cela, l'application utilise le système d'électeurs (**VOTERS**) de Symfony. Ils permettent de centraliser toute la logique d'autorisation, puis de les réutiliser à de nombreux endroits.

3.5.1 Autorisation sur la suppression d'une tâche

La sécurisation pour la suppression d'une tâche est gérée par l'électeur possédant l'attribut **TASK_DELETE**, c'est-à-dire l'électeur personnalisé **TaskVoter**.

src/Controller/TaskController.php

```php
#[Route(path: '/{id}/delete', name: 'task_delete', methods: 'POST')]
#[IsGranted('TASK_DELETE', subject: 'task')]
public function deleteTaskAction(Task $task, Request $request): RedirectResponse{...}
```

src/Security/Voter/TaskVoter.php

```php
if ($this->security->isGranted( attributes: 'ROLE_TASK_MANAGE')) {
    return true;
}

// ... (check conditions and return true to grant permission) ...
return match ($attribute) {
    self::DELETE => $user === $subject->getOwner(),
    self::EDIT => !$subject->isDone(),
    default => false,
};
```

L'électeur autorisera l'accès à la demande de suppression de la tâche si une des deux vérifications est valide :

- Si l'utilisateur possède le rôle **ROLE_TASK_MANAGE** (ce qui est le cas de l'administrateur via la hiérarchie de rôle)
- Si l'utilisateur est le **propriétaire** (owner) de la tâche

3.5.2 Autorisation sur la modification d'une tâche

La sécurisation pour la modification d'une tâche est gérée par l'électeur possédant l'attribut **TASK_EDIT**, c'est-à-dire l'électeur personnalisé **TaskVoter**.

src/Controller/TaskController.php

```php
#[Route(path: '/{id}/edit', name: 'task_edit', methods: ['GET', 'POST'])]
#[IsGranted('TASK_EDIT', 'task')]
public function editAction(Task $task, Request $request): RedirectResponse|Response{...}
```

src/Security/Voter/TaskVoter.php

```php
if ($this->security->isGranted( attributes: 'ROLE_TASK_MANAGE')) {
    return true;
}

// ... (check conditions and return true to grant permission) ...
return match ($attribute) {
    self::DELETE => $user === $subject->getOwner(),
    self::EDIT => !$subject->isDone(),
    default => false,
};
```

L'électeur autorisera l'accès à la demande de suppression de la tâche si une des deux vérifications est valide :

- Si l'utilisateur possède le rôle **ROLE_TASK_MANAGE** (ce qui est le cas de l'administrateur via la hiérarchie de rôle)
- Si la tâche est marquée comme « **non terminée** »

3.5.3 Autorisation sur la suppression d'un utilisateur

La sécurisation pour la suppression d'un utilisateur est gérée par l'électeur possédant l'attribut **USER_DELETE**, c'est-à-dire l'électeur personnalisé **UserVoter**.

src/Controller/UserController.php

```
83      #[Route(path: '/{id}/delete', name: 'user_delete', methods: 'POST')]
84      #[IsGranted('USER_DELETE', 'user')]
85      public function deleteTaskAction(User $user, Request $request): RedirectResponse{...}
```

src/Security/Voter/UserVoter.php

```
41          // ... (check conditions and return true to grant permission) ...
42          return match ($attribute) {
43              self::DELETE => $subject !== $this->security->getUser(),
44              default => false,
45          };
```

L'électeur autorisera l'accès à la demande de suppression de l'utilisateur si l'utilisateur authentifié n'est pas identique à celui qui souhaite être supprimé.