“TODO&CO”

Implémentation de l’authentification et l’autorisation

Partie 1 : Les bundles utilisés :

En premier lieux nous avons besoin du Security Bundle de Symfony, ce bundle nous permettra de gérer l’authentification des utilisateurs et les authorisation (si un utilisateur connecté à le droit ou pas d'accéder à une page ou utiliser une fonctionnalité).

Doctrine nous permet de gérer la création et l’édition des entités de notre application, mais pour créer une entité ‘User’ qui représente un compte utilisateur nous avons aussi besoin du MakerBundle.

Partie 2 : L’implémentation de l’authentification :

Premièrement nous installons les bundles utilisées, ensuite nous créons l’entité ‘User’ avec le MakerBundle qui permet de créer et de gérer les comptes utilisateurs.

L’utilisation de Maker Bundle et Security Bundle nous permettra d’avoir une configuration de sécurité par défaut établie dans ‘/config/packages/security.yaml’, dont la configuration qui indique à Symfony l’entité qui servira à gérer toutes les fonctionnalité liés aux comptes utilisateurs.

```yaml
    providers:
        users_in_memory: { memory: null }
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
```

Ainsi que le cryptage utilisé pour crypter et décrypter les mot de passe :

```yaml
password_hashers:
    App\Entity\User: 'auto'
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
        algorithm: 'auto'
        cost: 15
```

Nous configurons les routes qui gères la connexion *(login_path)*, la vérification de la connexion *(check_path)* et la déconnexion *(logout)* d'un utilisateur :

```yaml
        main:
            lazy: true
            provider: app_user_provider
            form_login:
                    login_path: login
                    check_path: login
            logout:
                path: logout
```

Ces trois routes correspondent au trois méthodes dans *App/Controller/SecurityController* :

```php
class SecurityController extends AbstractController
{

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

        return $this->render( view: 'security/login.html.twig', array(
            'last_username' => $lastUsername,
            'error' => $error
        ));
    }

    #[Route('/login_check', name: 'login_check')]
    public function loginCheck()
    {
        // This code is never executed.
    }

    #[Route('/logout', name: 'logout')]
    public function logoutCheck()
    {
        // This code is never executed.
    }
}
```

Les deux méthodes *loginCheck* & *logoutCheck* ne requiert aucune configuration supplémentaire, dans la méthode *loginAction* vous pouvez configurer la vue et les informations qui seront affichées à l'utilisateur au moment d'une connexion réussie ou échouée.

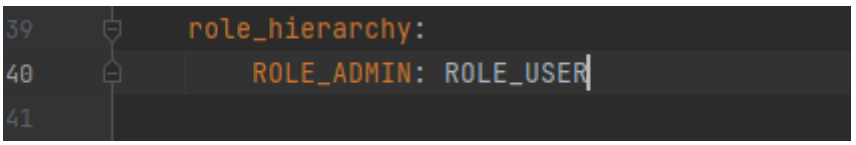
Partie 3 : L'implémentation de l'autorisation :

L'autorisation dans l'application se gère en assignant des rôles à un utilisateur, l'application TODO&CO propose deux rôles :

- ROLE_ADMIN
- ROLE_USER

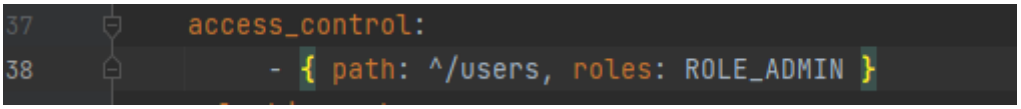
Si des rôles supplémentaires doivent être intégrés dans l'application, pour respecter la convention de nommage des rôles dans une application symfony tous les rôles doivent commencer par *'ROLE_'.*

Ces rôles sont gérés dans *'config/packages/security.yaml'* :



Ici nous déclarons les deux rôles, en utilisant la hiérarchie des rôles du security bundle pour indiquer que : ROLE_ADMIN hérite de tous les droits qui sont accordés au ROLE_USER (mais ROLE_USER ne hérite pas des droits de ROLE_ADMIN).

Dans *'config/packages/security.yaml'* nous pouvons indiquer des routes qui requiert une autorisation très simplement, par exemple n'autoriser que les utilisateurs avec le *'ROLE_ADMIN'* d'accéder à toutes les routes commençant par *'/users'* :



Nous pouvons également déclarer ces vérifications d'autorisation au début de une méthode auquel vous souhaitez assigner une autorisation il vous suffit d'utiliser la méthode `$this->denyAccessUnlessGranted('ROLE_ADMIN')` qui permet de donner l'autorisation à l'utilisateur seulement si le rôle passer en paramètre fait partie des rôles qui sont assignés à l'utilisateur (ou si l'utilisateur détient un rôle héritant des droits d'autorisation du rôle passer en paramètre).

Prenons un exemple de vérification d'autorisation plus complexe : la fonctionnalité d'édition d'une tâche *(Controller/TaskController/editAction)*.

```php
#[Route('/tasks/{id}/edit', name: 'task_edit')]
public function editAction(Task $task, Request $request, ManagerRegistry $managerRegistry,): RedirectResponse|Response
{
    $this->denyAccessUnlessGranted( attribute: 'edit', $task);

    $form = $this->createForm( type: TaskType::class, $task, [
        'validation_groups' => ['edit']
    ]);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $managerRegistry->getManager()->flush();

        $this->addFlash( type: 'success', message: 'La tâche a bien été modifiée.');

        return $this->redirectToRoute( route: 'task_list');
    }

    return $this->render( view: 'task/edit.html.twig', [
        'form' => $form->createView(),
        'task' => $task,
    ]);
}
```

A la ligne 66 nous invoquons la méthode `$this->denyAccessUnlessGranted`, auquel on passe deux paramètres *'edit'* (l'action que l'on veut exécuter) et l'objet `$task` (la tâche que l'on souhaite modifier).

Cette utilisation de *denyAcessUnlessGranted* invoquera la classe *Voter* correspondant à l'action et à l'objet passé en paramètre, c'est donc la classe *App/Security/TaskVoter* qui est appelée.

D'abord nous déclarons dans des constantes les différents types d'actions sur lequel la classe *TaskVoter* va pouvoir s'exécuter, ainsi que l'entité qui est passer en deuxieme parametre :

```php
const DELETE = 'delete';
const EDIT = 'edit';
const CREATE = 'create';
const TOGGLE = 'toggle';

/**
 * @inheritDoc
 */
protected function supports(string $attribute, $subject): bool
{
    // if the attribute isn't one we support, return false
    if (!in_array($attribute, [self::DELETE, self::EDIT, self::CREATE,  self::TOGGLE])) {
        return false;
    }

    // only vote on `Task` objects
    if (!$subject instanceof Task) {
        return false;
    }

    return true;
}
```

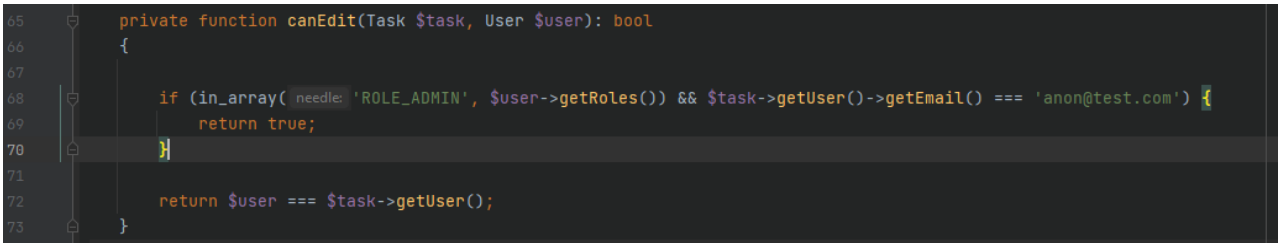
La méthode *supports()* vérifie que la classe contient l'action demandé dans ces constantes ou si l'objet qui est passer en deuxieme parametre est une instance de l'entité *Task*, si ce n'est pas le cas le voter indique qu'il ne pourra pas se prononcer sur la décision d'autoriser ou pas l'utilisateur en retournant *false*.

Si le *supports()* renvoient true nous passons à la méthode *voteOnAttribute()* :

```php
42    protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
43    {
44        $user = $token->getUser();
45
46        if (!$user instanceof User) {
47            // the user must be logged in; if not, deny access
48            return false;
49        }
50
51        // you know $subject is a Task object, thanks to `supports()`
52        /** @var Task $task */
53        $task = $subject;
54
55        switch ($attribute) {
56            case self::DELETE || self::EDIT:
57                return $this->canEdit($task, $user);
58            case self::CREATE || self::TOGGLE:
59                return true;
60        }
61
62        throw new \LogicException( message: 'This code should not be reached!');
63    }
```

D’abord à la ligne 46 nous vérifions que `$user` soit bien une instance de l’entité User, si ce n’est pas le cas on refuse l’accès sans faire de vérification supplémentaire.

Le switch case à la ligne 55 regarde le contenue de `$attribute` (l’action ‘edit’ dans cet exemple), si l’utilisateur connecté souhaite créer une tâche ou changer le statut d’une tâche, le voter autorise l’accès car tous les utilisateurs authentifié peuvent exécuter ces actions sans vérifications supplémentaires. Mais pour les actions ‘*delete’* & *‘edit’* le voter appelle la méthode *canEdit()* :



Ici nous vérifions deux cas, d’abord si l’utilisateur à le rôle d’un administrateur *(ROLE_ADMIN)* et que la tâche que l’on souhaite modifier à été créer pour l’utilisateur anonyme. Si ce n’est pas le cas nous vérifions que la tâche que l’on souhaite modifier a été créée par l’utilisateur connecté.

Toutes modifications / ajouts de condition d'autorisation supplémentaire sur les actions concernant les tâches peuvent donc être établies dans *TaskVoter.*