Fonctionnement de la connexion sur To Do and Co

Le système de connexion à To Do and Co repose sur deux commandes :

- `symfony console make:user`
- `symfony console make:auth`

`symfony console make:user` permet de créer une entité qui servira au utilisateurs.
`symfony console make:auth` permet de mettre en place un système de connexion lié à une entité, ici notre entité `User`.

Comment fonctionne l'authentification ?

Regardons de plus prêt le fichier `config/packages/security.yaml` :

```yaml
security:
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            custom_authenticator: App\Security\AppAuthenticator
            logout:
                path: app_logout
    access_control:
        - { path: ^/tasks, roles: ROLE_USER }
        - { path: ^/users, roles: ROLE_ADMIN }

when@test:
    security:
        password_hashers:
                algorithm: auto
                cost: 4 # Lowest possible value for bcrypt
                time_cost: 3 # Lowest possible value for argon
                memory_cost: 10 # Lowest possible value for argon
```

`symfony console make:auth` vient modifier ce fichier à différent endroit pour mettre en place la connexion. Si l'on jette un oeil dans la partie `provider`, nous pouvons voir un provider nommé `app_user_provider`, utilisant notre entité `User` et la propriété `email`.

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

Qu'est-ce que cela signifie concrètement ? C'est notre entité `User` qui servira pour la partie sécurité de l'application.

Si on regarde maintenant du côté du firewall :

```yaml
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            custom_authenticator: App\Security\AppAuthenticator
            logout:
                path: app_logout
```

On retrouve dans la partie `provider` notre `app_user_provider`, mais aussi `App\Security\AppAuthenticator` dans la partie `custom_authenticator`. C'est cette classe qui sera en charge de la connexion.

Si on regarde cette classe, c'est elle qui est en charge d'authentifier notre utilisateur en lui créant un `Passport`. On peut voir dans la méthode `authenticate` qu'on récupère bien la propriété `email` pour l'utiliser avec notre entité `User`, mais aussi son mot de passe pour voir s'il match bien avec le hash stocké en base de données.

```php
<?php

namespace App\Security;

use Symfony\Bundle\SecurityBundle\Security;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Http\Authenticator\AbstractLoginFormAuthenticator;
use Symfony\Component\Security\Http\Authenticator\Passport\Badge\CsrfTokenBadge;
use Symfony\Component\Security\Http\Authenticator\Passport\Badge\UserBadge;
use Symfony\Component\Security\Http\Authenticator\Passport\Credentials\PasswordCredentials;
use Symfony\Component\Security\Http\Authenticator\Passport\Passport;
use Symfony\Component\Security\Http\Util\TargetPathTrait;

class AppAuthenticator extends AbstractLoginFormAuthenticator
{
    use TargetPathTrait;

    public const LOGIN_ROUTE = 'app_login';

    public function __construct(private readonly UrlGeneratorInterface $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);
        }

        return new RedirectResponse($this->urlGenerator->generate('homepage'));
    }

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

Pour finir, nous allons voir le `SecurityController` qui est en charge de rendre la page sur le site :

```php
<?php

namespace App\Controller;
```

```php
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class SecurityController extends AbstractController
{
    #[Route(path: '/login', name: 'app_login')]
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        $lastUsername = $authenticationUtils->getLastUsername();

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

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

Ici deux routes sont définies, `/login` et `/logout`. Intéressons nous à la vue de la méthode `login` :

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

{% block title %}Log in!{% endblock %}

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

        {% if app.user %}
            <div class="mb-3">
                You are logged in as {{ app.user.userIdentifier }},
                <a href="{{ path('app_logout') }}">Logout</a>
            </div>
        {% endif %}

        <h1 class="h3 mb-3 font-weight-normal">Connexion</h1>
        <div class="mb-3">
            <label for="inputEmail">Email</label>
            <input
                    type="email" value="{{ last_username }}"
                    name="email" id="inputEmail"
                    class="form-control" autocomplete="email"
                    required autofocus
                >
        </div>
        <div class="mb-3">
            <label for="inputPassword">Mot de passe</label>
            <input
                    type="password" name="password"
                    id="inputPassword" class="form-control"
                    autocomplete="current-password" required
                >
        </div>
        <input type="hidden" name="_csrf_token"
                value="{{ csrf_token('authenticate') }}">

        <button class="btn btn-primary" type="submit">Se connecter</button>
    </form>{% endblock %}
```

Hormis l'input pour le token CSRF, on retrouve un formulaire de connexion classique.

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

Une fois le formulaire soumis, c'est alors notre `App\Security\AppAuthenticator` qui prend le relais pour s'assurer que les informations soumise sont corrects, dans le cas échéant, un ou des messages d'erreurs seront affichées à l'utilisateur.

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

Une dernière chose avant de conclure, dans notre fichier `config/packages/security.yaml` nous définissons également la route en charge de déconnecter l'utilisateur.

```
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            custom_authenticator: App\Security\AppAuthenticator
            logout:
                path: app_logout  # Ici nous définissons la route servant à la déconnexion.
```

Cette route correspond à notre route `/logout` de la méthode `logout` du contrôleur `SecurityController`.

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

Pour conclure

Les fichiers importants sont les suivants :

- `config/packages/security.yaml` contient la configuration liée à la sécurité.
- `src/Security/AppAuthentificator.php` contient le code en charge de connecter ou non l'utilisateur.
- `src/controller/SecurityController` contient la route de connexion et de déconnexion.
- `templates/security/login.html.twig` contient la vue en charge de rendre la page de connexion.