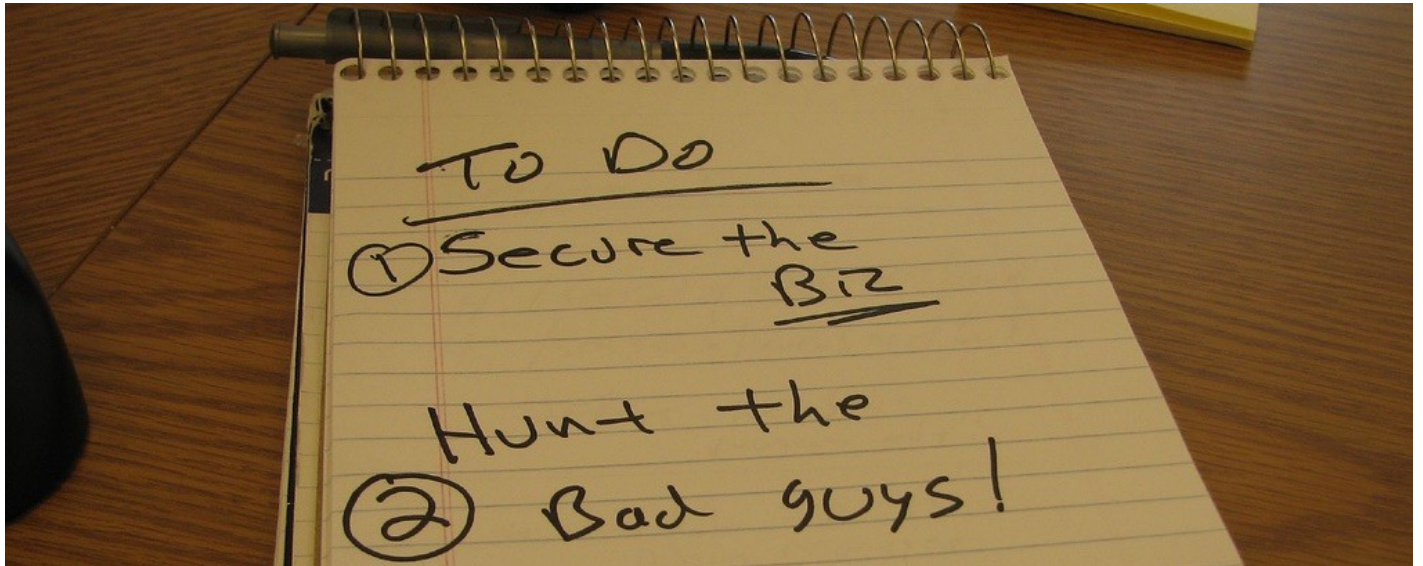


# Documentation technique



## Table des matières

Nicolas de Fontaine ToDo & Co – TodoList.....	1
2. Générer l'entity User.....	2
3. Implémenté l'authentification.....	3
4 Fonctionnement de l'authentification .....	4
4.1 Les providers.....	4
.....	4
4.2 les encoders ou password_hashers.....	4
.....	4
4.3 Le firewalls.....	5
4.4 Access Control.....	5
.....	6
4.5 La class User (src/Entity/User).....	6
4.7 Le AppAuthenticator.php.....	7
4.8 La vue login.html.twig.....	8

## 2. Générer l'entity User

Avant toute chose, il est nécessaire de définir une entité qui représentera l'utilisateur connecté. Cette classe doit implémenter de l'interface `UserInterface` pour ajouter les différentes méthodes définies dans celle-ci.

Pour générer l'entity user vous pouvez utiliser le `MakerBundel` :

**`php bin/console make:user`**

## 3. Implémenté l'authentification

Symfony embarque la mise en place de l'authentification à travers le `MakerBundel`

```
netrestore-3c0754698637:securitybundle utilisateur$ php bin/console make:auth

What style of authentication do you want? [Empty authenticator]:
[0] Empty authenticator
[1] Login form authenticator
> 1

The class name of the authenticator to create (e.g. AppCustomAuthenticator):
> LoginFormAuthenticator

Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
>

Do you want to generate a '/logout' URL? (yes/no) [yes]:
>

created: src/Security/LoginFormAuthenticator.php
updated: config/packages/security.yaml
created: src/Controller/SecurityController.php
created: templates/security/login.html.twig

Success!

Next:
- Customize your new authenticator.
- Finish the redirect "TODO" in the App\Security\LoginFormAuthenticator::onAuthenticationSuccess() method.
- Review & adapt the login template: templates/security/login.html.twig.
```

La commande : **`php bin/console make:auth`**

Le maker va vous poser plusieurs questions :

« What style of authentication do you want ? » permet de choisir le type d'authentification.

« The class name of the authenticator create » Le nom de class Authenticator qui va gérer l'authentification des utilisateurs.

« Choose a name for the controller class » Nom du controller qui va contenir les méthodes login, register etc.

« Do you want to generate a /logout URL » Permet de générer la route de déconnexion.

## 4 Fonctionnement de l'authentification

### 4.1 Les providers

Le provider va nous permettre d'indiquer où se situent les informations que l'on souhaite utiliser pour l'authentification de l'utilisateur. Dans le cas de ToDoList on indique qu'on récupérera les utilisateurs via Doctrine (base de données) grâce à l'entité User et que la propriété email sera utilisé pour s'authentifier sur le site.

Plus d'infos : <https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider>

```
# 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
```

### 4.2 les encoders ou password\_hashers

La section password\_hashers va simplement nous permettre de déterminer quel algorithme nous allons utilisée lors de l'encodage de certaine information.

Dans le cas de ToDoList, on utilisera l'algorithme bcrypt lorsque que quelque chose doit être encodée dans l'entité App\Entity\User via le PasswordAuthenticatedUserInterface, dans ce cas-ci cela concerne le mot de passe.

```
# 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
```

## 4.3 Le firewalls

Un firewall va définir comment nos utilisateurs vont être authentifiés sur certaines parties du site. Le firewall dev ne concerne que le développement ainsi que le profiler et ne devra à priori pas être modifié.

```
firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  main:
    lazy: true
    provider: app_user_provider
    custom_authenticator: App\Security\AppAuthenticator
    pattern: ^/
    logout:
      path: app_logout
      # where to redirect after logout
      # target: app_any_route
```

Le firewall main englobe l'entièreté du site à partir de la racine défini via `pattern: ^/`, l'accès y est autorisé en anonyme c-à-d sans être authentifié, on y indique que c'est le provider "app\_user\_provider" qui sera utilisé ainsi que notre AppAuthenticator généré précédemment.

## 4.4 Access Control

L'access control va définir les limitations d'accès à certaines parties du site. Dans ce cas-ci, on indique que :

- L'url /login est accessible sans authentification.
- Tout le reste de l'application nécessite une authentification,

```
# 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: /login, roles: [IS_AUTHENTICATED_ANONYMOUSLY] }
  - { path: ^/, roles: [IS_AUTHENTICATED_FULLY] }
```

il y a également une autre méthode pour protéger les accès : Dans le cas de ToDoList le controller User est accessible seulement aux administrateurs, de ce fait vous-pouvez également utilisé les annotations en tête de votre controller :

```
/**
 * @Route("/user")
 * @IsGranted("ROLE_ADMIN")
 */
class UserController extends AbstractController
{
```

## 4.5 La class User (src/Entity/User)

La class User représente la table où sont stockés les utilisateurs, elle implémente la UserInterface qui contient les méthodes minimales pour l'authentification.

## 4.6 Le SecurityController

Le SecurityController contient les points d'entrées des routes liées à l'authentification :

La méthode **login()** ne contient pas toute la logique pour l'authentification, elle permet de rendre la vue du formulaire et de récupérer certaines informations comme par exemple les erreurs ou le dernier nom d'utilisateur utilisé pour pré-remplir le formulaire.

```
/**
 * @Route("/login", name="login")
 */
public function login(AuthenticationUtils $authenticationUtils, EntityManagerInterface $manager, UserPasswordEncoderInterface $encoder): Response
{
    if ($this->getUser()) {
        return $this->redirectToRoute('route: '/'');
    }
    // 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', ['last_username' => $lastUsername, 'error' => $error]);
}
```

La méthode **logout()** permet simplement d'effectuer certaine action lors de la déconnexion, par exemple rediriger l'utilisateur vers la page d'accueil.

```
/**
 * @Route("/logout", name="app_logout")
 */
public function logout(): RedirectResponse
{
    return $this->redirectToRoute('route: '/'');
}
```

## 4.7 Le AppAuthenticator.php

L'authenticator permet à l'utilisateur de démarrer le processus d'authentification. La classe en charge de cette authentification devra implémenter l'interface [AuthenticatorInterface](#) qui dans ce cas présent est chargée par la class parente `AbstractLoginFormAuthenticator`.

La class prend en paramètre dans son constructeur `UrlGeneratorInterface` qui permet de générer des URL à partir des routes,

```
private UrlGeneratorInterface $urlGenerator;

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

La méthode `authenticate` permet de lancer le processus d'authentification et renvoie objet de type `PassportInterface`,

Le passeport contient les informations d'identification et toutes les informations supplémentaire qui doivent être vérifiés par le système Symfony Security.

Par exemple, pour une connexion l'authenticator renverra un passeport contenant l'utilisateur et le mot de passe soumis par le formulaire et la valeur du jeton CSRF.

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

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

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



La méthode `onAuthenticationSuccess` permet de rediriger l'utilisateur une fois connecté.

```
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( name: self::LOGIN_ROUTE));
}
```

## 4.8 La vue `login.html.twig`

La vue `login.html.twig` contient tout les éléments du formulaire de connexion au langage HTML elle contient 2 conditions :

`if error` permettra d'afficher les erreurs liées au formulaire

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

`if app,user` permet d'informer l'utilisateur qui arrive sur la page qu'il est déjà connecté

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