

## Étape 6 : Authentification

L'objectif est de mettre en place un **système d'authentification** sur notre API et donc pour le **front-end Angular**.

Les utilisateurs pourront :

- Voir la liste des catégories à tout moment.
- Voir les articles uniquement s'ils sont identifiés (`ROLE_USER` ou `ROLE_ADMIN`).
- Ajouter un article uniquement s'ils sont identifiés en tant qu'administrateurs (`ROLE_ADMIN`).

---

### ⚠ Attention :

- Dans cet exemple, nous allons mettre en place une **authentification basée sur les tokens JWT** côté **Symfony**, mais nous allons utiliser les tokens JWT comme de **simples tokens**.
  - Un **"simple" token** est juste une chaîne de caractères obtenue généralement à l'aide d'un algorithme de hashage (*un code, en somme*).
  - Un **token JWT** (*JSON Web Token*) sert de **code d'identification unique** et peut également inclure des **données personnalisées** (*claims*), permettant d'**identifier un utilisateur** et de **transporter des informations spécifiques** de manière sécurisée.
  - Il existe des mécanismes de **renouvellement des tokens JWT**, souvent mis en œuvre avec les concepts de **Refresh Token** et **Access Token**.
  - **Nous n'allons pas utiliser ces mécanismes maintenant**, mais nous en parlerons dans un prochain module de S6.

**Pourquoi utiliser des tokens JWT si nous voulons simplement des tokens classiques ?**

Actuellement, avec **Symfony**, le module **JWT** est **plus simple à mettre en œuvre** qu'une authentification basée sur un **token classique**.

🔗 Si vous maîtrisez JWT ou un autre mécanisme d'authentification, vous pouvez parfaitement l'utiliser dans votre projet.

---

## 1. Configuration de PHP

Nous avons besoin du module **Sodium** pour PHP, qui **n'a pas été activé au démarrage du projet**.

- Éditer votre fichier **php.ini**

Vérifiez que la ligne suivante **n'est pas commentée** (*pas de ; au début de la ligne*) :  
extension=sodium

○

- **Après activation de Sodium :**

Mettez à jour les dépendances avec :  
composer update

○

**Redémarrez le serveur Symfony :**

symfony server:start --port=8008

○

---

## 2. Configuration de la Sécurité dans Symfony

Nous allons installer et configurer le **bundle lexik/jwt-authentication-bundle**, puis définir :

- Une **route d'identification** (`/api/login`).
- Une **route pour récupérer les informations de l'utilisateur connecté** (`/api/user/me`).
- Des **règles d'accès aux routes** en fonction des rôles.

---

### Installation du module JWT

composer require lexik/jwt-authentication-bundle

---

### Génération des clés SSH

⚠ Utilisez **GIT BASH** au lieu de **PowerShell** pour être sûr d'avoir **openssl**.

# Dans Git Bash !

mkdir -p config/jwt

openssl genrsa -out config/jwt/private.pem -aes256 4096

openssl rsa -pubout -in config/jwt/private.pem -out config/jwt/public.pem

📌 OpenSSL va demander une **passphrase**. **Notez-la bien**, car elle sera nécessaire pour la configuration du bundle.

(Dans cet exemple, nous supposons que la passphrase est *PassPhraseDeFabrice*, mais vous devez utiliser la vôtre.)

♦ **Alternative :**

Il est aussi possible d'utiliser la commande suivante (si *openssl* est bien installé sur votre machine) :

```
php bin/console lexik:jwt:generate-keypair
```

---

## Configuration du bundle JWT

Dans le fichier **.env**, remplacez la **passphrase** par la vôtre :

```
###> lexik/jwt-authentication-bundle ###
JWT_SECRET_KEY=%kernel.project_dir%/config/jwt/private.pem
JWT_PUBLIC_KEY=%kernel.project_dir%/config/jwt/public.pem
JWT_PASSPHRASE=PassPhraseDeFabrice
###< lexik/jwt-authentication-bundle ###
```

Dans **config/packages/lexik\_jwt\_authentication.yaml**, configurez les chemins vers les **clés** et l'**identifiant utilisateur** (*email dans notre cas*) :

```
lexik_jwt_authentication:
  secret_key: '%env(resolve:JWT_SECRET_KEY)%'
  public_key: '%env(resolve:JWT_PUBLIC_KEY)%'
  pass_phrase: '%env(JWT_PASSPHRASE)%'
  token_ttl: 3600 # Durée de vie du token en secondes (1h)
  user_identity_field: email
```

---

## Configuration des Firewalls

♦ L'ordre des **firewalls** est important :

- **login** → Doit être en **premier**.
- **api** → Juste après.
- **Les autres (dev, main...)** suivent.

📌 Éditez **config/packages/security.yaml** :

```
firewalls:
  login:
```

```
pattern: ^/api/login
stateless: true
json_login:
    check_path: /api/login
    username_path: email
    success_handler: lexik_jwt_authentication.handler.authentication_success
    failure_handler: lexik_jwt_authentication.handler.authentication_failure
api:
    pattern: ^/api
    stateless: true
    jwt: ~
# Autres firewalls (ex: dev, main)
```

---

## Configuration des règles d'accès (**access\_control**)

Toujours dans **config/packages/security.yaml**, définissez les restrictions d'accès aux endpoints :

```
access_control:
    - { path: ^/admin, roles: ROLE_ADMIN }
    - { path: ^/api/login, roles: PUBLIC_ACCESS }
    - { path: ^/api/articles, roles: ROLE_USER }
    - { path: ^/api/article, roles: ROLE_ADMIN }
    - { path: ^/api, roles: PUBLIC_ACCESS }
```

---

## Création d'un Contrôleur pour l'Authentification

Ce contrôleur permettra de :

- Vérifier les informations de l'utilisateur (**/api/user/me**).
- Définir une route **/api/login** (*même si elle est gérée automatiquement par **lexik\_jwt\_authentication***).

**Générez le contrôleur :**

```
php bin/console make:controller AuthController --no-template
```

**Modifiez le fichier **src/Controller/AuthController.php** :**

```
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Bundle\SecurityBundle\Security;
```

```

use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class AuthController extends AbstractController
{
    public function __construct(
        private Security $security
    ) {}

    #[Route('/api/user/me', name: 'api_user_me', methods: ['GET'])]
    public function me()
    {
        $user = $this->security->getUser();
        if (!empty($user)) {
            return new JsonResponse([
                'email' => $user->getUserIdentifier(),
                'roles' => $user->getRoles(),
            ]);
        } else {
            return new JsonResponse(null);
        }
    }

    #[Route('/api/login', name: 'api_login', methods: ['POST'])]
    public function login(): Response
    {
        // L'utilisateur est authentifié à ce stade
        // lexik/jwt-authentication-bundle s'occupe de retourner le JWT
        return new Response('Logged !');
    }
}



```

---

### 3. Tests avec Postman

Ces tests permettent de vérifier le bon fonctionnement de l'authentification.

#### 1 Tester l'authentification (**/api/login**)

 **Requête POST** sur : <https://localhost:8008/api/login>  
 **Données envoyées (JSON)** :

```

{
    "email": "admin@articles.fr",
    "password": "admin"
}

```

```
}
```

✓ Réponse attendue :

- Un JSON contenant un **token JWT**
  - **Copiez ce token**, nous en aurons besoin pour les requêtes suivantes.
- 

## 2 Accéder aux catégories (libre accès)

📌 Requête **GET** sur : <https://127.0.0.1:8008/api/categories>

✓ Réponse attendue :

- La liste des catégories
  - **Pas besoin d'envoyer de token**
- 

## 3 Accéder aux articles (**ROLE\_USER** ou **ROLE\_ADMIN** requis)

📌 Requête **GET** sur : <https://127.0.0.1:8008/api/articles>

✗ Sans token :

- **Erreur 401 Unauthorized**
- Message : "JWT Token not found"

✓ Avec un token JWT (**Authorization: Bearer <token>**) :

- La liste des articles s'affiche
- 

## 4 Ajouter un article (**ROLE\_ADMIN** requis)

📌 Requête **POST** sur : <https://127.0.0.1:8008/api/article>

📌 Données envoyées (JSON) :

```
{  
  "titre": "Nouveau Titre",  
  "description": "Contenu de l'article",  
  "categorie": { "id": 1 }  
}
```

✗ Avec un compte **ROLE\_USER** :

- Erreur **403 Forbidden**

✓ Avec un compte **ROLE\_ADMIN** :

- L'article est ajouté avec succès
- 

## 4. Gestion de l'Authentification sur le Front Angular

Nous allons maintenant gérer l'authentification côté **Angular** avec :

- Un **service AuthService** pour gérer les connexions et les tokens.
  - Un **interceptor HTTP** pour ajouter automatiquement le token JWT aux requêtes.
  - Un **formulaire de connexion (LoginComponent)**.
  - Une **mise à jour du menu** pour afficher "Login" ou "Logout".
  - Des **protections sur les composants Angular (ArticlesListComponent, AddArticleComponent)**.
- 

### 4.1. Création du Service d'Authentification (AuthService)

Nous allons :

1. **Créer un modèle AuthUser** contenant les informations de l'utilisateur.
2. **Stocker et gérer le token JWT** dans **localStorage**.
3. **Mettre à jour l'utilisateur courant après connexion/déconnexion**.

**Générer le service AuthService**

ng generate service services/auth

**Code src/app/services/auth.service.ts**

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable, map } from 'rxjs';
```

```
export class AuthUser {
  constructor(
    public email: string = "",
    public roles: string[] = []
  ) {}

  isAdmin(): boolean {
    return this.roles.includes("ROLE_ADMIN");
  }
}
```

```

isLogged(): boolean {
  return this.email.length > 0;
}
}

@Injectable({
  providedIn: 'root'
})
export class AuthService {

  private apiUrlLogin = 'https://localhost:8008/api/login';
  private apiUrlUserInfo = 'https://localhost:8008/api/user/me';

  private localStorageToken = 'currentToken';

  private currentTokenSubject: BehaviorSubject<string | null>;
  public currentToken: Observable<string | null>;
  public get currentTokenValue(): string | null { return this.currentTokenSubject.value; }

  private currentAuthUserSubject: BehaviorSubject<AuthUser>;
  public currentAuthUser: Observable<AuthUser>;
  public get currentAuthUserValue(): AuthUser { return this.currentAuthUserSubject.value; }

  constructor(private http: HttpClient) {
    this.currentTokenSubject = new BehaviorSubject<string | null>(null);
    this.currentToken = this.currentTokenSubject.asObservable();
    this.currentAuthUserSubject = new BehaviorSubject(new AuthUser());
    this.currentAuthUser = this.currentAuthUserSubject.asObservable();

    const storedToken: string | null = localStorage.getItem(this.localStorageToken);
    this.updateUserInfo(storedToken);
  }

  private updateUserInfo(token: string | null) {
    this.currentTokenSubject.next(null);
    this.currentAuthUserSubject.next(new AuthUser());

    if (token) {
      const headers = new HttpHeaders({ 'Authorization': `Bearer ${token}`, 'skip-token': 'true'
});
      this.http.get<AuthUser>(this.apiUrlUserInfo, { headers }).subscribe({
        next: data => {
          if (data.email) {
            this.currentTokenSubject.next(token);
            this.currentAuthUserSubject.next(new AuthUser(data.email, data.roles));
          }
        }
      });
    }
  }
}

```



```

    });
  }
}

public login(email: string, password: string): Observable<boolean> {
  return this.http.post<any>(this.apiUrlLogin, { email, password })
    .pipe(map(response => {
      if (response.token) {
        this.updateUserInfo(response.token);
        return true;
      } else {
        return false;
      }
    }));
}

public logout() {
  this.updateUserInfo(null);
}
}

```

---

## 4.2. Création de l'Interceptor HTTP (AuthInterceptor)

Un **Interceptor** permet d'**ajouter automatiquement** le token JWT aux requêtes HTTP.

### Générer l'interceptor

ng generate interceptor services/auth

### Code `src/app/services/auth.interceptor.ts`

```

import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from
'@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    if (!request.headers.has('skip-token')) {
      let currentToken = this.authService.currentTokenValue;
      if (currentToken) {
        request = request.clone({

```

```

        setHeaders: {
          Authorization: `Bearer ${currentToken}`
        }
      });
    }
  } else {
    request = request.clone({
      headers: request.headers.delete('skip-token')
    });
  }

  return next.handle(request);
}
}

```

📌 Ajoutez cet Interceptor dans **app.module.ts** :

```

providers: [
  {
    provide: HTTP_INTERCEPTORS,
    useClass: AuthInterceptor,
    multi: true
  }
],

```

---

### 4.3. Création du Formulaire de Connexion (LoginComponent)

#### Générer le composant

ng generate component login

#### Code **src/app/login/login.component.ts**

```

import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { AuthService } from '../services/auth.service';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent {
  model: any = {};
}

```

```

constructor(
  private authService: AuthService,
  private router: Router
) {}

onSubmit() {
  this.authService.login(this.model.email, this.model.password).subscribe({
    next: () => this.router.navigate(['/']),
    error: err => console.error('Erreur lors de la connexion', err)
  });
}
}

```

#### Code `src/app/login/login.component.html`

```

<h2>Connexion</h2>

<form (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="email">Email :</label>
    <input type="email" class="form-control" id="email" [(ngModel)]="model.email"
name="email" required>
  </div>
  <div class="form-group">
    <label for="password">Mot de Passe :</label>
    <input type="password" class="form-control" id="password"
[(ngModel)]="model.password" name="password" required>
  </div>
  <button type="submit" class="btn btn-primary">Connexion</button>
</form>

```

---

## 4.4. Mise à jour du Menu

Ajoutez **Login/Logout** dans le menu de navigation.

#### Code `src/app/app.component.ts`

```

export class AppComponent {
  constructor(
    public authService: AuthService,
    private router: Router
  ) {}

  logout() {
    this.authService.logout();
    this.router.navigateByUrl('/login');
  }
}

```

```
}  
}
```

#### Code `src/app/app.component.html`

```
<li class="nav-item" *ngIf="authService.currentAuthUserValue.isLogged()">  
  <button class="nav-link" (click)="logout()">  
    Logout  
    <i *ngIf="authService.currentAuthUserValue.isAdmin()">ADMIN</i>  
    <i *ngIf="!authService.currentAuthUserValue.isAdmin()">user</i>  
  </button>  
</li>  
<li class="nav-item" *ngIf="!authService.currentAuthUserValue.isLogged()">  
  <a class="nav-link" routerLink="/login">Login</a>  
</li>
```

---

## 5. Protection des Composants Angular avec AuthService

Dans chaque composant soumis à authentification, on peut facilement vérifier les conditions d'utilisation :

- en injectant AuthService et Router
  - en vérifiant les condition d'usage dans ngOnInit()
  - en redirigeant éventuellement l'utilisateur si il ne remplit pas les conditions
    - *on pourrait aussi générer un message d'erreur ou juste limiter les fonctionnalités dans la vue*
- 

### 5.1. Sécuriser l'accès à la liste des articles (`ArticlesListComponent`)

#### Code `src/app/articles-list/articles-list.component.ts`

```
import { Component, OnInit } from '@angular/core';  
import { Router } from '@angular/router';  
import { Article } from '../models/article';  
import { ApiService } from '../services/api.service';  
import { AuthService } from '../services/auth.service';  
  
@Component({  
  selector: 'app-articles-list',  
  templateUrl: './articles-list.component.html',  
  styleUrls: ['./articles-list.component.css']  
})  
export class ArticlesListComponent implements OnInit {  
  articles: Article[] = [];
```

```

constructor(
  private apiService: ApiService,
  private authService: AuthService,
  private router: Router
) {}

ngOnInit(): void {
  if (!this.authService.currentUserValue.isLogged()) {
    // Si l'utilisateur n'est pas connecté, rediriger vers la page de connexion
    this.router.navigate(['/login']);
    return;
  }

  this.apiService.getArticles().subscribe((data: Article[]) => {
    this.articles = data;
  });
}
}

```

---

## 5.2. Sécuriser l'accès à l'ajout d'article (AddArticleComponent)

Code **src/app/add-article/add-article.component.ts**

```

import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { Article } from '../models/article';
import { Categorie } from '../models/categorie';
import { ApiService } from '../services/api.service';
import { AuthService } from '../services/auth.service';

@Component({
  selector: 'app-add-article',
  templateUrl: './add-article.component.html',
  styleUrls: ['./add-article.component.css']
})
export class AddArticleComponent implements OnInit {
  article: Article = new Article(0, "", "", new Date(), false, new Categorie(0, "", "", 0));
  categories: Categorie[] = [];
  info: string = "";

  constructor(
    private apiService: ApiService,
    private authService: AuthService,
    private router: Router
  ) {}

```

```

ngOnInit(): void {
  if (!this.authService.currentAuthUserValue.isAdmin()) {
    // Si l'utilisateur n'est pas admin, rediriger vers la page de connexion
    this.router.navigate(['/login']);
    return;
  }

  this.apiService.getCategories().subscribe((data: Categorie[]) => {
    this.categories = data;
  });
}

onSubmit() {
  this.apiService.addArticle(this.article).subscribe(result => {
    console.log('Article ajouté', result);
    this.info = 'Article No ' + result.id + ' créé';
  });
}
}

```

---

## 6. Conclusion

Nous avons mis en place une **authentification complète** avec **Symfony et Angular** :

### ✅ Côté Symfony

- Installation et configuration du **bundle JWT**.
- Sécurisation des routes avec des **règles d'accès basées sur les rôles**.
- Création d'un **contrôleur d'authentification**.

### ✅ Côté Angular

- Création d'un **service d'authentification** pour gérer le **token JWT**.
- Création d'un **interceptor** pour ajouter automatiquement le token aux requêtes.
- Mise en place d'un **formulaire de connexion**.
- Mise à jour du **menu de navigation** avec **Login** et **Logout**.
- Sécurisation des composants en fonction des **rôles utilisateurs**.

### 📌 Prochaines améliorations possibles :

- Gestion du **rafraîchissement des tokens JWT** (*Refresh Token*).
- Implémentation d'un **système de récupération de mot de passe**.
- Affichage de **messages d'erreur utilisateur-friendly** lors de la connexion.