Documentation technique

Scénario d’utilisation

1. **Titre : Inscription d'un nouvel utilisateur :**

L'utilisateur envoie ses informations de manière sécurisée via une requête POST à l'API d'inscription `/api/inscription` en format JSON :

- `email` : L'adresse email de l'utilisateur.
- `password` : Le mot de passe choisi par l'utilisateur.
- `nom` : Le nom d'utilisateur choisi par l'utilisateur.

Le système valide ces informations, crée un utilisateur avec un mot de passe haché, génère un token JWT pour l'activation, puis envoie un email de confirmation à l'utilisateur.

L'utilisateur clique sur le lien d'activation dans son email, ce qui redirige vers l'API pour vérifier le token et activer son compte.

2. **Titre : Connexion du nouveau utilisateur :**

- **Connexion initiale** : L'utilisateur saisit son nom d'utilisateur et son mot de passe. Le système vérifie ces informations.
    - Si valide, un code PIN est généré et envoyé par email pour une vérification supplémentaire.
    - Si invalide, une erreur est retournée (mot de passe incorrect, compte non vérifié, ou trop de tentatives).

- **Validation du code PIN** : L'utilisateur saisit le code PIN reçu. Le système vérifie sa validité.
    - Si le PIN est valide, un **token JWT** est généré pour authentifier l'utilisateur pour ses prochaines requêtes.
    - Si invalide ou expiré, une erreur est retournée.
- **Accès à l'application** : Une fois le token JWT validé, l'utilisateur peut accéder aux ressources sécurisées de l'application.

Fonctionnalites

1. Fonction “InscriptionController”: #pour l’inscription:

- Validation : `email`, `utilisateur`, `mot de passe`, `erreurs`
    - Methodes: `ValidatorInterface::validate()` : Valide les entités (email, utilisateur) pour détecter les erreurs.

- Creation d’utilisateur : `User`, `roles`, `mot de passe haché`
    - Methodes: `UserPasswordHasherInterface::hashPassword()` : Hache le mot de passe de l'utilisateur pour la sécurité.
    - `setIdEmail()`, `setUsername()`, `setRoles()` : Définir les propriétés de l'utilisateur.
    - `EntityManagerInterface::persist()` : Persiste les objets `User`, `Email` et `Token` dans la base de données.

- Token JWT : `génération`, `expiration`, `payload`, `JWTManager`
    - Methodes : `JWTManager::create()` : Crée un token JWT pour l'utilisateur avec un payload (expiration incluse).

- Erreur: `validation des données`, `message d'erreur`, `retour JSON`
    - Methodes:

- `JsonResponse::serialize()` : Sérialise les erreurs en format JSON pour les renvoyer dans la réponse HTTP.
- `count()` : Vérifie le nombre d'erreurs de validation.

2. **Fonction “LoginController”:#pour connection**

- `login()`
  - **Validation** : Vérification du nom d'utilisateur et du mot de passe.
  - **Envoi du PIN** : Un code PIN est généré et envoyé par email.
  - Gestion des tentatives échouées (limitation à 3 tentatives).

- `pin_verify()`
  - **Validation** : Vérification du PIN et gestion des tentatives échouées.
  - **Génération du token** : Si le PIN est correct, un token JWT est généré et retourné.

3. Fonction “UserController”:#activites user

-

Collection postman

1. Inscription :

- Utilisez l'endpoint `/api/inscription` pour créer un utilisateur.
  - **Méthode** : `POST`
  - **URL** : `{{base_url}}/api/inscription`
  - **Body** (en format JSON) :

```
{
  "email": "user@example.com",
  "username": "username123",
  "password": "password123"
}
```

2. **Authentification** :

- Utilisez l'endpoint `/api/login` pour envoyer un PIN à l'utilisateur après une tentative de connexion.
  - **Méthode** : `POST`
  - **URL** : `{{base_url}}/api/login`
  - **Body** (en format JSON) :

```
{
  "username": "username123",
  "password": "password123"
}
```

3. **Vérification du PIN** :

- Une fois que l'utilisateur a reçu le code PIN par email, il devra le valider via l'endpoint `/api/pin_verification`.
  - **Méthode** : `POST`
  - **URL** : `{{base_url}}/api/pin_verification`
  - **Body** (en format JSON) :

```
{
  "pin": 12345
}
```

4. **Vérification de l'email** :

- Après l'inscription, l'utilisateur recevra un email avec un lien de vérification. Utilisez cet endpoint pour valider l'email.
    - **Méthode** : `GET`
    - **URL** : `{{base_url}}/api/verification/{{token}}`
    - Remplacez `{{token}}` par le token obtenu dans l'email.

Instruction pour lancer API:

Pour lancer votre **API Symfony**, commencez par **cloner** votre projet et **installer les dépendances** avec la commande `composer install`.

Ensuite, configurez le fichier `.env` pour les informations de connexion à la **base de données** et à d'autres services comme le **mailer**.

Créez la base de données avec `php bin/console doctrine:database:create` et appliquez les **migrations** si nécessaire avec `php bin/console doctrine:migrations:migrate`.

Vous pouvez également créer le **schéma de la base de données** en exécutant `php bin/console doctrine:schema:create`.

Ensuite, pour vider le cache, exécutez `php bin/console cache:clear` pour assurer que toutes les modifications prennent effet. Lancez le **serveur Symfony** avec `symfony server:start` pour accéder à votre API via `http://127.0.0.1:8000`.

Utilisez `php bin/console debug:router` pour vérifier que les **routes** de l'API sont correctement configurées. Pour tester l'API, importez la **collection Postman** et configurez un environnement dans Postman, en utilisant des variables comme `base_url`, `username`, et `password`.

En cas d'erreur, consultez les **logs** dans `var/log/dev.log` ou utilisez le **profiler Symfony** à `http://127.0.0.1:8000/_profiler` pour le débogage.

Enfin, vous pouvez consulter la **documentation Symfony** et des bundles comme **LexikJWTAuthenticationBundle** pour l'authentification JWT.