



Y

SYMFON



Sommaire

- Introduction
- Installation de postman
- Création de projet
- Création d'endpoints
- Gestions des entités
- Sérialisation et désérialisation
- Gestions des requêtes
- Sécurité de l'API



Introduction

QU'EST-CE QU'UNE API ?

Une API (Application Programming Interface) est un ensemble de règles et de protocoles qui permet à différentes applications de communiquer entre elles.

Un intermédiaire qui facilite les échanges de données ou de fonctionnalités entre deux systèmes, même s'ils sont développés dans des langages de programmation différents.

Les APIs jouent un rôle crucial dans la séparation du back-end et du front-end. Cette séparation permet aux développeurs front-end de construire des applications réactives et interactives tout en se connectant à des services ou à des bases de données via l'API, sans avoir besoin de connaître les détails du back-end.

Grâce à ce principe, une API permet également de rendre les services disponibles à différents types de clients : applications web, mobiles, ou autres systèmes.

LES TYPES D'API

REST (Representational State Transfer)

- Le type d'API le plus courant. Utilise les méthodes HTTP (GET, POST, PUT, DELETE) pour accéder et manipuler les ressources.
- Repose sur une structure simple basée sur des URL, permettant de manipuler des données au format JSON ou XML.
- Respecte les principes de stateless (sans état), ce qui signifie que chaque requête contient toutes les informations nécessaires pour être traitée.

LES TYPES D'API

SOAP (Simple Object Access Protocol)

- Protocole plus ancien, principalement basé sur XML pour l'échange de données. Utilisé dans des environnements nécessitant une sécurité et une fiabilité renforcées (exemple : services bancaires).

GraphQL

- Développé par Facebook, GraphQL permet aux clients de spécifier exactement les données qu'ils souhaitent recevoir.
- Offre une flexibilité accrue, permettant de récupérer uniquement les champs nécessaires, ce qui optimise la quantité de données transférées.

LES PRINCIPES D'UNE API REST

Endpoints :

- Les endpoints sont des URL spécifiques qui permettent d'accéder à des ressources particulières. Chaque ressource est identifiée par un identifiant unique.
Exemple d'endpoint : `https://api.exemple.com/users/1` pour accéder à l'utilisateur ayant l'ID 1.

Méthodes HTTP :

- GET : Récupérer des données (lecture seule).
- POST : Envoyer des données pour créer une nouvelle ressource.
- PUT : Mettre à jour une ressource existante.
- DELETE : Supprimer une ressource.

LES PRINCIPES D'UNE API REST

Codes de statut HTTP :

Les API REST utilisent des codes de statut pour indiquer le résultat d'une requête.

- 200 OK : Requête réussie.
- 201 Created : Ressource créée avec succès.
- 404 Not Found : Ressource non trouvée.
- 500 Internal Server Error : Erreur côté serveur.

Liste complète de tous les codes de statut HTTP : [ici](#)

AVANTAGES DES APIS

Modularité : Permet de découper une application en plusieurs services indépendants. Chaque service peut être développé, maintenu et mis à jour séparément.

Interopérabilité : Les APIs permettent aux applications d'interagir, même si elles sont développées avec des technologies différentes.

Réutilisabilité : Les APIs peuvent être réutilisées dans différents projets, ce qui facilite le développement et réduit les coûts.

Scalabilité : Les services peuvent être répartis sur plusieurs serveurs, ce qui permet de mieux gérer les pics de trafic.

Les APIs sont devenues essentielles dans le développement d'applications modernes. Elles offrent une manière structurée et efficace de gérer les échanges de données. Dans ce support, nous allons nous concentrer sur la création d'une API REST.

AVANTAGES DES APIS

Modularité : Permet de découper une application en plusieurs services indépendants. Chaque service peut être développé, maintenu et mis à jour séparément.

Interopérabilité : Les APIs permettent aux applications d'interagir, même si elles sont développées avec des technologies différentes.

Réutilisabilité : Les APIs peuvent être réutilisées dans différents projets, ce qui facilite le développement et réduit les coûts.

Scalabilité : Les services peuvent être répartis sur plusieurs serveurs, ce qui permet de mieux gérer les pics de trafic.

Les APIs sont devenues essentielles dans le développement d'applications modernes. Elles offrent une manière structurée et efficace de gérer les échanges de données. Dans ce support, nous allons nous concentrer sur la création d'une API REST.

Installation Postman

POSTMAN C'EST QUOI

Postman est un outil qui permet de tester et de manipuler les APIs REST de manière simple et efficace. Grâce à son interface intuitive, il offre la possibilité d'envoyer des requêtes HTTP (GET, POST, PUT, DELETE, etc.) aux endpoints d'une API et de visualiser les réponses renvoyées par celle-ci, notamment au format JSON.

Postman est un outil essentiel pour les développeurs qui souhaitent tester rapidement leur API sans avoir besoin de créer une interface utilisateur.

Pourquoi utiliser Postman ?

- Tester facilement les différentes méthodes HTTP (GET, POST, PUT, DELETE).
- Visualiser les réponses des endpoints sous différents formats (JSON, XML).
- Simuler les requêtes envoyées par le front-end, tout en ayant un contrôle complet sur les paramètres et les headers.
- Debugger les API en examinant les requêtes et les réponses directement.

INSTALLATION

Pour installer Postman :

Téléchargement : Rendez-vous sur le site officiel <https://www.postman.com/downloads/> .

Download Postman

Download the app to get started using the Postman API Platform today. Or, if you prefer a browser experience, you can try the web version of Postman.

The Postman app

Download the app to get started with the Postman API Platform.

Windows 64-bit

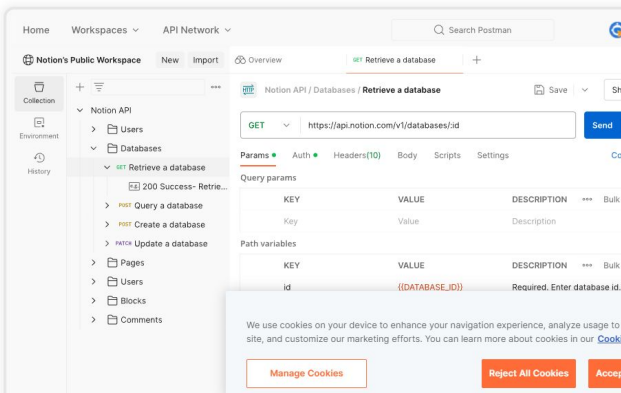
By downloading and using Postman, I agree to the [Privacy Policy](#) and [Terms](#).

[Release Notes](#) →

Not your OS? Download for Mac ([Intel Chip](#), [Apple Chip](#)) or Linux ([x64](#), [arm64](#))

Postman on the web

Access the Postman API Platform through your web browser. Create a free account, and you're in.



Création Projet

CRÉATION PROJET

Deux moyens pour créer vos projets api :

Ouvrez votre terminal et rendez-vous à l'endroit où vous souhaitez créer votre projet.

Avec composer :

composer create-project symfony/skeleton:"6.4.*" my_project_directory

Explication :

- symfony/skeleton : Le squelette de base pour un nouveau projet Symfony.
- "6.4.*" : Spécifie la version LTS 6.4 de Symfony.
- my_project_directory : Nom du dossier de votre projet.

Vous pouvez également utiliser l'outil en ligne de commande Symfony :

symfony new my_project_directory --version=lts

Explication :

- symfony new : Crée un nouveau projet Symfony.
- my_project_directory : Nom du dossier de votre projet.
- --version=lts : Indique que le projet doit utiliser la version LTS (dans ce cas, la 6.4.*).

CRÉATION PROJET

Deux moyens pour créer vos projets api :

Ouvrez votre terminal et rendez-vous à l'endroit où vous souhaitez créer votre projet.

Avec composer :

composer create-project symfony/skeleton:"6.4.*" my_project_directory

Explication :

- symfony/skeleton : Le squelette de base pour un nouveau projet Symfony.
- "6.4.*" : Spécifie la version LTS 6.4 de Symfony.
- my_project_directory : Nom du dossier de votre projet.

Vous pouvez également utiliser l'outil en ligne de commande Symfony :

symfony new my_project_directory --version=lts

Explication :

- symfony new : Crée un nouveau projet Symfony.
- my_project_directory : Nom du dossier de votre projet.
- --version=lts : Indique que le projet doit utiliser la version LTS (dans ce cas, la 6.4.*).

INSTALLATION DES BUNDLES

Le skeleton de Symfony est une version minimaliste, idéale pour commencer un projet d'API, mais il ne contient que les composants de base. Pour développer une API complète, nous avons besoin d'ajouter certains bundles

Doctrine ORM : Gérer les entités et la base de données

Pour interagir avec une base de données et manipuler les entités, nous devons installer Doctrine ORM.

`composer require symfony/orm-pack`

Cette commande installe Doctrine et les outils nécessaires pour interagir avec une base de données relationnelle.

INSTALLATION DES BUNDLES

Maker Bundle : Génération automatique de code Le Maker Bundle est un outil extrêmement pratique qui permet de générer du code répétitif (entités, contrôleurs, etc.) et facilite ainsi le développement d'une API.

Commande d'installation :

`composer require symfony/maker-bundle --dev`

Cette commande installe le Maker Bundle, uniquement pour l'environnement de développement.

INSTALLATION DES BUNDLES

Serializer : Transformer les objets en JSON, Le Serializer est un composant essentiel pour une API, car il permet de convertir les objets PHP (comme des entités) en JSON, qui est le format utilisé pour la communication entre le back-end et le front-end.

Commande d'installation :

composer require symfony/serializer-pack

Le Serializer sera utilisé pour convertir les données renvoyées par votre API en JSON, et inversement pour décoder les requêtes entrantes.

PRÉPARATION DE L'ENVIRONNEMENT

Pour créer la base de données :

- Ouvrir le fichier « .env » qui se trouve à la racine du projet et modifier la propriété DATABASE_URL avec les bons paramètres (user_name, password, hoost, db_name)

```
DATABASE_URL=mysql://root:@127.0.0.1:3306/db_hello_world?serverVersion=5.7
```

- Lancer la commande suivante (toujours se positionner sur la racine du projet):

php bin/console doctrine:database:create

```
$ php bin/console doctrine:database:create  
Created database 'db_hello_world' for connection named default
```

Créations Endpoints

CRÉATIONS D'ENDPOINTS

Pour créer des endpoints dans Symfony, nous utilisons les contrôleurs. Chaque méthode du contrôleur correspond à un endpoint de l'API qui peut répondre à différentes requêtes HTTP comme GET, POST, PUT, ou DELETE.

Commande pour générer un nouveau contrôleur dédié à l'API :

php bin/console make:controller ApiController

Cette commande génère automatiquement un fichier ApiController.php dans le répertoire src/Controller. Nous allons ensuite y ajouter des méthodes pour les différentes requêtes HTTP.

METHODE GET

GET est utilisé pour récupérer des données d'un serveur. Dans une API, il permet de renvoyer des informations sous forme de JSON.

Exemple d'utilisation : Récupérer la liste des articles, utilisateurs, ou tout autre ensemble de données.

Dans Symfony, nous utilisons des routes (`#[Route]`) et une méthode de contrôleur dédiée pour répondre aux requêtes GET.

METHODE GET

```
#[Route('/api')]
class ApiController extends AbstractController
{
    no usages
    #[Route('/items', name: 'api_get_items', methods: ['GET'])]
    public function getItem(): JsonResponse
    {
        $items = [
            ['id' => 1, 'name' => 'Item 1'],
            ['id' => 2, 'name' => 'Item 2'],
        ];

        return $this->json($items);
    }
}
```

`#[Route]` :

Cette annotation définit l'URL de l'endpoint (/items) et donne un nom unique à la route (api_get_items).

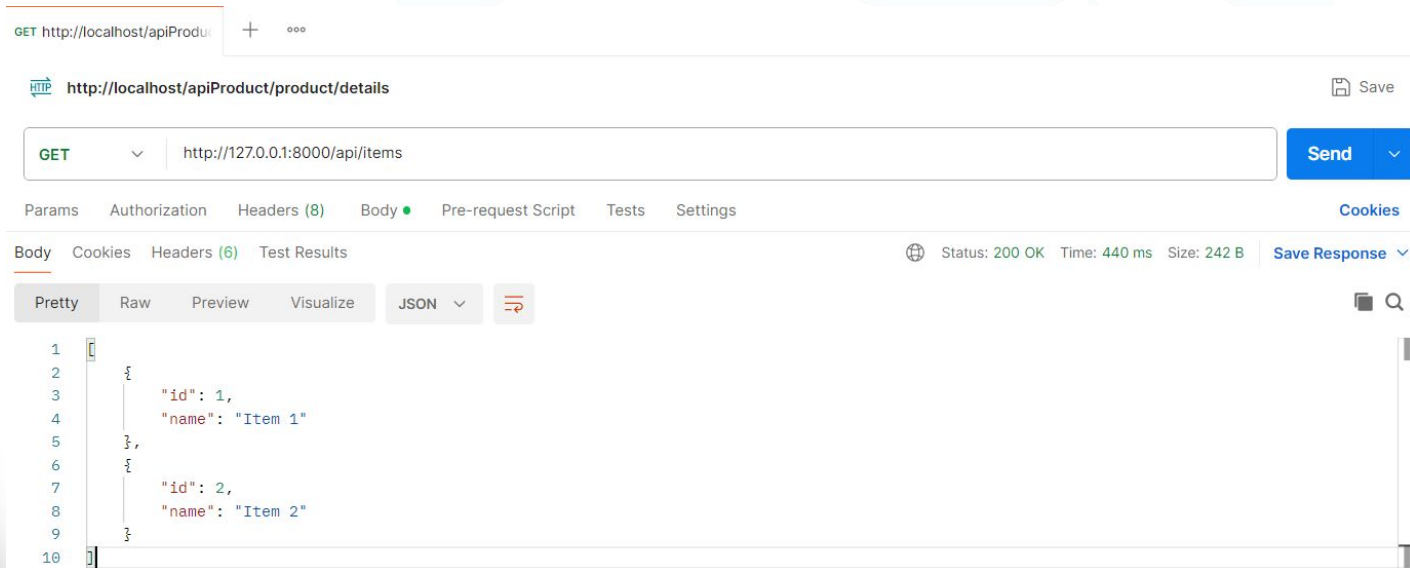
`methods: ['GET']` :

Spécifie que cette route répond uniquement aux requêtes HTTP de type GET. C'est essentiel pour que Symfony sache comment traiter cette requête.

`$this->json()` : Convertit le tableau \$items en réponse JSON et le renvoie au client.

METHODE GET

Tester l'url suivant sur Postman :
<http://127.0.0.1:8000/api/items>



GET http://localhost/apiProduct

HTTP http://localhost/apiProduct/product/details Save

GET http://127.0.0.1:8000/api/items Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (6) Test Results Status: 200 OK Time: 440 ms Size: 242 B Save Response

Pretty Raw Preview Visualize JSON

```

1  [
2    {
3      "id": 1,
4      "name": "Item 1"
5    },
6    {
7      "id": 2,
8      "name": "Item 2"
9    }
10 ]

```

METHODE POST

POST permet de créer de nouvelles données sur le serveur. Les informations sont généralement envoyées sous forme de JSON dans le corps de la requête.

```
#[Route('/items', name: 'api_post_item', methods: ['POST'])]  
public function postItem(Request $request): JsonResponse  
{  
    $data = json_decode($request->getContent(), associative: true);  
  
    // Traitement pour créer un nouvel élément (simulé ici)  
    $newItem = ['id' => 3, 'name' => $data['name']];  
  
    return $this->json($newItem, status: 201);  
}
```

methods: ['POST'] :

Indique que cette méthode de contrôleur répond aux requêtes POST. Les autres types de requêtes seront ignorés.

\$request->getContent() :

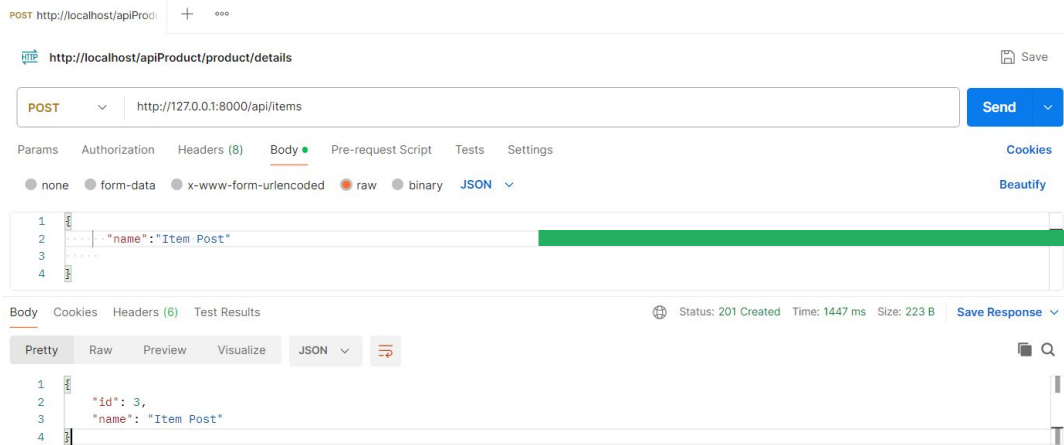
Récupère le contenu brut de la requête, généralement au format JSON, et le décode en tableau PHP.

\$this->json() :

Retourne la nouvelle ressource créée avec le code de statut 201 (Created)

METHODE POST

Tester l'url suivant sur Postman :
<http://127.0.0.1:8000/api/items>



Il faut rajouter dans
le body de la
requête un objet
Json contenant les
données.

METHODE PUT

PUT est utilisé pour mettre à jour des données existantes. On envoie des données dans le corps de la requête, souvent au format JSON.

Exemple d'utilisation : Mettre à jour les informations d'un utilisateur, modifier les détails d'un produit, etc.

L'annotation `[methods: ['PUT']]` spécifie que la route ne doit répondre qu'aux requêtes PUT.

METHODE PUT

```
#[Route('/items/{id}', name: 'api_put_item', methods: ['PUT'])]  
public function putItem(int $id, Request $request): JsonResponse  
{  
    $data = json_decode($request->getContent(), associative: true);  
  
    // Traitement pour mettre à jour l'élément (simulé ici)  
    $updatedItem = ['id' => $id, 'name' => $data['name']];  
  
    return $this->json($updatedItem);  
}
```

#[Route] :

Inclut un paramètre dynamique {id} dans l'URL, permettant de cibler un élément spécifique.

methods: ['PUT'] :

Cette méthode répond uniquement aux requêtes HTTP de type PUT.

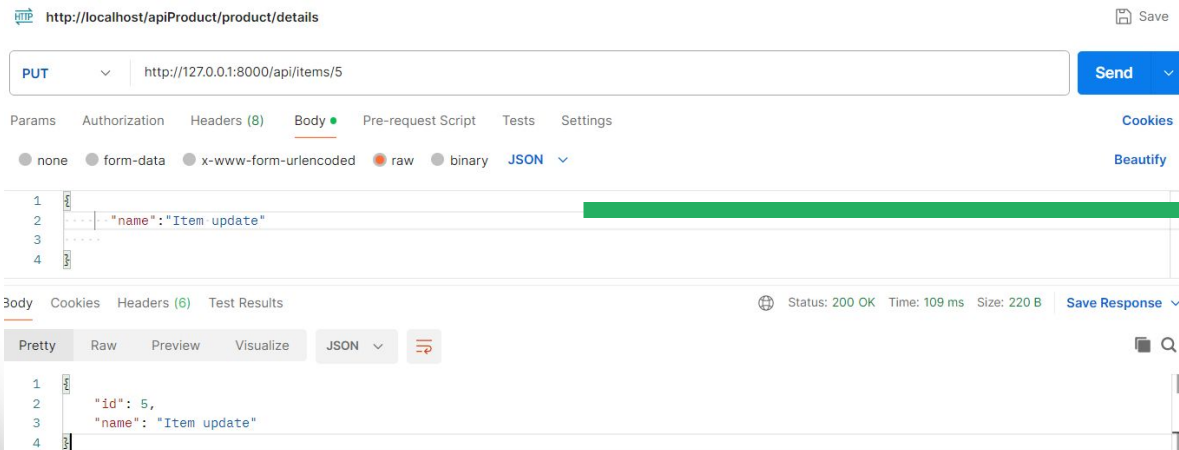
\$request->getContent() :

Récupère et décode les nouvelles données envoyées dans le corps de la requête.

\$this->json() : Renvoie l'élément mis à jour sous forme de JSON.

METHODE PUT

Tester l'url suivant sur Postman :
<http://127.0.0.1:8000/api/items/5>



Il faut rajouter dans le body de la requête un objet Json contenant les données.

METHODE DELETE

DELETE permet de supprimer des ressources sur le serveur.

Exemple d'utilisation : Supprimer un utilisateur, retirer un article de la base de données, etc.

L'annotation [methods: ['DELETE']] indique que la méthode de contrôleur répond uniquement aux requêtes DELETE.

```
#[Route('/items/{id}', name: 'api_delete_item', methods: ['DELETE'])]  
public function deleteItem(int $id): JsonResponse  
{  
    // Traitement pour supprimer l'élément (simulé ici)  
    return $this->json(['message' => 'Item deleted'], status: 204);  
}
```

#[Route] :

Utilise {id} pour identifier l'élément à supprimer.

methods: ['DELETE'] :

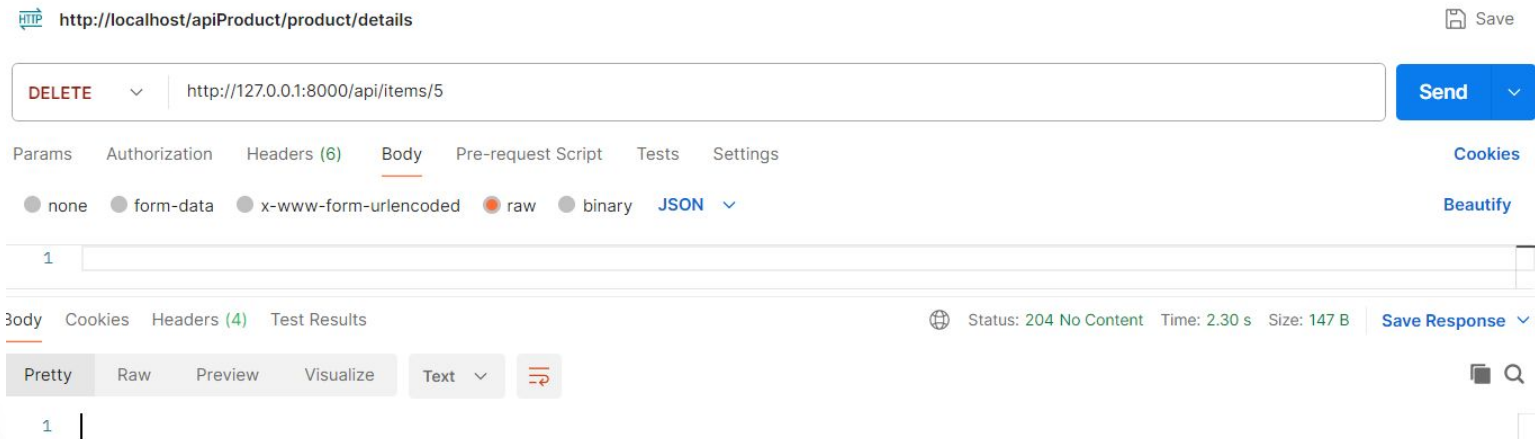
Précise que cette méthode de contrôleur répond uniquement aux requêtes HTTP de type DELETE.

return \$this->json() :

Retourne un message JSON et un code de statut 204 pour indiquer que l'opération s'est bien déroulée et qu'aucun contenu n'est renvoyé.

METHODE DELETE

Tester l'url suivant sur Postman :
<http://127.0.0.1:8000/api/items/5>



The screenshot shows the Postman interface for a DELETE request. The URL bar at the top displays `http://localhost/apiProduct/product/details`. Below it, the request configuration shows the method **DELETE** and the URL `http://127.0.0.1:8000/api/items/5`. The **Body** tab is selected, showing a single line of text: `1`. The response section at the bottom shows a status of **204 No Content**, a time of **2.30 s**, and a size of **147 B**. The response body is empty, with a single line of text: `1`.

Gestion des Entités

ENTITÉS

Une entité est une classe PHP qui représente un objet métier ou un concept du monde réel dans votre application. Dans le contexte de Symfony, une entité est utilisée pour définir des modèles de données et correspond généralement à une table dans la base de données.

Pourquoi utiliser des entités ?

- **Modélisation des données** : Les entités permettent de structurer les données sous forme d'objets PHP.
Par exemple, une entité "Auteur" peut représenter un auteur avec des attributs tels que "nom", "biographie", etc.
- **ORM (Object Relational Mapping)** : Symfony utilise Doctrine pour mapper ces entités aux tables de la base de données, simplifiant ainsi les interactions avec les données sans écrire de SQL. Doctrine va automatiquement gérer les opérations d'insertion, de mise à jour et de suppression de données.

ENTITÉS

- **Attributs** : Chaque entité possède des propriétés qui correspondent aux colonnes de la table dans la base de données.
 Par exemple, une entité Livre peut avoir des attributs comme titre, datePublication, etc.
- **Annotations** : Les entités utilisent des annotations (ou attributs PHP) pour indiquer à Doctrine comment mapper la classe et ses propriétés à la base de données.
 Cela inclut des informations comme le type de chaque champ et les relations entre entités.
- **Relations** : Doctrine permet de définir des relations entre plusieurs entités, comme "Un auteur écrit plusieurs livres" (OneToMany) ou "Un livre peut avoir plusieurs auteurs" (ManyToMany).

QUELQUES ANNOTATIONS

Symfony, via Doctrine ORM, utilise des annotations pour définir les propriétés des entités, leur mapping avec la base de données, et les relations entre elles. Voici une liste des principales annotations que vous rencontrerez dans vos projets Symfony.

- **@ORM\Entity :**

Définit la classe comme une entité, ce qui signifie qu'elle est mappée à une table de la base de données.

```
1 namespace App\Entity;  
2  
3 use Doctrine\ORM\Mapping as ORM;  
4  
5 no usages  
6  
7 #[ORM\Entity]  
8  
9 class Authors  
10 {  
11  
12 }  
13 }
```

QUELQUES ANNOTATIONS

- **@ORM\Table :**

Permet de spécifier des options sur la table associée à l'entité, comme le nom de la table ou des index.

Exemple :

```
3 namespace App\Entity;
4 use Doctrine\ORM\Mapping as ORM;
5
6 #[ORM\Entity]
7 #[ORM\Table(name: "author")]
8 class Authors
9 {
10 }
```

- **@ORM\Id :**

Indique que la propriété associée est la clé primaire de l'entité.

Exemple :

```
class Authors
{
    no usages
    #[ORM\Id]
    private ?int $id = null;
}
```

QUELQUES ANNOTATIONS

- **@ORM\GeneratedValue :**

Utilisée avec @ORM\Id, cette annotation indique que la valeur de la clé primaire est générée automatiquement (par auto-incrémentation ou via une séquence).

Exemple :

```
class Authors
{
    no usages
    #[ORM\Id]
    #[ORM\GeneratedValue]
    private ?int $id = null;
}
```

- **@ORM\Column :**

Définit une colonne de la base de données. On peut spécifier le type, la longueur et si elle peut être nulle.

Exemple :

```
no usages
#[ORM\Column(type: "string", length: 100)]
private ?string $name = null;
```

QUELQUES ANNOTATIONS ENTRE ENTITÉS

- **@ORM\OneToMany:**

Définit une relation "un à plusieurs" entre deux entités. Une entité peut avoir plusieurs autres entités liées.

```
E  
#[ORM\OneToMany(targetEntity: Book::class, mappedBy: "author")]  
private Collection $books;
```

mappedBy : indique la propriété dans l'entité liée (l'autre côté de la relation) qui fait référence à l'entité actuelle.

targetEntity : indique la classe de l'entité cible (celle avec laquelle la relation est établie).

QUELQUES ANNOTATIONS ENTRE ENTITÉS

- **@ORM\ManyToOne** :

Définit une relation "plusieurs à un". Plusieurs entités peuvent être liées à une seule entité.

Exemple :

```
#[ORM\Entity(repositoryClass: BookRepository::class)]  
class Book  
{  
  
    // autres Propriétés de l'entité  
    no usages  
    #[ORM\ManyToOne(targetEntity: Author::class, inversedBy: "books")]  
    private ?Author $author = null;  
}
```

targetEntity : indique la classe de l'entité liée.

inversedBy : fait référence à la propriété dans l'entité liée qui représente cette relation inverse (l'autre côté).

QUELQUES ANNOTATIONS ENTRE ENTITÉS

- **@ORM\ManyToOne :**

Permet de définir une relation "plusieurs à plusieurs" entre deux entités. Chaque entité peut avoir plusieurs liens avec l'autre.

```
#[ORM\Entity(repositoryClass: BookRepository::class)]
class Book
{
    // autres Propriétés de l'entité
    no usages
    #[ORM\ManyToOne(targetEntity: Tag::class)]
    #[ORM\JoinTable(name: "book_tags")]
    private Collection $tags;
```

targetEntity : indique l'entité liée.

mappedBy (optionnel dans certains cas) : déclare le côté propriétaire de la relation dans l'entité liée.

inversedBy (optionnel) : côté inverse de la relation.

QUELQUES ANNOTATIONS ENTRE ENTITÉS

- **@ORM\JoinColumn :**

Utilisée pour spécifier des options supplémentaires sur une relation, notamment la colonne de la clé étrangère.

```
4 usages
#[ORM\Entity(repositoryClass: BookRepository::class)]
class Book
{
    // autres Propriétés de l'entité
    no usages
    #[ORM\ManyToOne(targetEntity: Category::class)]
    #[ORM\JoinColumn(name: "category_id", referencedColumnName: "id")]
    private ?Category $category = null;
```

name : le nom de la colonne dans la base de données qui contiendra la clé étrangère.

referencedColumnName : la colonne de l'entité cible (normalement la clé primaire) vers laquelle cette clé étrangère fait référence.

QUELQUES ANNOTATIONS ENTRE ENTITÉS

Ces annotations vous permettent de configurer précisément le comportement de vos entités et leurs relations avec la base de données. Elles sont essentielles pour le développement avec Doctrine ORM dans Symfony.

Vous pouvez aussi construire votre entité via le maker bundle en utilisant la commande :

Symfony console make:entity

Ce qui permet de créer votre entité propriété par propriété en sélectionnant directement le bon type de donnée pour chaque attribut

```
PS C:\Env\workspace\formation\novao\demo> symfony console make:entity

Class name of the entity to create or update (e.g. GrumpyChef):
> Category

created: src/Entity/Category.php
created: src/Repository/CategoryRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> title

Field type (enter ? to see all types) [string]:
> ?

Main Types
* string or ascii_string
* text
* boolean
* integer or smallint or bigint
* float

Relationships/Associations
* relation a wizard will help you build the relation
* ManyToOne
```

EXEMPLE DE CRÉATION D'ENTITÉ

Passons a un exemple de création de l'entité auteur et livre via la commande du `make:entity`

- Ouvrez votre terminal à la racine de votre projet Symfony.
- Tapez la commande suivante : ***Symfony console make:entity Author***

```
PS C:\Env\workspace\formation\novao\demo> symfony console make:entity Author
created: src/Entity/Author.php
created: src/Repository/AuthorRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> name

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Author.php

Add another property? Enter the property name (or press <return> to stop adding fields):
```

EXEMPLE DE CRÉATION D'ENTITÉ

On va rajouter un autre attribut , l'attribut biographie de type string non nullable

```
New property name (press <return> to stop adding fields):  
> bio  
  
Field type (enter ? to see all types) [string]:  
>  
  
Field length [255]:  
>  
  
Can this field be null in the database (nullable) (yes/no) [no]:  
>  
  
updated: src/Entity/Author.php
```

Une fois les propriétés définies, l'entité Author et son repository sera générée avec ces propriétés dans la classe correspondante.

EXERCICE DE CRÉATION D'ENTITÉ

Maintenant à vous de jouer , vous allez devoir créer l'entité Book avec les attributs suivants :

- Titre de type string, non nul, longueur max 100
- Prix de type float, non nul
- Description de type string , peut être nul, longueur max 200
- Date de publication de type date , non nul

MIGRATIONS

N'oubliez pas de créer votre base de données si cela n'est pas déjà fait , effectuer la commande
Symfony console doctrine:database:create

Une fois les entités créées, n'oubliez pas de mettre à jour la base de données pour appliquer les changements. Utilisez la commande suivante :

Symfony console make:migration

Symfony console doctrine:migrations:migrate

Sérialisation et Désérialisation

QU'EST-CE QUE LA SÉRIALISATION ET DÉSÉRIALISATION ?

- **Sérialisation** : La transformation d'un objet en un format (souvent JSON ou XML) qui peut être facilement transmis ou stocké.
- **Désérialisation** : Le processus inverse, à savoir la conversion d'un format structuré (JSON, XML) en objet utilisable par le programme.
- **Utilité** : Ces processus permettent de communiquer entre le back-end (Symfony) et le front-end (ou d'autres systèmes) via des données structurées.

Exemple d'application :

- Récupérer des entités (comme Author) sous forme de JSON pour les exposer dans une API.
- Recevoir des données au format JSON depuis une requête pour créer ou modifier une entité.

DÉSÉRIALISATION

La désérialisation est le processus de transformation de données structurées (comme du JSON) en objets PHP. Cela permet de recevoir des données au format **JSON** ou **XML** et de les convertir directement en entités ou objets PHP.

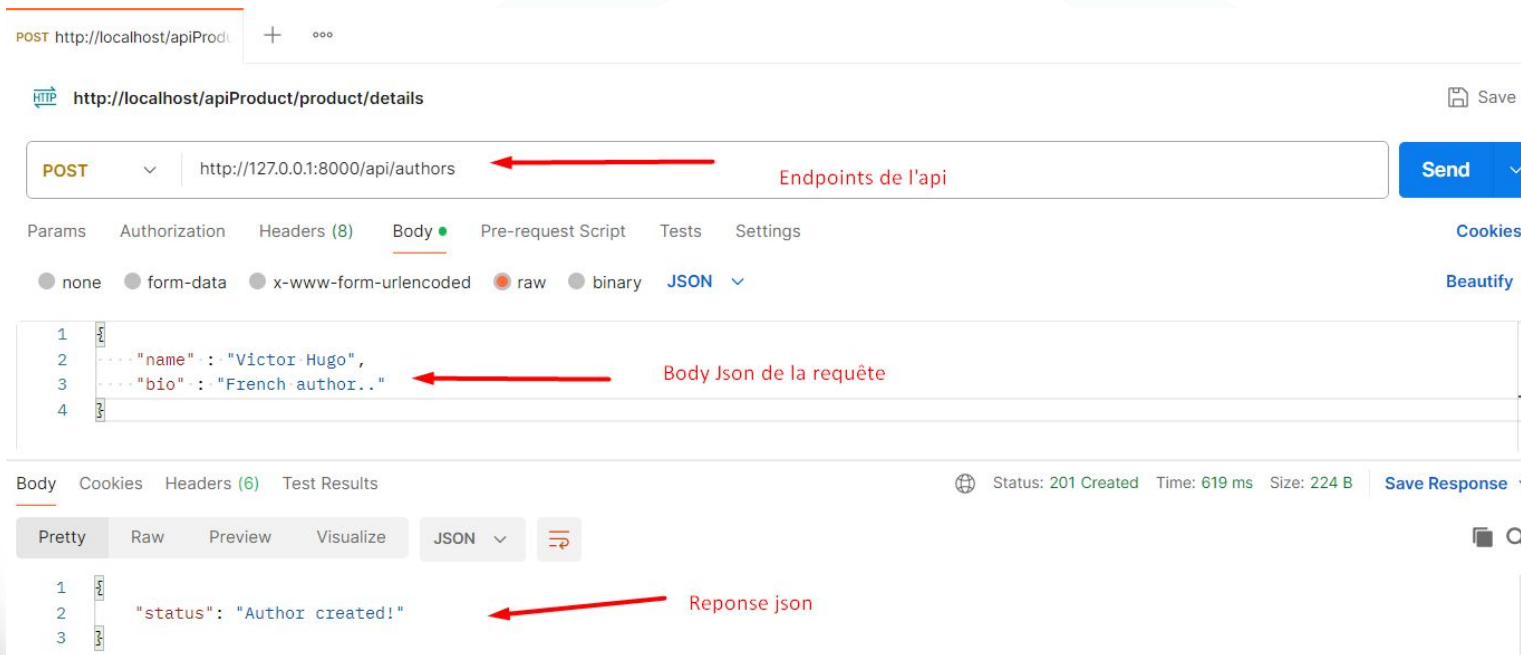
```
#[Route('/api/authors', name: 'create_author', methods: ['POST'])]
public function createAuthor(Request $request, EntityManagerInterface $entityManager): JsonResponse
{
    // Récupération des données JSON du corps de la requête
    $data = json_decode($request->getContent(), associative: true);
    // Créer une nouvelle entité Author
    $author = new Author();
    $author->setName($data['name']);
    $author->setBio($data['bio']);
    // Persister l'auteur dans la base de données
    $entityManager->persist($author);
    $entityManager->flush();
    // Retourner une réponse JSON
    return new JsonResponse(['status' => 'Author created!'], status: 201);
}
```

Récupération des données : Le contenu JSON est récupéré avec `$request->getContent()`, puis il est transformé en tableau associatif grâce à `json_decode($data, true)`.

Assignment des données à l'entité : Les champs du tableau sont utilisés pour définir les propriétés de l'entité Author.

Persistence : L'entité est persistée dans la base de données avec `persist()` et `flush()` de l'**EntityManager**

TEST DE LA REQUÊTE SUR POSTMAN



The screenshot shows the Postman interface for a POST request. The URL bar shows the endpoint `http://127.0.0.1:8000/api/authors`, which is highlighted by a red arrow and labeled "Endpoints de l'api". The request body is set to JSON and contains the following content:

```
1 {
2   "name": "Victor Hugo",
3   "bio": "French author.."
4 }
```

This body content is also highlighted by a red arrow and labeled "Body Json de la requête". The response section shows a status of 201 Created, with a time of 619 ms and a size of 224 B. The response body is displayed in JSON format:

```
1 {
2   "status": "Author created!"
3 }
```

This response content is highlighted by a red arrow and labeled "Reponse json". The interface includes tabs for Params, Authorization, Headers (8), Body (selected), Pre-request Script, Tests, and Settings. The Body tab is further divided into none, form-data, x-www-form-urlencoded, raw, and binary, with JSON selected. The response section also includes tabs for Body, Cookies, Headers (6), and Test Results, with the Body tab selected and the response displayed in JSON format.

DÉSÉRIALISATION AVEC LE SERIALIZER

Symfony offre un composant Serializer puissant, qui permet de sérialiser et désérialiser des objets facilement et de manière flexible.

Installation :

composer require symfony/serializer

```
#[Route('/api/authors', name: 'create_author', methods: ['POST'])]  
public function createAuthor(Request $request, SerializerInterface $serializer, EntityManagerInterface $em): JsonResponse  
{  
    $jsonData = $request->getContent();  
    // Désérialisation du JSON en objet Author  
    $author = $serializer->deserialize($jsonData, type: Author::class, format: 'json');  
    // Sauvegarder l'auteur dans la base de données  
    $em->persist($author);  
    $em->flush();  
    return new JsonResponse(['status' => 'Author created!'], status: 201);  
}
```

DÉSÉRIALISATION AVEC LE SERIALIZER

- **`$request->getContent()`** : Récupère le contenu brut de la requête (généralement en **JSON**).
- **`$serializer->deserialize()`** : Convertit le **JSON** reçu en un objet de type **Author**.
- **`$jsonData`** : Les données **JSON** à désérialiser.
- **`Author::class`** : La classe cible de l'entité.
- **`'json'`** : Le format d'entrée, ici **JSON**.
- **`$em->persist($author)`** : Prépare l'enregistrement de l'entité dans la base de données.
- **`$em->flush()`** : Sauvegarde effectivement l'entité en base de données.

Avec cette désérialisation, vous pouvez facilement convertir des données JSON en entités ou objets PHP, facilitant ainsi la gestion des requêtes API POST dans votre application Symfony.

UTILISATION DES GROUPES DE SÉRIALISATION

Les annotations **@Groups** permettent de définir quelles propriétés de l'entité doivent être exposées lors de la sérialisation.

Cela évite d'exposer des données **sensibles** ou **inutiles** dans les réponses API.

```
#[ORM\Column(length: 255)]  
#[Groups(['author:read'])]  
private ?string $name = null;
```

2 usages

```
#[ORM\Column(length: 255)]  
#[Groups(['author:read'])]  
private ?string $bio = null;
```

Les groupes permettent de contrôler finement les données envoyées. Ici, seules le name et la bio sont exposés dans le groupe author:read.

SÉRIALISATION

- Pour la sérialisation en JSON, Symfony fournit une méthode pratique via **`$this->json()`**.
- Exemple de méthode pour récupérer tous les auteurs sous forme de JSON.

```
namespace App\Controller;

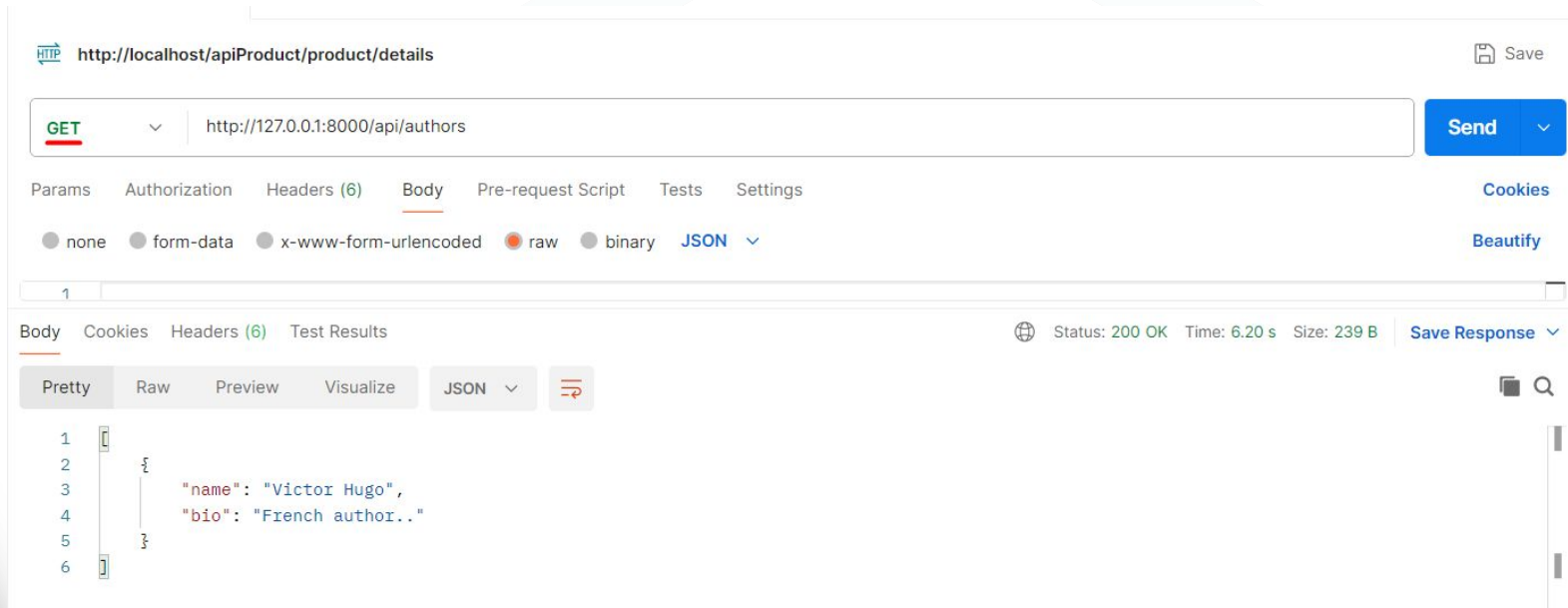
use ...

no usages
class AuthorController extends AbstractController
{
    no usages
    #[Route('/api/authors', name: 'api_authors_get', methods: ['GET'])]
    public function getAllAuthors(AuthorRepository $authorRepository): JsonResponse
    {
        // Récupération de tous les auteurs
        $authors = $authorRepository->findAll();
        // Retourne les données sous forme de JSON
        return $this->json($authors, status: 200, [], ['groups' => 'author:read']);
    }
}
```

La méthode **`findAll()`** de **`AuthorRepository`** permet de récupérer tous les auteurs.

La méthode **`$this->json()`** permet de convertir les entités en JSON, avec une gestion des groupes de sérialisation via **`['groups' => 'author:read']`**.

TEST DE LA REQUÊTE SUR POSTMAN



HTTP <http://localhost/apiProduct/product/details> Save

GET <http://127.0.0.1:8000/api/authors> Send

Params Authorization Headers (6) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON**

1

Body Cookies Headers (6) Test Results Status: 200 OK Time: 6.20 s Size: 239 B Save Response

Pretty Raw Preview Visualize **JSON**

```

1 {
2   "name": "Victor Hugo",
3   "bio": "French author.."
4 }
5
6

```


Gestion des Requêtes

INTRODUCTION À LA GESTION DES REQUÊTES

Dans une API, la gestion des requêtes est essentielle pour traiter les données envoyées par le client et fournir des réponses appropriées.

Lorsqu'un client envoie une requête HTTP à notre API, il peut y inclure des données, souvent sous forme de JSON. Symfony permet de récupérer et traiter ces données de manière efficace, notamment grâce à l'objet Request.

Pour récupérer les données JSON envoyées dans le corps d'une requête, on utilise l'objet Request de Symfony.

Il est possible d'extraire ces données en utilisant la méthode `getContent()` pour manipuler le contenu brut ou `toArray()` pour convertir automatiquement le JSON en tableau associatif.

Le reste ne change pas vraiment comparé à une application web avec symfony.

LE CRUD : CONCEPT FONDAMENTAL DES API

Le **CRUD** est un acronyme qui représente les quatre opérations de base nécessaires pour gérer les données dans une application.

Que signifie **CRUD** ?

- **Create (C) :**

- Action : Ajouter une nouvelle entrée dans la base de données.
- Méthode HTTP : POST
- Exemple : Ajouter un nouvel auteur dans l'application.

- **Read (R) :**

- Action : Lire ou récupérer des données de la base.
- Méthode HTTP : GET
- Exemple : Récupérer la liste de tous les auteurs ou un auteur spécifique.

LE CRUD : CONCEPT FONDAMENTAL DES API

- **Update (U) :**
 - Action : Modifier une entrée existante dans la base de données.
 - Méthode HTTP : PUT ou PATCH
 - Exemple : Mettre à jour les informations d'un auteur.
- **Delete (D):**
 - Action : Supprimer une entrée de la base de données.
 - Méthode HTTP : DELETE
 - Exemple : Supprimer un auteur de l'application.

Le **CRUD** est la base de la gestion des ressources dans une **API**.

Chaque opération correspond à un cycle de vie d'une ressource (création, lecture, mise à jour, suppression).

EXEMPLE CRUD READ

Nous allons voir un exemple de récupération d'un auteur par son id, cet Endpoint de recherche correspond à une lecture (read)

```
#[Route('/api/authors/{id}', methods: ['GET'])]  
public function getAuthorById(int $id, AuthorRepository $authorRepository): JsonResponse  
{  
    $author = $authorRepository->find($id);  
    if (!$author) {  
        return $this->json(['error' => 'Author not found'], status: Response::HTTP_NOT_FOUND);  
    }  
    return $this->json($author);  
}
```

Paramètre int \$id : Représente l'identifiant de l'auteur que l'on souhaite récupérer.

Réponse JSON : Si l'auteur est trouvé, retourne les détails de l'auteur sous forme de JSON.

GET ⌵

Send ⌵

EXEMPLE CRUD CREATE

Nous allons voir revoir l'exemple de création d'un auteur, cet Endpoint de correspond à une création (create)

```
#[Route('/api/authors', name: 'create_author', methods: ['POST'])]  
public function createAuthor(Request $request, SerializerInterface $serializer, EntityManagerInterface $em): JsonResponse  
{  
    $jsonData = $request->getContent();  
    // Désérialisation du JSON en objet Author  
    $author = $serializer->deserialize($jsonData, type: Author::class, format: 'json');  
    // Sauvegarder l'auteur dans la base de données  
    $em->persist($author);  
    $em->flush();  
    return new JsonResponse(['status' => 'Author created!'], status: 201);  
}
```

Récupération des données : Le contenu JSON est récupéré avec `$request->getContent()`.

Assignation des données à l'entité : à l'aide de la méthode `deserialize()` du `Serializer`.

Persistence : L'entité est persistée dans la base de données avec `persist()` et `flush()` de l'`EntityManager`

EXAMPLE CRUD UPDATE

Nous allons voir revoir l'exemple de mise à jour d'un auteur, cet Endpoint de correspond à une mise à jour (update)

```
#[Route('/api/authors/{id}', methods: ['PUT'])]  
public function updateAuthor(  
    int $id, Request $request, AuthorRepository $authorRepository, SerializerInterface $serializer,  
    EntityManagerInterface $entityManager): JsonResponse  
{  
    $author = $authorRepository->find($id);  
    // Vérification de l'existence de l'auteur  
    if (!$author) {  
        return $this->json(['error' => 'Author not found'], status: Response::HTTP_NOT_FOUND);  
    }  
    // Désérialisation des données JSON dans l'objet Author existant  
    $serializer->deserialize($request->getContent(), type: Author::class, format: 'json', ['object_to_populate' => $author]);  
    // Enregistrement des modifications  
    $entityManager->flush();  
    // Retour des détails de l'auteur mis à jour  
    return $this->json($author);  
}
```

EXEMPLE CRUD UPDATE

Paramètre int \$id :

Représente l'identifiant de l'auteur dont les informations doivent être mises à jour.

AuthorRepository :

Utilisé pour récupérer l'auteur de la base de données.

Gestion des erreurs :

Vérifie si l'auteur existe (if (!\$author)) et renvoie un message d'erreur avec un code HTTP 404 (Not Found) si ce n'est pas le cas.

Désérialisation :

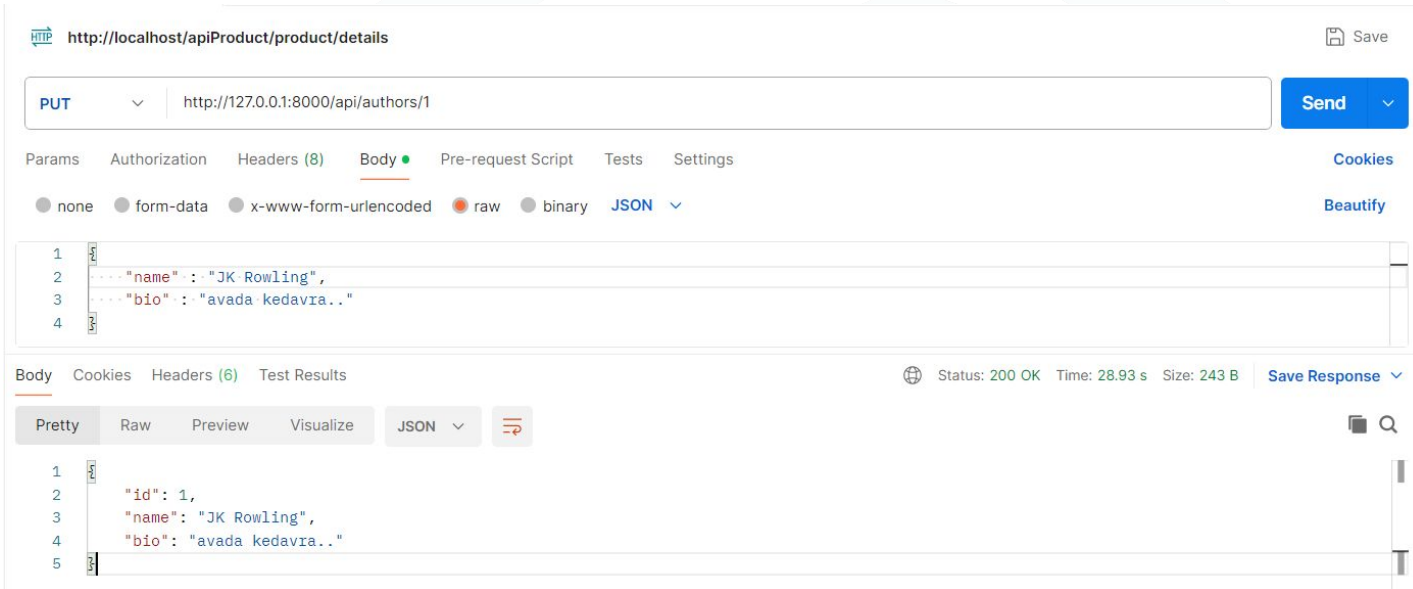
Utilise le Serializer pour désérialiser directement les données JSON dans l'objet Author existant. La clé object_to_populate permet de remplir l'instance existante de l'auteur avec les nouvelles valeurs.

Enregistrement des Changements :

Utilise \$entityManager->flush() pour enregistrer les modifications dans la base de données.

Réponse JSON : Renvoie les détails de l'auteur mis à jour sous forme de JSON.

EXEMPLE CRUD UPDATE



The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost/apiProduct/product/details`
- Method:** PUT
- Request URL:** `http://127.0.0.1:8000/api/authors/1`
- Body (Request):**

```

1 {
2   "name": "JK Rowling",
3   "bio": "avada kedavra.."
4 }
```
- Response:**
 - Status:** 200 OK
 - Time:** 28.93 s
 - Size:** 243 B
 - Body (Response):**

```

1 {
2   "id": 1,
3   "name": "JK Rowling",
4   "bio": "avada kedavra.."
5 }
```

EXEMPLE CRUD DELETE

Nous allons voir revoir l'exemple de suppression d'un auteur, cet Endpoint de correspond à suppression (delete)

```
#[Route('/api/authors/{id}', methods: ['DELETE'])]  
public function deleteAuthor(int $id, AuthorRepository $authorRepository, EntityManagerInterface $entityManager): JsonResponse  
{  
    $author = $authorRepository->find($id);  
    // Vérification de l'existence de l'auteur  
    if (!$author) {  
        return $this->json(['error' => 'Author not found'], status: Response::HTTP_NOT_FOUND);  
    }  
    // Suppression de l'auteur  
    $entityManager->remove($author);  
    $entityManager->flush();  
    // Retour d'un message de confirmation  
    return $this->json(['message' => 'Author deleted successfully'], status: Response::HTTP_NO_CONTENT);  
}
```

EXEMPLE CRUD UPDATE

Paramètre int \$id :

Représente l'identifiant de l'auteur à supprimer.

AuthorRepository :

Utilisé pour récupérer l'auteur de la base de données.

Gestion des erreurs :

Vérifie si l'auteur existe (if (!\$author)) et renvoie un message d'erreur avec un code HTTP 404 (Not Found) si ce n'est pas le cas.

Suppression de l'Auteur :

Utilise \$entityManager->remove(\$author) pour marquer l'auteur pour suppression.
Appelle \$entityManager->flush() pour exécuter la suppression dans la base de données.

Réponse JSON :

Renvoie un message de confirmation que l'auteur a été supprimé avec succès, avec un code HTTP 204 (No Content) pour indiquer qu'aucun contenu supplémentaire n'est renvoyé.

EXEMPLE CRUD UPDATE

DELETE [Send](#)

Params Authorization Headers (6) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary [JSON](#)

[Cookies](#)

[Beautify](#)

1

Body Cookies Headers (4) Test Results 🌐 Status: 204 No Content Time: 1918 ms Size: 147 B [Save Response](#)

Pretty Raw Preview Visualize **Text**

1

EXEMPLE CRUD UPDATE

DELETE ⌵ **Send** ⌵

Params Authorization Headers (6) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** ⌵

⌵

Body Cookies Headers (4) Test Results ⌐ Status: 204 No Content Time: 1918 ms Size: 147 B **Save Response** ⌵

Pretty Raw Preview Visualize **Text** ⌵

1

EXERCICE CRUD

Maintenant à vous de jouer , vous allez devoir mettre en place le CRUD de l'entité Book, Commencer par la création d'un controller BookController

- Prévoir un endpoint de créations d'un livre
- Prévoir un endpoint de récupération d'un livre par son titre
- Prévoir un endpoint de récupérations de plusieurs livres à partir d'une certaine date de parution
- Prévoir un endpoint de mise à jour d'un livre
- Prévoir un endpoint de suppression d'un livre via son id

VALIDATION DES DONNÉES

La validation des données est essentielle pour garantir l'intégrité et la qualité des informations reçues par votre API. Dans Symfony, cela peut être effectué directement dans les entités à l'aide des assertions.

Avant d'utiliser les assertions pour la validation des données, il est nécessaire d'installer le composant Validator de Symfony.

Pour ce faire, exécutez la commande suivante dans votre terminal :

composer require symfony/validator

VALIDATION DES DONNÉES

```
use Symfony\Component\Validator\Constraints as Assert;
```

6 usages

```
#[ORM\Entity(repositoryClass: AuthorRepository::class)]
```

```
class Author
```

```
{
```

1 usage

```
#[ORM\Id]
```

```
#[ORM\GeneratedValue]
```

```
#[ORM\Column]
```

```
private ?int $id = null;
```

2 usages

```
#[ORM\Column(length: 255)]
```

```
#[Groups(['author:read'])]
```

```
#[Assert\NotBlank]
```

```
#[Assert\Length(min: 2)]
```

```
private ?string $name = null;
```

#[Assert\NotBlank]:

Garantit que le champ name n'est pas vide.

#[Assert\Length(min: 2)]:

Assure que le name a au moins 2 caractères.

Liste des Assertions disponible :

<https://symfony.com/doc/6.4/reference/constraints.html>

VALIDATION DES DONNÉES

Pour valider une entité, vous pouvez utiliser le service ValidatorInterface. Voici comment procéder

```
#[Route('/api/authors', name: 'create_author', methods: ['POST'])]  
public function createAuthor(Request $request, SerializerInterface $serializer, EntityManagerInterface $em, ValidatorInterface $validator): JsonResponse {  
    $jsonData = $request->getContent();  
    // Désérialisation du JSON en objet Author  
    $author = $serializer->deserialize($jsonData, type: Author::class, format: 'json');  
    // Validation de l'auteur  
    $errors = $validator->validate($author);  
    if (count($errors) > 0) {  
        // Gérer les erreurs de validation  
        return $this->json(['errors' => (string) $errors], status: 400);  
    }  
    // Sauvegarder l'auteur dans la base de données  
    $em->persist($author);  
    $em->flush();  
    return new JsonResponse(['status' => 'Author created!'], status: 201);  
}
```

VALIDATION DES DONNÉES

Le ValidatorInterface est injecté dans la méthode pour permettre la validation des entités.

Validation des Données :

Après la désérialisation, l'objet Author est validé avec `$validator->validate($author)`.
Si des erreurs sont détectées, une réponse JSON contenant les erreurs est renvoyée avec le code HTTP 400.

Sauvegarde :

Si aucune erreur n'est trouvée, l'auteur est persistant dans la base de données comme avant.
Cette approche garantit que seules les entités valides sont enregistrées dans la base de données, améliorant ainsi l'intégrité des données de votre application.

Sécurité de l'API

INTRODUCTION

La sécurité est cruciale pour protéger l'accès aux données et fonctionnalités sensibles.

Symfony permet de gérer cette sécurité via la configuration des firewalls et des contrôles d'accès dans `security.yaml`.

Nous allons configurer l'authentification avec JWT et sécuriser les endpoints grâce à l'utilisation des rôles et des Voters.

Pour configurer la sécurité de notre API avec JWT, nous avons besoin de certains bundles :

- `lexik/jwt-authentication-bundle` pour la gestion des tokens JWT :

Composer require lexik/jwt-authentication-bundle

- `symfony/security-bundle` pour la gestion des utilisateurs et des rôles :

Composer require security

C'EST QUOI JWT ?

JWT signifie **JSON Web Token**, un standard ouvert (RFC 7519) utilisé pour transférer des informations de manière sécurisée entre deux parties.

Il est généralement utilisé pour **authentifier** un utilisateur et lui permettre d'accéder à des ressources protégées dans une **API**.

Le JWT est composé de **trois parties** encodées en **base64** et séparées par des points (.) :

- **Header** (En-tête)
- **Payload** (Corps)
- **Signature**

LES COMPOSANTS D'UN JWT

Header (En-tête):

Contient deux informations clés :

alg : l'algorithme utilisé pour signer le token (par exemple, HS256 ou RS256).

typ : le type de token (qui est toujours "JWT").

Payload (Corps):

Contient les claims, c'est-à-dire les informations que l'on souhaite transmettre. Ces informations peuvent inclure :

L'identifiant de l'utilisateur.

Ses rôles ou permissions.

La date d'expiration du token (exp).

Il est important de noter que le payload n'est pas chiffré, donc les informations ne doivent pas être sensibles.

Signature:

La signature est utilisée pour vérifier que le token n'a pas été altéré. Elle est générée en prenant le header, le payload, et une clé secrète convenue, puis en les combinant avec l'algorithme de signature spécifié.



CRÉATION DE L'ENTITÉ USER AVEC LE MAKER BUNDLE

Pour commencer, nous devons créer une entité User qui sera utilisée pour l'authentification de notre API.

Commande pour générer l'entité User :

symfony console make:user

L'entité générée implémente l'interface UserInterface et permet de gérer les rôles et le mot de passe.

```
#[ORM\Entity(repositoryClass: UserRepository::class)]
#[ORM\UniqueConstraint(name: 'UNIQ_IDENTIFIER_EMAIL', fields: ['email'])]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    1 usage
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    3 usages
    #[ORM\Column(length: 180)]
    private ?string $email = null;

    2 usages
    #[ORM\Column]
    private array $roles = [];
```

AJOUT D'UN UTILISATEUR VIA API

Après avoir créé l'entité User, nous créons un endpoint pour enregistrer un nouvel utilisateur dans la base de données.

Commande pour créer un contrôleur :

symfony console make:controller UserController

```
class UserController extends AbstractController
{
    no usages
    #[Route('/api/register', name: 'api_register', methods: ['POST'])]
    public function register(Request $request, UserPasswordHasherInterface $passwordHasher, EntityManagerInterface $em): JsonResponse
    {
        $data = json_decode($request->getContent(), associative: true);
        $user = new User();
        $user->setEmail($data['email']);
        $user->setPassword($passwordHasher->hashPassword($user, $data['password']));
        $user->setRoles(['ROLE_USER']);
        $em->persist($user);
        $em->flush();
        return new JsonResponse(['status' => 'User created'], status: 201);
    }
}
```


TEST DE L'AJOUT DE L'UTILISATEUR

POST

http://127.0.0.1:8000/api/register

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

JSON

Cookies

Beautify

```

1
2  ... "email": "test@test.fr",
3  ... "password": "123456789"
4

```

body

Cookies

Headers (6)

Test Results

Status: 201 Created

Time: 3.43 s

Size: 221 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

1

2 "status": "User created"

3

AUTHENTIFICATION JWT

L' Authentification JWT avec `lexik/jwt-authentication-bundle`

JWT (JSON Web Token) permet de sécuriser les API en utilisant des jetons d'authentification.

Pour fonctionner, Lexik a besoin de générer des clefs. Il y a une clef publique et une clef privée.

La clé publique permet d'encoder le token généré, et la clé privée permet de le décoder.

Comme la clé privée est... privée, même si quelqu'un intercepte le token, sans la clé privée il ne saura pas comment le décoder, et ne pourra donc rien en faire.

Voici comment créer ces clefs, toujours en ligne de commande :

```
mkdir -p config/jwt
```

```
openssl genpkey -out config/jwt/private.pem -algorithm RSA -pkeyopt rsa_keygen_bits:4096
```

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

AUTHENTIFICATION JWT

Une fois que la génération de clefs est effectuée vous allez devoir faire quelques petites modifications dans votre fichier d'environnement « .env »

```
###> 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=321c9c93fe67d5e9a23e3248075f03260471672636ca34009e1807a55001a0f0  
###< lexik/jwt-authentication-bundle ###  
|
```

← Modifier avec la passphrase que vous avez utiliser durant création des clefs

AUTHENTIFICATION JWT

Pour utiliser **JSON Web Tokens** (JWT) avec Symfony , nous devons configurer le fichier **security.yaml** pour permettre l'authentification via JWT avec le bundle **LexikJWTAuthenticationBundle**

```
# config/packages/security.yaml
security:
    enable_authenticator_manager: true

    # Gestion des hashages de mots de passe pour les utilisateurs
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'

    # Définition des fournisseurs d'utilisateurs (User Providers)
    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
```

enable_authenticator_manager :

Active le gestionnaire d'authentificateurs modernes dans Symfony .

password_hashers :

Définit les règles pour le hachage des mots de passe, utilisant ici une méthode automatique.

providers :

Le fournisseur d'utilisateurs est configuré pour charger les utilisateurs depuis l'entité User en utilisant leur adresse email comme identifiant.

AUTHENTIFICATION JWT

```
# Définition des firewalls pour gérer les différentes méthodes d'authentification
firewalls:
  # Firewall pour le mode de développement, sans sécurité
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false

  # Firewall pour le login, gestion des requêtes de connexion
  login:
    pattern: ^/api/login
    stateless: true
    provider: app_user_provider
    json_login:
      check_path: /api/login_check
      username_path: email
      password_path: password
      success_handler: lexik_jwt_authentication.handler.authentication_success
      failure_handler: lexik_jwt_authentication.handler.authentication_failure
```

dev :

Désactive la sécurité pour les actifs frontaux (JS, CSS, etc.) et les outils de développement.

login :

Gère l'authentification via une requête POST avec des informations d'identification JSON (/api/login).

AUTHENTIFICATION JWT

```
# Firewall pour protéger l'API avec JWT
api:
  pattern: ^/api
  stateless: true
  jwt: ~

# Firewall principal
main:
  lazy: true
  provider: app_user_provider

# Règles d'accès (access_control) pour sécuriser les routes
access_control:
  - { path: ^/api/register, roles: PUBLIC_ACCESS }
  - { path: ^/api/login, roles: PUBLIC_ACCESS }
  - { path: ^/api, roles: IS_AUTHENTICATED_FULLY }
```

api :

Gère la protection des routes API, requiert un token JWT valide pour accéder aux endpoints.

main :

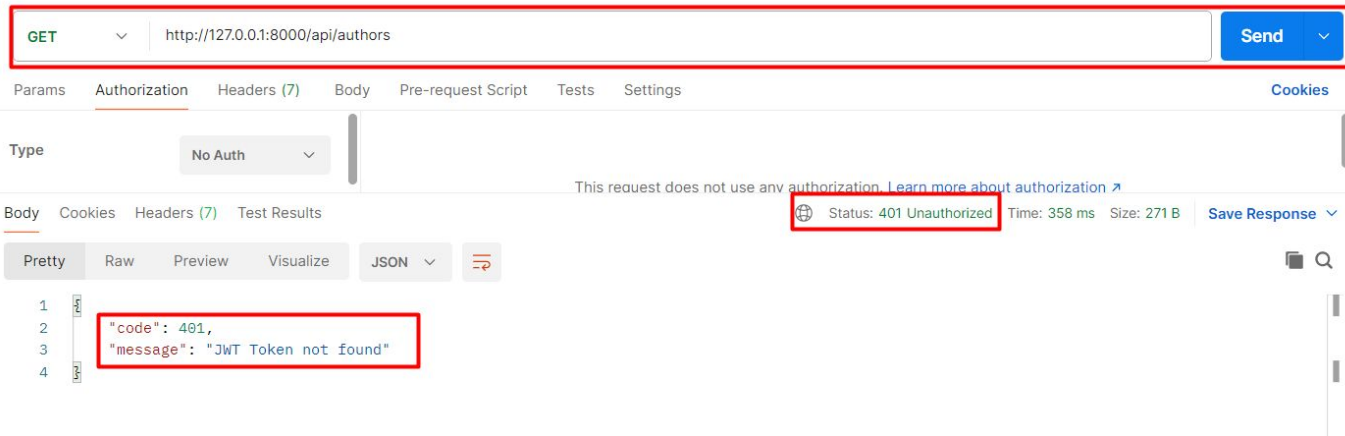
Firewall principal, configuré pour la gestion de l'utilisateur, mais n'utilisé que dans les autres contextes.

access_control :

Règles définissant les permissions d'accès aux différentes routes. Par exemple, seules les routes /api/login et /api/register sont accessibles sans authentification.

AUTHENTIFICATION JWT

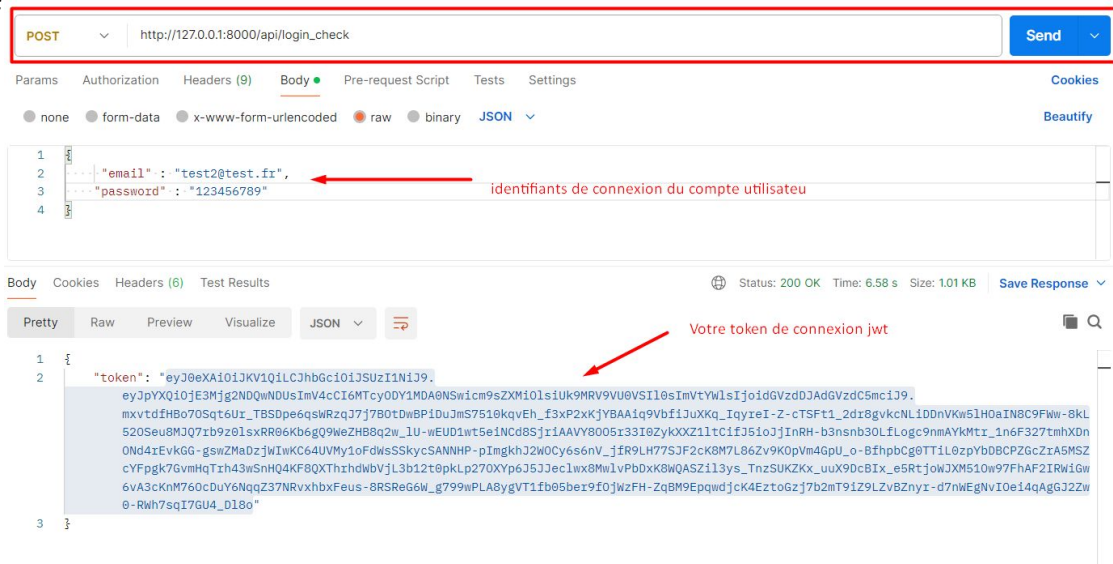
Maintenant si on essaye l'endpoint pour récupérer tous les auteurs on n'aura pas le résultat suivant :



Le statut 401 Unauthorized précise que nous n'avons pas donné le token JWT à notre requête.

AUTHENTIFICATION JWT

Pour avoir notre token jwt nous allons devoir effectuer la requête de connexion suivante :



Request:

```
POST http://127.0.0.1:8000/api/login_check
```

Body:

```
{
  "email": "test2@test.fr",
  "password": "123456789"
}
```

identifiants de connexion du compte utilisateur

Response:

```
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpYXQiOiJlMjg2NDQwNDUwImV4cCI6MTcyODY1MDA8NSw1cm9sZXN1bG10sImVtYWlsIjoIdGVzdD03AdGVzdC5mc1J9.mxvtdfh807Sg6Uir_TBSDpe6qsWRzqj7j780tdwBPiDuJmS7510kqvEh_f3p2xKjYBAAIq9VbfjJXKq_IqyreI-Z-cTSFT1_2dr8gvkcNL1DDnVKw5lHOaIN8C9Fww-8kL520Seu8MJQ7rb9z0lsxRR06Kb6g09WeZH8q2w_lU-wEUD1wt5e1Ncd8SjriAAVY8005i33I0ZyKXXZ11tcifj51oJjInRH-b3nsnb30Lfl0gc9nmAYkMtz_1n6F327tmhXDN0Nd4zEvkGg-gswZMaDzjwIwKC64UvMy1oFdWssSkycSANNHP-pImghJ2W0Cy6s6nV_jfr9LH7SjF2cK8M7L86Zv9K0pVm46pu_o-Bfhpbcg8TTiL0zpyDBDPCZGcZrASMSZcYFpgk7GvmHqTzh43wSnHq4KF8QXThzhdwVjL3b12tpkLp270XYp6J5Jec1wx8MwlvPbdxK8WQASZi13ys_TnzSUKZKx_uuX90cBIX_e5RtjowJXMS10w97FhAF2IRwiGw6vA3cKnM760duY6NqQZ37NRvxbxFeus-8RSRe6W_g799wPLAbygVT1fb05ber9f0jwzFH-ZqBM9EpqwdjK4EztoGzj7b2mT9I29LZV8Znyr-d7nWEgNvI0e14qAgG32Zw0-Rwh7sqI7GU4_D180"
}
```

Votre token de connexion jwt

Nous allons pouvoir copier ce token et l'utiliser dans le header de notre prochaine requête

AUTHENTIFICATION JWT

Maintenant nous allons pouvoir renseigner le **token** jwt dans la partie **Authorization**, puis on va sélectionner le type **Bearer Token** et enfin coller le token

GET http://127.0.0.1:8000/api/authors Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Type **Bearer Token** Token eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.e...

The authorization header will be automatically generated when you send the request.
[Learn more about authorization](#)

Body Cookies Headers (6) Test Results Status: 200 OK Time: 291 ms Size: 245 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "name": "Victor hugo",
3   "bio": "french author from..."
4 }
5
6
```

Notre endpoint est enfin sécurisé et nous savons comment y accéder.

AUTHENTIFICATION JWT

Le mécanisme utilisé repose sur le JSON Web Token (JWT), une méthode moderne et sécurisée d'authentification, fréquemment utilisée dans les applications web.

Le Processus Global :

- **Requête de Connexion** : L'utilisateur envoie une requête POST à l'endpoint `/api/login` en fournissant ses identifiants, généralement son email et son mot de passe.
- **Validation des Identifiants** : Le système compare l'email fourni avec celui de la base de données, puis vérifie que le mot de passe fourni correspond à celui stocké (de manière sécurisée) dans la base.
- **Génération du Token JWT** : Si les informations sont valides, un jeton JWT est généré. Ce jeton est une chaîne cryptée qui contient des informations sur l'utilisateur (comme son rôle ou son ID), et il est signé pour éviter les falsifications.
- **Retour du Token** : Le jeton JWT est renvoyé dans la réponse de l'API. L'utilisateur peut alors utiliser ce token pour accéder aux autres parties sécurisées de l'API, en le joignant aux requêtes futures dans les headers.

AUTHENTIFICATION JWT

L'Importance du JWT

- **Sans état :**

Le JWT permet à l'API de rester stateless, c'est-à-dire que chaque requête est autonome et contient toute l'information nécessaire pour l'authentification, sans avoir besoin de maintenir une session côté serveur.

- **Sécurité :**

Le jeton est signé, ce qui garantit que personne ne peut le modifier sans invalidation.

- **Accès Contrôlé :**

Le token peut contenir des informations spécifiques comme le rôle de l'utilisateur, permettant ainsi de restreindre l'accès à certaines routes en fonction des permissions.

Ce mécanisme simplifie l'authentification pour une API REST, tout en offrant un haut niveau de sécurité et de flexibilité