

Api - Symfony6

I Initialisation du projet Symfony - Api

Installation de symfony6

En considérant les prérequis pour l'installation de symfony sont déjà remplis.

Par défaut, le framework Symfony peut s'installer :

- en version Basique, avec uniquement les bundles requis pour un fonctionnement minimal
- ou en version web comme nous avons pu le voir précédemment (pour une utilisation de twig etc...;)

Afin de proposer une API avec Symfony 6, nous allons initialiser le projet avec la version basique et nous rajouterons au fur et à mesure les bundles adéquat.

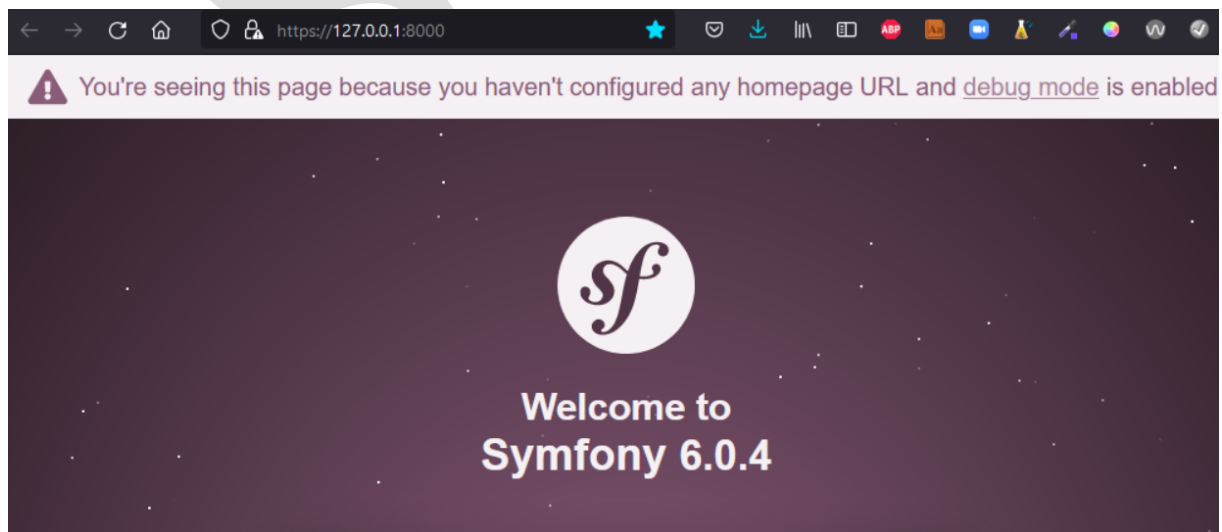
Pour cela nous allons dans un terminal et taper la commande suivante :

```
symfony new Books
```

Books est le nom de notre projet qui sera une application de gestion de livres.

Pour vérifier que l'installation s'est bien déroulée, toujours dans le terminal, vous pouvez vous rendre dans le dossier `Books` qui a été créé avec la commande `cd Books`, puis lancer le serveur de Symfony avec `symfony server:start`

Cela lancera le serveur de Symfony, et en vous rendant sur l'URL proposée, vous devriez voir quelque chose comme ceci :



II Créer la première entité

Notre framework est donc maintenant bien installé, on peut s'apercevoir que nous avons moins de fichiers que lorsque l'on crée un projet web Symfony6.

Nous souhaitons maintenant créer notre première **table** que l'on appellera Book qui contiendra quelques informations sur les livres que nous allons mettre à disposition.

Nous allons demander d'abord à symfony de nous créer des éléments comme des controllers, des entity etc.. Pour cela il nous faut le bundle **maker-bundle**.

Pour installer ce bundle nous devons taper la commande suivante dans le terminal de commande :

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

Le `-- dev` précise que l'outil ne sera disponible qu'en mode développement.

Puisque nous voulons créer une entité, et pouvoir la modifier tout au long de la création de l'API, nous devons donc installer le bundle Doctrine avec la commande suivante dans votre terminal:

```
composer require orm
```

Voilà Doctrine est maintenant installé dans votre projet. Nous allons pouvoir donc créer notre première entité qui sera **Book** avec la commande suivante:

```
php bin/console make:entity
```

```
PS D:\Openclassrooms\Cours API Symfony\Test\Books> php bin/console make:entity

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

created: src/Entity/Book.php
created: src/Repository/BookRepository.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]:
>

Field length [255]:
>

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

updated: src/Entity/Book.php

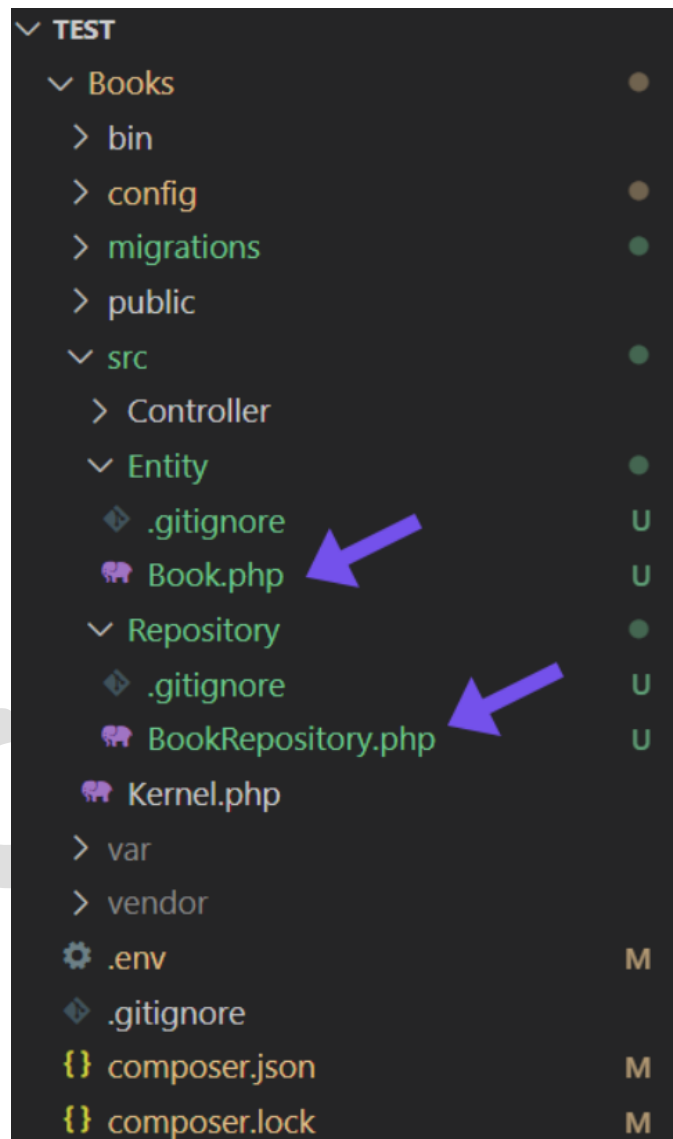
Add another property? Enter the property name (or press <return> to stop adding fields):
> 
```

Deux fichiers se sont créés suite à cette commande.

Le premier qui est l'entité, deux champs ont été également créé dans l'entité

- Title : le titre du livre en laissant toutes les options par défaut
- coverText : la quatrième de couverture, de type text et nullable

Et son repository qui est une sorte de manager de l'entité, qui nous permettra de faire les requêtes selon les besoins.



Dans Books.php nous avons ceci :

```
<?php
// src\Entity\Book.php

namespace App\Entity;

use App\Repository\BookRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: BookRepository::class)]
class Book
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private $id;

    #[ORM\Column(type: 'string', length: 255)]
    private $title;

    #[ORM\Column(type: 'text', nullable: true)]
    private $coverText;

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getTitle(): ?string
    {
        return $this->title;
    }

    public function setTitle(string $title): self
    {
        $this->title = $title;
        return $this;
    }

    public function getCoverText(): ?string
    {

```

```

        return $this->coverText;
    }

    public function setCoverText(?string $coverText): self
    {
        $this->coverText = $coverText;
        return $this;
    }
}

```

Pour le moment l'entité est créée mais pas envoyée en BDD. Il nous manque la configuration de la bdd dans le .env

Nous allons donc y procéder de la manière suivante, rendons nous dans le fichier .env, puisque nous travaillons avec mysql activons le database mysql et désactivons le postgresql :

```

# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
DATABASE_URL="mysql://root:@127.0.0.1:3306/books"
#
DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=14&
charset=utf8"
###< doctrine/doctrine-bundle ###

```

Le format de DATABASE_URL est le suivant :

`DATABASE_URL="mysql://pseudo:mdp@localhost:port/nomBDD"`

Nous pouvons donc envoyer notre entity dans notre BDD avec les deux commandes suivantes:

```
php bin/console make:migration
```

```
php bin/console doctrine:migrations:migrate
```

Une fois ces deux commandes saisies votre entity est arrivée dans votre BDD, nous sommes prêts pour envoyer de la donnée et la récupérer.

Par ailleurs voyons comment nous pouvons générer de la donnée avec les **fixtures**.

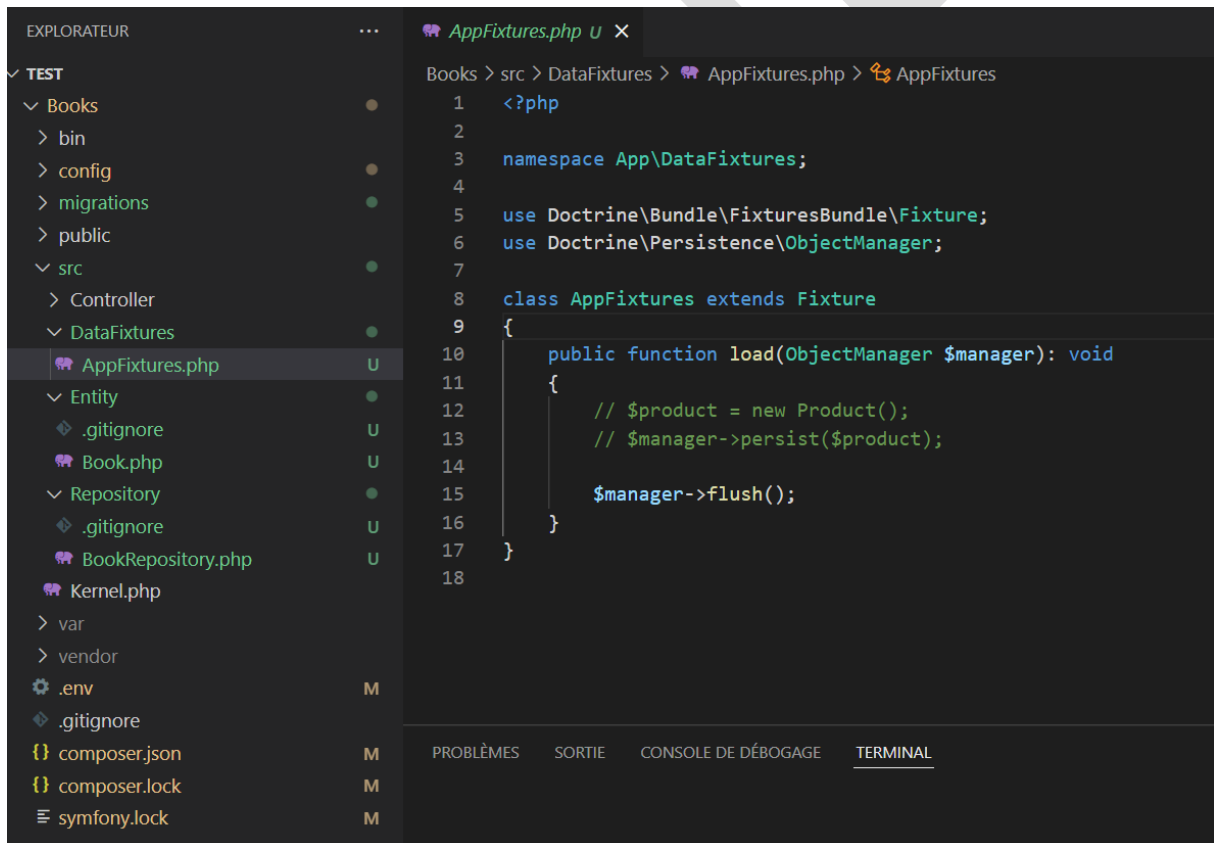
II Générer des données avec les fixtures

Tout d'abord nous allons installer via composer le bundle qui nous servira à utiliser les fixtures.

```
composer require orm-fixtures --dev
```

Avec ceci nous allons pouvoir générer du code pour envoyer de la donnée en bdd

Un nouveau dossier s'est créé suite à la commande réalisé **DataFixtures** et à l'intérieur de celui-ci nous pouvons retrouver un fichier nommé **AppFixtures.php**. C'est dans ce fichier que nous allons produire du code pour générer des données afin de tester notre api.



Nous allons donc faire en sorte que d'envoyer une vingtaine de livres en bdd de la façon suivante:

```
public function load(ObjectManager $manager): void
{
    // Création d'une vingtaine de livres ayant pour titre
    for ($i = 0; $i < 20; $i++) {
        $livre = new Book;
        $livre->setTitle('Livre ' . $i);
        $livre->setCoverText('Quatrième de couverture numéro :
' . $i);
        $manager->persist($livre);
    }

    $manager->flush();
}
```

On vient faire une boucle avec la **variable \$i** dans la condition on la met à 0 et on va lui faire 20 tours de boucle. On oublie pas **d'incrémenter \$i**.

A chaque tour de boucle :

- on crée un livre
- on envoie un titre, on fait une concaténation avec \$i pour avoir à chaque fois un livre différent
- on fait de même avec la couverture du livre
- on **persist \$livre**

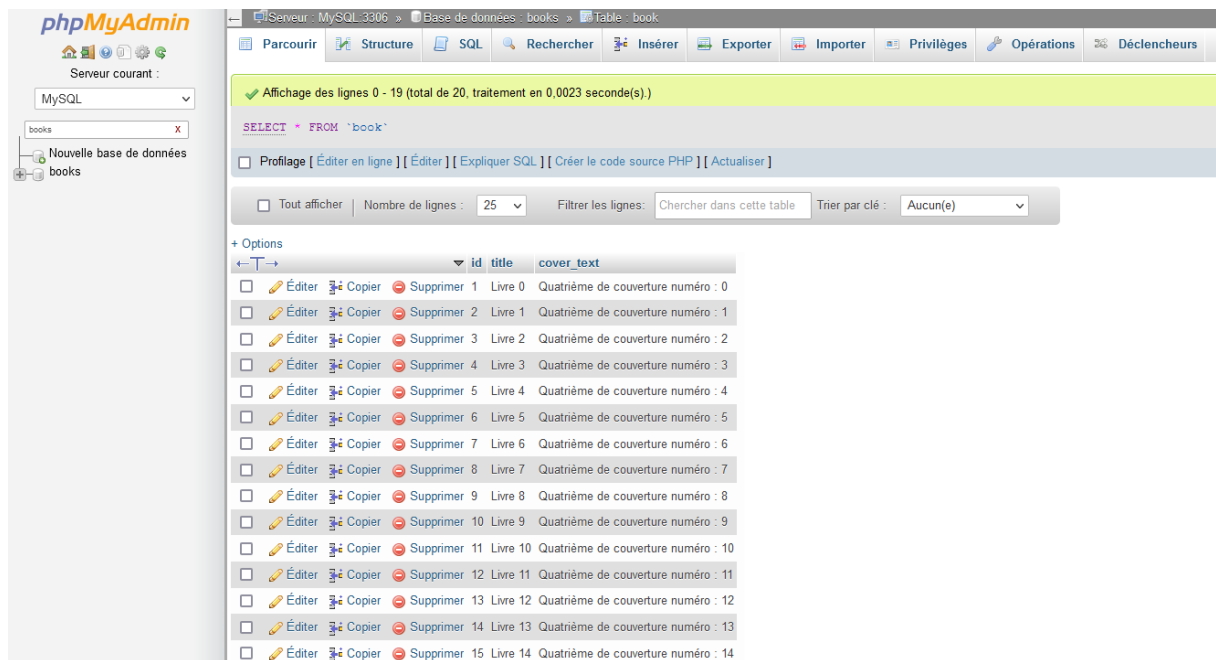
On sort de la boucle et on fait le **flush()**

Notre code est bien construit pour envoyer les données de 20 livres en bdd, il nous manque juste une commande ensuite pour l'envoyer. Pas besoin de faire une migration car ici on envoie de la donnée on ne touche pas à la structure de la BDD. Il nous suffit donc de faire la commande suivante :

```
php bin/console doctrine:fixtures:load
```

Ne pas oublier de faire "yes" car suite à cette commande on nous demande si nous sommes d'accord pour envoyer les données.

Vérifions donc que nos données se sont bien mises dans notre BDD



phpMyAdmin

Seigneur : MySQL:3306 » Base de données : books » Table : book

Parcourir Structure SQL Rechercher Insérer Exporter Importer Privilèges Opérations Déclencheurs

Affichage des lignes 0 - 19 (total de 20, traitement en 0,0023 seconde(s).)

SELECT * FROM `book`

Profilage [Éditer en ligne] [Éditer] [Expliquer SQL] [Créer le code source PHP] [Actualiser]

Tout afficher Nombre de lignes : 25 Filtrer les lignes : Chercher dans cette table Trier par clé : Aucun(e)

+ Options

				id	title	cover_text
<input type="checkbox"/>	Éditer	Copier	Supprimer	1	Livre 0	Quatrième de couverture numéro : 0
<input type="checkbox"/>	Éditer	Copier	Supprimer	2	Livre 1	Quatrième de couverture numéro : 1
<input type="checkbox"/>	Éditer	Copier	Supprimer	3	Livre 2	Quatrième de couverture numéro : 2
<input type="checkbox"/>	Éditer	Copier	Supprimer	4	Livre 3	Quatrième de couverture numéro : 3
<input type="checkbox"/>	Éditer	Copier	Supprimer	5	Livre 4	Quatrième de couverture numéro : 4
<input type="checkbox"/>	Éditer	Copier	Supprimer	6	Livre 5	Quatrième de couverture numéro : 5
<input type="checkbox"/>	Éditer	Copier	Supprimer	7	Livre 6	Quatrième de couverture numéro : 6
<input type="checkbox"/>	Éditer	Copier	Supprimer	8	Livre 7	Quatrième de couverture numéro : 7
<input type="checkbox"/>	Éditer	Copier	Supprimer	9	Livre 8	Quatrième de couverture numéro : 8
<input type="checkbox"/>	Éditer	Copier	Supprimer	10	Livre 9	Quatrième de couverture numéro : 9
<input type="checkbox"/>	Éditer	Copier	Supprimer	11	Livre 10	Quatrième de couverture numéro : 10
<input type="checkbox"/>	Éditer	Copier	Supprimer	12	Livre 11	Quatrième de couverture numéro : 11
<input type="checkbox"/>	Éditer	Copier	Supprimer	13	Livre 12	Quatrième de couverture numéro : 12
<input type="checkbox"/>	Éditer	Copier	Supprimer	14	Livre 13	Quatrième de couverture numéro : 13
<input type="checkbox"/>	Éditer	Copier	Supprimer	15	Livre 14	Quatrième de couverture numéro : 14

Nous voyons bien que nous avons nos données en BDD

III Création de notre premier Controller

Pour faire notre Api nous avons besoin de différentes routes, ces routes sont gérées dans les contrôleurs respectifs.*

Afin de pouvoir récupérer les données, nous avons donc besoin de la route. Pour cela, créons le contrôleur **BookController**

```
php bin/console make:controller
```

Saisir le nom du contrôleur : **Book**

Notre controller est désormais créé

```
<?php
// src\Controller\BookController.php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class BookController extends AbstractController
{
    #[Route('/book', name: 'book')]
    public function index(): Response
    {
        return $this->json([
            'message' => 'welcome to your new controller!',
            'path' => 'src/Controller/BookController.php',
        ]);
    }
}
```

Si nous faisons un controle du retour de notre controller dans Postman en tapant la route :

`http://localhost:8000/book`

Nous obtenons le résultat suivant

The screenshot shows the Postman interface. At the top, a GET request is made to `https://127.0.0.1:8000/book`. The 'Params' tab is selected, showing an empty table. The 'Body' tab is also selected, showing a JSON response in 'Pretty' view. The response is a JSON array with two objects: one with 'message' and 'path' keys, and another with 'path' key.

KEY	VALUE
Key	Value

```
1 {
2   "message": "welcome to your new controller!",
3   "path": "src/Controller/BookController.php"
4 }
```

Nous avons effectivement dans un JSON le retour du message et du path.

Nous allons donc changer la route avec pour valeur `api/books` et le nom de la fonction que nous allons appeler `getBookList()` voir l'exemple ci-dessous :

```
<?php
// src\Controller\BookController.php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Routing\Annotation\Route;

class BookController extends AbstractController
{
    #[Route('/api/books', name: 'book', methods: ['GET'])]
    public function getBookList(): JsonResponse
    {
        return new JsonResponse([
            'message' => 'welcome to your new controller!',
            'path' => 'src/Controller/BookController.php',
        ]);
    }
}
```

La route ici est bien changée ainsi que le name de la route et nous avons la method en **GET**.

Nous avons changé le return avec le `JsonResponse` afin d'avoir une meilleure lisibilité et prévoir les erreurs

Nous sommes prêts à produire du code pour récupérer la liste des livres générée en BDD.

Attention désormais pour vérifier ce que `BookController` nous retourne la route a changé :

```
http://localhost:8000/api/books
```

IV Récupération de nos premières données

Notre première route est prête, en état de fonctionner mais ne nous retourne pas encore la liste de nos livres donc pour cela il faudra créer une variable par exemple **\$bookList** on va faire intervenir le **BookRepository** pour récupérer la fonction **findAll()** :

```
<?php
// src\Controller\BookController.php
// ...

#[Route('/api/books', name: 'book', methods: ['GET'])]
public function getBookList(BookRepository $bookRepository):
    JsonResponse
    {
        $bookList = $bookRepository->findAll();

        return new JsonResponse([
            'books' => $bookList,
        ]);
    }
```

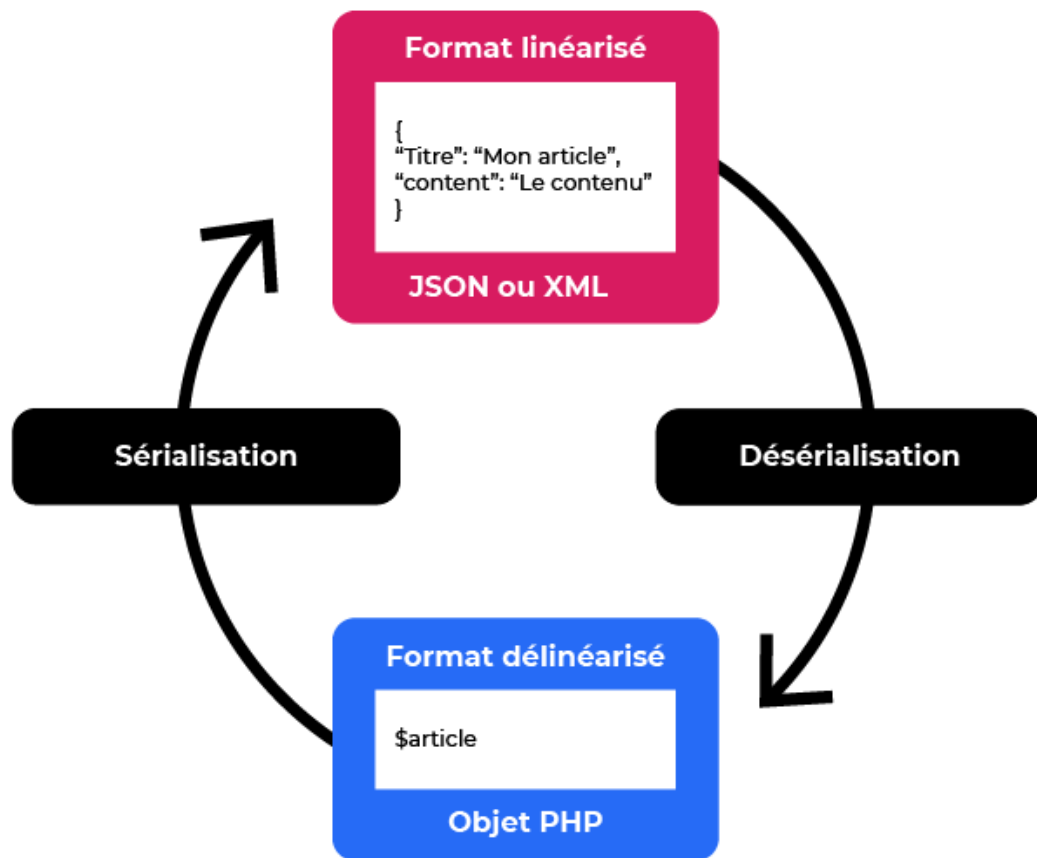
Si nous essayons dans postman cette fonction nous aurons un problème. Effectivement nous aurons un retour de 20 livres vides.

Pourquoi?

PHP ne sait pas convertir des objets (ici un tableau d'objets). Nous allons donc devoir utiliser un **SERIALIZER**.

Un serializer est un programme qui permet de **prendre des objets complexes et de les transformer en chaîne de caractère**, bien plus facile à transporter.

Les plus courants sont des serializers qui vont convertir des éléments vers du **JSON** ou du **XML**, mais il en existe de nombreux autres. Il existe également des désérialiseurs qui permettent l'opération inverse.



Source : OPEN CLASSROOM

Installons donc le bundle pour le SERIALIZER :

```
composer require symfony/serializer-pack
```

Ensuite mettons à jour notre méthode :

```
<?php
// src\Controller\BookController.php
// ...

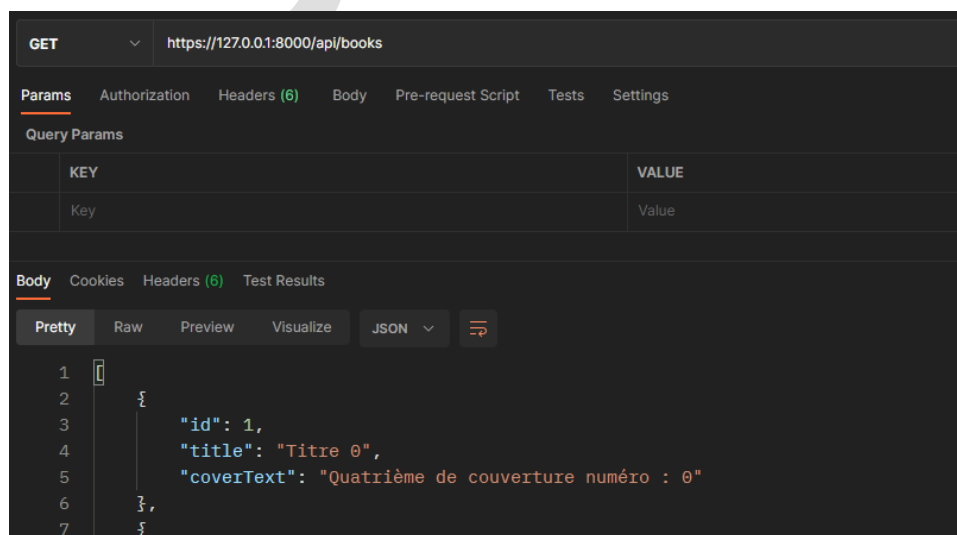
#[Route('/api/books', name: 'book', methods: ['GET'])]
public function getBookList(BookRepository $bookRepository,
SerializerInterface $serializer): JsonResponse
{
    $bookList = $bookRepository->findAll();
    $jsonBookList = $serializer->serialize($bookList, 'json');
    return new JsonResponse($jsonBookList, Response::HTTP_OK,
[], true);
}
```

N'oublions pas d'ajouter nos use correspondant aux outils utilisés au début de notre fichier.

Notez le changement sur notre **JsonResponse** qui prend désormais 4 arguments :

- Les données sérialisées
- Le code retour : `Response::HTTP_OK` qui sera le code status retourné et ici c'est le code 200
- les headers (qu'on laisse vide pour garder le comportement par défaut pour le moment)
- et un `true` qui signifie que nous avons déjà sérialisé les données et que nous n'avons donc plus rien à faire.

Voyons maintenant le retour de notre route dans Postman :



Nous constatons que notre route fonctionne comme nous le souhaitons.
Après avoir vu la liste de livre nous allons vouloir retourner un livre en fonction de son id.

V Récupération d'un livre en particulier

Nous allons donc rajouter une deuxième route dans notre BookController afin d'obtenir un livre en particulier. Pour cela nous aurons besoin de l'id du livre.

```
<?php
// src\Controller\BookController.php
// ...

#[Route('/api/books/{id}', name: 'detailBook', methods:
['GET'])]
public function getDetailBook(int $id, SerializerInterface
$serializer, BookRepository $bookRepository): JsonResponse {

    $book = $bookRepository->find($id);
    if ($book) {
        $jsonBook = $serializer->serialize($book, 'json');
        return new JsonResponse($jsonBook, Response::HTTP_OK,
[], true);
    }
    return new JsonResponse(null, Response::HTTP_NOT_FOUND);
}
```

Nous constatons donc que notre n'est pas la même, `/api/books/{id}`, que le nom de la route n'est pas la même `detailBook` par contre la méthode reste la même (ceci est normal car nous voulons récupérer une donnée, nous lisons simplement nous ne sommes pas en acte d'écriture en bdd)

Dans nos paramètres de la fonction nous faisons intervenir `$id` avec son type, nous appelons le `SerializerInterface`, le `BookRepository`.

Nous créons une variable que l'on appelle `$book` et nous faisons intervenir `BookRepository` pour reprendre la fonction `find()`, dans cette fonction `find` on passe l'`$id` du livre.

Nous faisons une condition qui nous dit que si `$book` existe alors on crée une variable `$jsonBook` et on vient appliquer le `serialize` sur la variable `$book` grâce au `SerializerInterface` et on retourne en Json la variable `$jsonBook` et le status `HTTP_OK`

Notez également que si le livre n'existe pas, alors nous retournons **une réponse vide avec un autre code d'erreur** : `Reponse::HTTP_NOT_FOUND`, la fameuse erreur `404` dont vous avez probablement déjà entendu parler.

Ici c'est une première façon de faire. Découvrons une autre façon de faire en utilisant le `ParamConverter`. Symfony nous permet d'utiliser un outil qui est donc le `ParamConverter`.

Le `ParamConverter` sert, comme son nom l'indique, à "convertir des paramètres". C'est-à-dire qu'il va tenter, à partir de l'id fourni, de trouver seul de quel livre il s'agit. De fait, il ne sera plus nécessaire de passer l'id et le repository, car le `ParamConverter` est capable, seul, de fournir directement le livre correspondant.

Installons maintenant cet outil avec la commande suivante :

```
composer require sensio/framework-extra-bundle
```

Regardons donc notre code de plus près pour voir ce qui a changé :

```
#[Route('/api/books/{id}', name: 'detailBook', methods: ['GET'])]
public function getDetailBook(Book $book, SerializerInterface
$jsonSerializer): JsonResponse
{
    $jsonBook = $jsonSerializer->serialize($book, 'json');
    return new JsonResponse($jsonBook, Response::HTTP_OK,
['accept' => 'json'], true);
}
```

Nos paramètres dans la fonction ont changé, nous n'avons plus l'`$id` ni le `BookRepository` mais nous l'entity `Book`

Notre fonction ne contient plus que deux lignes et notre code est donc optimisé.

Et pour la gestion d'erreur ? Si vous tentez de regarder un id invalide, sur Postman vous verrez un message d'erreur en HTML. Ignorez-le pour l'instant, nous verrons plus tard comment le rendre plus agréable à lire. Le plus important est notre code retour qui est le code **404**

VI Une deuxième entité avec une relation

Nous allons créer une deuxième entité avec une **relation** entre celle-ci et les livres. Ce sera l'**entité author**

Reprenons notre commande pour créer une entité

```
php bin/console make:entity
```

Nous allons y créer deux champs que l'on va appeler :

- firstName
- lastName

Un troisième champ qui sera en relation avec Book et ce champ s'appellera books

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

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

What class should this entity be related to?:
> Book

What type of relationship is this?
-----
Type      Description
-----
ManyToOne Each Author relates to (has) one Book.
           Each Book can relate to (can have) many Author objects
OneToMany Each Author can relate to (can have) many Book objects.
           Each Book relates to (has) one Author
ManyToMany Each Author can relate to (can have) many Book objects.
           Each Book can also relate to (can also have) many Author objects
OneToOne  Each Author relates to (has) exactly one Book.
           Each Book also relates to (has) exactly one Author.
-----

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> OneToMany

A new property will also be added to the Book class so that you can access and set the related Author object from it.

New field name inside Book [author]:
>

Is the Book.author property allowed to be null (nullable)? (yes/no) [yes]:
> yes

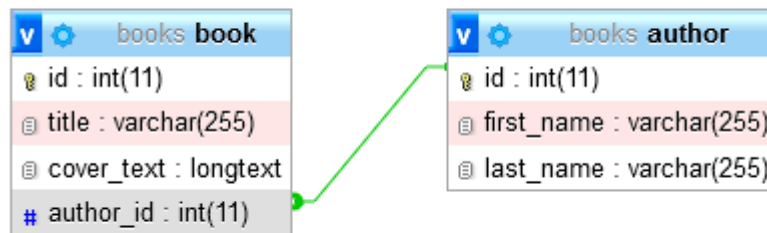
updated: src/Entity/Author.php
updated: src/Entity/Book.php
```

Un auteur peut être
lié à plusieurs livres
Un livre est lié à un
seul auteur.

On utilise la relation OneToMany ici car un auteur peut être lié à plusieurs livres et un livre est lié qu'à un seul auteur.

Nous pouvons à présent réaliser la migration avec les deux commandes vues précédemment.

On peut vérifier sur phpMyAdmin que tout est bon et utiliser le concepteur pour voir la relation



Nous Allons maintenant mettre à jour nos fixtures afin d'ajouter des auteurs au nombre de 10 dans la BDD.

Reprenons donc notre fichier AppFixture.php et ajoutons un peu de code pour générer de la donnée des auteurs

```
// Création des auteurs.
$listAuthor = [];
for ($i = 0; $i < 10; $i++) {
    // Création de l'auteur lui-même.
    $author = new Author();
    $author->setFirstName("Prénom " . $i);
    $author->setLastName("Nom " . $i);
    $manager->persist($author);
    // On sauvegarde l'auteur créé dans un tableau.
    $listAuthor[] = $author;
}
```

On crée d'abord un tableau pour y mettre les auteurs dedans et ensuite on y fait la boucle pour créer un auteur à chaque tour de boucles.

Dans les données des livres il faudra également envoyer l'auteur donc une ligne est à rajouter

```
// Création des livres
for ($i = 0; $i < 20; $i++) {
    $book = new Book();
    $book->setTitle("Titre " . $i);
    $book->setCoverText("Quatrième de couverture numéro : " .
    $i);

    // On lie le livre à un auteur pris au hasard dans le
    tableau des auteurs.

    $book->setAuthor($listAuthor[array_rand($listAuthor)]);
    $manager->persist($book);
}
```

On peut voir donc que nous avons rajouter
`$book->setAuthor($listAutor[array_rand($listAuthor)])`;
 On envoie un auteur en mode random sur chacun des livres.

On exécute les fixtures avec la commande :

```
php bin/console doctrine:fixtures:load
```

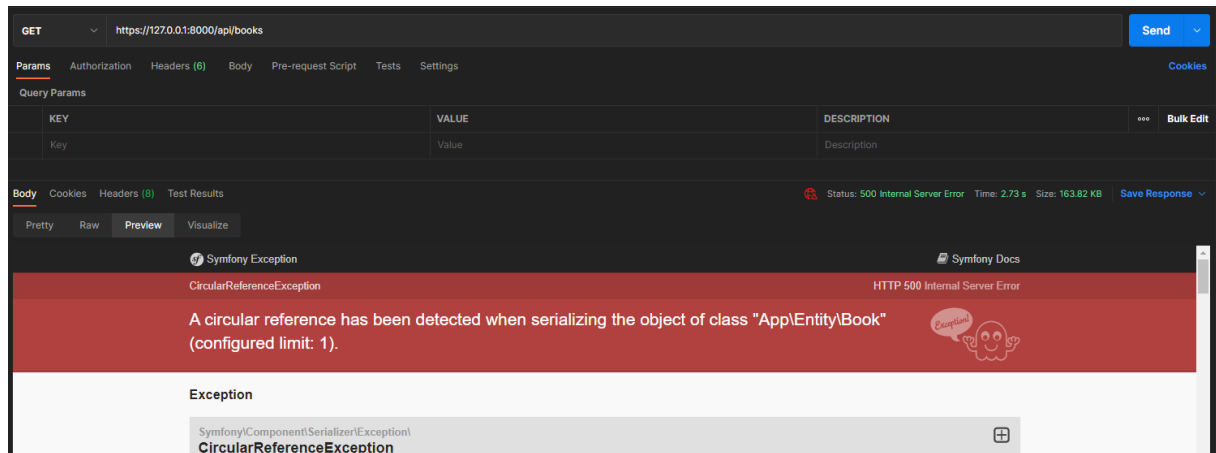
Vérifions que dans la base de données tout s'est bien passé, que nous avons bien nos 10 auteurs et que dans nos livres nos auteurs soient bien affectés à chaque livre.

						id	title	cover_text	author_id	
<input type="checkbox"/>		Éditer		Copier		Supprimer	41	Titre 0	Quatrième de couverture numéro : 0	15
<input type="checkbox"/>		Éditer		Copier		Supprimer	42	Titre 1	Quatrième de couverture numéro : 1	13
<input type="checkbox"/>		Éditer		Copier		Supprimer	43	Titre 2	Quatrième de couverture numéro : 2	12
<input type="checkbox"/>		Éditer		Copier		Supprimer	44	Titre 3	Quatrième de couverture numéro : 3	13
<input type="checkbox"/>		Éditer		Copier		Supprimer	45	Titre 4	Quatrième de couverture numéro : 4	15
<input type="checkbox"/>		Éditer		Copier		Supprimer	46	Titre 5	Quatrième de couverture numéro : 5	12
<input type="checkbox"/>		Éditer		Copier		Supprimer	47	Titre 6	Quatrième de couverture numéro : 6	13

Notez que **l'id des livres a changé**. En effet, lorsqu'on relance une fixture, la base de données est vidée, mais l'indice de l'id n'est pas remis à zéro. Cela peut être important lorsque vous testez vos routes, car après avoir relancé les fixtures, un id qui était valide initialement n'a aucune garantie de l'être encore après coup.

Notez la colonne `author_id` qui a été rajoutée automatiquement par Doctrine, et qui contient l'id de l'auteur qui a écrit le livre

Une fois ces données ajoutées dans notre BDD, nous souhaitons donc les récupérer. Tentons via Postman de les récupérer.



Cette erreur s'affiche donc à l'écran. Il s'agit d'une **référence circulaire**.

En gros, lorsque le serializer a voulu récupérer des données du livre, il a pris, comme avant, les données de l'id, **title** et **coverText**. Par contre, il a également pris les données de l'auteur.

Or, l'auteur possède comme donnée... une collection de livres qu'il a écrits. Ces livres ont des auteurs, qui ont des livres, qui ont des auteurs, qui ont des livres... vous voyez le problème.

Pour résoudre ce problème, il faudrait pouvoir **spécifier quels champs on veut récupérer dans quel cas**.

Symfony a prévu un mécanisme pour ça, les **Groups**

Reprenons l'entité **Book** dans le fichier **Book.php** :

```
<?php
//src\Entity\Book.php

namespace App\Entity;

use App\Repository\BookRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Serializer\Annotation\Groups;

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

```

{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    #[Groups(["getBooks"])]
    private $id;

    #[ORM\Column(type: 'string', length: 255)]
    #[Groups(["getBooks"])]
    private $title;

    #[ORM\Column(type: 'text', nullable: true)]
    #[Groups(["getBooks"])]
    private $coverText;

    #[ORM\ManyToOne(targetEntity: Author::class, inversedBy:
'Books')]
    #[Groups(["getBooks"])]
    private $author;

    // ...

```

Nous avons une nouvelle annotation qu'il faut donc rajouter :

```
#[Groups(["getBooks"])]
```

Celle-ci définit que le champ qui suit sera pris en compte si on demande à récupérer les éléments qui appartiennent à ce groupe. Les crochets indiquent que l'on peut spécifier plusieurs groupes différents (nous verrons cela plus loin).

Attention à ne pas oublier le **use** de **Groups** qui permet d'expliquer à symfony comment on l'utilise.

Ici, pas de problème, nous prenons tous les champs.

Voyons maintenant du côté de l'entity author :

```
<?php
// src\Entity\Author.php

namespace App\Entity;

use App\Repository\AuthorRepository;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Serializer\Annotation\Groups;

#[ORM\Entity(repositoryClass: AuthorRepository::class)]
class Author
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    #[Groups(["getBooks"])]
    private $id;

    #[ORM\Column(type: 'string', length: 255, nullable: true)]
    #[Groups(["getBooks"])]
    private $firstName;

    #[ORM\Column(type: 'string', length: 255)]
    #[Groups(["getBooks"])]
    private $lastName;

    #[ORM\OneToMany(mappedBy: 'author', targetEntity: Book::class)]
    private $books;

    // ...
}
```

Ici, nous récupérons l'id, le prénom, le nom, mais PAS la liste des livres. De ce fait, nous "cassons" la référence circulaire.

Lorsque l'on va demander des informations sur un livre, nous aurons les informations sur le livre, dont l'auteur, mais en récupérant les informations sur l'auteur, on s'arrête puisqu'on ne récupère pas la propriété `$books`

Il nous reste maintenant à spécifier dans le contrôleur que nous voulons les informations qui sont dans ce nouveau groupe `getBooks` que nous venons de créer.

```
<?php
// src\Controller\BookController.php

namespace App\Controller;

use App\Entity\Book;
use App\Repository\BookRepository;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Serializer\SerializerInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

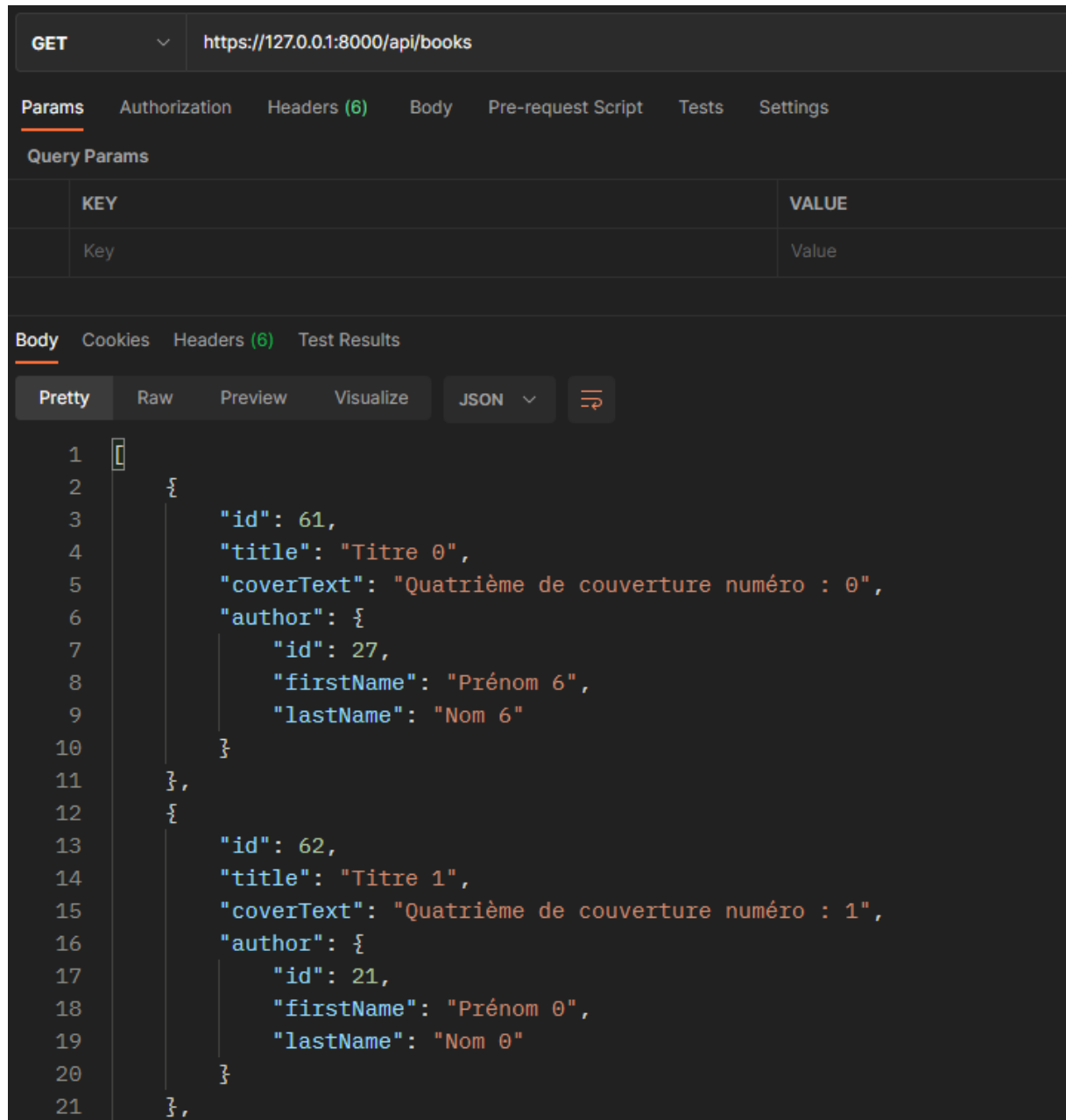
class BookController extends AbstractController
{
    #[Route('/api/books', name: 'books', methods: ['GET'])]
    public function getAllBooks(BookRepository $bookRepository,
        SerializerInterface $serializer): JsonResponse
    {
        $bookList = $bookRepository->findAll();

        $jsonBookList = $serializer->serialize($bookList, 'json',
            ['groups' => 'getBooks']);
        return new JsonResponse($jsonBookList, Response::HTTP_OK,
            [], true);
    }

    #[Route('/api/books/{id}', name: 'detailBook', methods:
        ['GET'])]
    public function getDetailBook(Book $book, SerializerInterface
        $serializer): JsonResponse
    {
        $jsonBook = $serializer->serialize($book, 'json', ['groups'
            => 'getBooks']);
        return new JsonResponse($jsonBook, Response::HTTP_OK, [],
            true);
    }
}
```

ci, la seule chose qui a changé est que nous passons une option supplémentaire au serializer : `['groups' => 'getBooks']`

Retournons sur postman pour voir le résultat :



Et voilà, lorsque nous demandons l'ensemble des livres, nous obtenons aussi des informations sur les auteurs. Nous avons bien cassé la référence circulaire !

EXERCICE

Créer le controller Author avec ses méthodes.

créer une fonction getAllAuthor qui nous récupèrera la liste des auteurs
créer une fonction getAuthor pour afficher un auteur en fonction de son id

Inspirez vous de ce qui a été fait avec les livres

VII Le CRUD

Après la lecture des données nous allons passé à l'écriture de celles-ci, c'est à dire que nous allons envoyer des données en BDD non plus de manière générique mais nous allons créer des méthodes différentes soit pour créer, soit pour modifier ou soit pour supprimer.

Bref une fois ce chapitre fini nous saurons faire un CRUD entier dans une api avec symfony6.

- **Le Delete**

Commençons par la suppression d'un élément. Avec cette méthode on viendra supprimer un élément de la base de donnée

Dans une API REST pour la suppression la méthode n'est plus **GET** mais **DELETE**

Regardons comment faire pour la suppression d'un élément :

Toujours dans le BookController réalisons cette méthode :

```
#[Route('/api/books/{id}', name: 'deleteBook', methods:
['DELETE'])]
    public function deleteBook(Book $book, EntityManagerInterface
$em): JsonResponse
    {
        $em->remove($book);
        $em->flush();

        return new JsonResponse(null, Response::HTTP_NO_CONTENT);
    }
```

- La route est donc ici **/api/books/{id}**, nous avons besoins effectivement de l'id
- Le name de la route est ici **deleteBook**
- La méthode est donc **DELETE**

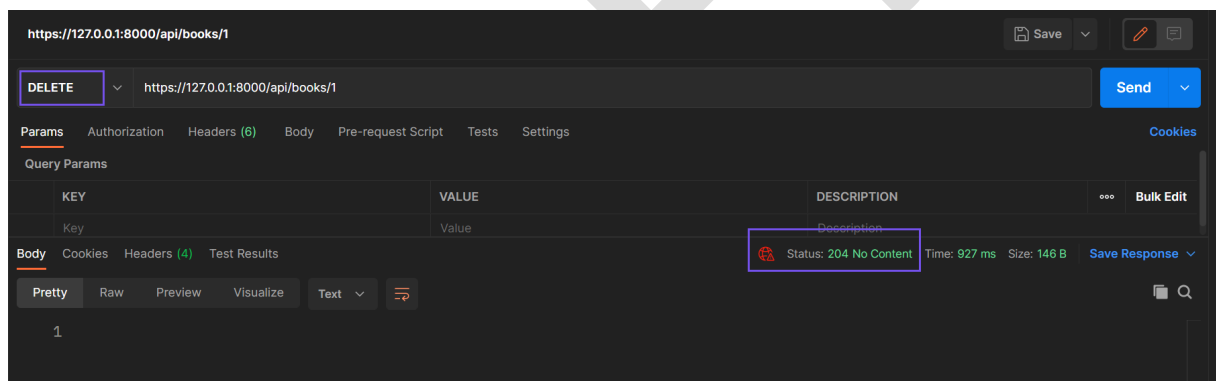
Nous avons créé la fonction deleteBook et dans les paramètres nous avons besoin de l'entity Book et de la class EntityManagerInterface

Nous utilisons le même **principe que pour retourner un élément**, sauf qu'au lieu de forger une réponse avec l'élément, on **le supprime grâce à l'EntityManagerInterface**, et on retourne une réponse **204 - No content**.

Notez qu'au second appel de cette route avec le même id, la réponse changera. En effet, une fois l'élément supprimé, au second appel nous obtiendrons une erreur **404 - Not found**, car l'élément ne sera plus là.

Retournons sur Postman

La route est la même que pour afficher un élément, sauf qu'au lieu d'utiliser un GET qui signifie que nous voulons récupérer les données, nous utilisons DELETE qui signifie que nous voulons effacer la donnée.



Regardons également dans PHPmyAdmin

+ Options

					id	author_id	title	cover_text
<input type="checkbox"/>	Éditer	Copier	Supprimer	2	5		Titre 1	Quatrième de couverture numéro : 1
<input type="checkbox"/>	Éditer	Copier	Supprimer	3	6		Titre 2	Quatrième de couverture numéro : 2
<input type="checkbox"/>	Éditer	Copier	Supprimer	4	8		Titre 3	Quatrième de couverture numéro : 3
<input type="checkbox"/>	Éditer	Copier	Supprimer	5	10		Titre 4	Quatrième de couverture numéro : 4

L'élément avec l'id 1 n'est plus présent, le premier possède l'id 2.

Le DELETE fonctionne très bien

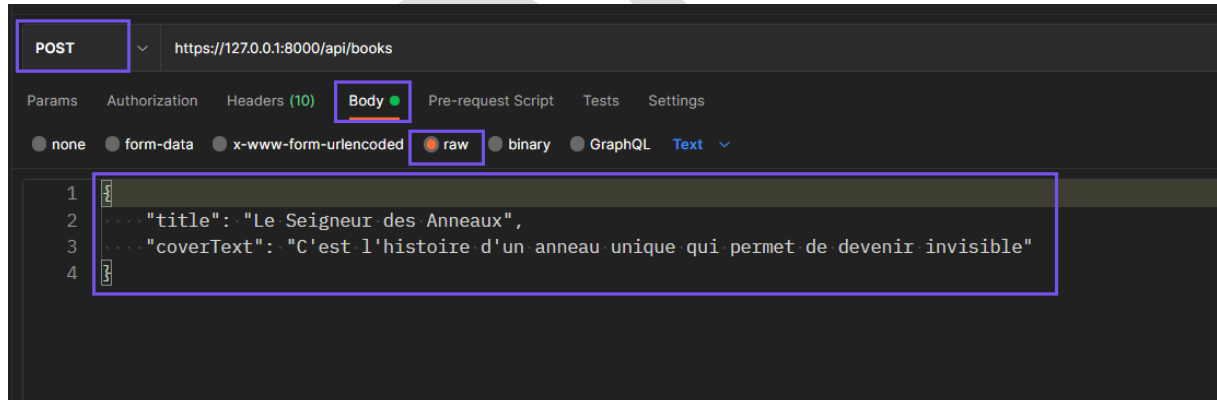
- **LE CREATE**

Maintenant que l'on arrive à supprimer, on va venir en ajouter un dans la base de donnée.

Pour ceux qui sont habitués à utiliser des formulaires en HTML, lorsque vous précisez dans la balise `form : method="post"`, il s'agit ici exactement de la même chose

Cependant il existe une différence majeure au niveau des données entre le DELETE et le POST. Cette fois, nous ne pouvons pas nous contenter d'envoyer une petite information comme un id directement dans l'URL. Nous devons, dans le corps du message Body, **envoyer les informations sur le livre** que nous voulons créer.

Une fois n'est pas coutume, je vais commencer par créer la requête sur Postman pour bien montrer ce que l'on vise :



Pour faire ceci dans postman on sélectionne le Post puis on va dans body et dans raw.

Puis c'est parti pour implémenter les données dans un JSON.

Si on envoie ceci bien évidemment cela ne fonctionnera pas car nous n'avons pas créé la route pour le POST dans le controller BookController.

Nous allons réaliser cela de suite :

```
#[Route('/api/books', name:"createBook", methods: ['POST'])]
public function createBook(Request $request,
SerializerInterface $serializer, EntityManagerInterface $em,
UrlGeneratorInterface $urlGenerator): JsonResponse
{
    $book = $serializer->deserialize($request->getContent(),
Book::class, 'json');
    $em->persist($book);
    $em->flush();

    $jsonBook = $serializer->serialize($book, 'json', ['groups'
=> 'getBooks']);

    $location = $urlGenerator->generate('detailBook', ['id' =>
$book->getId()], UrlGeneratorInterface::ABSOLUTE_URL);

    return new JsonResponse($jsonBook, Response::HTTP_CREATED,
["Location" => $location], true);
}
```

Le chemin est le même que pour la récupération de l'ensemble des livres, avec la précision POST pour la méthode.

L'idée est de **récupérer** ce que l'on reçoit dans la requête (traduit ici par `$request->getContent()`), et cette fois-ci de **désérialiser** l'élément, directement dans un objet de la classe `Book`

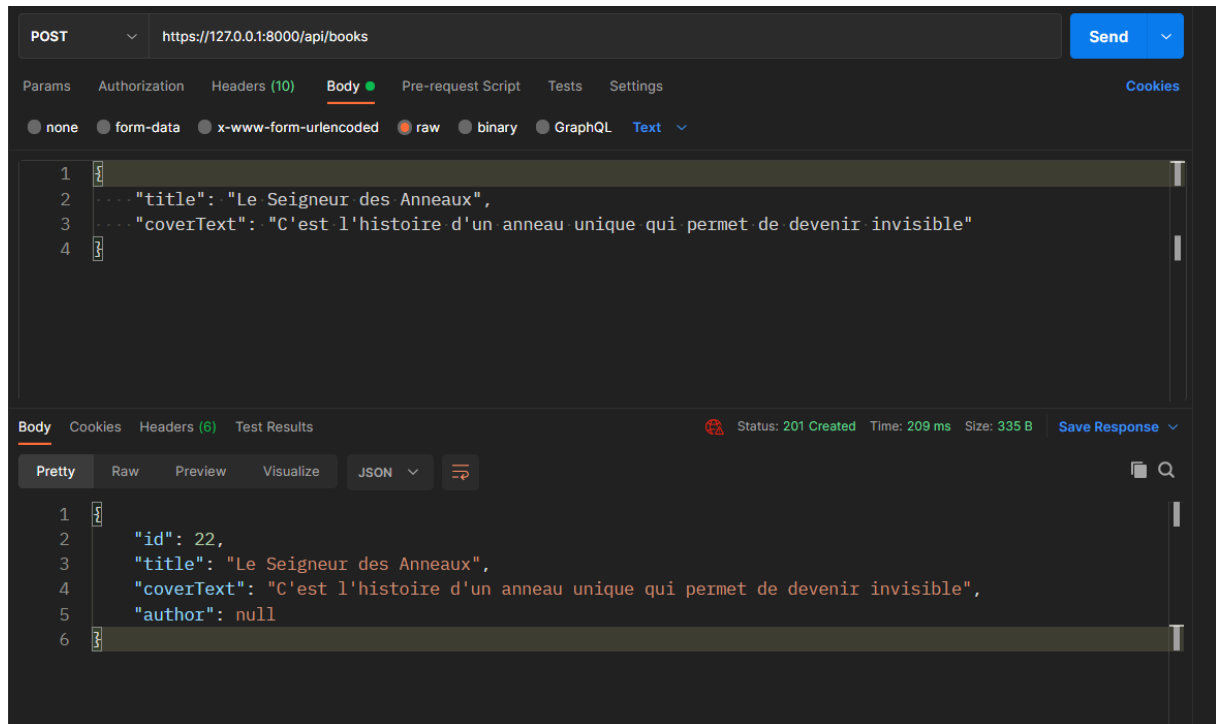
Dernier point, il est courant de retourner l'élément créé. Pour cela, on le **séréalise** et on l'envoie.

Notez également l'apparition d'un `$urlGenerator`, qui permet de générer la route qui pourrait être utilisée pour récupérer des informations sur le livre ainsi créé.

Là encore, lorsqu'on crée une ressource, il est d'usage de retourner un champ Location dans le header. Ce champ contient l'URL qui pourrait être interrogée si on voulait avoir plus d'informations sur l'élément créé.

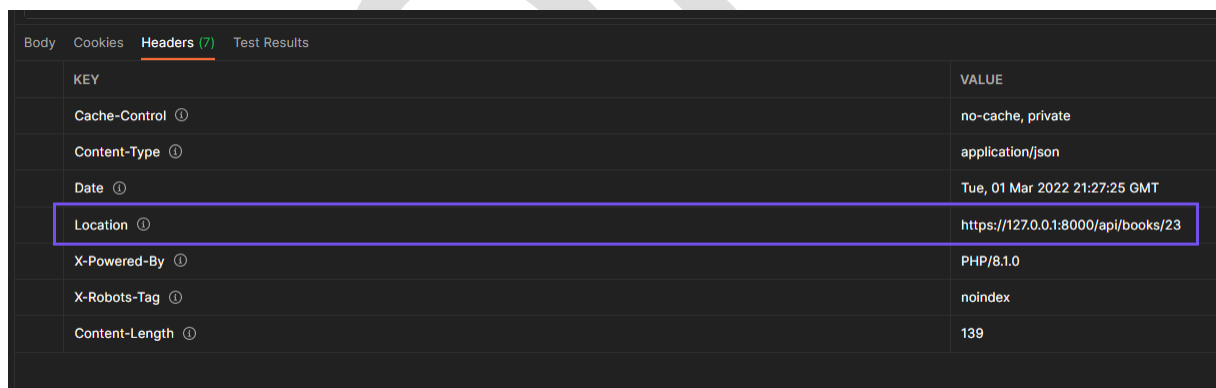
Attention, si vous avez la moindre erreur dans votre appel sur Postman, même juste **une virgule en trop**, alors la désérialisation ne se passera pas bien ! Il est fondamental d'être très rigoureux.

Faisons un petit tour sur Postman pour tester ce que nous venons de créer.



Notez en particulier l'id de l'élément créé (ici 22), et le status : **201 Created**

On peut aussi regarder l'onglet Headers et constater la présence de notre nouvelle entrée Location :



Nous pouvons remarquer que author est null il faudra donc ajouter un auteur au livre.

Plusieurs cas peuvent se présenter :

- L'auteur existe dans la BDD, il faudra juste y passer son id
- L'auteur n'existe pas dans la BDD
- on pourrait vouloir **le créer en même temps que le livre**, par exemple en passant un champ `author` qui contiendrait toutes les informations sur le nouvel auteur
- Il est également envisageable de ne rien rajouter ici pour **garder la création d'un livre simple**, et rajouter une nouvelle route `LinkBookAndAuthor` pour permettre de créer un lien entre un livre existant et un auteur existant.

Je vais ici opter pour une solution intermédiaire et me contenter de passer l'id d'un auteur existant. Si celui-ci n'existe pas, charge au client de le créer au préalable.

Dans postman nous rajouterons donc une ligne :

```
{
  "title": "Le Silmarillion",
  "coverText": "Beren et Lúthien et autres histoires",
  "idAuthor" : 5
}
```

Comme vous le voyez, j'ai passé un `idAuthor` . Mais celui-ci ne correspond pas à la désérialisation "classique".

Ici, le champ ajouté en plus s'appelle `idAuthor` et pas `author` , car il ne contient bien que l'id de l'auteur et pas sa description complète. C'est juste un moyen pour l'utilisateur de spécifier facilement l'id de l'auteur qui sera ajouté, et c'est dans le contrôleur que nous allons récupérer le véritable objet `Author` correspondant.

Nous allons donc devoir gérer ceci dans le controller :

```
#[Route('/api/books', name:"createBook", methods: ['POST'])]
public function createBook(Request $request,
SerializerInterface $serializer, EntityManagerInterface $em,
UrlGeneratorInterface $urlGenerator, AuthorRepository
$authorRepository): JsonResponse
{
    $book = $serializer->deserialize($request->getContent(),
Book::class, 'json');

    // Récupération de l'ensemble des données envoyées sous
forme de tableau
    $content = $request->toArray();

    // Récupération de l'idAuthor. S'il n'est pas défini, alors
on met -1 par défaut.
    $idAuthor = $content['idAuthor'] ?? -1;

    // On cherche l'auteur qui correspond et on l'assigne au
livre.
    // Si "find" ne trouve pas l'auteur, alors null sera
retourné.
    $book->setAuthor($authorRepository->find($idAuthor));

    $em->persist($book);
    $em->flush();

    $jsonBook = $serializer->serialize($book, 'json', ['groups'
=> 'getBooks']);

    $location = $urlGenerator->generate('detailBook', ['id' =>
$book->getId()], UrlGeneratorInterface::ABSOLUTE_URL);

    return new JsonResponse($jsonBook, Response::HTTP_CREATED,
["Location" => $location], true);
}
```

Notez comme le paramètre `idAuthor` a été récupéré directement, et comme nous l'avons utilisé pour aller chercher les informations sur l'entité `Author` pour l'assigner au `Book`.

- **Le PUT**

Nous avons donc réalisé le DELETE et le POST nous allons voir le PUT pour la modification d'un livre.

La méthode HTTP à utiliser sera donc PUT, et nous allons modifier une ressource avec un id donné dans l'URL.

Dans notre contrôleur BookController, nous pouvons créer une nouvelle méthode `updateBook` pour la mise à jour :

```
#[Route('/api/books/{id}', name:"updateBook", methods:['PUT'])]

    public function updateBook(Request $request,
    SerializerInterface $serializer, Book $currentBook,
    EntityManagerInterface $em, AuthorRepository $authorRepository):
    JsonResponse
    {
        $updatedBook =
        $serializer->deserialize($request->getContent(),
                                Book::class,
                                'json',
                                [AbstractNormalizer::OBJECT_TO_POPULATE =>
        $currentBook]);
        $content = $request->toArray();
        $idAuthor = $content['idAuthor'] ?? -1;

        $updatedBook->setAuthor($authorRepository->find($idAuthor));

        $em->persist($updatedBook);
        $em->flush();
        return new JsonResponse(null,
        JsonResponse::HTTP_NO_CONTENT);
    }
```

Ici nous pouvons voir une nouvelle option pour la désérialisation. Nous avons rajouté un paramètre, `[AbstractNormalizer::OBJECT_TO_POPULATE]`

L'idée ici est de **désérialiser** directement à l'intérieur de l'objet `$currentBook` , qui correspond au livre passé dans l'URL.

On gère toujours à la main l'**idAuthor**

Le PUT peut avoir plusieurs réponses possibles. Celle qui est recommandée est de retourner une réponse vide avec un **204 - No content**. Cela signifie que l'opération s'est bien passée.

Comme l'id de la ressource est déjà connu ainsi que les modifications demandées, il n'est pas nécessaire de retourner quoi que ce soit comme information. Le code HTTP, dans la série des **200**, est suffisant pour indiquer que tout fonctionne.

Exercice

Essayez de réaliser le CRUD pour l'auteur, cela permettra de voir si tout est bien compris.

Attention cependant, il existe une petite subtilité. Lorsqu'on veut supprimer un auteur, si des livres sont associés, la suppression sera refusée car la base de données interdit d'avoir un livre associé à un auteur qui n'existe pas !

Il vous faudra donc bien penser à faire une suppression dite *en cascade*, pour supprimer les livres en même temps que l'auteur lié.

Pensez également à **commenter votre projet** ! Cela vous permettra également de revenir sur les divers éléments, et de voir s'ils sont bien compris.

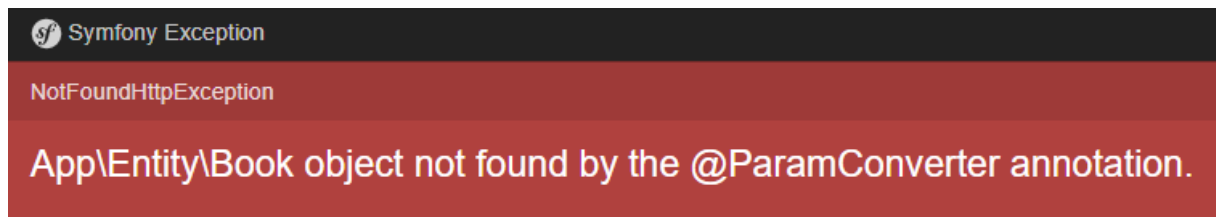
VIII La Gestion des Erreurs

Notre API est désormais fonctionnelle, nous savons faire un CRUD entier cependant il nous manque une compétence essentielle pour une API c'est la gestion des erreurs.

Nous allons voir comment centraliser la gestion des erreurs et une fois la gestion des erreurs faite nous verrons la validation des données afin de s'assurer que nos données respectent certains critères.

Il existe un mécanisme présent dans de nombreux langages, ce sont les **exceptions**. Lever une exception, c'est clamer au code "Au secours, il y a une erreur !" en espérant qu'un **try catch** va attraper cette exception et la traiter.

Dans Symfony en particulier, ce système est évidemment déjà mis en place. Si vous demandez à récupérer un livre avec un id invalide, vous obtenez ce message d'erreur :



Comme vous pouvez le voir, il s'agit d'une **NotFoundHttpException**, et Symfony l'a automatiquement traduite en page HTML, avec les informations nécessaires pour que vous puissiez comprendre le problème.

C'est très pratique lorsqu'on développe un site web, mais ici, nous travaillons avec les API. Nous souhaiterions donc avoir des informations, mais en JSON.

L'idée va donc être d'étendre le système d'exception de Symfony, et non pas de laisser Symfony faire par défaut. Nous allons décrire nous-mêmes le comportement attendu.

En créant un **ExceptionSubscriber**, c'est-à-dire un élément qui va écouter toutes les exceptions.

Comme souvent, on peut simplement commencer par une petite commande dans le terminal :

```
php bin/console make:subscriber
```

```
PS D:\Openclassrooms\Cours API Symfony\7709361-API-REST-Symfony> php bin/console make:subscriber

Choose a class name for your event subscriber (e.g. ExceptionSubscriber):
> ExceptionSubscriber

Suggested Events:
* console.command (Symfony\Component\Console\Event\ConsoleCommandEvent)
* console.error (Symfony\Component\Console\Event\ConsoleErrorEvent)
* console.terminate (Symfony\Component\Console\Event\ConsoleTerminateEvent)
* kernel.controller (Symfony\Component\HttpKernel\Event\ControllerEvent)
* kernel.controller_arguments (Symfony\Component\HttpKernel\Event\ControllerArgumentsEvent)
* kernel.exception (Symfony\Component\HttpKernel\Event\ExceptionEvent)
* kernel.finish_request (Symfony\Component\HttpKernel\Event\FinishRequestEvent)
* kernel.request (Symfony\Component\HttpKernel\Event\RequestEvent)
* kernel.response (Symfony\Component\HttpKernel\Event\ResponseEvent)
* kernel.terminate (Symfony\Component\HttpKernel\Event\TerminateEvent)
* kernel.view (Symfony\Component\HttpKernel\Event\ViewEvent)

What event do you want to subscribe to?:
> kernel.exception

created: src/EventSubscriber/ExceptionSubscriber.php

Success!

Next: Open your new subscriber class and start customizing it.
Find the documentation at https://symfony.com/doc/current/event\_dispatcher.html#creating-an-event-subscriber
PS D:\Openclassrooms\Cours API Symfony\7709361-API-REST-Symfony> 
```

Nous répondons aux questions, à savoir :

- Le nom de notre subscriber que nous allons laisser le nom par défaut mais il faut le réécrire, car cela est cohérent pour un outil qui va nous permettre de gérer les exceptions
- Les événements auxquels nous souhaitons nous connecter : Symfony6 nous donne une liste et nous choisissons ici **kernel.exception**

À ce stade, vous devriez avoir un nouveau fichier, **Exceptionssubscriber.php** , avec en particulier une méthode **onKernelException**

Cette méthode est un point de passage par lequel toutes les exceptions vont passer. C'est donc l'endroit parfait pour dire en une fois que nous voulons une réponse en JSON et pas en HTML.

```

<?php
// src\EventSubscriber\ExceptionSubscriber.php
// ...

public function onKernelException(ExceptionEvent $event): void
{
    $exception = $event->getThrowable();

    if ($exception instanceof HttpException) {
        $data = [
            'status' => $exception->getStatusCode(),
            'message' => $exception->getMessage()
        ];

        $event->setResponse(new JsonResponse($data));
    } else {
        $data = [
            'status' => 500, // Le status n'existe pas car ce
// n'est pas une exception HTTP, donc on met 500 par défaut.
            'message' => $exception->getMessage()
        ];

        $event->setResponse(new JsonResponse($data));
    }
}

```

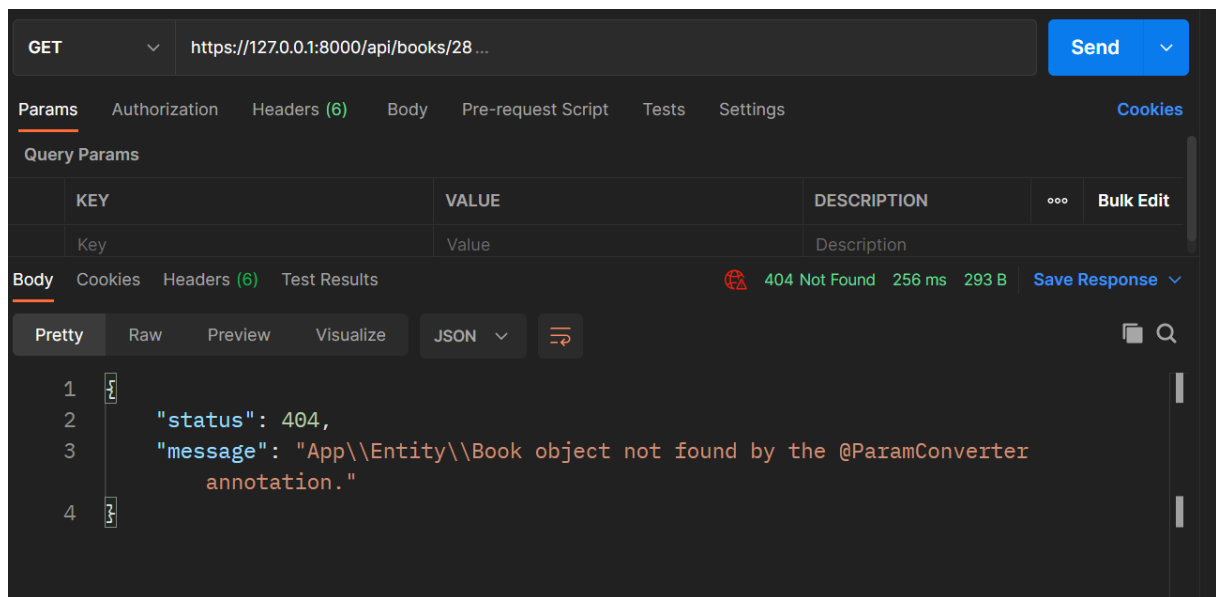
Si nous décortiquons un peu ce code, en fait, après avoir récupéré l'objet `exception`, nous vérifions si cet objet est une **HttpException** (qui donc correspond à un des codes HTTP, par exemple 404).

Si c'est le cas, nous récupérons le code et le message, puis nous changeons la réponse de l'événement pour la transformer en `JsonResponse`.

Au contraire, si ce n'est pas une `HttpException`, alors, du point de vue du client on va considérer que c'est une **erreur "générique"**. Le code associé à ce type d'erreur est 500. Donc on renseigne ce code ainsi que le message d'erreur, et on retourne à nouveau une `JsonResponse`.

Attention, comme d'habitude : n'oubliez pas les `use` qui correspondent aux classes que l'on utilise.

Regardons sur Postman, prenons un livre avec un id qui n'existe pas :



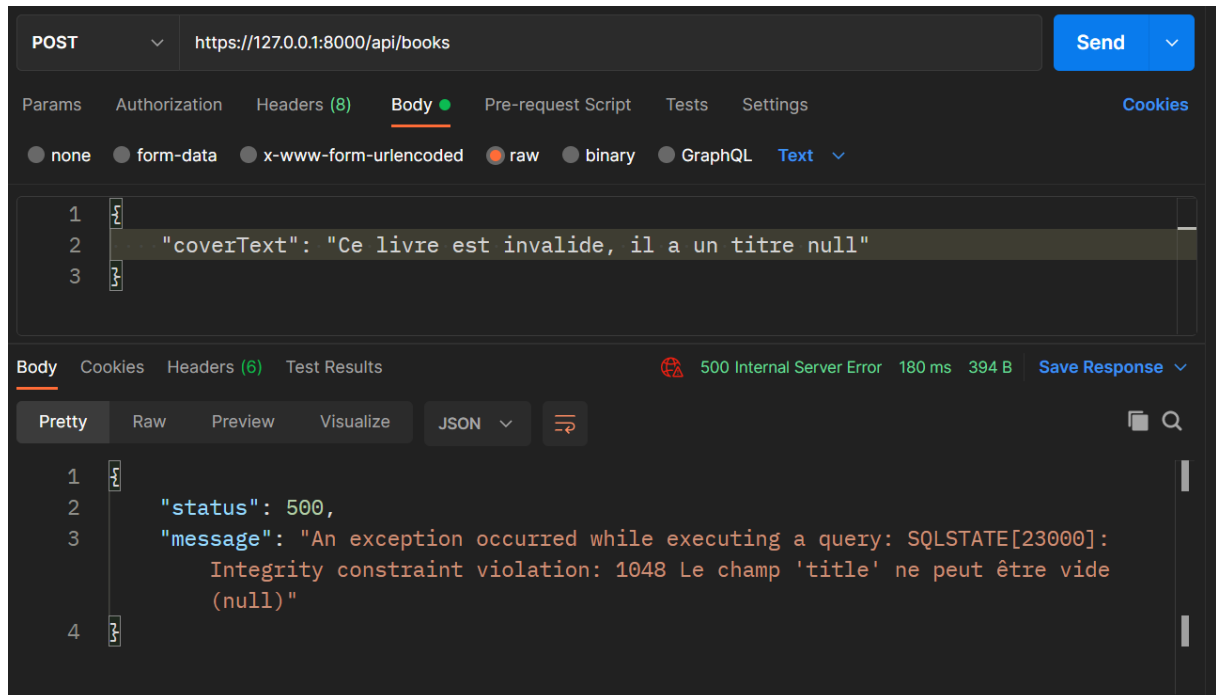
Nous avons bien une réponse JSON avec le status 404 et le message.

Notez qu'ici le traitement que j'ai fait dans la méthode `onKernelException` est simple, je me suis contenté de récupérer le code d'erreur et le message. Libre à vous de donner plus ou moins d'informations en fonction des besoins spécifiques de votre application.

IX La validation des données

Nous avons maintenant un système de gestion d'exception simple et centralisé, poussons les choses un peu plus loin.

Nous allons essayer d'envoyer un livre en oubliant son titre, nous analyserons le retour dans postman :



Lors de la création de l'entité, nous avons dit que le titre du livre ne pouvait être null, nous avons donc un retour de celle ci dans notre JSON. L'insertion du livre a donc été refusé.

Notez au passage le statut 500 qui prouve que notre système centralisé se comporte plutôt bien.

Cependant, du point de vue de l'utilisateur, ce message est très technique. Je dirais même qu'il ressemble presque à un bug. Ce que nous aimerions ici, ce n'est pas une erreur 500, mais **une vraie erreur HTTP** qui nous dirait **400 - Bad Request**. Cela indiquerait clairement à l'utilisateur que le problème ne vient pas du serveur, mais bien du fait qu'il a **mal forgé sa requête**.

En fait, il faudrait s'assurer que le livre que l'utilisateur veut créer est bien valide.

Symfony est un framework, et comme tout framework qui se respecte cela est prévu sous symfony.

Première étape, installez le package qui nous permet d'utiliser les validateurs conçus pour nous :

```
composer require symfony/validator doctrine/annotations
```

Ensuite, directement au niveau des entités, il va être possible d'écrire des **assert**. Ils vont **décrire les données que nous attendons**, et même le message d'erreur en cas d'échec.

Comme d'habitude, il faut un **use** pour utiliser ces **assert**, mais Symfony recommande de créer un "alias" plutôt que l'insertion classique :

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

Le mot **assert** est déjà utilisé dans d'autres langages et contextes pour une utilisation similaire, ce qui permet de rendre le code plus clair.

Exemple pour l'entité Book :

```
<?php
// src\Entity\Book.php
// ...
#[ORM\Column(type: 'string', length: 255)]
#[Groups(["getBooks", "getAuthors"])]
#[Assert\NotBlank(message: "Le titre du livre est obligatoire")]
#[Assert\Length(min: 1, max: 255, minMessage: "Le titre doit faire
au moins {{ limit }} caractères", maxMessage: "Le titre ne peut pas
faire plus de {{ limit }} caractères")]
private $title;
```

Nous avons indiqué deux éléments

- **Assert\NotBlank** : pour nous dire que le titre ne peut pas être null et le message qu'il doit nous retourner
- **Assert\Length** : Pour définir la taille maximal et minimal et le retour du message

Pour rappel, ces asserts sont présentés en mode php8, avec **#[Assert\Validation()]**. Vous pourrez également croiser une autre notation qui s'écrit sous forme de commentaires avec **@assert()**.

Il ne reste plus maintenant qu'à vérifier que nos entités sont valides juste avant la création, directement dans le contrôleur :

```
<?php
    // src\Controller\BookController.php
    // ...

#[Route('/api/books', name:"createBook", methods: ['POST'])]
    public function createBook(Request $request,
        SerializerInterface $serializer, EntityManagerInterface $em,
        UrlGeneratorInterface $urlGenerator, AuthorRepository
        $authorRepository, ValidatorInterface $validator): JsonResponse
    {
        $book = $serializer->deserialize($request->getContent(),
        Book::class, 'json');

        // On vérifie les erreurs
        $errors = $validator->validate($book);

        if ($errors->count() > 0) {
            return new JsonResponse($serializer->serialize($errors,
            'json'), JsonResponse::HTTP_BAD_REQUEST, [], true);
        }

        $em->persist($book);
        $em->flush();

        $content = $request->toArray();
        $idAuthor = $content['idAuthor'] ?? -1;

        $book->setAuthor($authorRepository->find($idAuthor));
        $jsonBook = $serializer->serialize($book, 'json', ['groups'
        => 'getBooks']);
        $location = $urlGenerator->generate('detailBook', ['id' =>
        $book->getId()], UrlGeneratorInterface::ABSOLUTE_URL);

        return new JsonResponse($jsonBook, Response::HTTP_CREATED,
        ["Location" => $location], true);
    }
```


Notez l'apparition dans les paramètres de la méthode d'un nouvel élément : le **ValidatorInterface**.

Ce validator va tout simplement **valider** notre entité et **retourner** un objet `error` contenant l'ensemble des erreurs

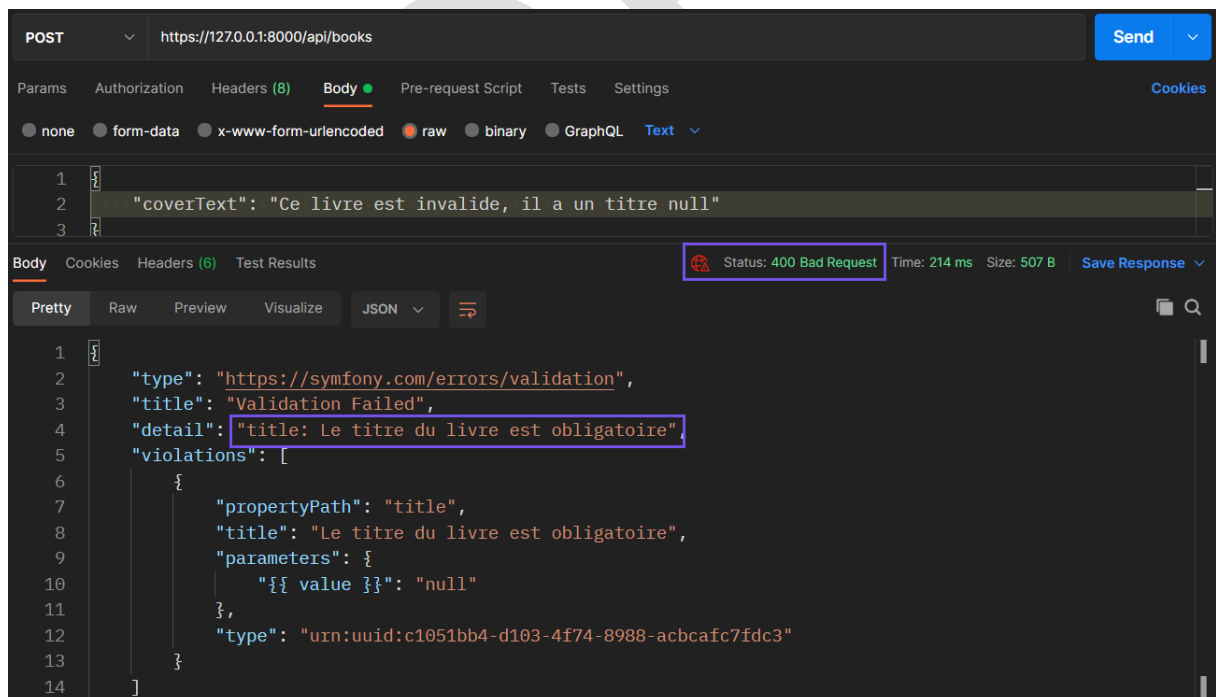
Si cet objet **error** n'est pas vide, c'est que la demande initiale n'était pas correcte. On retourne donc une `JsonResponse` avec notre erreur **400 - Bad Request**.

Grâce à notre subscriber créé ci-dessus, plutôt que de retourner l'erreur sérialisée, nous aurions pu aussi écrire :

```
throw new HttpException(JsonResponse::HTTP_BAD_REQUEST, "La requête est invalide");
```

Ici j'ai préféré retourner directement l'erreur, afin que chacun puisse visualiser exactement les informations à disposition en cas d'erreur de validation. Il est évident que c'est un code à adapter en fonction des projets et des besoins.

Retournons sur Postman pour voir le résultat :



Nous avons maintenant un statut **400 - Bad Request**, bien plus approprié, et même le message d'erreur.

Exercice

C'est désormais à votre tour de rajouter une ou plusieurs validations. En particulier, le nom d'un auteur ne peut pas être vide.

N'oubliez pas non plus, la création n'est pas le seul endroit où l'on peut spécifier des champs.

X La création d'utilisateur

Passons maintenant à **l'authentification**. En effet, il n'est pas nécessairement souhaitable que tout le monde puisse réaliser toutes les opérations.

Même si un simple utilisateur doit pouvoir **consulter** les titres de notre bibliothèque, il semble plutôt logique que seul le bibliothécaire, autrement dit l'administrateur, puisse **ajouter** de nouveaux livres.

L'authentification étant un mécanisme présent dans la plupart des projets, Symfony a déjà mis en place des éléments pour nous faciliter la vie.

La toute première étape est d'installer le composant **Security**. Saisissez la commande suivante:

```
composer require security
```

Ensuite, il nous faut stocker les utilisateurs. Pour cela, nous pourrions tout à fait créer une nouvelle entité `User` à la main, mais comme je l'ai mentionné plus tôt, il est tellement commun d'avoir à gérer des utilisateurs dans une application, que Symfony a carrément créé une commande spécifiquement pour ça :

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

```

The name of the security user class (e.g. User) [User]:
>

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
>

Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
>

Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed
by some other system (e.g. a single sign-on server).

Does this app need to hash/check user passwords? (yes/no) [yes]:
>

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml

Success!

Next Steps:
- Review your new App\Entity\User class.
- Use make:entity to add more fields to your User entity and then run make:migration.
- Create a way to authenticate! See https://symfony.com/doc/current/security.html

```

Symfony nous pose quelques questions, nous pouvons nous contenter d'accepter les choix par défaut, et nous voilà avec la nouvelle entité `User` créée.

Petit point de détail pour nous simplifier la vie plus tard, nous allons ajouter une méthode `getUsername()` dans l'entité `User.php`. Celle-ci sera utilisée plus tard par JWT pour nous authentifier.

```

<?php
//src\Entity\User.php
//...

/**
 * Méthode getUsername qui permet de retourner le champ qui est
 * utilisé pour l'authentification.
 *
 * @return string
 */
public function getUsername(): string {
    return $this->getUserIdentifier();
}

```

Cette méthode très simple est juste un alias vers `$this->getUserIdentifier()`.

Faisons maintenant la migration et rendez-vous ensuite sur phpMyAdmin :

← Serveur : MySQL:3306 » Base de données : books » Table : user

Parcourir

Structure



SQL

Rechercher

Insérer

Structure de table

Vue relationnelle

	#	Nom	Type	Interclassement	Attributs	Null	Valeur par d
<input type="checkbox"/>	1	id 	int(11)			Non	Aucun(e)
<input type="checkbox"/>	2	email 	varchar(180)	utf8mb4_unicode_ci		Non	Aucun(e)
<input type="checkbox"/>	3	roles	json			Non	Aucun(e)
<input type="checkbox"/>	4	password	varchar(255)	utf8mb4_unicode_ci		Non	Aucun(e)

La table est bien créée. Nous voyons en particulier l'`email` et le `password` qui serviront de champs pour s'authentifier, et un champ `roles` qui va contenir le rôle de notre user, afin de savoir s'il est admin ou simple utilisateur.

Puisque notre BDD est prête pour accueillir nos utilisateurs créons deux utilisateurs avec nos fixtures.

Rappel : **Lorsqu'on enregistre un mot de passe en base de données, celui-ci ne doit jamais être en clair !**

Quelle qu'en soit la raison, si quelqu'un accède à votre base, il n'aura ainsi pas accès au couple "nom d'utilisateur - mot de passe" de vos utilisateurs, et ne pourra pas se servir de ces informations à des fins malveillantes !

Mettons donc à jour notre code dans le fichier AppFixtures.php

```
<?php
    // src\DataFixtures\AppFixtures.php

namespace App\DataFixtures;

use App\Entity\Author;
use App\Entity\Book;
use App\Entity\User;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;
use
Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface
;

class AppFixtures extends Fixture
{
    private $userPasswordHasher;

    public function __construct(UserPasswordHasherInterface
$userPasswordHasher)
    {
        $this->userPasswordHasher = $userPasswordHasher;
    }

    public function load(ObjectManager $manager): void
    {
        // Création d'un user "normal"
        $user = new User();
        $user->setEmail("user@bookapi.com");
        $user->setRoles(["ROLE_USER"]);

        $user->setPassword($this->userPasswordHasher->hashPassword($user,
"password"));
        $manager->persist($user);

        // Création d'un user admin
        $userAdmin = new User();
        $userAdmin->setEmail("admin@bookapi.com");
        $userAdmin->setRoles(["ROLE_ADMIN"]);

        $userAdmin->setPassword($this->userPasswordHasher->hashPassword($us
erAdmin, "password"));
    }
}
```

```

$manager->persist($userAdmin);

// Création des auteurs.
$listAuthor = [];
for ($i = 0; $i < 10; $i++) {
    // Création de l'auteur lui-même.
    $author = new Author();
    $author->setFirstName("Prénom " . $i);
    $author->setLastName("Nom " . $i);
    $manager->persist($author);

    // On sauvegarde l'auteur créé dans un tableau.
    $listAuthor[] = $author;
}

for ($i = 0; $i < 20; $i++) {
    $book = new Book();
    $book->setTitle("Titre " . $i);
    $book->setCoverText("Quatrième de couverture numéro : "
. $i);
    $book->setAuthor($listAuthor[array_rand($listAuthor)]);
    $manager->persist($book);
}

$manager->flush();
}
}

```

Nous avons ajouté, dans la méthode `load`, deux éléments : le `ROLE_USER` pour créer un utilisateur normal, et un second, avec le `ROLE_ADMIN`.

Notez en particulier **comment le mot de passe a été encodé**. Symfony nous fournit un `UserPasswordEncoderInterface` qui est une classe qui nous permet de réaliser cet encodage.

Cependant, comme nous ne sommes pas dans un contrôleur, nous ne pouvons pas directement récupérer cet élément en le passant dans les paramètres. Il nous faut créer un **constructeur**, qui lui peut recevoir cet élément, le sauvegarder dans le membre local `$this->userPasswordEncoder`, et seulement après nous en servir dans la méthode `load`.

```
$this->userPasswordHasher->hashPassword($userAdmin, "password")
```

Cette instruction permet de retourner le mot de passe encodé de l'utilisateur.

Envoyons maintenant nos Fixtures en BDD :

```
php bin/console doctrine:fixtures:load
```

Normalement tout devrait bien se passer, il nous reste seulement à vérifier dans la base de données que nos utilisateurs sont présents :

id	email	roles	password
1	user@bookapi.com	["ROLE_USER"]	\$2y\$13\$yAIVE0f4wfTNI0FCoUy3j.yc7rZbRQ4osxUBSAkJ56A...
2	admin@bookapi.com	["ROLE_ADMIN"]	\$2y\$13\$vgY1ymwR/imb.W4fXDhn2eed82KOMumu.nW1O/dEy0c...

Notez le contenu du champ `password`. Alors que j'ai demandé dans mes fixtures à créer à chaque fois le même mot de passe (qui est "password"), dans la base est stockée deux **longues chaînes de caractères**, et en plus **différentes** pour les deux utilisateurs créés.

Cela permet de ne donner aucun indice (même le fait que les deux mots de passe sont les mêmes !) à un éventuel pirate qui aurait accès à notre base, et voudrait réutiliser le mot de passe ailleurs.

Notez également le contenu du champ `roles`, qui indique clairement quels rôles ont nos utilisateurs

XI Le JWT : La protection de l'API

Maintenant que nos utilisateurs existent, il va falloir créer un mécanisme pour que l'API puisse authentifier et savoir que cette authentification a réussi.

Rappelez-vous, une API REST est **stateless** (sans état), c'est-à-dire qu'à chaque appel, elle a complètement oublié ce qui a pu se passer à l'appel précédent, nous ne pouvons donc pas stocker les informations de l'utilisateur en session.

La solution pour pallier ce problème va être de faire un premier appel pour s'authentifier. Ce premier appel va retourner un **token**, encodé, qui va contenir les **informations sur la personne qui vient de s'authentifier**.

À l'appel suivant, dans le header Authorization, il suffira de **renvoyer ce token** pour dire à l'application "Je suis authentifié, voici la preuve grâce à ce token."

Ceci peut se faire grâce à **JWT**, qui signifie **JSON Web Token**, et qui est un standard qui définit les diverses étapes pour échanger les informations d'authentification de manière sécurisée.

La sécurité étant un point très important, nous allons nous aider d'un bundle très populaire, fiable et open source : LexikJWT.

Il faudra donc l'installer avec la commande suivante :

```
composer require lexik/jwt-authentication-bundle
```

Ensuite, pour fonctionner, Lexik a besoin de **générer des clefs**. Sans rentrer dans les détails, il y a une clef publique et une clef privée.

La clef **publique** permet d'encoder le token généré, et la clef **privée** permet de le décoder. Comme la clef privée est... privée, même si quelqu'un intercepte le token, sans la clef privée il ne saura pas comment le décoder, et ne pourra donc rien en faire.

Nous allons devoir créer les clefs avec les lignes de commandes suivantes :

```
openssl genpkey -out config/jwt/private.pem -aes256 -algorithm  
rsa-pkeyopt rsa_keygen_bits:4096
```

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

Une "passphrase" vous sera demandée. Cette passphrase va en quelque sorte servir de clef pour l'encodage/décodage du token. Elle doit rester secrète, lors de l'écriture aucun caractère n'apparaîtra c'est normal ...

Il arrive parfois que cette partie pose problème.

En effet, **openSSL** n'est pas nécessairement installé sur le terminal par défaut, en particulier sous Windows. En revanche, si vous avez installé Git, vous devriez probablement avoir accès à GitBash qui est également un terminal, mais avec openSSL installé.

Si cela ne marche pas, comme c'est un problème courant, suivant votre OS et votre configuration, il ne devrait pas être très compliqué de trouver un tutoriel pas à pas qui corresponde à votre cas, et bien sûr il y a toujours la [documentation officielle](#) !

Notez qu'ici nous enregistrons nos clefs dans le dossier `config/jwt`. Il faut que ce dossier existe, sinon la commande échouera.

Passons à la configuration de JWT:

Maintenant que JWT est installé et que nos clefs ont été créées, nous devons dire à Symfony où elles se trouvent.

Dans le fichier `.env.local`, nous pouvons écrire ceci :

```
###> 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=fdd719e8855fdf770a5141fd0afb817b

###< lexik/jwt-authentication-bundle ###
```

Le contenu de `JWT_PASSPHRASE` doit être ce que vous avez tapé en `passphrase` pendant la génération de vos clefs ! Cette information est importante et ne doit PAS être envoyée sur Git, ce pourquoi nous modifions le fichier `.env.local` et PAS le fichier `.env`.

Maintenant, nous devons indiquer à Symfony quelle est l'URL qui va être utilisée pour l'authentification, et lui dire également que nous voulons que ce soit LexikJWT qui s'occupe de tout.

Pour cela, il faut aller dans le fichier `security.yaml` et le mettre à jour :

```
#config/packages/security.yaml

security:
    enable_authenticator_manager: true
    #
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

    #
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
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

    # A METTRE EN COMMENTAIRE !
    # main:
    #     lazy: true
    #     provider: app_user_provider

    # activate different ways to authenticate
    #
https://symfony.com/doc/current/security.html#the-firewall

    #
https://symfony.com/doc/current/security/impersonating\_user.html
```

```

# switch_user: true

login:
    pattern: ^/api/login
    stateless: true
    json_login:
        check_path: /api/login_check
        success_handler:
lexik_jwt_authentication.handler.authentication_success
        failure_handler:
lexik_jwt_authentication.handler.authentication_failure
    api:
        pattern: ^/api
        stateless: true
        jwt: ~

# 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: ^/api/login, roles: PUBLIC_ACCESS }
    - { path: ^/api, roles: IS_AUTHENTICATED_FULLY }

when@test:
    security:
        password_hashers:
            # By default, password hashers are resource intensive and
take time. This is
            # important to generate secure password hashes. In tests
however, secure hashes
            # are not important, waste resources and increase test
times. The following
            # reduces the work factor to the lowest possible values.

Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInter
face:

    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

```

Attention : Le fichier YAML est très sensible, faites bien attention à l'indentation, elle est utilisée pour savoir quelles options sont dans quels groupes.

Ici, j'ai notamment ajouté un bloc `login` et un bloc `api` pour donner à Symfony l'URL que nous allons utiliser pour nous authentifier, et lui dire que nous voulons utiliser JWT.

J'ai également mis à jour le bloc `access_control` pour dire que toutes les routes doivent être authentifiées, sauf la route `/api/login` .

Dernier point, j'ai commenté le bloc `main` pour éviter un parasitage.

Il nous reste encore à aller dans le fichier `route.yaml` et à lui rajouter un bloc `api_login_check` pour créer la route elle-même.

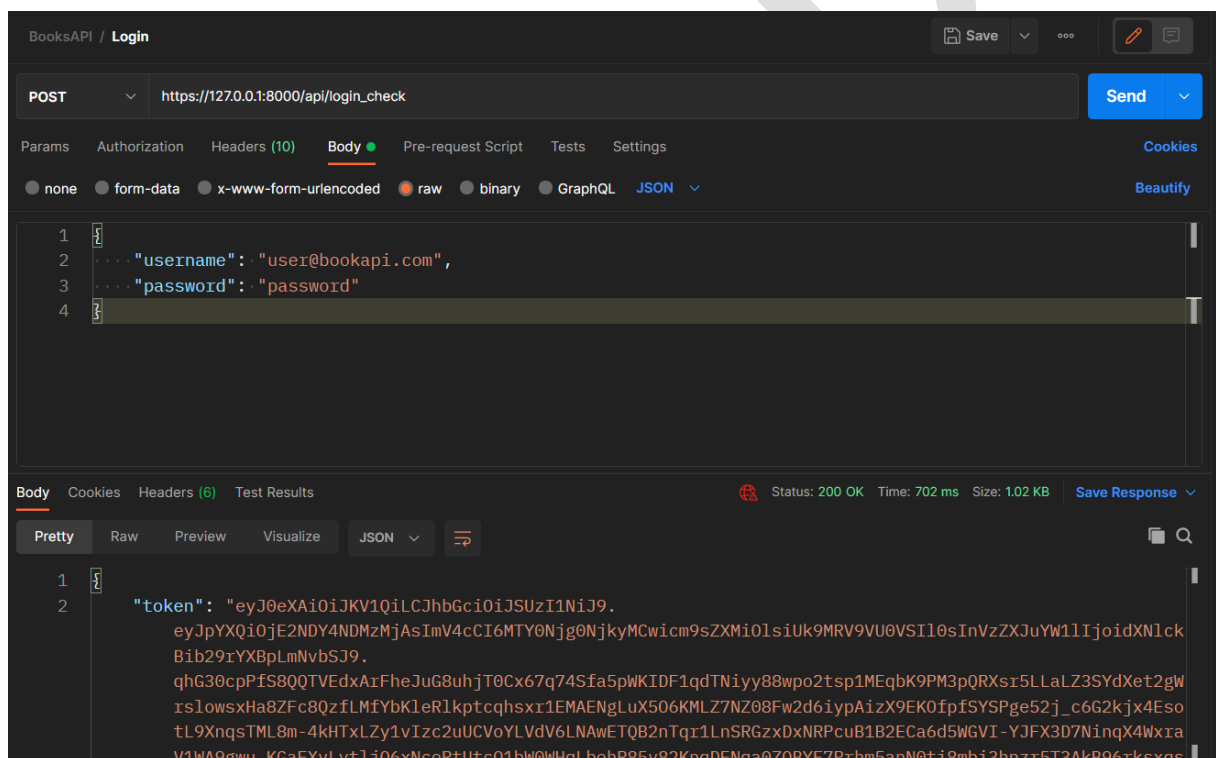
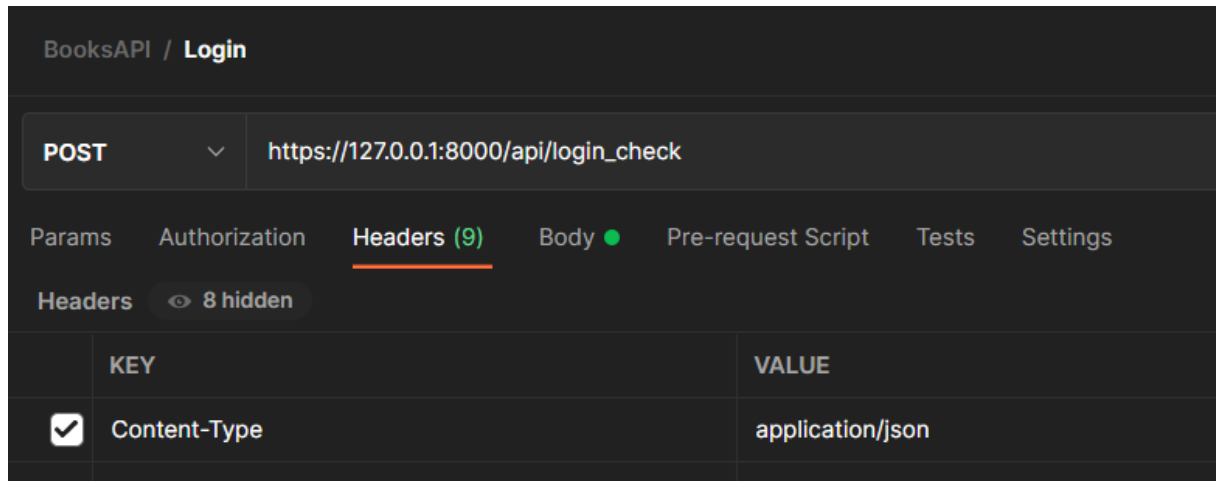
```
# config/routes.yaml
controllers:
    resource: ../src/Controller/
    type: annotation

kernel:
    resource: ../src/Kernel.php
    type: annotation

api_login_check:
    path: /api/login_check
```

Il ne nous reste plus qu'à tester tout ça dans postman :

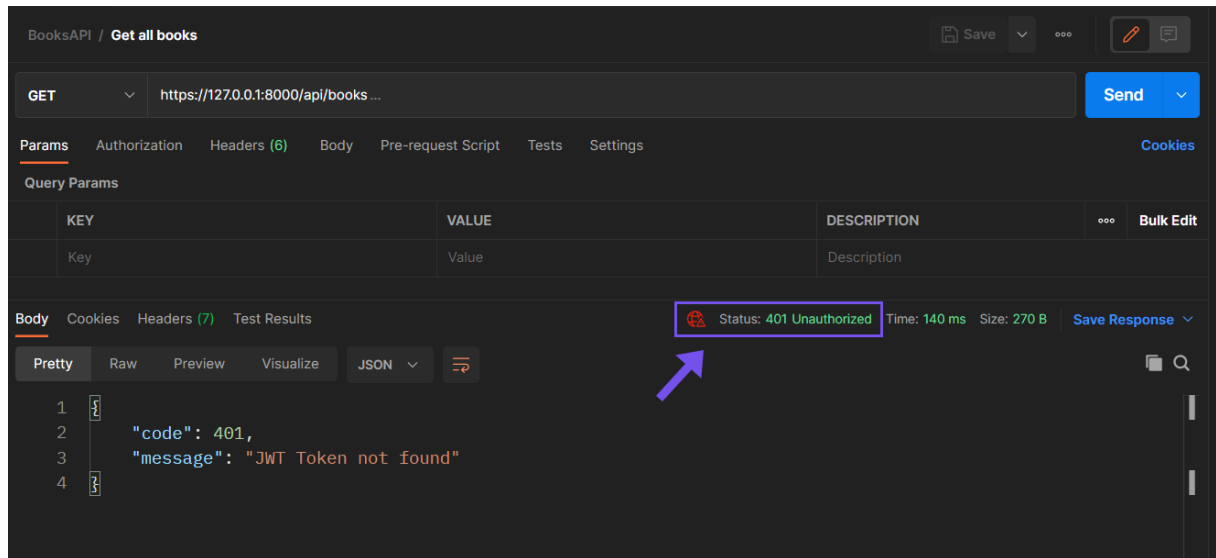
Avec Postman, en mode POST, il va falloir appeler l'URL `login_check` et passer dans le body les informations liées à l'authentification. Il faut également configurer le header pour passer une information en plus, `Content-Type` : `application/json` .



Nous voyons que nous récupérons bien le token en fonction du user@bookapi.com

XII La Gestion des droits

Nous avons créé notre token. Que se passe-t-il désormais si nous essayons, par exemple, d'accéder à la liste des livres ?

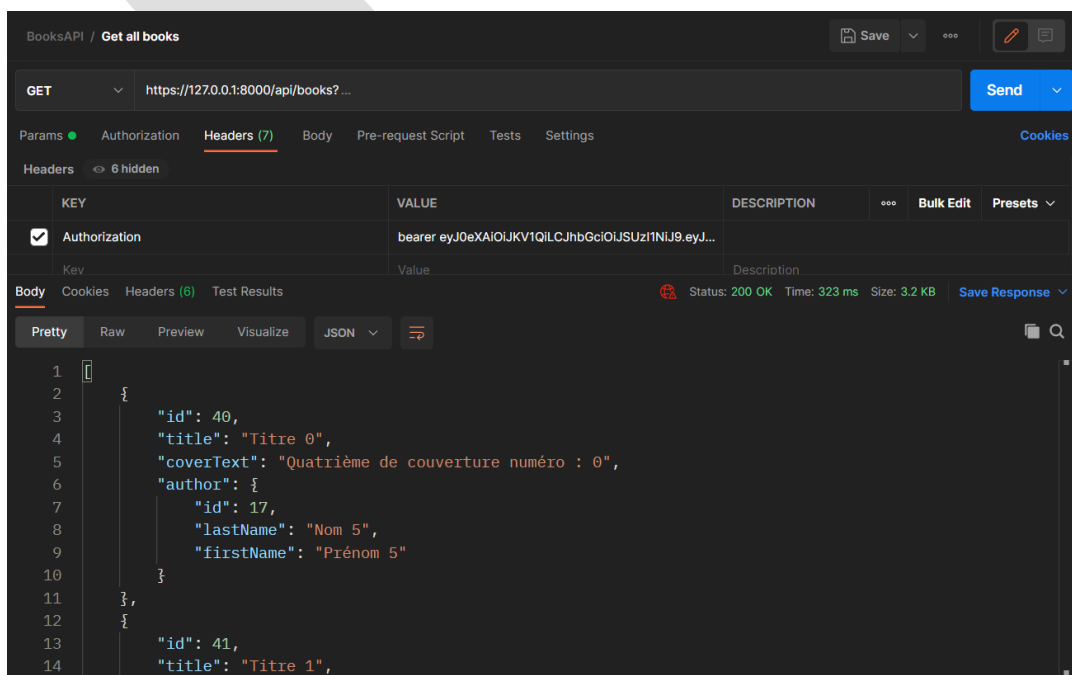


Nous avons un retour **401 unauthorized**

Cela signifie que notre application est bien **protégée**. Il n'est plus possible d'accéder aux ressources sans être authentifié, sous peine d'obtenir un code de retour.

Souvenez-vous, les API REST sont stateless. C'est-à-dire que s'être authentifié et avoir reçu le token n'est pas suffisant, il nous faudra désormais **renvoyer ce token à chaque demande** pour prouver que nous sommes bien authentifiés.

Pour cela, il faut aller dans l'onglet Headers et rajouter une nouvelle entrée : **Authorization** . Et pour la valeur, il faut écrire le mot **Bearer** suivi d'un **espace** et du **token** que vous aurez copié-collé.



C'est fait maintenant les données apparaissent vous êtes authentifié.

Poussons plus loin et nous souhaitons définir les ressources en fonction des rôles. Effectivement un user n'aura pas les mêmes droits qu'un admin.

Essayons maintenant de faire en sorte que seul un utilisateur avec le rôle administrateur puisse créer un nouveau livre.

En fait, maintenant que le système est en place, c'est très facile à faire. Il suffit d'ajouter une **annotation** directement au-dessus de la méthode pour dire quel rôle est autorisé à accéder à la méthode, ainsi qu'un message d'erreur pour les utilisateurs qui n'ont pas les autorisations requises.

Regardons ça de plus près dans notre BookController :

```
<?php
// src\Controller\BookController.php
// ...

#[Route('/api/books', name:"createBook", methods: ['POST'])]
#[IsGranted('ROLE_ADMIN', message: 'Vous n\'avez pas les droits
suffisants pour créer un livre')]
public function createBook(Request $request,
SerializerInterface $serializer, EntityManagerInterface $em,
UrlGeneratorInterface $urlGenerator, AuthorRepository
$authorRepository, ValidatorInterface $validator): JsonResponse
{
    $book = $serializer->deserialize($request->getContent(),
Book::class, 'json');

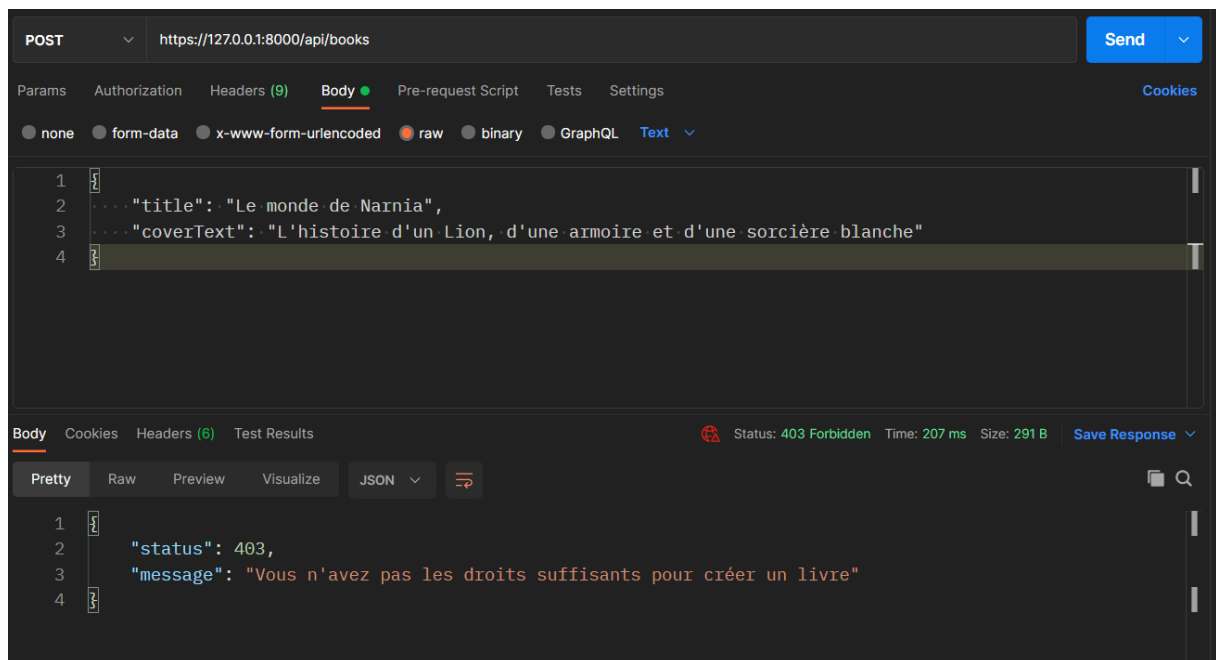
    (...)
}
```

Comme toujours, n'oubliez pas le `use` , ici :

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
```

Pour tester, essayez d'abord de générer un token pour un simple `user` . Il faut donc que le `password` et le `username` correspondent à un utilisateur ayant le rôle `ROLE_USER` .

Avec nos fixtures, c'est l'utilisateur `user@bookapi.com` :



Et si vous testez, alors vous recevez le message d'erreur que nous avons spécifié directement avec nos annotations, et à nouveau un code de retour **403 - Forbidden**

Testez à nouveau avec un token généré pour un utilisateur qui possède le **ROLE_ADMIN** . Avec nos fixtures, c'est celui qui a pour username : `admin@bookapi.com` .

Bravo tout fonctionne

XIII La Pagination

Au plus on avance dans la construction d'une application au plus elle devient grande et nous devons créer de l'optimisation

Les plus classiques sont la mise en place d'un **système de pagination**, ce qui permet de récupérer les données petit à petit plutôt qu'en bloc, et la mise en place d'un **système de cache**, qui permet d'économiser du temps de traitement si plusieurs requêtes identiques sont faites à la suite.

Occupons nous de la pagination et nous nous occuperons du cache par la suite.

La pagination permet d'éviter de retourner toutes les données d'un coup. Si vous avez une base de données avec un million de livres, on peut facilement imaginer que notre `getAllBooks` va retourner une quantité considérable de données, et que cela risque de poser des problèmes de performance.

Pour pallier ce problème, au lieu de tout retourner, ne vont être retournés que les X premiers éléments à partir de la page Y.

Pour l'instant, lorsqu'on demande la liste des livres, nous utilisons la méthode toute faite `findAll()` fournie par Doctrine. Ici, l'idée va être non pas de tout récupérer, mais de récupérer **un certain nombre** d'éléments à partir d'un début donné.

Pour cela, la première étape va être de nous créer une nouvelle méthode `findAllWithPagination` dans le `BookRepository`, qui attend 2 paramètres, à savoir le numéro de page et le nombre de livres à récupérer :

```
<?php
// src\Repository\BookRepository.php
// ...

public function findAllWithPagination($page, $limit) {
    $qb = $this->createQueryBuilder('b')
        ->setFirstResult(($page - 1) * $limit)
        ->setMaxResults($limit);
    return $qb->getQuery()->getResult();
}
```

Ici nous utilisons le `QueryBuilder` de Doctrine pour créer une nouvelle requête qui va retourner l'ensemble des livres, en spécifiant :

- le `firstResult` : à partir de quand nous récupérons les livres
- le `maxResult` : le nombre de livres que l'on souhaite retourner

Ensuite il nous suffit, dans le contrôleur, de remplacer le `findAll` initial par notre `findAllWithPagination` :

```
<?php
// src\Controller\BookController.php
// ...

#[Route('/api/books', name: 'books', methods: ['GET'])]
public function getAllBooks(BookRepository $bookRepository,
SerializerInterface $serializer, Request $request): JsonResponse
{
    $page = $request->get('page', 1);
    $limit = $request->get('limit', 3);
    $bookList = $bookRepository->findAllWithPagination($page,
$limit);

    $jsonBookList = $serializer->serialize($bookList, 'json',
['groups' => 'getBooks']);

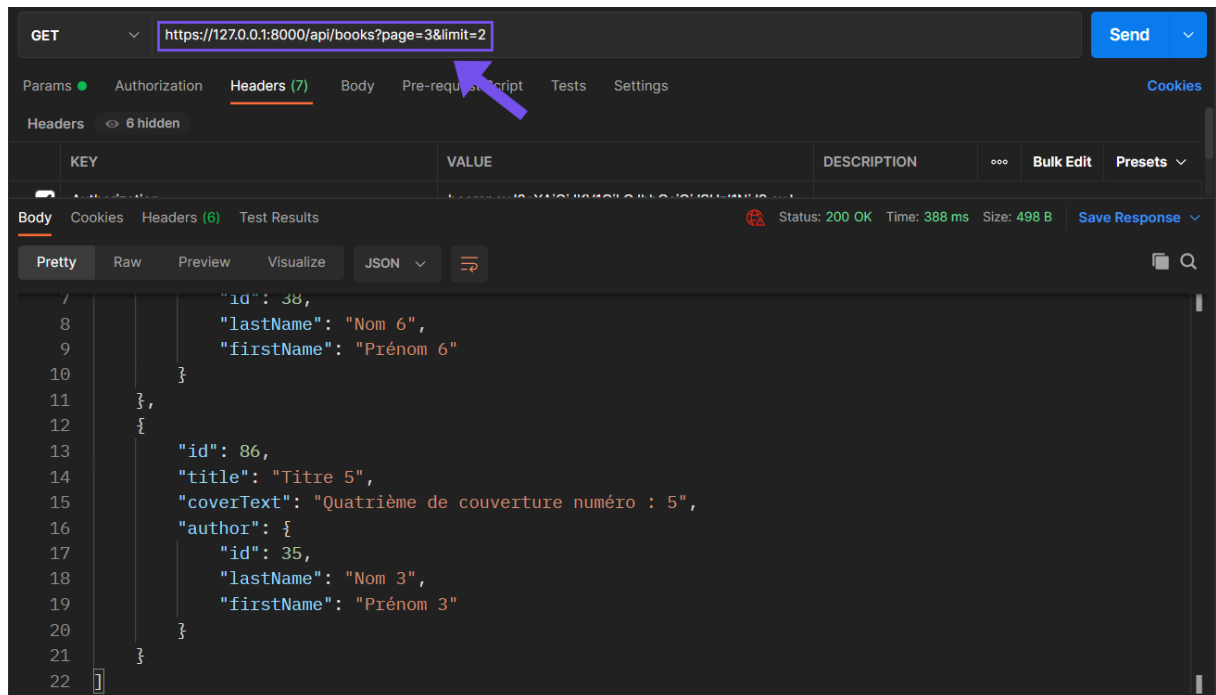
    return new JsonResponse($jsonBookList, Response::HTTP_OK,
[], true);
}
```

Notez la récupération dans la requête des paramètres `page` et `limit` avec une valeur par défaut. Cela signifie que si on ne spécifie rien pour le paramètre `page` , alors c'est la première `page` qui sera retournée. Si on ne spécifie rien pour le paramètre `limit` , alors trois éléments seront retournés.

Ici, j'ai utilisé comme noms de paramètres `page` et `limit` , mais dans l'usage courant, il se peut que vous croisie le mot `offset` à la place de `page` .

À ce stade, si nous demandons avec Postman, nous pouvons passer un paramètre `page` et un paramètre `limit` .

Notre URL serait donc : <https://127.0.0.1:8000/api/books?page=3&limit=2>.



Et là, nous obtenons bien seulement 2 livres à partir de la page 3.

Si effectivement vous désirez aller plus loin, il va alors devenir pertinent d'utiliser une librairie comme par exemple [Knnpaginator](#) ou encore [PagerFanta](#).

XIV Le cache

La pagination est ok, voyons maintenant le cache.

Le principe d'un système de cache est d'éviter de recalculer la réponse à chaque fois, lorsque les mêmes requêtes sont réalisées plusieurs fois. Au lieu de cela, on va **reprendre directement ce qui a été enregistré dans le cache**. Ainsi, la récupération des livres en base de données ne sera faite qu'à la première requête. Pour les requêtes suivantes, les données seront directement récupérées depuis le cache.

Reprenons notre fonction `getAllBooks` dans le `BookController.php` pour ajouter le cache :

```
<?php
// src\Controller\BookController.php
// ...

#[Route('/api/books', name: 'books', methods: ['GET'])]
public function getAllBooks(BookRepository $bookRepository,
SerializerInterface $serializer, Request $request,
TagAwareCacheInterface $cachePool): JsonResponse
{
    $page = $request->get('page', 1);
    $limit = $request->get('limit', 3);

    $idCache = "getAllBooks-" . $page . "-" . $limit;
    $bookList = $cachePool->get($idCache, function
(ItemInterface $item) use ($bookRepository, $page, $limit) {
        $item->tag("booksCache");
        return $bookRepository->findAllWithPagination($page,
$limit);
    });

    $jsonBookList = $serializer->serialize($bookList, 'json',
['groups' => 'getBooks']);
    return new JsonResponse($jsonBookList, Response::HTTP_OK,
[], true);
}
```

J'ai créé ici un identifiant `idCache` . Il est construit ici avec le mot `getAllBooks` auquel j'ai ajouté les valeurs de `page` et `limit` , séparés par des tirets.

Cela permettra de faire une mise en cache différenciée par page et limit. Par exemple, la première fois qu'on demandera la 3ème page, avec 4 éléments par page, le calcul sera fait, et stocké sous l'identifiant "getAllBooks-3-4".

Le prochain appel avec ces mêmes paramètres retournera directement le résultat sans même avoir besoin d'interroger la base de données.

Ensuite, je demande à mon cache, fourni par `CacheInterface` de Symfony (n'oubliez pas le `use`), de me retourner l'élément mis en cache, ici la liste de mes livres. Cela se fait avec le `$cachePool->get()` .

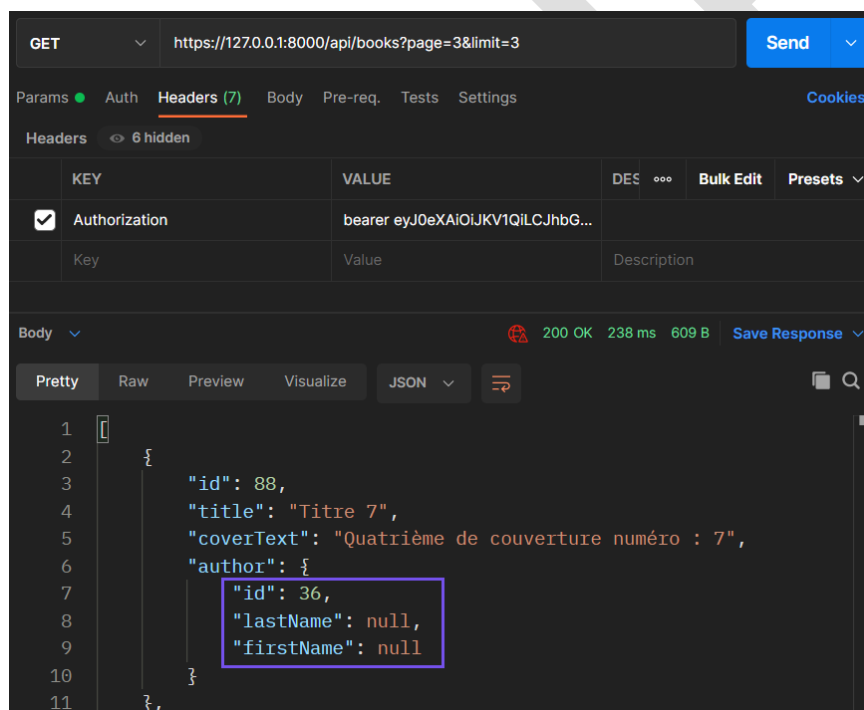
Si rien n'a été mis en cache, alors la méthode GET ne peut rien retourner. Donc la fonction passée en second argument fait le calcul (ici la récupération de mes données), met le résultat en cache, et le retourne.

Notez la présence de `$item` (qui représente l'item mis en cache) et le fait que je lui passe un tag, `"booksCache"`. Ce tag nous sera utile plus tard pour nettoyer le cache.

Lors des tests, vu que la mise en cache a été réalisée, il faut penser à **vider le cache** après chaque appel. Une des méthodes (nous en verrons une autre juste en dessous) sera tout simplement d'utiliser une nouvelle ligne de commande :

```
php bin/console cache:clear
```

XIV Gestion du Lazy Loading de Doctrine



Cela est dû à ce qu'on appelle le **lazy loading**, ou chargement fainéant, en français.

Pour des raisons de performances qui en temps ordinaire sont complètement transparentes pour nous, lorsque Doctrine nous retourne des données, en réalité ce n'est pas l'intégralité des données qui sont retournées. Les sous-entités ne sont pas chargées avant qu'on essaie de les lire, pour les afficher ou, dans notre cas, qu'on essaie de les transformer en JSON.

Ceci permet dans de nombreux cas d'économiser du temps de traitement, mais ici, comme nous mettons en cache la réponse directement sans au préalable accéder aux données, nous ne les avons pas.

Doctrine nous retourne un objet **author** avec certes l'**id** , mais hélas aussi avec **null** pour l'ensemble des autres champs.

L'option la plus simple pour résoudre ce problème est donc de faire la mise en cache **après** avoir converti les données en JSON.

Regardons la nouvelle version de notre méthode **getAllBooks** du fichier **BookController.php** :

```
<?php
// src\Controller\BookController.php
// ...

#[Route('/api/books', name: 'books', methods: ['GET'])]
public function getAllBooks(BookRepository $bookRepository,
SerializerInterface $serializer, Request $request,
TagAwareCacheInterface $cache): JsonResponse
{
    $page = $request->get('page', 1);
    $limit = $request->get('limit', 3);

    $idCache = "getAllBooks-" . $page . "-" . $limit;

    $jsonBookList = $cache->get($idCache, function
(ItemInterface $item) use ($bookRepository, $page, $limit,
$serializer) {
        $item->tag("booksCache");
        $bookList =
$bookRepository->findAllWithPagination($page, $limit);
        return $serializer->serialize($bookList, 'json',
['groups' => 'getBooks']);
    });

    return new JsonResponse($jsonBookList, Response::HTTP_OK,
[], true);
}
```

Cette solution est plus pertinente à tout point de vue, car le cache va non seulement optimiser la requête à la base de données, mais également le temps de conversion des données en JSON.

Neutralisez le mécanisme de lazy loading de Doctrine

Il existe également une autre solution, sans toucher au contrôleur. Il faut dire à Doctrine, dans la méthode `findAllWithPagination`, que nous ne voulons pas un lazy loading, mais bien un **chargement complet** des données pour que la mise en cache puisse se faire sans souci.

Voici ce que nous aurions pu écrire dans `BookRepository.php` pour mettre à jour notre méthode `findAllWithPagination` :

```
<?php
// src\Repository\BookRepository.php
// ...

public function findAllWithPagination($page, $limit) {
    $qb = $this->createQueryBuilder('b')
        ->setFirstResult(($page - 1) * $limit)
        ->setMaxResults($limit);

    $query = $qb->getQuery();
    $query->setFetchMode(Book::class, "author",
        \Doctrine\ORM\Mapping\ClassMetadata::FETCH_EAGER);
    return $query->getResult();
}
```

Le `queryBuilder` ne change pas, mais nous récupérons la requête pour lui préciser son `fetchMode`, c'est-à-dire son mode de récupération des données. Puis on lui spécifie `FETCH_EAGER` pour lui dire de tout charger directement, en opposition à `FETCH_LAZY` qui est présent par défaut.

Les deux méthodes sont correctes, mais il faut savoir que la première est plus optimisée.

Faites attention à la synchronisation du cache

Mettre en place un système de cache n'est jamais anodin, car on retourne des données sauvegardées au lieu des données réelles.

Par exemple, si ici nous supprimons tous les éléments de la base de données, le cache retournera quand même les livres qu'il a en mémoire, ce qui est un problème.

Donc mettre en place un cache signifie également se poser la question du **moment où il faut le rafraîchir**. Cela peut être après un certain temps, ou alors lorsqu'une opération a eu lieu.

Par exemple, si on utilise la route `delete`, comme les données ont changé, supprimer le cache pour forcer une relecture complète des données réelles peut être une bonne idée.

```
<?php
// src\Controller\BookController.php
// ...

#[Route('/api/books/{id}', name: 'deleteBook', methods:
['DELETE'])]
#[IsGranted('ROLE_ADMIN', message: 'Vous n\'avez pas les droits
suffisants pour supprimer un livre')]
public function deleteBook(Book $book, EntityManagerInterface
$em, TagAwareCacheInterface $cachePool): JsonResponse
{
    $cachePool->invalidateTags(["booksCache"]);
    $em->remove($book);
    $em->flush();
    return new JsonResponse(null, Response::HTTP_NO_CONTENT);
}
```

Ici, dans la méthode `delete`, nous avons à nouveau utilisé notre cache. Cette fois-ci, ce n'est pas pour récupérer des données, mais pour **invalid**er le tag `bookCache`.

De ce fait, tous les éléments que nous aurons créés en utilisant ce `bookCache`, à savoir toutes les pages de livres, seront supprimés, et au prochain appel un rechargement sera créé.

Nous aurions également pu régler manuellement le temps d'expiration du cache grâce à la méthode "expiresAfter". Au moment de créer le cache, dans la méthode `getAllBooks`, nous avons spécifié le tag, nous aurions pu ajouter :

```
$item->expiresAfter(60);
```

Ici, je précise que le cache va durer 60 secondes.

Exercice

Mettez en place une pagination et le système de cache sur les auteurs. Surtout, n'oubliez pas de mettre en place des méthodes pour vider le cache, afin de garantir que vos données seront toujours cohérentes.

XV Rendre l'API autodécouvrable

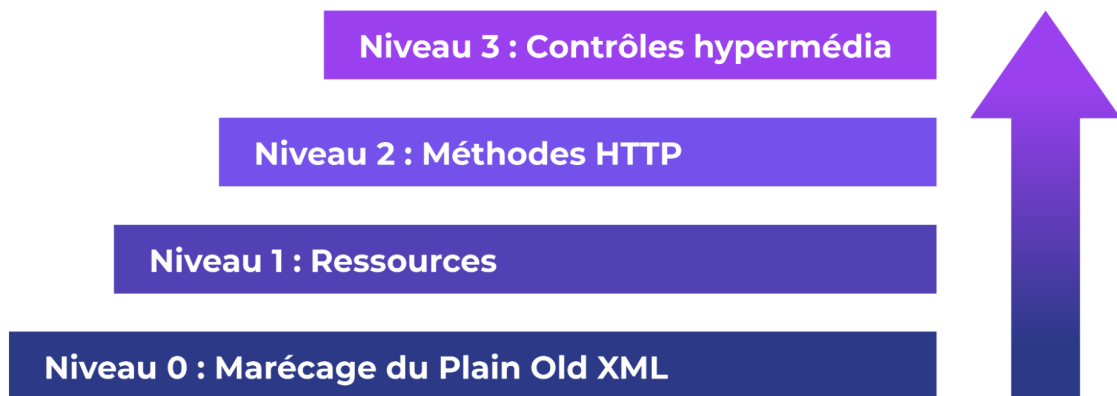
Pour avoir une API totalement REST, il faut respecter tous les niveaux du modèle de maturité de Richardson et, en particulier, il faut que notre API soit **autodécouvrable**.

En d'autres termes, lorsque nous accédons à une ressource, nous devons **avoir directement les URL pour explorer cette ressource**.

Un exemple typique : lorsque nous récupérons la liste des livres à disposition, nous devons, pour respecter cette contrainte, également récupérer les URL de chacun de ces livres, ainsi que, le cas échéant, les URL pour les mettre à jour et les modifier.

C'est ce qu'on appelle "The Glory of REST" (en français, la gloire du REST).

Le modèle de maturité de Richardson



Source : OpenClassroom

Toutes les API ne respectent pas ce dernier niveau : avoir une documentation externe complète et à jour est souvent largement suffisant pour la plupart des API. Ce dernier niveau n'est mis en œuvre qu'en cas de besoins spécifiques.

C'est d'autant plus vrai dans un monde où la **sobriété numérique** va gagner en importance, et où envoyer systématiquement des informations d'autodécouvrabilité risque de générer un trafic dont on pourrait se passer facilement.

Nous pourrions mettre ce dernier niveau en œuvre à la main, en récupérant nous-mêmes les routes que nous voulons indiquer, et en les sérialisant en même temps que les données.

Cependant, c'est assez fastidieux à faire, et nous risquerions de ne pas respecter les standards... De plus, les projets nécessitant le niveau 3 de Richardson sont souvent des projets relativement complexes, et il est bien plus pratique de **déléguer** cette tâche à un outil externe.

Cet outil va être dans notre cas : [HATEOAS](#).

Cet outil a cependant besoin d'un serializer un peu plus abouti que celui de Symfony pour fonctionner, le [JMSSerializer](#).

Heureusement, tout est fourni dans un seul et même bundle, pour l'installer il suffit de suivre la commande suivante :

```
composer require willdurand/hateoas-bundle
```

Composer va vous demander si vous voulez exécuter des recettes, celles-ci permettent de créer notamment les fichiers de configuration nécessaires à ces librairies, dites oui.

Maintenant que le bundle est installé, et avant de regarder le fonctionnement de HATEOAS, il va falloir utiliser le nouveau serializer.

Un nouveau fichier, `jms_serializer.yaml`, devrait avoir été créé. Nous allons commencer par le mettre à jour :

```
# config/packages/jms_serializer.yaml

jms_serializer:
  visitors:
    xml_serialization:
      format_output: '%kernel.debug%'
  property_naming:
    id: jms_serializer.identical_property_naming_strategy
```

Le début devrait normalement être présent, mais nous allons ajouter la propriété `property_naming`

Celle-ci nous permet de continuer à utiliser le camel case dans nos appels (`coverText` et pas `cover_text`), comme nous l'a proposé Symfony par défaut jusqu'à présent.

Il faut désormais mettre à jour les controllers.

Commençons par le fichier `BookController.php` .

Il va falloir remplacer le `use` concernant le serializer de Symfony par un `use` qui concerne JMSSerializer.

```
<?php
// src\Controller\BookController.php
// ...

// Supprimez cette ligne :
use Symfony\Component\Serializer\SerializerInterface;

// Ajoutez celle ci :
use JMS\Serializer\Serializer;
```

Mon éditeur de texte me souligne en rouge certains éléments :

```
#[Route('/api/books/{id}', name: 'detailBook', methods: ['GET'])]
public function getDetailBook(Book $book, SerializerInterface $serializer): JsonResponse {
    $jsonBook = $serializer->serialize($book, 'json', ['groups' => 'getBooks']);
    return new JsonResponse($jsonBook, Response::HTTP_OK, [], true);
}
```

C'est parce que même si on utilise toujours un `$serializer` qui vient de `SerializerInterface` , il ne s'agit plus du même serializer.

On est passé à **JMSSerializer**, et donc son fonctionnement diffère légèrement. Notamment en ce qui concerne le contexte de sérialisation, c'est-à-dire les informations qui permettent de savoir comment sérialiser (ici, la gestion des groupes, par exemple).

JMSSerializer étant construit de la même manière, nous aurons peu de choses à changer, mais parmi elles, il faut mettre à jour la syntaxe pour utiliser les "groups".

Donc le code change, voici le nouveau :

```
<?php
// src\Controller\BookController.php
// ...

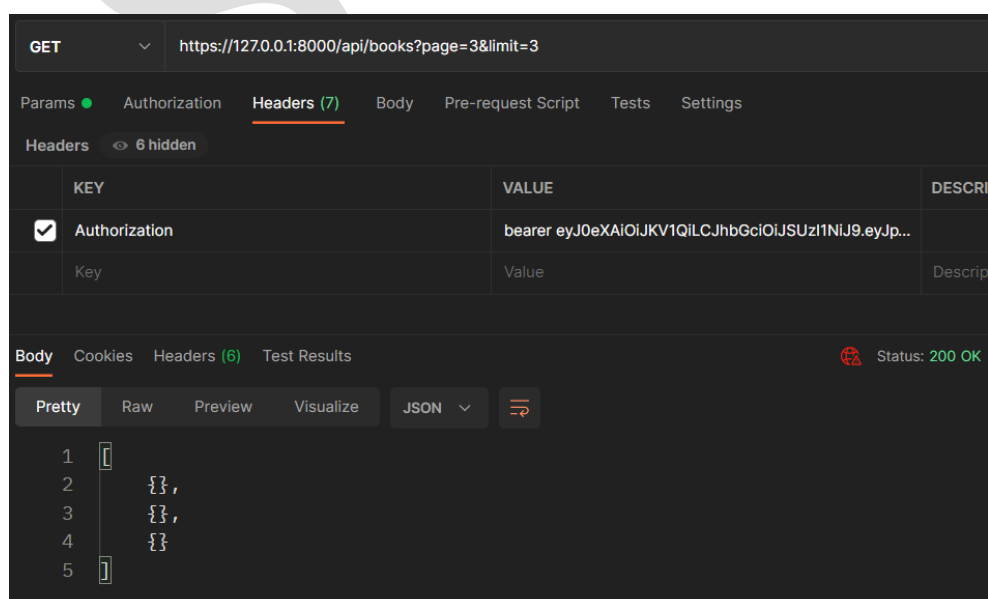
use JMS\Serializer\SerializationContext;
use JMS\Serializer\SerializerInterface;

// ...

#[Route('/api/books/{id}', name: 'detailBook', methods:
['GET'])]
public function getDetailBook(Book $book, SerializerInterface
$serializer): JsonResponse
{
    $context =
SerializationContext::create()->setGroups(['getBooks']);
    $jsonBook = $serializer->serialize($book, 'json',
$context);
    return new JsonResponse($jsonBook, Response::HTTP_OK, [],
true);
}
```

Pour faire simple, JMS crée un objet **SerializationContext** et lui passe le groupe : c'est le même principe qu'avant, mais avec une écriture différente.

Si on teste sur Postman, nous obtiendrons ceci :



Nous avons bien nos trois éléments comme demandé, mais ils sont vides. Cela ressemble très fort à ce qui se passerait si on indiquait un mauvais nom de groupe.

C'est presque le cas ! En fait, vu que nous sommes passés sur JMSSerializer, nous devons maintenant également modifier les `use` qui sont dans les entités.

```
<?php
// src\Entity\Book.php et src\Entity\Author.php
// ...

// Annotation à supprimer
use Symfony\Component\Serializer\Annotation\Groups;

// Annotation à rajouter :
use JMS\Serializer\Annotation\Groups;
```

On relance, et maintenant nous avons le bon résultat :

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

- Method:** GET
- URL:** https://127.0.0.1:8000/api/books?page=3&limit=3
- Headers:** 7 headers are listed, including an Authorization header with a bearer token.
- Body:** The response is a JSON array. The first element is a book object with the following data:

```
{
  "id": 88,
  "title": "Titre 7",
  "cover_text": "Quatrième de couverture numéro : 7",
  "author": {
    "id": 36,
    "last_name": "Nom 4",
    "first_name": "Prénom 4"
  }
},
{
  "id": 89,
  "title": "Titre 8",
  "cover_text": "Quatrième de couverture numéro : 8",
  "author": {
    "id": 37,
    "last_name": "Nom 5",
    "first_name": "Prénom 5"
  }
}
```
- Status:** 200 OK

Si après modification vous voyez encore l'ancienne version, peut-être est-ce dû au cache qui n'a pas été vidé. Cette erreur est très commune.

```
php bin/console cache:clear
```

Maintenant il faut gérer le PUT, souvenez-vous, dans le cas de l'update, nous récupérons l'entité visée, et nous nous étions arrangés pour désérialiser directement à l'intérieur.

Par exemple, quand nous éditons le livre avec l'id 42, Symfony nous fournissait l'entité correspondant à l'id 42 et désérialisait en utilisant cette entité pour base.

C'est faisable également avec `jms_serializer`, mais c'est moins pratique à mettre en place, comme vous pouvez le voir dans [la documentation de JMSSerializer](#) (en anglais). Nous allons voir une autre technique, plus simple mais très classique.

Elle consiste tout simplement à :

- Récupérer l'entité qui correspond à son id. (grâce au ParamConverter, nous l'obtenons directement dans `$currentBook`) ;
- Désérialiser cette entité dans une nouvelle entité vierge (dans `$newbook`) ;
- Recopier les éléments de l'une dans l'autre (`$currentBook->set...`).

```
<?php
// src\Controller\BookController.php
// ...

#[Route('/api/books/{id}', name:"updateBook", methods:['PUT'])]
#[IsGranted('ROLE_ADMIN', message: 'Vous n\'avez pas les droits
suffisants pour éditer un livre')]
public function updateBook(Request $request,
SerializerInterface $serializer, Book $currentBook,
EntityManagerInterface $em, AuthorRepository $authorRepository,
ValidatorInterface $validator, TagAwareCacheInterface $cache):
JsonResponse
{
    $newBook = $serializer->deserialize($request->getContent(),
Book::class, 'json');
    $currentBook->setTitle($newBook->getTitle());
    $currentBook->setCoverText($newBook->getCoverText());
```

```

        // On vérifie les erreurs
        $errors = $validator->validate($currentBook);
        if ($errors->count() > 0) {
            return new JsonResponse($serializer->serialize($errors,
'json'), JsonResponse::HTTP_BAD_REQUEST, [], true);
        }

        $content = $request->toArray();
        $idAuthor = $content['idAuthor'] ?? -1;

$currentBook->setAuthor($authorRepository->find($idAuthor));

        $em->persist($currentBook);
        $em->flush();

        // On vide le cache.
        $cache->invalidateTags(["booksCache"]);

        return new JsonResponse(null,
JsonResponse::HTTP_NO_CONTENT);
    }

```

JMS_Serializer est en place.

Maintenant il faut mettre en place HATEOAS, jusqu'ici, nous avons certes remplacé le serializer de Symfony par JMSSerializer, mais sur notre projet, cela ne change encore rien. Nous l'avons fait uniquement parce que HATEOAS réclame ce nouveau serializer pour fonctionner.

Le niveau 3 du modèle de Richardson explique que lorsqu'on récupère une entité, on doit en même temps récupérer des informations sur **les autres routes possibles** pour cette entité.

Voyons comment mettre cela en place avec HATEOAS, sur notre entité Book :

```
<?php
// src\Entity\Book.php
// ...

use Hateoas\Configuration\Annotation as Hateoas;

// ...

/**
 * @Hateoas\Relation(
 *     "self",
 *     href = @Hateoas\Route(
 *         "detailBook",
 *         parameters = { "id" = "expr(object.getId())" }
 *     ),
 *     exclusion = @Hateoas\Exclusion(groups="getBooks")
 * )
 *
 */
#[ORM\Entity(repositoryClass: BookRepository::class)]
```

Ici nous avons des annotations "à l'ancienne", avec des @ directement dans les commentaires. J'ai préféré garder ce style d'annotation ici car ce sont encore celles qui sont dans la [documentation officielle](#) de HATEOAS.

Les annotations de HATEOAS définissent une **relation**, c'est-à-dire un nouveau lien qui va apparaître. Le premier lien est **self** , il s'agit du lien vers la ressource que l'on est en train de regarder.

On ne le répète jamais assez, comme toujours, faites attention aux **use** :

```
use Hateoas\Configuration\Annotation as Hateoas;
```

Le paramètre **href** va contenir la **route** concernée. Ici, nous avons **detailBook** , car c'est la route pour avoir le détail d'un livre. Nous passons le **paramètre id** à cette route.

Nous voyons également ici un paramètre : **exclusion** . Celui-ci permet de déterminer **dans quel cas les informations vont apparaître ou pas**. Ici, nous précisons simplement le groupe.

Faisons pareil pour le **delete** et le **update** , et ajoutons ces éléments sous le premier:

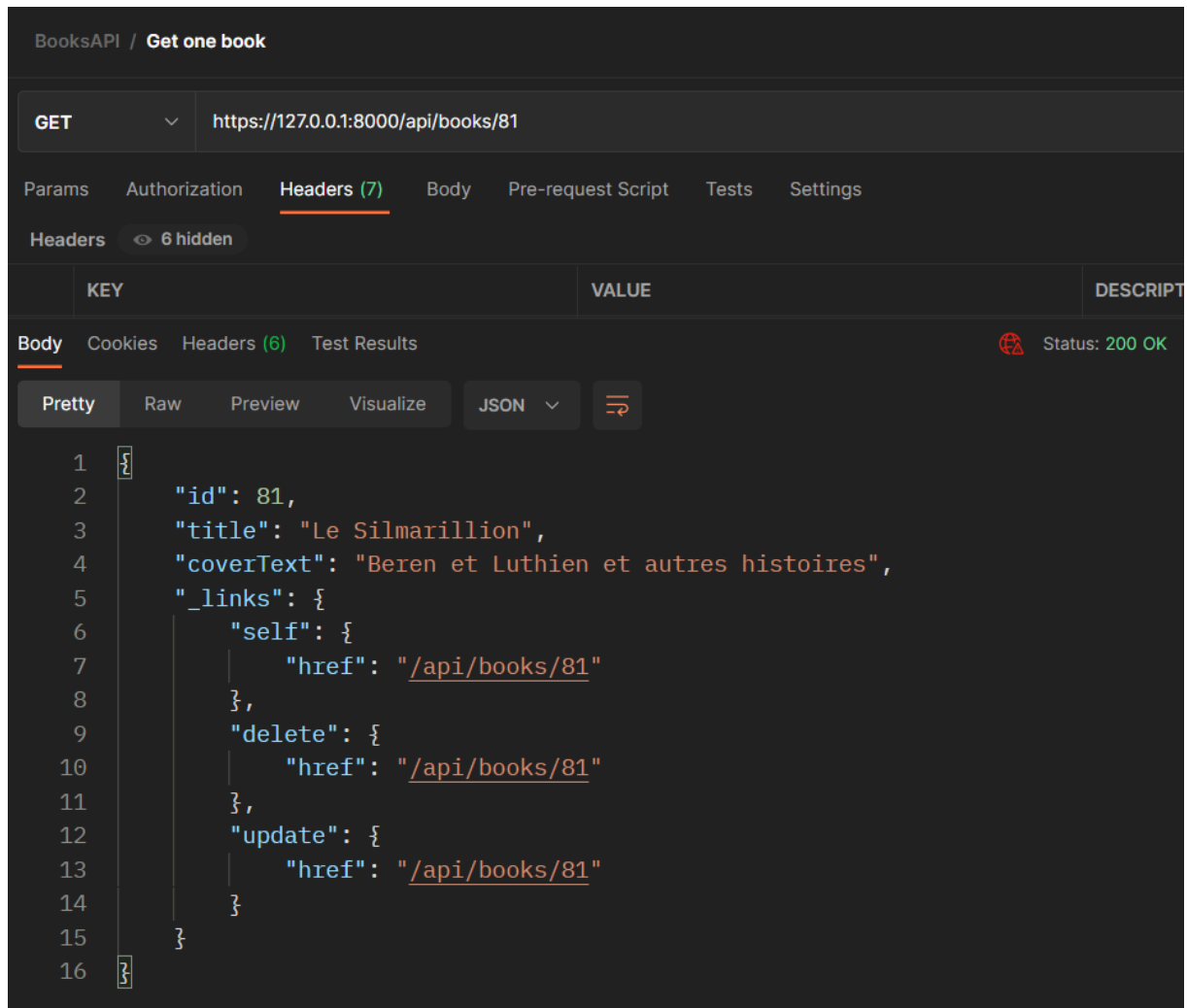
```
<?php
    // src\Entity\Book.php
    // ...

/*
 * @Hateoas\Relation(
 *     "delete",
 *     href = @Hateoas\Route(
 *         "deleteBook",
 *         parameters = { "id" = "expr(object.getId())" },
 *     ),
 *     exclusion = @Hateoas\Exclusion(groups="getBooks", excludeIf
= "expr(not is_granted('ROLE_ADMIN'))"),
 * )
 *
 * @Hateoas\Relation(
 *     "update",
 *     href = @Hateoas\Route(
 *         "updateBook",
 *         parameters = { "id" = "expr(object.getId())" },
 *     ),
 *     exclusion = @Hateoas\Exclusion(groups="getBooks", excludeIf
= "expr(not is_granted('ROLE_ADMIN'))"),
 * )
 *
 */
#[ORM\Entity(repositoryClass: BookRepository::class)]
#[ApiResponse()]
class Book

(...)
```

Notez le paramètre “exclusion” qui évolue, nous précisons maintenant que le lien sera exclu de la liste si l'utilisateur ne possède pas le `ROLE_ADMIN` . En effet, seul l'administrateur peut effectuer un `delete` ou un `update` .

Regardons à nouveau le résultat dans Postman :



Vous pouvez constater l'apparition d'un nouvel élément, `_links`, qui contient directement les liens vers les ressources `self` , `delete` et `update` .

Regardons maintenant si l'on récupère tous les livres :

```

{
  "id": 81,
  "title": "Le Silmarillion",
  "coverText": "Beren et Luthien et autres histoires",
  "_links": {
    "self": {
      "href": "/api/books/81"
    },
    "delete": {
      "href": "/api/books/81"
    },
    "update": {
      "href": "/api/books/81"
    }
  }
},
{
  "id": 82,
  "title": "Titre 1",
  "coverText": "Quatrième de couverture numéro : 1",
  "author": {
    "id": 36,
    "lastName": "Nom 4",
    "firstName": "Prénom 4"
  },
  "_links": {
    "self": {
      "href": "/api/books/82"
    }
  }
}

```

Les liens sont bien mis à jour et en plus, le tout respecte le standard HAL qui est un des plus utilisés pour mettre en œuvre le niveau 3 du modèle de Richardson.

Vous pouvez consulter [la documentation HAL](#) (en anglais) pour en savoir plus sur le standard.

Exercice

Nous avons mis à jour notre projet pour utiliser JMS et HATEOAS pour les livres, mais il reste les auteurs à gérer de la même manière. N'hésitez pas à essayer d'autres options grâce à la [documentation officielle](#) pour avoir une idée des possibilités offertes par cette librairie.

XVI Le versionning

Notre API est désormais complète : elle gère le CRUD, la sécurité, la mise en cache et respecte même les 3 niveaux du modèle de Richardson.

Si nous voulons faire vivre cette API, la faire évoluer il faut utiliser la gestion du versionning.

Notre API est basique. Les livres ne possèdent que peu d'informations : un titre, la quatrième de couverture et un auteur.

Imaginons que nous voulions ajouter une information, par exemple un commentaire du bibliothécaire. Cela va faire évoluer notre API et avec la même route, nous allons avoir des informations différentes.

Mais rien ne dit que ceux qui appellent notre API aient prévu ce qu'il faut pour gérer ces nouvelles informations. L'idée est donc de fournir un moyen à ces clients de **garder l'ancienne API**, alors que la nouvelle est déjà fonctionnelle, pour leur laisser le temps de se mettre à jour.

Toutes les API ne gèrent pas le versioning. Gérer trop de versions peut mener à un code et une base de données **difficiles à maintenir**. Comme pour le cache ou l'autodécouvrabilité, c'est un choix qui doit être fait en conscience pour répondre à un besoin réel de l'application.

Nous allons rajouter un champ `comment` dans la table `book`. Rien de neuf ici, nous avons déjà fait des choses similaires dans les chapitres précédents.

```
php bin/console make:entity Book
```

Symfony reconnaît que l'entité existe déjà, et on se contente de rajouter notre nouveau champ.

Pensez à ajouter l'annotation **Groups** sur ce champ dans le fichier **Book.php** :

```
<?php
// src\Entity\Book.php
// ...

#[ORM\Column(type: 'text', nullable: true)]
#[Groups(["getBooks"])] // Groupe à ajouter
private $comment;
```

N'oubliez pas de faire votre migration.

On met à jour nos fixtures dans AppFixture.php :

```
<?php
// src\DataFixtures\AppFixtures.php
// ...

for ($i = 0; $i < 20; $i++) {
    $book = new Book();
    $book->setTitle("Titre " . $i);
    $book->setCoverText("Quatrième de couverture numéro : " .
    $i);
    $book->setComment("Commentaire du bibliothécaire " . $i);
    $book->setAuthor($listAuthor[array_rand($listAuthor)]);
    $manager->persist($book);
}
```

Ici je me suis contenté de rajouter la ligne avec le **setComment** dans la boucle de création du livre.

Et finalement, nous rejouons nos fixtures :

```
php bin/console doctrine:fixtures:load
```

À ce stade, si tout se passe bien, votre API est déjà mise à jour et vous devriez voir votre commentaire dans Postman :

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

- Method:** GET
- URL:** https://127.0.0.1:8000/api/books/110
- Headers:** 7 headers, with 'Authorization' set to 'bearer eyJ0eXAI0iJKV1QiLCJhbG...'
- Body:** Pretty-printed JSON response:

```
1 {
2   "id": 110,
3   "title": "Titre 0",
4   "coverText": "Quatrième de couverture numéro : 0",
5   "author": {
6     "id": 52,
7     "lastName": "Nom 9",
8     "firstName": "Prénom 9"
9   },
10  "comment": "Commentaire du bibliothécaire 0",
11  "_links": {
12    "self": {
13      "href": "/api/books/110"
14    },
15    "delete": {
```
- Status:** 200 OK, 283 ms, 683 B

Maintenant, l'idée est de dire que ce champ correspond à une version 2 de notre API. Il se trouve que nous avons installé JMSSerializer, et que JMS gère nativement le versioning.

Symfony a prévu d'intégrer cette fonctionnalité avec la même syntaxe, directement dans le serializer natif de Symfony, mais au moment où ce cours est écrit, cette option n'est pas encore déployée.

Nous allons simplement ajouter dans notre entité une annotation précisant la version à partir de laquelle le champ est présent :

```
<?php
    // src\Entity\Book.php
    // ...
use JMS\Serializer\Annotation\Since;

    // ...

#[ORM\Column(type: 'text', nullable: true)]
#[Groups(["getBooks"])]
#[Since("2.0")]
private $comment;
```

`#[Since("2.0")]` , c'est-à-dire que ce champ n'existe qu'à partir de la version 2.

Et dans le contrôleur, lorsque nous récupérons nos données, nous n'avons plus qu'à ajouter une ligne au contexte pour lui dire quelle est la version de l'API que l'on veut traiter :

```
<?php
    // src\Controller\BookController.php
    // ...

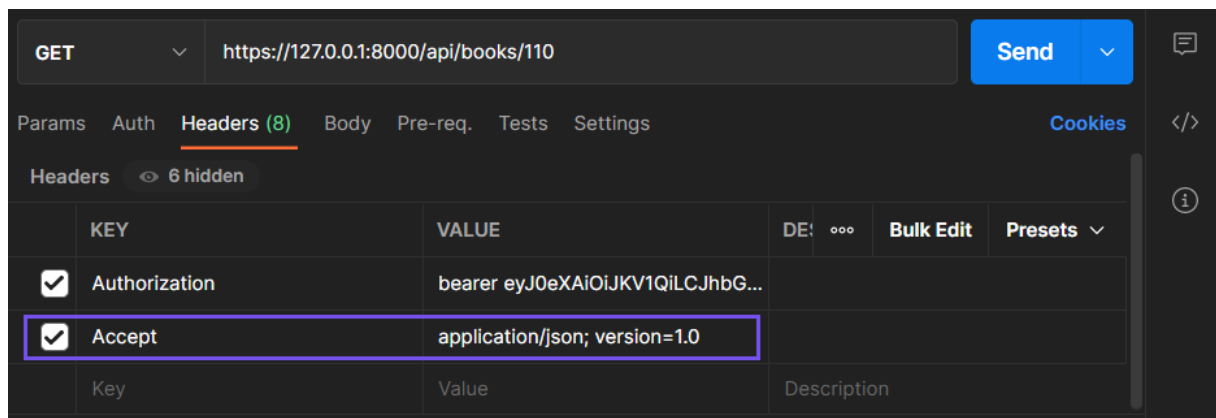
#[Route('/api/books/{id}', name: 'detailBook', methods:
['GET'])]
public function getDetailBook(Book $book, SerializerInterface
$serializer): JsonResponse
{
    $context =
SerializationContext::create()->setGroups(["getBooks"]);
    $context->setVersion("1.0");
    $jsonBook = $serializer->serialize($book, 'json',
$context);
    return new JsonResponse($jsonBook, Response::HTTP_OK, [],
true);
}
```

Ici, j'ai écrit 1.0, donc mon nouveau champ ne va pas être visible. En revanche, si je remplace ce numéro par 2.0, alors on verra mon nouveau champ.

Oui mais... Ici, j'ai écrit directement le numéro de version que je veux, comment faire pour le choisir depuis l'API ?

Il existe plusieurs techniques pour **envoyer** ce numéro de version, mais l'idée est toujours la même. Il faut que le client (ici, Postman) envoie l'information "Je veux le numéro de version xxx", et que le contrôleur écoute cette information.

Une des méthodes les plus propres consiste à **passer cette information dans le header**, et plus précisément dans le champ **Accept** .



Ici, j'ai écrit `application/json; version=1.0` .

Je précise donc que je veux récupérer du JSON, et que je veux l'API en version 1.0.

Maintenant il nous reste à récupérer cette information.

Nous pourrions lire ce header directement dans le contrôleur, mais cela nécessite un peu de traitement, et sera utile pour plusieurs de nos routes. Du coup, nous allons plutôt créer un **service** qui va nous permettre de **récupérer directement l'information, et qui pourra être appelé par tous les contrôleurs qui en ont besoin**.

Si vous ne savez pas ce qu'est un service, je vous renvoie sur la [documentation officielle de Symfony](#) (en anglais).

Cette fois-ci, pas de make, nous devons le créer à la main. Dans le répertoire `src`, créez un répertoire `Service` et un nouveau fichier `VersioningService.php` .


```

<?php
// src\Service\VersioningService.php

// Création d'un service Symfony pour pouvoir récupérer la version
contenue dans le champ "accept" de la requête HTTP.
namespace App\Service;

use
Symfony\Component\DependencyInjection\ParameterBag\ParameterBagInterface;
use Symfony\Component\HttpFoundation\RequestStack;

class VersioningService
{
    private $requestStack;

    /**
     * Constructeur permettant de récupérer la requête courante
     (pour extraire le champ "accept" du header)
     * ainsi que le ParameterBagInterface pour récupérer la version
     par défaut dans le fichier de configuration
     *
     * @param RequestStack $requestStack
     * @param ParameterBagInterface $params
     */
    public function __construct(RequestStack $requestStack,
ParameterBagInterface $params)
    {
        $this->requestStack = $requestStack;
        $this->defaultVersion =
$params->get('default_api_version');
    }

    /**
     * Récupération de la version qui a été envoyée dans le header
     "accept" de la requête HTTP
     *
     * @return string : le numéro de la version. Par défaut, la
     version retournée est celle définie dans le fichier de
     configuration services.yaml : "default_api_version"
     */
    public function getVersion(): string

```

```

{
    $version = $this->defaultVersion;

    $request = $this->requestStack->getCurrentRequest();
    $accept = $request->headers->get('Accept');
    // Récupération du numéro de version dans la chaîne de
    caractères du accept :
    // exemple "application/json; test=bidule; version=2.0" =>
    2.0

    $entete = explode(';', $accept);

    // On parcourt toutes les entêtes pour trouver la version
    foreach ($entete as $value) {
        if (strpos($value, 'version') !== false) {
            $version = explode('=', $value);
            $version = $version[1];
            break;
        }
    }
    return $version;
}
}

```

En décortiquant un peu ce code, nous pouvons voir plusieurs éléments.

D'abord le **constructeur**, celui-ci prend deux paramètres :

- La **RequestStack**, qui va nous permettre de récupérer le header de la requête envoyée ;
- Le **parameterBag**, qui va nous permettre de récupérer le numéro de version par défaut de notre API. Ainsi ce numéro pourra être configuré facilement depuis le fichier **services.yaml**, sans modifier le code.

Ensuite, nous avons la **méthode getVersion** proprement dite, qui commence par lire l'information **Accept** dans le header, la découpe suivant le caractère ;, et récupère la version.

Si la version n'est pas trouvée, c'est la version par défaut qui est retournée.

À ce stade, il faut encore modifier le fichier `services.yaml` pour ajouter le numéro de version par défaut :

```
parameters:
    default_api_version: "2.0"
```

Et il faut faire appel à notre service depuis le contrôleur :

```
<?php
// src\Repository\BookRepository.php
// ...

use App\Service\VersioningService;

// ...

#[Route('/api/books/{id}', name: 'detailBook', methods:
['GET'])]
public function getDetailBook(Book $book, SerializerInterface
$serializer, VersioningService $versioningService): JsonResponse
{
    $version = $versioningService->getVersion();
    $context =
SerializationContext::create()->setGroups(["getBooks"]);
    $context->setVersion($version);
    $jsonBook = $serializer->serialize($book, 'json',
$context);
    return new JsonResponse($jsonBook, Response::HTTP_OK, [],
true);
}
```

Ici, nous avons récupéré directement le service dans les paramètres de la méthode, et nous n'avons plus qu'à nous en servir pour récupérer la version.

Nous voyons ici tout l'intérêt d'utiliser un service, cette méthode `getVersion` pourra être réutilisée pour chaque méthode le nécessitant.

Quelques tests pour vérifier, n'hésitez pas à spécifier le numéro de version dans le `Accept`, l'enlever, et le passer dans le `service.yaml` à 1.0 puis 2.0 pour bien voir toutes les possibilités.

Pour ce qui est des traitements plus complexes (ajout d'une entité `Editor` par exemple, qui aurait un impact majeur sur la création et la mise à jour des livres), comme nous avons à disposition un moyen simple de récupérer le numéro de version, c'est très facile. Nous pourrions différencier les comportements avec un simple `if` dans les méthodes du contrôleur.

Ceci étant, comme je l'ai précédemment mentionné, il faut cependant être prudent et le faire en conscience pour ne pas rendre le code trop compliqué à maintenir.

XVII Documenter son code avec Nelmio

Une API, aussi bien pensée soit-elle, n'est rien sans une bonne documentation. Une option pourrait être de documenter votre API à la main. C'est tout à fait faisable, et suffisant dans de nombreux cas.

Ceci étant, la documentation étant un problème qui se pose pratiquement pour toutes les API, vous vous en doutez, un bundle existe précisément pour répondre à cette demande : [Nelmio](#).

L'idée de ce bundle va être de **générer un site web** qui va **reprendre l'intégralité de nos routes** pour les documenter, mais qui va même nous fournir un outil pour **tester** ces routes directement.

Nous installons donc Nelmio avec la commande suivante :

```
composer require nelmio/api-doc-bundle
```

Pensez à accepter la recette proposée pour que les fichiers de configuration soient automatiquement créés.

Une fois **Nelmio** installé, le fichier `nelmio_api_doc.yaml` devrait avoir été créé dans le dossier `routes` de la configuration.

```
# config/routes/nelmio_api_doc.yaml

# Expose your documentation as JSON swagger compliant
app.swagger:
  path: /api/doc.json
  methods: GET
  defaults: { _controller: nelmio_api_doc.controller.swagger }

## Requires the Asset component and the Twig bundle
## $ composer require twig asset
app.swagger_ui:
  path: /api/doc
  methods: GET
  defaults: { _controller: nelmio_api_doc.controller.swagger_ui }
```

Nous allons juste décommenter les dernières lignes pour activer la route `/api/doc`

Notez le commentaire juste au-dessus qui nous précise que pour fonctionner, cette route a besoin d'installer **Twig**, le moteur de template.

Appliquons la commande d'installation :

```
composer require twig asset
```


À ce stade, la documentation existe déjà sur la route `/api/doc`, il nous faut juste **rendre cette route accessible**. En effet, dans le fichier `security.yaml`, nous avons dit qu'il fallait un token pour toutes les routes, sauf celle qui permet d'obtenir le token.

Nous allons donc ajouter une seconde exception pour la route `^/api/doc`, qui doit avoir un accès public :

```
# config/packages/security.yaml

(...)
access_control:
    - { path: ^/api/login, roles: PUBLIC_ACCESS }
    - { path: ^/api/doc, roles: PUBLIC_ACCESS }
    - { path: ^/api, roles: IS_AUTHENTICATED_FULLY }
(...)
```

Celle-ci sera accessible normalement à l'URL
:https://127.0.0.1:8000/api/doc.

 NelmioApiDocBundle

My App

1.0.0 OAS3

This is an awesome app!

default			^
GET	/api/doc.json		✓
GET	/api/authors	Cette méthode permet de récupérer l'ensemble des auteurs.	✓
POST	/api/authors	Cette méthode permet de créer un nouvel auteur. Elle ne permet pas d'associer directement des livres à cet auteur.	✓
GET	/api/authors/{id}	Cette méthode permet de récupérer un auteur en particulier en fonction de son id.	✓
PUT	/api/authors/{id}	Cette méthode permet de mettre à jour un auteur.	✓
DELETE	/api/authors/{id}	Cette méthode supprime un auteur en fonction de son id.	✓
GET	/api/books	Cette méthode permet de récupérer l'ensemble des livres.	✓
POST	/api/books	Cette méthode permet d'insérer un nouveau livre.	✓
		Cette méthode permet de récupérer un livre en particulier en fonction	

Nous y voyons les diverses **routes disponibles**, et quelle **méthode HTTP** doit être utilisée.

La première route `/api/doc.json` est une route permettant d'obtenir la même documentation, mais en JSON.

Nelmio nous propose non seulement une documentation, mais ce site est également un outil pour **tester** nos routes comme nous le faisons avec Postman.

En cliquant, par exemple, sur Author pour avoir la liste des auteurs, on accède à une interface qui permet d'**exécuter la requête**. Mais si on le fait, on se heurte à un petit souci : **Erreur 401** : Pas de token JWT.

The screenshot shows the Nelmio API documentation interface for the `GET /api/books` endpoint. The description states: "Cette méthode permet de récupérer l'ensemble des livres." The interface includes a "Parameters" section with "No parameters" and a "Cancel" button. Below this are "Execute" and "Clear" buttons. The "Responses" section displays the following information:

- Curl**:

```
curl -X 'GET' \
  'https://127.0.0.1:8000/api/books' \
  -H 'accept: */*'
```
- Request URL**:

```
https://127.0.0.1:8000/api/books
```
- Server response**:

Code	Details
401	Error: Unauthorized
- Response body**:

```
{
  "code": 401,
  "message": "JWT Token not found"
}
```

At the bottom right of the response body, there are "Download" and "Copy" buttons.

C'est parfaitement logique, nous n'avons pas récupéré de token. Plus encore, l'authentification étant gérée par Symfony et LexikJWTAuthenticator, nous n'avons même pas la route ici présente.

Nous allons donc expliquer à Nelmio qu'une route existe, et qu'il faut la présenter. Dans le dossier `config/packages` se trouve un fichier `nelmio_api_doc.yaml` qui sert précisément à cela :

```
# config/packages/nelmio_api_doc.yaml

nelmio_api_doc:
  documentation:
    info:
      title: Books
      description: Une API d'OpenClassrooms avec des
livres, des autrices et des auteurs !
      version: 2.0.0
  paths:
    /api/login_check:
      post:
        operationId: postCredentialsItem
        tags:
          - Token
        summary: Permet d'obtenir le token JWT pour se logger.
        requestBody:
          description: Crée un nouveau token JWT
          content:
            application/json:
              schema:
                $ref:
'#/components/schemas/Credentials'
        responses:
          '200':
            description: Récupère le token JWT
            content:
              application/json:
                schema:
                  $ref: '#/components/schemas/Token'

  components:
    schemas:
      Token:
```



```

        type: object
        properties:
            token:
                type: string
                readOnly: true
    Credentials:
        type: object
        properties:
            username:
                type: string
                default: admin@bookapi.com
            password:
                type: string
                default: password
    securitySchemes:
        bearerAuth:
            type: apiKey
            in: header
            name: Authorization # or another header name
    security:
        - bearerAuth: []
    areas: # to filter documented areas
    path_patterns:
        - ^/api(?!/doc$) # Accepts routes under /api except
/api/doc

```

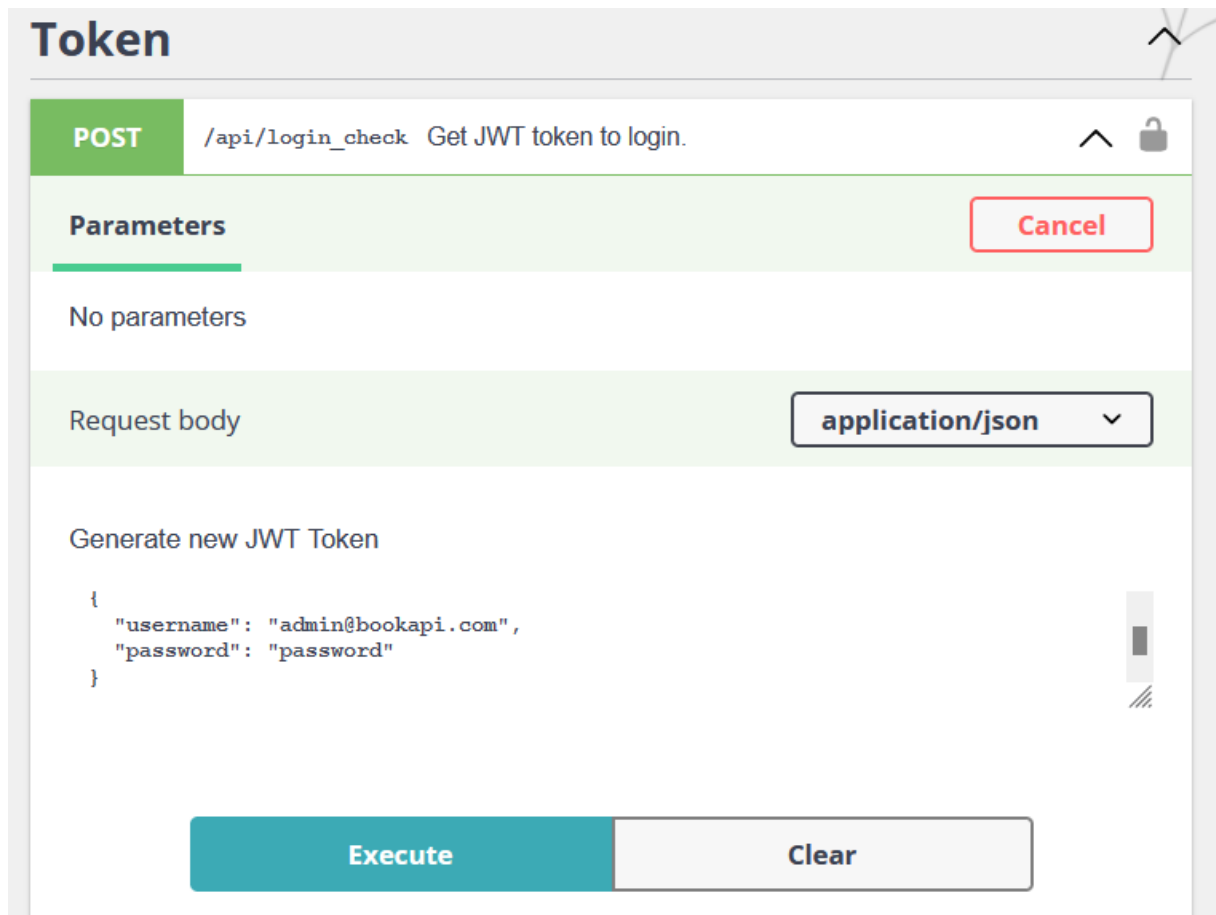
Ce fichier précise de nombreuses choses, en particulier des informations sur l'application et la route `login_check` elle-même. Si vous voulez en savoir plus, comme toujours la [documentation officielle sur la sécurité](#) .

Dans ce fichier nous pouvons donc voir :

- info : cette section contient des informations générales sur notre application ;
- path : précise la route pour réaliser la récupération de notre token et des informations de documentation (summary, requestBody, responses...) ;
- components : décrit ce qui a été appelé dans la partie path, en particulier :
 - token : dit que l'on veut un token, chaîne de caractères,
 - credential : précise les informations de connexion (en particulier username et password),
 - securitySchemes : explique que nous voulons un champ Authorization dans le header de la requête.

J'ai mis ici, dans les `default`, les vraies valeurs de mes fixtures pour me connecter à l'application, afin de faciliter les tests ; dans la documentation finale, il est évident que ces valeurs doivent être des exemples, et en aucun cas les vraies valeurs qui permettent de se connecter.

Et c'est tout ! Nous pouvons maintenant tester. La première chose va être de récupérer le token, mais une nouvelle route vient d'apparaître : `/api/login_check`



The screenshot shows a REST client interface with a title bar "Token". The main area displays a POST request to `/api/login_check` with the description "Get JWT token to login.". The "Parameters" tab is selected, showing "No parameters" and a "Cancel" button. The "Request body" tab is also visible, showing a JSON object:

```
{  "username": "admin@bookapi.com",  "password": "password"}
```

 with a dropdown menu set to "application/json". At the bottom, there are "Execute" and "Clear" buttons.

Et avec l'exécution, nous obtenons notre code :

Code

Details

200

Response body

```
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpYXQiOiJlNDg1NjM5NDUsImV4cCI6MTY0ODU2NzU0NSwicm9sZXMiOiJlUk9MRV9BRE1JTlIsIlJPTEVfVWVhbnV4Ij0iLCJ1c2VybmFtZSI6ImFkbWluQGJvb2thcGkuY29tIn0.US1Tlxsj-n4ouqCcZasJT10gq1QL1-KSTPPsvRQdrf1bZsPMHFq4JTkB06UBdMw5yErX1obySLgMFbNpxom436pc3EI7V36tzkIWVPqnVqJDpoNQWp-sUqTrfc9rOVcTc6MacAUyW8nVzRGxh_a-Mn1X1ZE5M5FvptuyxYuMhNN1LFi4rN4R-JItGmHENB1Ch2fIBVolju4-zdEBdlK0jircNqnigONOHaoicu0nhI7x6J_Kbi_cTBRvg0lt1EBmFDsnV3Nn-CZAZA5N2cong67_kuTGNgXFvDS2cii742RX1IB1mnJQw1joQYHKJH5Tp_jqcAfKHD_VwkDzy9nCatBNrCZTcgGjg90KaKJzvnH5TnZe64SCwKpu6NP7Y0Tqr9iib22W2Vz8Zmg7K5EocFmni5GFtifgnNZNMf1HT0iJRD-jtWhRehbhDDw2W_r97RJScKPt6mi5gxVZmdexMkJrNAk-oE-sTphpNU7yENUKvhsYCCz15NHuMXiiubLpemGLVA8zSDg6adj4bJk-AybKgh_s72hLoNMcupqtHYHWS0aZnA9x0oQ_LLOwszm04-F2rUAYhuh9CV03Q3_mRgB8aF20aJ7jZqN9_iVWjgdthrk0WkuOehfjs9P5Dy35_IJm8YSjBqD6A0C1x02-dQbGzBHXn62RVfbVBj2sTs"
}
```



Download

Response headers

```
cache-control: no-cache, private
content-length: 868
content-type: application/json
date: Tue, 29 Mar 2022 14:25:45 GMT
x-firefox-spdy: h2
x-powered-by: PHP/8.1.0
x-robots-tag: noindex
```

Il faut maintenant spécifier ce code en cliquant sur le champ Authorize, qui est apparu tout en haut du formulaire, sans oublier, comme avec Postman, de faire précéder ce code de **Bearer** :

Authorize

Available authorizations

bearerAuth (apiKey)

Name: Authorization

In: header

Value:

bearer eyJ0eXAiOiJKV1QiLC

Authorize

Close

Et voilà, nous sommes désormais identifiés, et nous pouvons tester nos routes.

Il est temps d'améliorer les informations sur les ressources dans la documentation.

Nous avons accès à nos routes, mais les **commentaires** sont un peu légers, les **réponses possibles** restent inconnues, et les **paramètres** difficiles à comprendre.

Là encore, Nelmio a tout prévu, et avec quelques annotations directement dans les contrôleurs, il est possible de tout paramétrer, ou presque !

Prenons par exemple la route qui retourne l'ensemble des livres. Pour l'instant, aucune information sur la pagination n'est fournie.

Améliorons cela dans le fichier `BookController.php`. D'abord, nous renseignons les `use` qui seront utiles pour les annotations.

```
<?php
// src\Controller\BookController.php
// ...

use Nelmio\ApiDocBundle\Annotation\Model;
use Nelmio\ApiDocBundle\Annotation\Security;
use OpenApi\Annotations as OA;
```

Et ensuite, nous pouvons annoter chaque méthode, ici `getAllBooks`.

```
<?php
// src\Controller\BookController.php
// ...

/**
 * Cette méthode permet de récupérer l'ensemble des livres.
 *
 * @OA\Response(
 *     response=200,
 *     description="Retourne la liste des livres",
 *     @OA\JsonContent(
 *         type="array",
 *         @OA\Items(ref=@Model(type=Book ::class,
groups={"getBooks"}))
 *     )
 * )
```

```

* @OA\Parameter(
*     name="page",
*     in="query",
*     description="La page que l'on veut récupérer",
*     @OA\Schema(type="int")
* )
*
* @OA\Parameter(
*     name="limit",
*     in="query",
*     description="Le nombre d'éléments que l'on veut
récupérer",
*     @OA\Schema(type="int")
* )
* @OA\Tag(name="Books")
*
* @param BookRepository $bookRepository
* @param SerializerInterface $serializer
* @param Request $request
* @return JsonResponse
*/
#[Route('/api/books', name: 'books', methods: ['GET'])]
public function getAllBooks(BookRepository $bookRepository,
SerializerInterface $serializer, Request $request,
TagAwareCacheInterface $cache): JsonResponse
{
    // ...

```

Ici nous pouvons voir que sont utilisées les “anciennes annotations”, en commentaire. Ce choix a été fait car c’est, au moment de l’écriture de ce cours, ce qui est encore recommandé dans la [documentation de Nelmio](#).

Ici sont définis les réponses (en l'occurrence, la réponse 200) et les paramètres disponibles (en l'occurrence, `limit` et `page`).

Regardons les éléments présents :

- `@OA\Response` : définit les réponses possibles (ici un code de retour 200) :
 - `@OA\JsonContent` : explique le type du contenu retourné (ici un tableau d’entité (`@model`) `Book`).
- `@OA\Parameter` : définit les paramètres acceptés par la méthode :
 - `@OA\Schema` : le type du paramètre, ici, tout simplement un chiffre.

- `@OA\Tag` : le groupe de la méthode, ce qui permet de classer les méthodes pour l'affichage par la suite.

Nous aurions également pu ajouter la réponse **401 Unauthorized**, qui est une des réponses possibles. Cela aurait été possible en ajoutant une autre annotation `@OA\Response`, comme nous l'avons fait pour le stat.

Regardons maintenant le résultat :

The screenshot shows the Swagger UI for the **Books** API. The top bar displays the **GET** method for the `/api/books` endpoint, with a description: "Cette méthode permet de récupérer l'ensemble des livres." Below this, the **Parameters** tab is active, showing two query parameters:

Name	Description
<code>page</code> <small>int (query)</small>	La page que l'on veut récupérer
<code>limit</code> <small>int (query)</small>	Le nombre d'éléments que l'on veut récupérer

Each parameter has an input field containing its name. At the bottom, there are **Execute** and **Clear** buttons.

Nos deux nouveaux paramètres sont désormais présents. Notez également que la route a été rangée dans **Books** grâce au tag qui a été précisé en commentaire.

En conclusion :

Vous avez maintenant les bases pour construire une API, à vous d'aller dans les documentations de chaque outils pour aller plus loin.

Bon Courage.

SOFIP