Les API

May the force be with you !

Sommaire

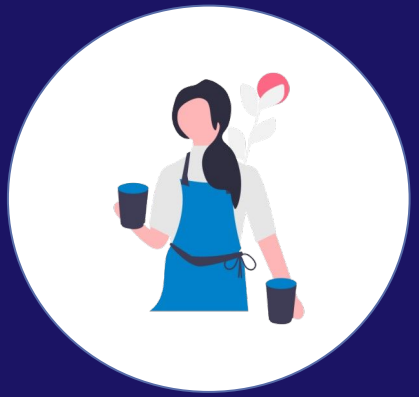
- Qu'est-ce qu'une API ?
  - SOAP
  - REST
  - GraphQL
- Comment ça fonctionne ?
- Verbes HTTP et leurs Utilisations
- Codes de Statut HTTP
- Le Json Web Token
  - A quoi ça sert ?
  - A quoi ça ressemble ?
- Postman / Insomnia

Qu'est-ce qu'une API ?

Définition :

Une API (Application Programming Interface, ou interface de programmation d'applications) est un ensemble de règles et de protocoles qui permet à différents logiciels de communiquer entre eux. Elle définit les méthodes et les données que les développeurs peuvent utiliser pour interagir avec un service, une application ou une bibliothèque.

Le client



La serveuse

La cuisine

Qu'est-ce qu'une API ?

API REST

Une API REST (Representational State Transfer) est une interface de programmation d'application (API) qui permet d'établir une communication entre plusieurs logiciels. Grâce à elle, des logiciels d'applications utilisant différents systèmes d'exploitation peuvent interagir et partager des informations par l'intermédiaire du protocole HTTP.

Principe : Basé sur des principes architecturaux (client-serveur) et utilise le protocole HTTP.

Verbes HTTP : GET, POST, PUT, DELETE, PATCH pour manipuler les ressources.

Format de données : JSON ou XML.

Une API SOAP

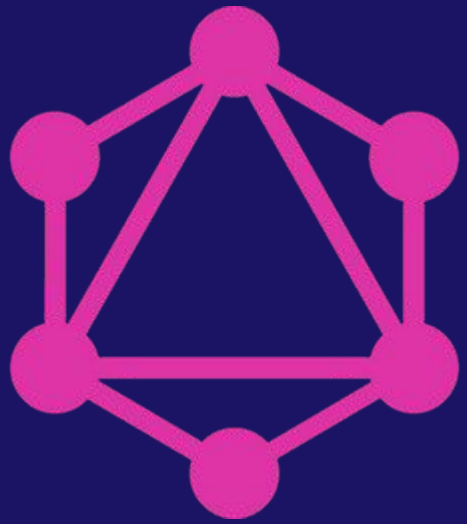
Définition

Une API SOAP (Simple Object Access Protocol) est un protocole de communication pour les services web qui permet l'échange d'informations structurées dans des environnements distribués. Il repose sur XML pour le format des messages et utilise principalement le protocole HTTP ou SMTP pour le transport des messages.

Principe : Utilise le protocole HTTP et XML pour l'échange de messages.

Caractéristiques : Plus rigide et nécessite un formatage spécifique des messages.

Qu'est-ce qu'une API ?



Graphql

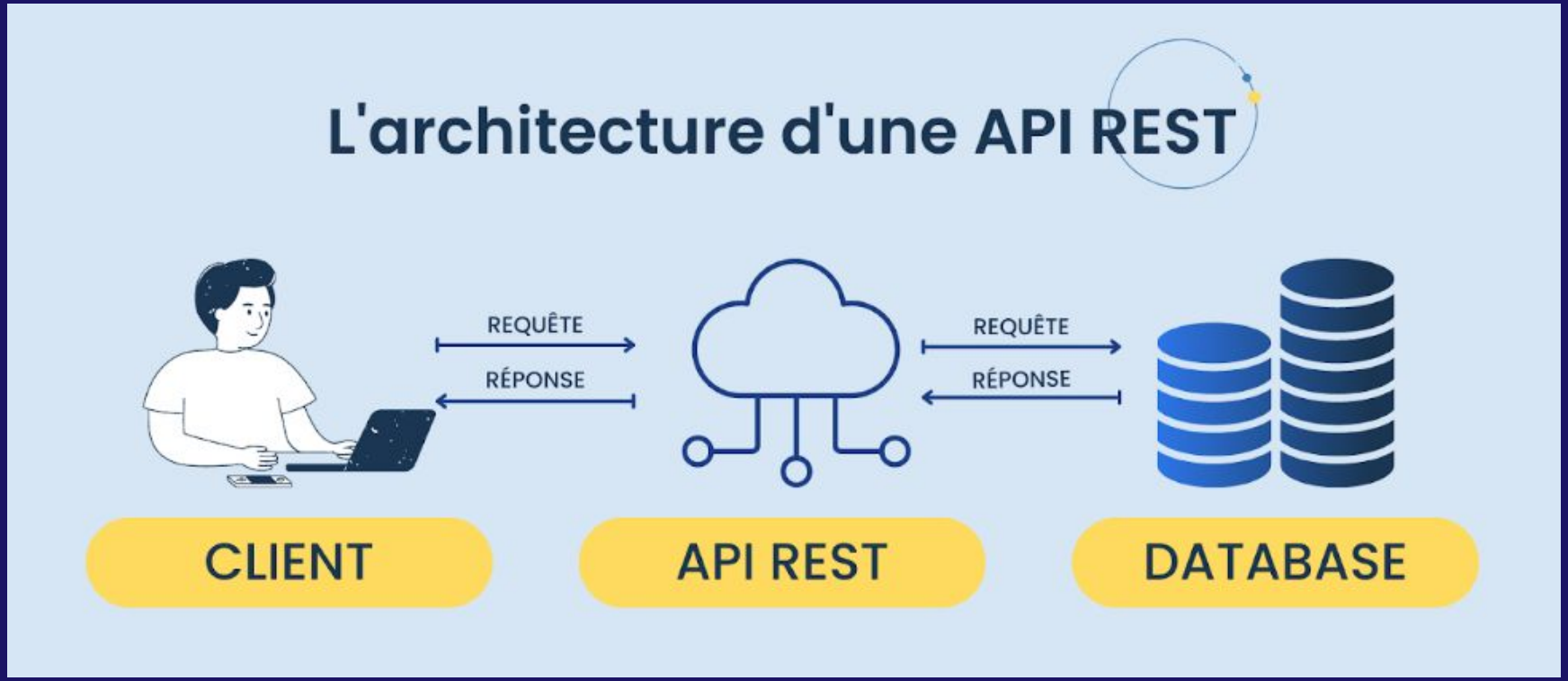
API Graphql

GraphQL est un langage de requête pour les API développé par Facebook en 2012 et ouvert au public en 2015. Contrairement à REST, qui utilise des endpoints prédéfinis pour accéder aux données, GraphQL permet aux clients de définir la structure des données requises, et le serveur répond alors avec les données exactes demandées

GraphQL révolutionne la manière dont les données sont gérées dans les interactions client-serveur, en offrant une flexibilité, une efficacité et une rapidité accrues pour les développeurs d'applications modernes. Il convient particulièrement bien aux applications nécessitant de grandes quantités de données interdépendantes et où la performance du réseau est une préoccupation.

Format de données : JSON ou XML. *https://graphql.org/learn/*

Comment ça fonctionne ?



Verbes HTTP et leurs Utilisations

GET

Récupérer des ressources sans les modifier.
Ex : GET /users - Récupère la liste de tous les utilisateurs.

POST

Créer une nouvelle ressource.
Ex : POST /users - Crée un nouvel utilisateur avec les données fournies dans le corps de la requête.

PUT

Mettre à jour une ressource existante.
Ex : PUT /users/{id} - Met à jour les informations de l'utilisateur avec l'ID spécifié.

DELETE

Supprimer une ressource.
Ex : DELETE /users/{id} - Supprime l'utilisateur avec l'ID spécifié.

PATCH

Mettre à jour partiellement une ressource.
Ex : PATCH /users/{id} - Met à jour partiellement les informations de l'utilisateur avec l'ID spécifié.

Codes de Statut HTTP

Les codes de statut HTTP sont des codes numériques retournés par un serveur web pour indiquer le résultat d'une requête HTTP. Ils sont divisés en cinq catégories, chacune représentant un type différent de réponse. Ces codes aident les clients (comme les navigateurs web ou les applications) à comprendre comment la requête a été traitée par le serveur.

Code 200 : Succès

- 200 OK : Requête réussie.
- 201 Created : Ressource créée avec succès.
- 204 No Content : Requête réussie sans contenu à retourner.

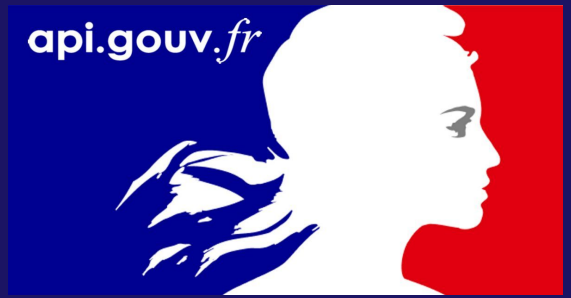
Code 400 : Erreur client

- 400 Bad Request : Requête mal formée.
- 401 Unauthorized : Authentification requise.
- 403 Forbidden : Accès refusé.
- 404 Not Found : Ressource non trouvée.
- 429 Too Many Requests : Le client a envoyé trop de requêtes dans un laps de temps donné.

Code 500 : Erreur Serveur

- 500 Internal Server Error : Erreur interne du serveur.
- 503 Service Unavailable : Service indisponible.

API Rest publique



Définition

Les API publiques sont accessibles au grand public sans restriction majeure. Elles permettent aux développeurs d'accéder aux données et aux services d'une organisation sans avoir besoin d'une authentification stricte.

Caractéristiques :

- Tout le monde peut les utiliser.
- Documentation publique
- Bien que certaines limitations ou quotas puissent s'appliquer

API Rest avec Clé

Définition

Ces API nécessitent l'utilisation d'une clé d'API pour authentifier les requêtes. La clé est généralement obtenue après une inscription gratuite ou payante auprès du fournisseur de l'API.

Caractéristiques :

- Contrôle d'accès par clés
- Les fournisseurs peuvent imposer des limites/forfaits sur le nombre de requêtes.
- Bien que certaines limitations ou quotas puissent s'appliquer

API Rest avec authentification d’utilisateur

Définition

Ces API nécessitent que l'utilisateur s'authentifie, souvent pour accéder à des données personnelles ou pour effectuer des actions en son nom.

Caractéristiques :

- Possibilité de définir des permissions (scopes) ou des niveaux d'accès.
- Authentification par JWT ou par OAuth 2.0

Le Json Web Token

JWT, ou JSON Web Token, est un standard ouvert (RFC 7519) qui permet d'échanger des informations de manière sécurisée entre deux parties sous forme de JSON. Ces informations peuvent être vérifiées et fiables car elles sont signées numériquement.

A quoi ça sert ?

L'utilisation principale des JWT est l'authentification. Lorsqu'un utilisateur se connecte avec ses informations d'identification, le serveur vérifie ces informations et, si elles sont correctes, génère un JWT. Ce token est ensuite envoyé au client et est utilisé pour authentifier les requêtes suivantes.

L'authentification avec JWT est **stateless**, ce qui signifie que le serveur n'a pas besoin de conserver l'état de la session.

Le Json Web Token

A quoi ça ressemble ?

Un JWT bien formé se compose de trois chaînes de caractères concaténées codées en Base64, séparées par un point.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

= Header

= Payload (Body)

= Signature

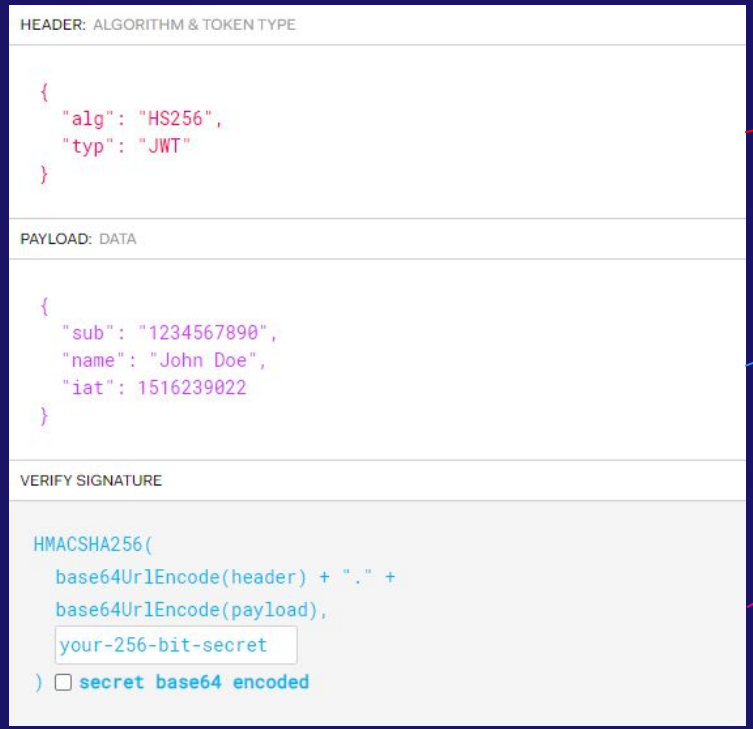
Il contient des informations sur le type de token et l'algorithme de signature utilisé.

Elle contient les données que vous souhaitez transmettre, également appelé claims.

Elle permet de vérifier que le message n'a pas été altéré. Elle est générée en prenant l'encodage base64url du header et du payload, puis en les signant avec un algorithme spécifié dans le header (par exemple, HMAC SHA256) en utilisant une clé secrète.

Le Json Web Token

Décodé, ça donne quoi ?



Il contient des informations sur le type de token et l'algorithme de signature utilisé.

Elle contient les données que vous souhaitez transmettre, également appelé claims.

Elle permet de vérifier que le message n'a pas été altéré. Elle est générée en prenant l'encodage base64url du header et du payload, puis en les signant avec un algorithme spécifié dans le header (par exemple, HMAC SHA256) en utilisant une clé secrète.

Postman / Insomnia

Qu'est ce que c'est ?

Postman et Insomnia sont des clients HTTP graphiques qui permettent aux développeurs de tester, de déboguer et de documenter des APIs.

Ils offrent une interface conviviale pour construire et envoyer des requêtes HTTP, analyser les réponses, et automatiser les tests.

Faire un appel API en PHP avec la bibliothèque Curl

```php
// Initialisation de cURL
$ch = curl_init();

// Configuration des options cURL
$apiUrl = "https://wttr.in/{$city}?format=j1&lang=fr";
curl_setopt($ch, option: CURLOPT_URL, $apiUrl);
curl_setopt($ch, option: CURLOPT_RETURNTRANSFER, value: true);

// Exécution de la requête
$response = curl_exec($ch);

// Vérification des erreurs
if ($response === false) {
    $error = curl_error($ch);
    curl_close($ch);
    die("Erreur lors de la récupération des données : {$error}");
}

// Fermeture de la session cURL
curl_close($ch);
```

Cas pratique:
Utiliser l'API de Spotify

Objectif : Cet exercice a pour but de vous familiariser avec le concept d'API en créant une application permettant de récupérer des information depuis l'api de Spotify.

Consignes:

- Créer un compte "Spotify for développer" & récupérer un client_id et un client_secret
- Tester la récupération de token avec Postman ou Insomnia
- Tester le token sur deux requête de l'API Postman

- Créer une application qui permet de récupérer les informations d'un profil utilisateur spotify
- Sauvegarder le token pour ne pas demander l'authentification systématiquement
- Gérer la durée de vie du token
- Récupérer la liste des artiste suivis par l'utilisateur avec illustrations & lien vers spotify
- Inclure une recherche de musique avec possibilité de lecture de la musique

- (Optionnel) Recherche un artiste et pouvoir soit le suivre, soit ne plus le suivre

Lien vers la documentation: https://developer.spotify.com/documentation/web-api

Symfony 7 & API Platform

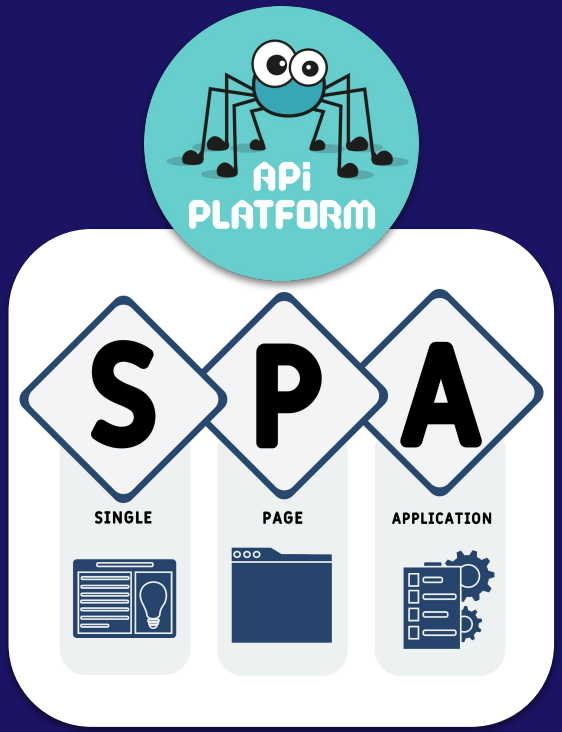
API Platform, qu'est ce que c'est ?

C'est un framework open-source puissant et complet pour la création d'APIs web modernes. Il est basé nativement basé sur le framework Symfony mais depuis peu est disponible sur Laravel également. Il fournit un ensemble d'outils pour construire rapidement des APIs conformes aux standards, performantes et sécurisées.

Les avantages :

- **Génération automatique de l'API** à partir de vos entités Doctrine.
- **Documentation interactive**.
- **Gestion automatique de la pagination, des filtres, et des tris**.
- **Validation des données** via les contraintes de validation Symfony.
- **Support de l'authentification** via JWT, OAuth, etc.
- **Optimisation des performances.**

Symfony 7 & API Platform



Quand l'utiliser ?

Attention, utiliser Api Platform sur des petites infrastructure de données peu être surdimensionné.

Si vous avez besoin d'un contrôle extrêmement précis sur chaque aspect de l'API, écrire votre propre implémentation sans framework est préférable.

Aujourd'hui, il est de plus en plus courant d'utiliser Symfony + API Platform dans le cadre de **SPA** avec React ou Vue js.

Qu'est ce qu'une SPA ?

Une Single Page Application est une application web qui fonctionne à l'intérieur d'une seule page web en chargeant dynamiquement les contenus nécessaires au fur et à mesure des interactions de l'utilisateur, sans recharger entièrement la page.

Cette approche offre une expérience utilisateur plus fluide et réactive, similaire à celle des applications de bureau ou mobiles.

Créer une App Symfony 7

Prérequis :

- PHP >= 8.2
- Composer 2
- Wamp / Lamp / Xamp
- **Symfony CLI**

Symfony CLI : c'est un outil en ligne de commande pour les développeurs qui utilisent le framework Symfony. Il permet d'automatiser certaines tâches courantes telles que la création de nouveaux projets, la génération de code, la gestion des dépendances, etc.

Poun installer Symfony CLI (Pour **C**ommand-**L**ine **I**nterface) :

sous Windows :

Pour installer Symfony CLI sous Windows, il vous faudra au préalable installer scoop, un gestionnaire de paquets pour Windows.

Puis lancer la commande suivante :

```
scoop install symfony-cli
```

sous Linux || Mac :

via wget:

```
wget https://get.symfony.com/cli/installer -O - | bash
```

via curl:

```
curl -sS https://get.symfony.com/cli/installer | bash
```

via Homebrew (Homebrew requis):

```
brew install symfony-cli/tap/symfony-cli
```

Environnement de Travail

Si “Symfony CLI” est correctement installé, vous pouvez lancer la commande:

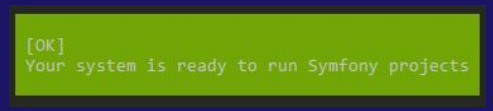
```
symfony help
```

Cette commande vous affichera votre version Symfony et la liste des commandes disponibles.

Il est nécessaire de les vérifier avant de démarrer un nouveau projet ou de mettre à jour un projet existant.

Afin de valider que tous les prérequis sont installés et configurés correctement, vous pouvez lancer la commande suivante:

```
symfony check:requirements
```



Environnement de Travail

Création d'un projet symfony :

Grâce à Symfony CLI, nous sommes désormais capables de créer un projet grâce à la commande suivante.

```
symfony new `Nom_du_projet`
```

Il est possible de personnaliser l'installation d'un projet avec les options suivantes:

`--version="X.X.XX"`	Permet de spécifier la version
`--webapp`	Créer un projet complet avec toutes les fonctionnalités nécessaires pour développer une application web
`--no-git`	Cela indique à Symfony CLI de ne pas initialiser un dépôt Git pour le nouveau projet.
`--demo`	Permet de créer un projet de démonstration avec des exemples de code.

Afin de connaître toutes options d'une commande, vous pouvez faire :

```
symfony help `command`
```

Exemple :

```
symfony help new
```

Environnement de Travail

Pour lancer votre serveur pour le projet Symfony, vous devez fair la commande suivante:

```
symfony serve
```

Vous pouvez aussi utiliser l'option `-d`, pour lancer votre serveur pour le lancer en fond.

Il est désormais possible d'accéder à votre projet en vous à l'adresse suivante: 127.0.0.1:8000

Si le port 8000 n'est pas disponible, votre serveur prendra directement le suivant.

Pour couper votre serveur, il faudra faire :

```
symfony server:stop
```

Installation de API Platform

Afin d'installer API Platform à votre projet Symfony, vous pouvez faire la commande suivante:

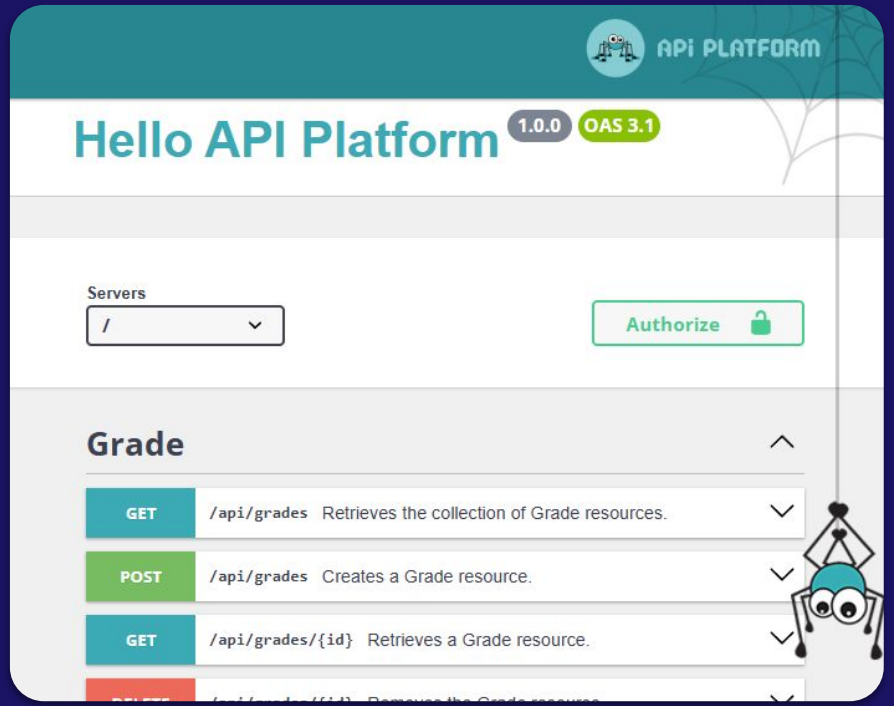
```
symfony composer require api
```

Ensuite, vous pouvez créer votre base de données avec la commande suivante si vous avez bien renseigné "**DATABASE_URL**" dans votre fichier d'environnement.

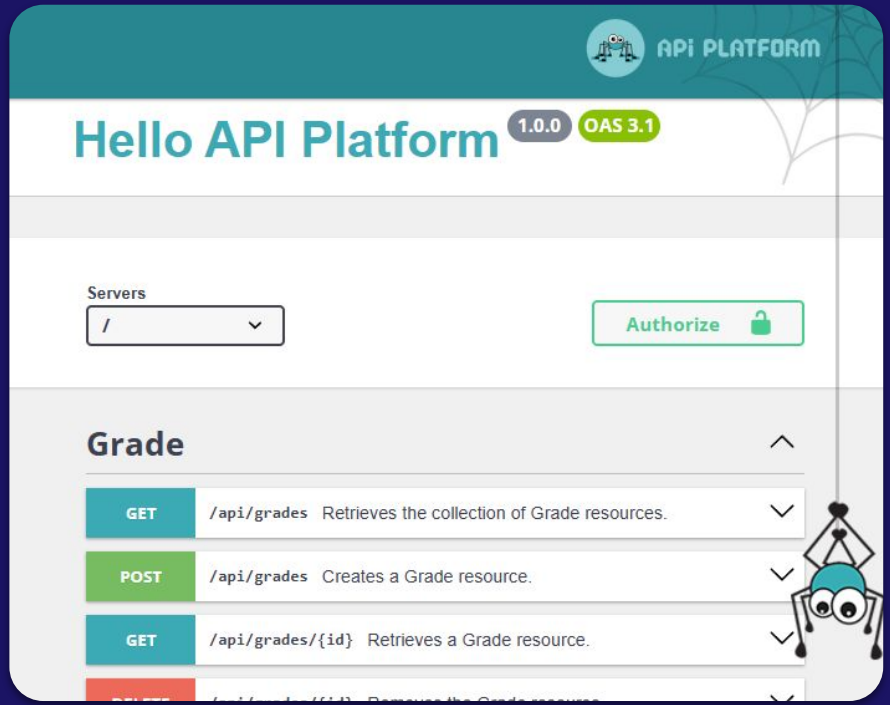
```
symfony console doctrine:database:create
```

Si vous utilisez le MakerBundle pour créer vos entitées, vous pour ajouter ce préfixe pour intégrer directement les annotations de API Platform.

```
php bin/console make:entity --api-resource
```



Installation de API Platform



Vous pouvez désormais accéder à l'interface Swagger UI de API Platform à partir de l'URL suivantes:

https://localhost:8000/api

Swagger UI est un outil qui génère automatiquement une documentation conviviale pour votre API, offrant aux développeurs et aux utilisateurs une façon simple et intuitive d'interagir avec les endpoints disponibles.

Créer une route personnalisé

Comment faire ?

Avec API Platform, il est possible de créer des routes personnalisé en passant par des controlleur. Pour que API Platform puisse intégrer des route personnalisé, il vous faudra mettre à jour votre fichier ***api_platform.yaml*** et lui préciser :

Ensuite, il va falloir créer le contrôleur qui va lancer notre action personnalisé avec la méthode __invoke.

packages\api_platform.yaml

```yaml
api_platform:
    title: Hello API Platform
    version: 1.0.0
    use_symfony_listeners: true
    defaults:
        stateless: true
        cache_headers:
            vary: ['Content-Type', 'Authorization', 'Origin']
```

src\Controller\CustomController.php

```php
#[AsController]
class CustomController extends AbstractController
{
    no usages
    public function __invoke(): JsonResponse
    {
        $data = [
            'message' => 'Ceci est une route personnalisée.',
            'timestamp' => time(),
        ];

        return new JsonResponse($data);
    }
}
```

Ajouter la route au Swagger

Comment faire ?

Pour ajouter la route à notre Swagger, il va nous falloir compléter la liste des opérations disponibles dans l'annotation **[#ApiResource].**

Néanmoins, il nous faudra préciser les routes par défauts car sinon ces dernières ne s'afficheront plus.

Voici un exemple de routes personnalisée:

```
22 usages
#[ORM\Entity(repositoryClass: GradesRepository::class)]
#[ApiResource(operations: [
    new Get(),                  // GET /grades/{id}
    new Put(),                  // PUT /grades/{id}
    new Patch(),                // PATCH /grades/{id}
    new Delete(),               // DELETE /grades/{id}
    new GetCollection(),        // GET /grades
    new Post(),                 // POST /grades
    new Post(
        uriTemplate: '/grades/custom',
        controller: CustomController::class,
        input: false,
        output: false,
        read: false,
        deserialize: false,
        validate: false,
        name: 'custom'
    )
])]
class Grade {
```

Ajouter de l'authentification avec un token JWT

Installation de Lexik JWT AuthBundle:

Afin de pouvoir gérer l'authentification en JWT nous allons avoir besoin du paquet suivant

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

Ensuite il va vous falloir générer les clés privée & public avec la commande suivante:

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

Le Lexik JWT AuthBundle utilise par défaut une signature asymétrique avec l'algorithme RS256 (RSA avec SHA-256).

Les clés privé & publique seront générés et stocké dans le dossier "jwt" dans le dossier config et le chemin pour accéder sera stocké dans une variable d'environnement.

A quoi servent les clés dans ce contexte ?

Clé privée

- Elle est utilisée uniquement pour signer les tokens JWT lors de leur génération.
- Elle doit être conservée secrète et ne jamais être exposée publiquement. Elle est généralement stockée sur le serveur d'authentification.
- La sécurité du système repose en grande partie sur la confidentialité de cette clé. Si elle est compromise, un attaquant pourrait signer des tokens valides.

Clé publique

- La clé publique est utilisée pour vérifier la signature des tokens JWT lors des requêtes entrantes.
- Elle peut être partagée librement avec les parties qui ont besoin de vérifier les tokens, par exemple d'autres services
- Comme elle ne permet pas de signer des tokens, sa divulgation ne compromet pas la sécurité du système.

Et dans un contexte de signature symétrique

Clé secrète unique

- Elle est utilisée uniquement pour signer les tokens JWT lors de leur génération et pour vérifier la signature des tokens JWT lors des requêtes entrantes.
- Ce schéma de signature est simple. Les signatures symétriques empêchent le partage du JWT avec un autre service.
- Pour vérifier l'intégrité du JWT, tous les services devraient accéder à la même clé secrète. Toutefois, la possession d'une clé secrète suffit pour générer des JWT arbitraires avec une signature valide.

La phrase secrète

Qu'est ce que c'est ? A quoi ça sert ?

- La passphrase (ou phrase secrète) est une chaîne de caractères utilisée pour protéger votre clé privée.
- Dans le contexte du lexik/jwt-authentication-bundle, la passphrase sert à chiffrer la clé privée qui est utilisée pour signer les tokens JWT.
- Elle ajoute une couche supplémentaire de sécurité en cas de compromission de la clé privée.

Création de la classe “User”

Utilisation du MakerBundle

Au lieu de créer notre classe User comme on créer n’importe quelle autre table on va utiliser la commande suivante du MakerBundle.

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

Cela permet d’incorporer directement la “UserInterface” de Symfony.

N’oubliez pas de créer un fichier de migrations une fois la classe créés

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

Et de l’appliquer sur votre base de données.

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

Configurer le SecurityBundle

Afin de configurer correctement de “SecurityBundle”, on va se rediriger la documentation officielle de API Platform:

https://api-platform.com/docs/core/jwt/#jwt-authentication

Tester de vous authentifier

Création d'un utilisateur

Afin de rendre votre classe accessible via le Swagger d'API Platform, n'oubliez pas d'y ajouter l'annotation suivante en amont de la classe

```
#[ApiResource]
```

Votre IDE devrait automatiquement vous ajouter l'import associé.

Vous pouvez désormais créer un nouvelle utilisateur avec la méthode POST depuis votre Swagger ou depuis un outil comme POSTMAN ou Insomnia.

Sinon vous pouvez créer un utilisateur à la main dans votre base de données. Néanmoins vous ne pourrez pas saisir de mot de passe en clair, donc pour connaître le Hash du mot passe que vous souhaitez créer, vous pouvez utiliser la commande suivante:

```
php bin/console security:hash-password
```

Comment restreindre routes ?

Gestion des rôles

Vous pouvez restreindre l'accès à vos route, en utilisant les roles, par défaut “ROLE_USER” & “ROLE_ADMIN” au niveau de l’entité. Ainsi toutes les routes qui en découlent nécessitent d’avoir le role requis ou niveau d’autorisation requis.

```
16 usages
#[ORM\Entity(repositoryClass: StudentRepository::class)]
#[ApiResource(
    security: "is_granted('IS_AUTHENTICATED_FULLY')",
    securityMessage: "Vous devez être authentifié pour accéder à cette ressource."
)]
class Student
    {
```

Ou vous pouvez spécifier les habilitations requises directement au niveau de la route comme ci-contre :

```
new Post(
    uriTemplate: '/grades/custom',
    controller: CustomController::class,
    security: "is_granted('ROLE_USER')",
    input: false,
    output: false,
    read: false,
    deserialize: false,
    validate: false,
    name: 'custom'
)
```

Les filtres

Qu'est ce que c'est ?

Les filtres sont un mécanisme d'API Platform qui vous permet, par le biais de simples paramètres dans l'URL, de restreindre, trier ou sélectionner plus précisément les données que votre API retourne.

En voici quelques uns parmis les plus utilisé

SearchFilter	Permet de filtrer les ressources en fonction de certains champs. Par exemple, vous pouvez chercher tous les livres dont le titre contient un certain mot.
OrderFilter	Permet de trier les résultats selon un ou plusieurs champs (ex. ?order[title]=asc).
RangeFilter	Permet de filtrer selon des valeurs minimales et/ou maximales (ex. filtrer tous les produits dont le prix est entre 10€ et 200€).

Les filtres

Comment les utiliser ?

Vous devez annoter votre classe (entité) avec **#[ApiFilter]**.

```php
#[ORM\Entity(repositoryClass: StudentRepository::class)]
#[ApiResource(
    normalizationContext: ['groups' => ['read']],
    denormalizationContext: ['groups' => ['write']],
)]
#[ApiFilter(SearchFilter::class, properties: ['name' => 'partial'])]
class Student
    {
```

Ajuster le comportement ?

Vous pouvez spécifier le comportement de la recherche par propriété

exact
L'attribut doit correspondre exactement.

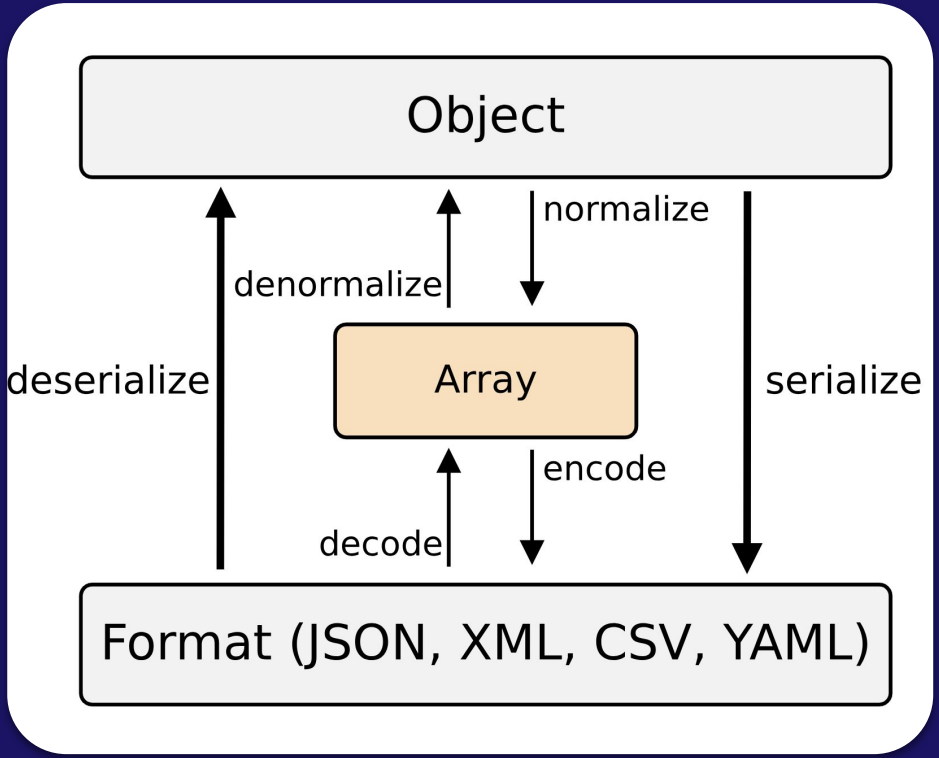
partial
Recherche partielle (LIKE %term%).

start
Recherche partielle au début (term%).

end
Recherche partielle à la fin (%term).

word_start
Recherche par mots.

La sérialisation



Définition

La sérialisation est le processus de conversion d'objets en un format qui peut être facilement stocké ou transmis, comme JSON, XML, ou YAML. Dans Symfony, la sérialisation est souvent utilisée pour convertir des entités en réponses JSON dans des API.

Le normaliseur

Le rôle de ce normaliseur est de transformer un objet spécifique en un tableau associatif personnalisé avant la sérialisation finale.

Normaliser une date

```php
#[ORM\Column(type: Types::DATE_MUTABLE, nullable: true)]
#[Context([DateTimeNormalizer::FORMAT_KEY => 'Y-m-d'])]
#[Groups(['read', 'write'])]
private ?\DateTimeInterface $birthdayDate = null;
```

Comment faire ?

Normaliser une date revient à passer un objet DateTime au serializer qui, grâce au **DateTimeNormalizer**, le convertira en une chaîne de caractères. Vous pouvez personnaliser le format via le contexte.

Faire des groupes de Sérialisation

Qu'est-ce que c'est ?

Les groupes de sérialisation sont un mécanisme fourni par le composant Serializer de Symfony qui permet de contrôler quelles propriétés d'un objet sont sérialisées (converties en JSON, XML, etc.) lors de la réponse de votre API.

En utilisant des groupes, vous pouvez inclure ou exclure certaines propriétés selon le contexte, comme le type de requête (GET, POST), le rôle de l'utilisateur, ou tout autre critère.

A quoi ça sert ?

- Limiter l'exposition de certaines propriétés sensibles (comme les mots de passe, les données personnelles).
- Réduire la taille des réponses en n'incluant que les données nécessaires.
- Adapter les données attendues lors de la création ou de la mise à jour d'une ressource.

Exemple de groupe de Sérialisation

```
16 usages
#[ORM\Entity(repositoryClass: StudentRepository::class)]
    #[ApiResource(
        normalizationContext: ['groups' => ['read']],
        denormalizationContext: ['groups' => ['write']],
    )]
    class Student
    {
    1 usage
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    #[Groups(['write'])]
    private ?int $id = null;

    2 usages
    #[ORM\Column(length: 255)]
    #[Groups(['read', 'write'])]
    private ?string $name = null;

    2 usages
    #[ORM\Column(length: 255)]
    #[Groups(['read', 'write'])]
    private ?string $firstname = null;
```

Sérialiser des relations

Il est également possible de sérialiser les relations d'une entités, dans ce cas il faut donner aux propriétées de l'entité correspondante les même groupes de sérialisation.

```
class Grade {
    1 usage
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    2 usages
    #[ORM\Column(nullable: true)]
    #[Groups(['read'])]
    private ?float $score = null;

    2 usages
    #[ORM\Column]
    #[Groups(['read'])]
    private ?float $maxScore = null;
```

Les Assertions

Qu'est-ce que c'est ?

Les assertions (ou contraintes de validation) dans Symfony sont des annotations que vous ajoutez à vos propriétés d'entité ou à vos DTO pour contrôler la validité des données.

Elles sont gérées par le Composant Validator de Symfony.

Lorsqu'une requête entrante (POST, PUT, PATCH, etc.) tente de créer ou de modifier une ressource, le Validator vérifie que les données fournies respectent les contraintes définies.

```
2 usages
#[ORM\Column(length: 255)]
#[Assert\NotBlank(message: "Le nom de l'examen est requis")]
#[Assert\Length(max: 255, maxMessage: "Le nom de l'examen ne peut dépasser 255 caractères.")]
private ?string $name = null;
```

TP: Créer une API à l’aide de API Platform et Symfony

Objectif :

Cet exercice a pour but de vous familiariser avec la mise en place d'une API RESTful avec Symfony et API Platform.

Consignes:

Vous devez développer une API pour une application de gestion d'une bibliothèque en ligne.

- S’inscrire et se connecter via une authentification JWT.
- Consulter la liste des livres. (avec l’id, le prénom et nom de l’auteur).
- Consulter la liste des auteurs. (avec la liste de ces livre, titre et id uniquement).
- Ajouter, modifier et supprimer des auteurs (pour les utilisateurs authentifiés).
- Ajouter, modifier et supprimer des livres (pour les utilisateurs authentifiés).
- Rechercher des livres par titre, auteur ou genre.
- Faite une route personnalisé pour avoir les nouveaux livres (Créée il y a moins de 7 jours)
- Utiliser des groupes de sérialisation pour contrôler les données exposées.
- Documenter votre API

- (Optionnel) Gérer des catégories par livres M2M

TP: Créer une API à l'aide de API Platform et Symfony

L'entité Book

id	title	author_id	genre	summary	publicationDate	createdAt	updatedAt

L'entité Author

id	name	firstname	birthDate	deathDate	nationality	createdAt	updatedAt

Créer une API avec Symfony 7

Créer une API sans API Platform ?

L'intérêt de « faire à la main » est d'avoir un contrôle total sur la logique métier, l'architecture, la structure des endpoints, ainsi que de maîtriser chaque couche de l'API. C'est un bon choix pour un projet sur mesure ou lorsque vous voulez éviter la surcouche et la « magie » d'un outil comme API Platform.

Néanmoins, si le projet nécessite un CRUD rapide, la documentation automatique ou la gestion avancée des relations entre entités, API Platform reste un choix très efficace.

Symfony

La structure de Symfony

bin

Si vous avez réalisé une installation avec l'option `*--webapp*`, vous trouverez les deux fichiers suivant dans ce dossier:

- console
- phpunit

Le fichier `*console*` est un script de ligne de commande (CLI) utilisé pour exécuter les tâches du projet Symfony. Vous pouvez accéder à la liste des commandes disponible grâce à:

`*php bin/console*` ou `*symfony*`

Le fichier `phpunit` est un script de ligne de commande (CLI) qui permet de lancer les tests unitaires d'une application Symfony en utilisant le framework de tests unitaires PHPUnit.

Vous pouvez exécuter vos test à l'aide de la commande suivante: `*php bin/phpunit*`

La structure de Symfony

config

Il contient les fichiers de configuration de votre application au format **Yaml***.

Les fichiers de configuration sont généralement découpés en plusieurs fichiers distincts pour une gestion plus claire et plus facile des différentes configurations du projet.

Il est également possible de définir des configurations spécifiques à des environnements tels que le développement, la production, etc... avec la syntaxe suivante:

```
when@dev
when@prod
when@test
```

Cela permet de séparer les configurations en fonction de l'environnement, facilitant ainsi la gestion et le déploiement de l'application.

Yaml (Yet Another Markup Language): C'est une alternative populaire aux formats de configuration plus complexes tels que XML et JSON.

Dans YAML, les données sont représentées sous forme de clés et de valeurs, séparées par des deux-points et organisées en blocs indentés.

La structure de Symfony

- bin
- config
- migrations
- public
- src
- templates
- tests
- translations
- var
- vendor
- .env
- composer.json
- composer.lock

migrations

Ce dossier contient les scripts de migration de base de données. Les migrations de base de données sont utilisées pour gérer les modifications apportées à la structure de la base de données d'une application au fil du temps.

Pour rappel, il est interdit de modifier directement la BDD. Il faut toujours passer par des migrations.

Une migration décrit les modifications à apporter à la base de données et peut être exécutée pour mettre à jour la base de données.

Les migrations peuvent également être annulées pour restaurer la base de données à un état antérieur.

Créer une migration : `symfony console make:migration`

Checker le status de nos migrations : `symfony console doctrine:migrations:status`

Lancer une migrations : `symfony console doctrine:migrations:migrate`

La structure de Symfony

public

Il contient les fichiers qui sont accessibles directement depuis le navigateur web. C'est le point d'entrée pour les requêtes HTTP et le contenu qui est servi aux utilisateurs.

On y retrouve le fichier `index.php` qui a pour but de créer une nouvelle instance de **Kernel**.

Le Kernel est le cœur d'une application Symfony. Il s'agit d'une classe qui définit le fonctionnement général de l'application et qui est utilisée pour traiter les requêtes HTTP.

Le Kernel est responsable de :

- Charger les différents fichiers configurations de l'application.
- Enregistrer les services nécessaires pour le fonctionnement de l'application.
- Enregistrer les routes de l'application et les lier à des contrôleurs.
- Gérer les erreurs et les exceptions pendant l'exécution de l'application.

La structure de Symfony

src

C'est dans ce dossier que 80% de votre code va se trouver car on y retrouve :

- Vos Modèles dans le dossier `Entity`.
- Vos Contrôleurs dans le dossier du même nom.

On va également trouver un dossier Repository qui va contenir l'ensemble des requêtes vers la BDD que l'on pourra appeler depuis un contrôleur.

C'est aussi ici que se trouve la classe Kernel dont nous avons parlé précédemment.

template

Il contient toutes les Vues de notre application, au format twig.

C'est donc ici que la structure des pages HTML de votre application va se trouver.

Le format twig permet d'y ajouter des variables récupérer de vos contrôleurs et des instructions conditionnelles.

La structure de Symfony

translations

Il sert à stocker les fichiers de traduction pour les différentes langues prises en charge par l'application.

L'utilisation de fichiers de traduction permet une meilleure flexibilité dans la prise en charge des différentes langues et une maintenance plus facile des traductions.

var

On y trouve les dossiers suivants:

cache Symfony va venir stocker certaines informations ici pour charger plus rapidement.

⚠ `symfony console cache:clear`

log Permet de récupérer l'historique des logs de l'application, on peut également les récupérer dans la console avec :

`symfony server:log`

La structure de Symfony

vendor

Il contient toutes les dépendances externes utilisées par l'application, telles que les librairies PHP. Il est généré lors de l'installation de ces dépendances en utilisant un outil de gestion de paquets, comme Composer.

Il est interdit de modifier directement les fichiers du dossier `*vendor*`, car ces modifications seront perdues lors de la prochaine mise à jour des paquets.

La structure de Symfony

.env

Il servir à stocker toutes les informations qui vont varier en fonction de l'environnement, comme les informations nécessaire pour se connecter à la base de données:

```
DATABASE_URL="mysql://root:root@127.0.0.1:3306/app?serverVersion=5.7.24 &charset=utf8"
```

Étant donné que ce fichier peut contenir des données sensibles, il est recommandé de créer une copie de ce fichier qui s'appellera `.env.local`.

Votre application utilisera par défaut les variable d'environnement présente dans `.env.local`. Il ne faudra pas comité ce dernier car c'est maintenant lui qui contient des données sensibles, mais vous pourrez comité le fichier `.env` en y indiquant toutes les variables nécessaires pour faire fonctionner votre application avec des valeurs par défaut.

```
DATABASE_URL="TYPE://USERNAME:PASSWORD@URL:PORT/app?serverVersion=X.X.XX &charset=utf8"
```

C'est dans ce fichier que vous allez définir si vous êtes en `prod` ou en `dev`:

```
APP_ENV=dev
```

La structure de Symfony

composer.json

Il regroupe toutes les dépendances de votre projet. Il indique les paquets PHP que votre projet a besoin pour fonctionner et les versions de ces paquets que vous souhaitez utiliser.

composer.lock

Il contient les informations exactes sur les versions des paquets installés. Il est utilisé pour garantir que toutes les installations sont reproductibles et que les dépendances sont installées de manière cohérente.

Il est généré automatiquement par Composer lors de l'installation ou de la mise à jour de packages.

Initialisation de l'API

Création du projet

Pour créer le squelette de notre API, nous allons ici initier à l'aide de Symfony CLI un nouveau projet.

```
symfony new `Nom_du_projet`
```

Etant dans le cadre de la création d'une API, nous n'allons pas utiliser l'option "--webapp" qui est utilisée pour configurer rapidement un projet orienté application web classique.

Pour rappel "--webapp" ajoute des packages spécifiques comme la gestion du moteur de templating twig, les gestions des sessions et des formulaires, la gestion des assets et tant d'autre chose qui sont inutile dans le cadre de la création API.

Initialisation de l'API

Création d'un fichier .env.local

Nous venons ensuite créer et compléter notre fichier .env.local afin de pouvoir nous connecter à notre base de données (voir page 54).

Dans la mesure ou vous n'avez pas créé votre base de données en amont, vous pouvez directement la créer à partir des données saisies dans la variable "**DATABASE_URL**" à l'aide de la commande suivante :

```
php bin/console doctrine:database:create
```

N'oubliez d'installer Doctrine en amont à l'aide la commande suivante:

```
composer require symfony/orm-pack
```

Toute les variables d'environnement sont chargées par Symfony au démarrage pour les rendre accessibles via la variable "**$_ENV**"

Doctrine

Doctrine est un **ORM** (Object-Relational Mapper) intégré à Symfony, qui permet de gérer les interactions entre une base de données relationnelle et les objets de ton application.

Au lieu d'écrire directement des requêtes SQL, tu manipules des entités PHP, et Doctrine se charge de convertir ces objets en requêtes SQL pour interagir avec la base de données.

Initialisation de l'API

Création des entités

Nous allons maintenant pouvoir passer à la création de notre première entitée. Pour ce faire nous allons avoir besoin d'installer le Maker Bundle

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

Nous pouvons désormais commencer la création de nos entités avec la commande suivante:

```
symfony console make:entity
```

Si vous souhaitez modifier une entitée existante, vous pouvez utilisez la même commande en spécifiant l'entité à modifier.

```
symfony console make:entity <Nom de l'entité>
```

NB: Cette commande va également vous créer le Repository associé.

--dev

L'option --dev est utilisée lors de l'installation de dépendances avec Composer pour indiquer que la dépendance doit être installée uniquement dans l'environnement de développement et non dans l'environnement de production.

Elles ne seront pas installées lorsque tu exécutes: **composer install --no-dev**

Ce qui est souvent utilisé en production pour réduire la taille du projet et éviter les paquets inutiles.

Initialisation de l'API

Création de relations

Pendant le processus de création des propriétés, vous pouvez également définir vos relations entre vos tables.

Pour ce faire, vous n'avez qu'à créer vos propriétés correspondant à vos relations, et leur donner le type : "relation". Suite à quoi, vous n'aurez plus qu'à saisir la table avec laquelle vous voulez créer une relations puis indiquer le type de relation.

Petit plus, vous aurez également l'options de créer la relation inverse dans l'autre table.

Répercuter mes modifications en BDD

Pour répercuter vos modifications mettre à jours votre base de données, en vous basant sur vos table, vous pouvez utiliser la commande :

```
php bin/console doctrine:schema:update --force
```

```
Class name of the entity to create or update (e.g. DeliciousElephant):
 > Task
Task

 Your entity already exists! So let's add some new fields!

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

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

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

What type of relationship is this?
 ------------ ---------------------------------------------------------------------
  Type         Description
 ------------ ---------------------------------------------------------------------
  ManyToOne    Each Task relates to (has) one Project.
               Each Project can relate to (can have) many Task objects.

  OneToMany    Each Task can relate to (can have) many Project objects.
               Each Project relates to (has) one Task.

  ManyToMany   Each Task can relate to (can have) many Project objects.
               Each Project can also relate to (can also have) many Task objects.

  OneToOne     Each Task relates to (has) exactly one Project.
               Each Project also relates to (has) exactly one Task.
 ------------ ---------------------------------------------------------------------

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

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

 Do you want to add a new property to Project so that you can access/update Task obj
 >

 A new property will also be added to the Project class so that you can access the r

 New field name inside Project [tasks]:
 >

 updated: src/Entity/Task.php
 updated: src/Entity/Project.php
```

Initialisation de l’API

Utiliser des fichiers de migrations

Il est généralement préférable de passer par les migrations plutôt que d'utiliser directement la commande ***doctrine:schema:update*** pour répercuter les modifications de tes entités sur tes tables, voici les avantages des migrations:

- Suivi des modifications
- Contrôle total des changements
- Rétrocompatibilité (Rollbacks)
- Collaboration en équipe

Afin de créer une migration pour entre entité récemment créé, vous pouvez lancer la commande suivante:

```
symfony console make:migration
```

Pour exécuter la migration :

```
symfony console doctrine:migrations:migrate
```

Pour RollBack

```
symfony console doctrine:migrations:execute “Version” --down
```

Initialisation de l'API

```php
7 usages
class ProjectFixtures extends Fixture implements FixtureGroupInterface
{
    2 usages
    public const PROJECT_REF = 'project_';

    no usages
    public static function getGroups(): array
    {
        return ['projects'];
    }

    public function load(ObjectManager $manager): void
    {
        $faker = Factory::create( locale: 'fr_FR');

        for ($i = 1; $i <= 5; $i++) {
            $project = new Project();
            $project->setName($faker->company());
            $project->setDescription($faker->paragraph( nbSentences: 2));

            $manager->persist($project);

            $this->addReference( name: self::PROJECT_REF . $i, $project);
        }

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

Génération de Datafixtures

Les DataFixtures sont un mécanisme essentiel pour initialiser ou peupler ta base de données avec des données de test ou de démo.

Voici la commande qui permet d'installer le le paquet:

```
composer require –dev doctrine/doctrine-fixtures-bundle
```

Cela va vous ajouter la classe **AppFixtures** dans "*src/DataFixtures*". C'est dans cette classe que vous allez pouvoir insérer votre logique pour créer vos données de test.

Pour créer d'autre classe de Fixtures:

```
symfony console make:fixtures
```

Puis lancer la génération:

```
symfony console doctrine:fixtures:load
```

Pour la génération, vous pouvez rajouter l'option "**--append**", pour ne pas écrasez les données existante.

Vous pouvez également utiliser l'option "--group" pour lancer spécifiquement une Fixture.

src/DataFixtures/ProjectFixtures.php

```php
7 usages
class ProjectFixtures extends Fixture implements FixtureGroupInterface
{
    2 usages
    public const PROJECT_REF = 'project_';

    no usages
    public static function getGroups(): array
    {
        return ['projects'];
    }

    public function load(ObjectManager $manager): void
    {
        $faker = Factory::create( locale: 'fr_FR');

        for ($i = 1; $i <= 5; $i++) {
            $project = new Project();
            $project->setName($faker->company());
            $project->setDescription($faker->paragraph( nbSentences: 2));

            $manager->persist($project);

            $this->addReference( name: self::PROJECT_REF . $i, $project);
        }

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

src/DataFixtures/TaskFixtures.php

```php
usages
class TaskFixtures extends Fixture implements DependentFixtureInterface, FixtureGroupInterface
{
    no usages
    public static function getGroups(): array
    {
        return ['tasks'];
    }

    no usages
    public function getDependencies(): array
    {
        return [
            ProjectFixtures::class,
        ];
    }

    public function load(ObjectManager $manager): void
    {
        $faker = Factory::create( locale: 'fr_FR');

        for ($i = 1; $i <= 10; $i++) {
            $task = new Task();
            $task->setName($faker->sentence( nbWords: 3));
            $task->setDescription($faker->paragraph());

            $projectNumber = rand(1, 5);
            $projectRef = ProjectFixtures::PROJECT_REF . $projectNumber;

            $task->setProject($this->getReference($projectRef, class: Project::class));

            $manager->persist($task);
        }

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

Initialisation de l'API

Création du premier "Controller"

Un controller dans Symfony est une classe qui reçoit les requêtes HTTP, exécute la logique métier nécessaire, puis retourne une réponse (page web, JSON, etc.). Il agit comme intermédiaire entre le modèle (données) et la vue (interface utilisateur).

Pour créer notre premier controller, nous pouvons utiliser le Maker-Bundle

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

```php
no usages
class ProjectController extends AbstractController
{
    no usages
    #[Route('/project', name: 'app_project')]
    public function index(): JsonResponse
    {
        return $this->json([
            'message' => 'Welcome to your new controller!',
            'path' => 'src/Controller/ProjectController.php',
        ]);
    }
}
```

```php
no usages
#[Route('/api/v1/projects', name: 'projects', methods: ['GET'])]
public function getProducts(ProjectRepository $projectRepository): JsonResponse
{

    $productList = $projectRepository->findAll();

    return $this->json($productList, status: Response::HTTP_OK, [], []);
}
```

Voici notre première route légèrement modifier pour correspondre aux besoins de notre API.

On y à modifier le endpoint, ou on vient préfixer par "api" et par la version afin que l'utilisation de l'API soit maintenable.

On précise également verbe HTTP.

Initialisation de l'API

Format de la réponse

Pour une API REST classique, on retourne généralement des données au format JSON avec un code HTTP adéquat.

```php
return new JsonResponse($productList, status: Response::HTTP_OK, [], json: true);
```

Ou

```php
return $this->json($productList, status: Response::HTTP_OK, [], ['groups' => 'getProject']);
```

```php
no usages
#[Route('/api/v1/projects', name: 'projects', methods: ['GET'])]
public function getProducts(ProjectRepository $projectRepository, SerializerInterface $serializer): JsonResponse
{
    $projectList = $projectRepository->findAll();

    $jsonProjects = $serializer->serialize($projectList, format: 'json', ['groups' => 'getProject']);

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

En général, on préférera utiliser la méthode "**json()**", car elle permet de sérializer notre résultat automatique.

Mais uniquement si on l'a installer au préalable :

composer require symfony/serializer-pack

Initialisation de l'API

Les Groups

Pensez à utiliser les groupe de sérialisation afin de filtrer les données que va renvoyer votre API.

Cela va également vous permettre d'avoir des **erreurs de références circulaires** sur vos relations.

Voir page 41

Gérer les exceptions

Pour le moment, si vous tester l'API avec Postman ou Insomnia, vous pouvez facilement vous rendre compte que en cas d'erreur, l'API vous retourne une page HTML.

On va modifier ce comportement pour que les erreurs soit également renvoyer au format JSON. Pour ce faire nous avons besoin de créer un ***"suscriber"*** :

php bin/console make:subscriber Ou *php bin/console make:listener*

Un subscriber dans Symfony est une classe plug-and-play qui intercepte certains événements du cycle de vie (requête, réponse, sécurité, Doctrine, etc.).

```
Suggested Events:
* console.command (Symfony\Component\Console\Event\ConsoleCommandEvent)
* console.error (Symfony\Component\Console\Event\ConsoleErrorEvent)
* console.signal (Symfony\Component\Console\Event\ConsoleSignalEvent)
* 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)
```

Initialisation de l'API

Gérer les exceptions

Maintenant que votre subscriber (listener) est créé, vous allez le retrouver dans :
*src/EventSubscriber/**ExceptionSubscriber**.php*

Ici nous avons complété la fonction **onKernelException** afin de renvoyer une JsonResponse en cas d'erreur.

C'est ici que l'on pourrait intégrer toutes logique en cas d'erreur, enregistrer l'exceptions en BDD, envoyer un rapport avec la stackTrace de l'erreur au responsable du projet ...

```
no usages
public function onKernelException(ExceptionEvent $event): void
{
    $exception = $event->getThrowable();

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

    } else {
        $data = [
            'status' => 500,
            'message' => $exception->getMessage()
        ];

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

Initialisation de l'API

```
no usages
#[Route('/api/project/{id}',
    name: 'project',
    requirements: ['id' => Requirement::DIGITS],
    methods: ['GET']
)]
public function getProduct(int $id, ProjectRepository $projectRepository): JsonResponse {

    $project = $projectRepository->find($id);
    if ($project) {
        return $this->json($project, status: Response::HTTP_OK, [], ['groups' => 'getProject']);
    }
    return new JsonResponse( data: null, status: Response::HTTP_NOT_FOUND);
}
```

Récupération simple d'un élément

Voici un exemple classique de route permettant de récupérer un Objet en passant par un id contenu dans la route.

Nous avons juste ajouter le paramètre "requirements" qui permet vérifier le format des paramètre de la route grâce à une REGEX.

Le "Param Converter"

Il y a une option dans symfony qui permet de directement "caster" l'objet souhaité en convertissant notre $**id** en **$project**.

Cela simplifie encore plus notre code.

```
no usages
#[Route('/api/project/{id}',
    name: 'project',
    requirements: ['id' => Requirement::DIGITS],
    methods: ['GET']
)]
public function getProduct(Project $project, ProjectRepository $projectRepository): JsonResponse {
        return $this->json($project, status: Response::HTTP_OK, [], ['groups' => 'getProject']);
}
```

Initialisation de l'API

La priorité des routes

Lorsque l'on commence à mettre des paramètre, il est pas rare que le nommage de nos routes commence à se confondre.

Heureusement depuis Symfony 5, il est possible de définir un ordre de passage prioritaire dans nos route, avant il fallait les écrire dans le bonne ordres.

Pour ce faire, il faut préciser la priorité comme ci-contre:

```
no usages
#[Route('/project/{id}', name: 'detail', methods: ['GET'])]
public function demoV1()
{
    // Logique métier ...
}

no usages
#[Route('/project/list', name: 'list', methods: ['GET'], priority: 20)]
public function demoV2()
{
    // Logique métier différente ...
}
```

Lien vers la doc

Initialisation de l'API

Utilisation des verbes HTTP:

Pensez à bien utilisez les bon verbes HTTP en fonction du comportement attendu par vos route.

Cela contribue à respecter les bonnes pratique des API REST.

Exemple de suppression avec le verbe DELETE

```php
no usages
#[Route('/api/project/{id}', name: 'deleteProject', methods: ['DELETE'])]
public function deleteProject(Project $project, EntityManagerInterface $em): JsonResponse
{
    $em->remove($project);
    $em->flush();

    return $this->json(['status' => 'success'], status: Response::HTTP_OK);
}
```

Verbe	Description
GET	Accède à une ressource (liste ou élément) ou appelle une fonction
POST	Ajoute une ressource ou exécute une action
PUT	Met à jour une ressource complète en la remplaçant par une nouvelle version (nécessite de renseigner toutes les propriétés de la ressource)
PATCH	Met à jour une partie d'une ressource en envoyant le différentiel (nécessite de renseigner uniquement les propriétés à modifier)
DELETE	Supprime une ressource

Initialisation de l'API

```
no usages
#[Route('/api/v1/project', name:"createProject", methods: ['POST'])]
public function createProject(
    Request $request,
    SerializerInterface $serializer,
    EntityManagerInterface $em,
    UrlGeneratorInterface $urlGenerator
): JsonResponse
{
    $project = $serializer->deserialize($request->getContent(), type: Project::class, format: 'json');
    $em->persist($project);
    $em->flush();

    $location = $urlGenerator->generate(
        name: 'project',
        ['id' => $project->getId()],
        referenceType: UrlGeneratorInterface::ABSOLUTE_URL
    );

    return $this->json(
        $project, status: Response::HTTP_CREATED,
        ["Location" => $location], ['groups' => 'getProject']
    );
}
```

Création d'un nouvel élément

Pour l'enregistrement, ici on part du principe que la requête contient dans son "*body*" contient un json avec les même propriétés que celle de la classe qu'on enregistrer.

Le **serializer** lit les clés/valeurs JSON, regarde les propriétés de la classe Project, et tente de les faire correspondre.

À l'aide de l'interface URLGenerator, je viens créer et renseigner dans le header de ma réponse l'url pour accéder à la donner que l'on vient de créer.

Initialisation de l'API

```
no usages
#[Route('/api/v1/project/{id}', name:"updateProject", methods:['PUT'])]
public function updateProject(
    Request $request, SerializerInterface $serializer, Project $currentProject,
    EntityManagerInterface $em, UrlGeneratorInterface $urlGenerator): JsonResponse
{
    $updatedProject = $serializer->deserialize($request->getContent(),
        type: Project::class,
        format: 'json',
        [AbstractNormalizer::OBJECT_TO_POPULATE => $currentProject]);

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

    $location = $urlGenerator->generate(
        name: 'project', ['id' => $updatedProject->getId()],
        referenceType: UrlGeneratorInterface::ABSOLUTE_URL
    );

    return $this->json(['status' => 'success'], status: Response::HTTP_OK, ["Location" => $location]);
}
```

Modification d'un élément existant

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 ***$currentProject*** , qui correspond au projet passé dans l'URL.

Initialisation de l'API

La validations des données avec les Asserts

Les ***Asserts*** sont des vérifications effectuées pour s'assurer que certaines conditions sont remplies dans le code.

Dans Symfony, elles sont utilisées pour vérifier les entrées de données et garantir que les valeurs attendues sont présentes et correctes.

Les assertions peuvent être utilisées pour vérifier la validité des champs de formulaire, des variables, des objets, etc.

Si l'assert échoue, une exception est levée et un message d'erreur est affiché.

Voici la commande pour les intégrés à votre projet :

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

```php
#[ORM\Column(length: 255)]
#[Assert\Length(
    min: 5,
    max: 50,
    minMessage: 'Le champ nom ne dois pas être inférieur à {{ limit }} caractères',
    maxMessage: 'Le champ nom ne dois pas dépassé {{ limit }} caractères',
)]
private ?string $name = null;
```

On va venir compléter nos fonction de création et d'édition avant de persist & flush pour vérifier que les propriétés de l'objet qu'on s'apprête à enregistrer respecte les règles que nous avons définis:

```php
$errors = $validator->validate($project);

if ($errors->count() > 0) {
    return $this->json($errors, status: Response::HTTP_BAD_REQUEST);
}
```

Lien vers la doc

Initialisation de l'API

Authentification de Lexik JWT AuthBundle:

La toute première étape est d'installer le composant Security .

composer require security

Afin de pouvoir gérer l'authentification en JWT nous allons avoir besoin du paquet suivant

composer require lexik/jwt-authentication-bundle

Ensuite il va vous falloir générer les clés privée & public avec la commande suivante:

php bin/console lexik:jwt:generate-keypair

Ensuite nous allons créer notre classe "**User**" :

php bin/console make:user

NB : N'oublier d'effectuer la migration pour la nouvelle classe "**User**"

config/packages/security.yaml

```yaml
security:
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'

    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        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: ~

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

Initialisation de l'API

config/routes.yaml

```yaml
controllers:
    resource:
        path: ../src/Controller/
        namespace: App\Controller
    type: attribute

api_login_check:
    path: /api/login_check
```

Configuration de l'authentification:

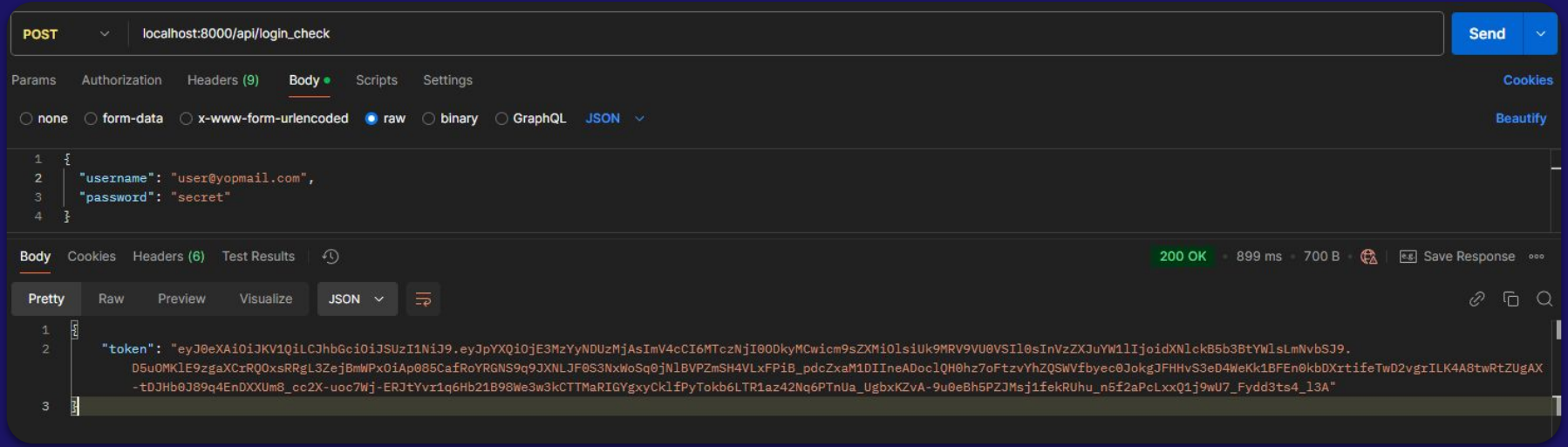
Une fois que vous avez terminé de compléter "service.yaml" et "routes.yaml".

Vous pouvez créer votre première utilisateur soit en dur soit à l'aide de Fixtures pour valider via Postman ou Insomnia que vous arrivez à récupérer un JWT Token. La commande suivante vous permettra de générer un mot de passe Hashé:

```
php bin/console security:hash-password
```

Initialisation de l'API

Résultat attendu sur POSTMAN



Initialisation de l’API

Gestion des droits

Maintenant que nous avons un système d’authentification fonctionnelle, nous allons nous pencher sur comment restreindre l’accès de nos route à certains utilisateurs ayant un profile/ rôle spécifique. Et pour ce faire nous allons utiliser la méthode “isGranted” à savoir que cette méthode peut directement être implémenté dans le controlleur de Symfony 5.2.

```php
no usages
#[Route('/api/product/{id}', name: 'deleteProduct', methods: ['DELETE'])]
public function deleteProduct(Product $product, EntityManagerInterface $em): JsonResponse
{
    if (!$this->isGranted( attribute: 'ROLE_ADMIN')){
        return new JsonResponse(['status' => 'Forbidden'], status: Response::HTTP_FORBIDDEN);
    }
    $em->remove($product);
    $em->flush();

    return new JsonResponse(['status' => 'success'], status: Response::HTTP_OK);
}
```

```php
no usages
#[Route('/api/product/{id}', name: 'deleteProduct', methods: ['DELETE'])]
#[IsGranted('ROLE_ADMIN', message: 'Vous n\'êtes pas autorisé à supprimer cet élément')]
public function deleteProduct(Product $product, EntityManagerInterface $em): JsonResponse
{

    $em->remove($product);
    $em->flush();

    return new JsonResponse(['status' => 'success'], status: Response::HTTP_OK);
}
```

Lien vers la doc

Initialisation de l'API

Optimiser et mettre en place de la pagination

Lorsqu'on traite des listes volumineuses, il est inefficace et peu ergonomique de tout renvoyer en une seule fois.
La pagination permet de récupérer les données en petits morceaux, par exemple 10, 20 ou 50 éléments à la fois.

Voici les avantages:

- **Limite la charge du serveur**
- **Réduit le temps de réponse**
- **Limite également la taille des réponses renvoyés**

ProjectRepository.php

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

ProjectController.php

```php
no usages
#[Route('/api/v1/projects', name: 'projects', methods: ['GET'])]
public function getProducts(ProjectRepository $projectRepository, Request $request): JsonResponse
{
    $page = $request->get( key: 'page', default: 1);
    $limit = $request->get( key: 'limit', default: 3);

    $productList = $projectRepository->findAllWithPagination($page, $limit);

    return $this->json($productList, status: Response::HTTP_OK, [], ['groups' => 'getProject']);
}
```

Initialisation de l'API

```php
no usages
#[Route('/api/v1/projects', name: 'projects', methods: ['GET'])]
public function getProjects(ProjectRepository $projectRepository, Request $request,
                            TagAwareCacheInterface $cachePool): JsonResponse
{
    $page = $request->get( key: 'page', default: 1);
    $limit = $request->get( key: 'limit', default: 3);

    $cacheIdentifier = "getAllProjects-" . $page . "-" . $limit;
    $projectList = $cachePool->get($cacheIdentifier,
        function (ItemInterface $item) use ($projectRepository, $page, $limit) {
            $item->tag( tags: "projectCache");
            return $projectRepository->findAllWithPagination($page, $limit);
        }
    );

    return $this->json($projectList, status: Response::HTTP_OK, [], ['groups' => 'getProject']);
}
```

Gestion du cache

Ici dans un premier temps, on détermine une clé de cache puis on demande ensuite au **TagAwareCacheInterface** ($cachePool) de récupérer l'éventuel contenu déjà mis en cache pour cette clé.

Si rien n'a été mis en cache sous cette clé, alors on passe dans la CallBack pour récupérer les données et on les enregistre dans le cache au passage sous cet identifiant et avec un tag spécifique.

Pour tester, n'oublier de vider votre cache :

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

Initialisation de l'API

Gestion du cache

Néanmoins, avec l'utilisation du cache, il faut être vigilant, car désormais si je fais le même appel deux fois d'affilés et que les données ont été modifiés, je ne verrais pas les modifications.

Pour corriger ce problème, il faut penser à supprimer le cache en nous servant des tag à tous les endroits ou nous interagissons avec notre entités, soit la suppression, l'édition, la création et d'autres endroits selon le contexte.

```
no usages
#[Route('/api/project/{id}', name: 'deleteProject', methods: ['DELETE'])]
    public function deleteProject(Project $project, EntityManagerInterface $em,
                                  TagAwareCacheInterface $cachePool): JsonResponse
    {
        $cachePool->invalidateTags(["projectCache"]);
        $em->remove($project);
        $em->flush();

        return $this->json(['status' => 'success'], status: Response::HTTP_OK);
    }
```

Initialisation de l'API

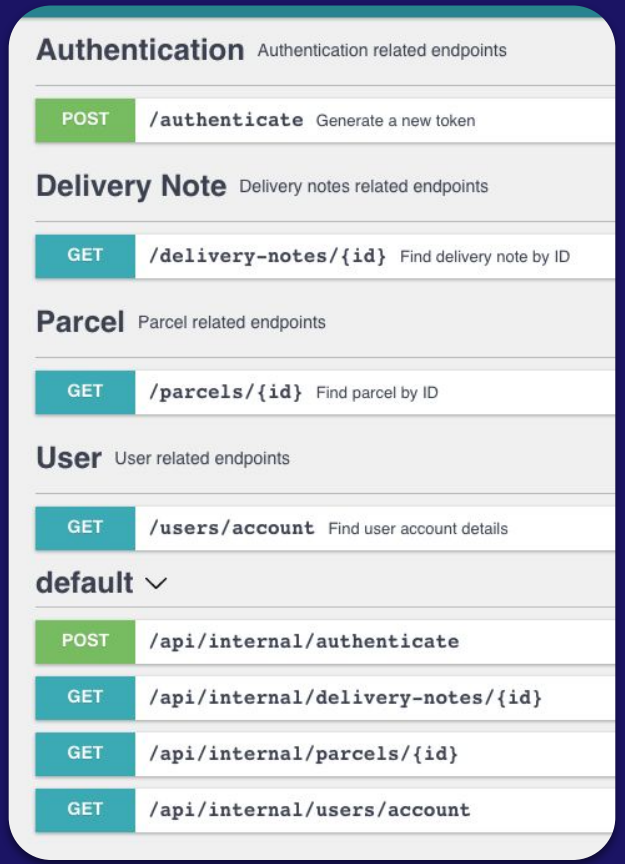
Nelmio

C'est un bundle pour Symfony qui permet de faciliter la création, la gestion et la documentation des API. Il permet d'intégrer un Swagger et est souvent utilisé pour améliorer l'expérience des développeurs lors de la construction d'API RESTful ou GraphQL, tout en fournissant une documentation automatique et des outils pour faciliter l'authentification, la validation et la gestion des erreurs.

Vous pouvez l'intégrer dans votre projet avec la commande suivante :

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

Lien vers la doc



TP: Créer une API Symfony

Objectif :

Cet exercice a pour but de vous familiariser avec la mise en place d'une API RESTful avec Symfony.

Consignes:

Vous devez développer une API pour une application permettant de récupérer des informations sur des jeux vidéos.

Vos entitées seront :

- **VideoGame** (title, releaseDate, description).
- **Category** (name).
- **Editor** (name, country).

Il faudra mettre en place

- Des fichiers de migrations
- Des CRUD pour chacun des entités avec gestion des FK.
    - Avec vérification du format des données (Asserts)
- De l'authentification, seul un ADMIN peut éditer, créer ou supprimer
- Un CRUD pour la table User
- Générer un jeu d'essai pertinent à l'aide des DataFixtures
- Gérer les exceptions
- De la pagination avec gestion du cache
- Documenter votre API à l'aide Postman ou d'un Swagger (Nelmio)