# Programmation avancée & Symfony

BUT Informatique - 3ème année (*R5A.05*)

# TP 7

## *API et Logs*



Version : **01/12/2025 1432**

## Objectifs du TP 7

PROJET SYMFONY

- Comprendre ce qu'est une **API**, son utilité
- Savoir développer les webservices d'une API REST (en JSON)
- Comprendre les mécanismes de **gestion des logs**
- Installer et utiliser **`Monolog`**
- Mettre en place la **rotation des logs**

# Sommaire

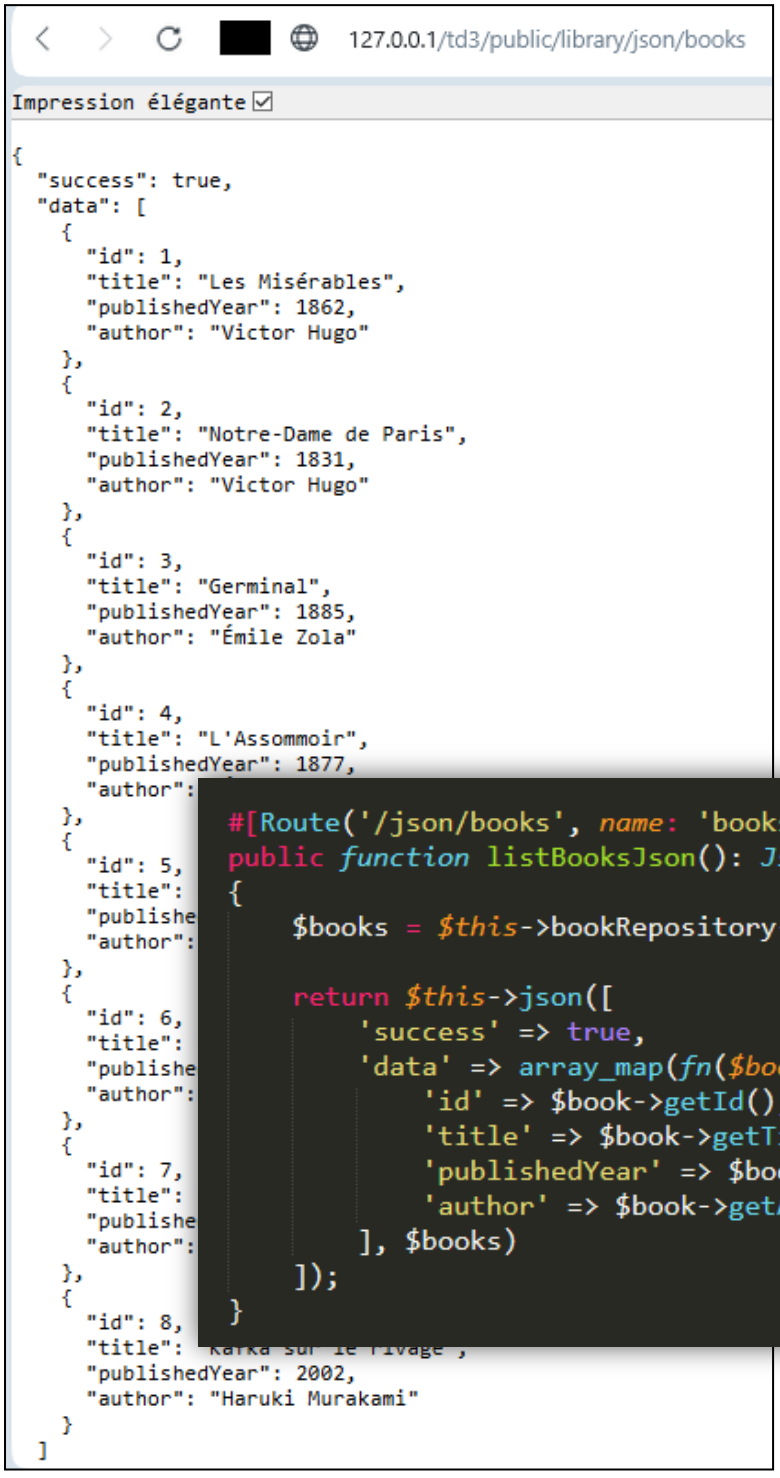
# 1. API

Durant le **TD 4**, nous avons développé des méthodes de contrôleur renvoyant du JSON.

Exemple 1 :

Lister tous les livres en **JSON**

Route **:** /library**/json/books**
Méthode **:** public function **listBooksJson**(): **JsonResponse**




```php
#[Route('/json/books', name: 'books_list_json', methods: ['GET'])]
public function listBooksJson(): JsonResponse
{
    $books = $this->bookRepository->findAll();

    return $this->json([
        'success' => true,
        'data' => array_map(fn($book) => [
            'id' => $book->getId(),
            'title' => $book->getTitle(),
            'publishedYear' => $book->getPublishedYear(),
            'author' => $book->getAuthor()?->getFullName()
        ], $books)
    ]);
}
```

*Correction TD4 - Lister tous les livres en **Json***

Exemple 2 :

Lister tous les auteurs et leurs livres en **JSON**

Route **:** /library**/json/authors**
Méthode **:** public function **listAuthorsJson**(): **JsonResponse**


*Correction TD4 - Lister tous les auteurs et leurs livres en Json*

```php
/**
 * Liste tous les auteurs et leurs livres, en JSON
 * @return JsonResponse
 */
#[Route('/json/authors/', name: 'authors_list_json', methods: ['GET'])]
public function authors(): JsonResponse
{
    $authors = $this->authorRepository->findAuthorsWithBooks();

    return $this->json([
        'success' => true,
        'count' => count($authors),
        'data' => array_map(fn($author) => [
            'id' => $author['id'],
            'fullName' => $author['firstName'].' '.$author['lastName'],
            'birthdate' => $author['birthDate']->format('Y-m-d'),
            'nationality' => $author['nationality'],
            'books' => $author['books']
        ], $authors)
    ]);
}
```

**Renvoyer des réponses JSON => manière de développer les services d'une API REST.**

## 1.1 📖 Qu'est-ce qu'une API

### API

Une **API** (***Application Programming Interface***) est une **interface** qui permet à deux composants informatiques de **communiquer** entre eux. C'est un ensemble de règles et de protocoles qui définit comment ces composants de SI peuvent interagir.

Exemples :

- ➔ Quand vous utilisez une application météo sur votre téléphone, elle interroge l'API d'un service météorologique pour récupérer les données.
- ➔ Les boutons de type "Se connecter avec Google" utilisent l'API de Google pour authentifier les utilisateurs
- ➔ Une application de voyage comparant des prix utilise les API de différentes compagnies, (par exemple compagnies aériennes) afin de récupérer leurs données

Généralement, on distingue donc le **fournisseur** de l'API, **exposant** son API, et le **consommateur** de l'API (**client** utilisateur de l'API).

📖 **Wikipédia > Interface de programmation :** https://fr.wikipedia.org/wiki/Interface_de_programmation

### API REST

Une **API REST** (***Representational State Transfer***) est une interface de programmation qui permet à différentes applications de communiquer entre elles, d'échanger des données, **via Internet en utilisant le protocole HTTP (ou HTTPS)**.

**Principes clés**

- ➢ **Architecture client-serveur** : Le client (par exemple, une application mobile) effectue des requêtes au serveur (fournisseur) qui lui renvoie des données, **généralement au format JSON**.
  Supporte plusieurs formats : **JSON (le plus courant)**, XML, HTML ou texte brut
  (à la différence des *API SOAP* : uniquement XML avec une structure d'enveloppe obligatoire)
- ➢ **Sans état (stateless)** : Chaque requête est indépendante et contient toutes les informations nécessaires. Le serveur ne conserve pas d'informations entre deux requêtes.
- ➢ **Ressources identifiées par URL** : Chaque élément (utilisateur, produit, article) est accessible via une URL unique, appelée **endpoint**. Par exemple :
  *https://api.exemple.com/users/123 pour les infos de l'utilisateur n°123*
  *https://api.exemple.com/products pour la liste des produits*

- ➢ **Méthodes HTTP**

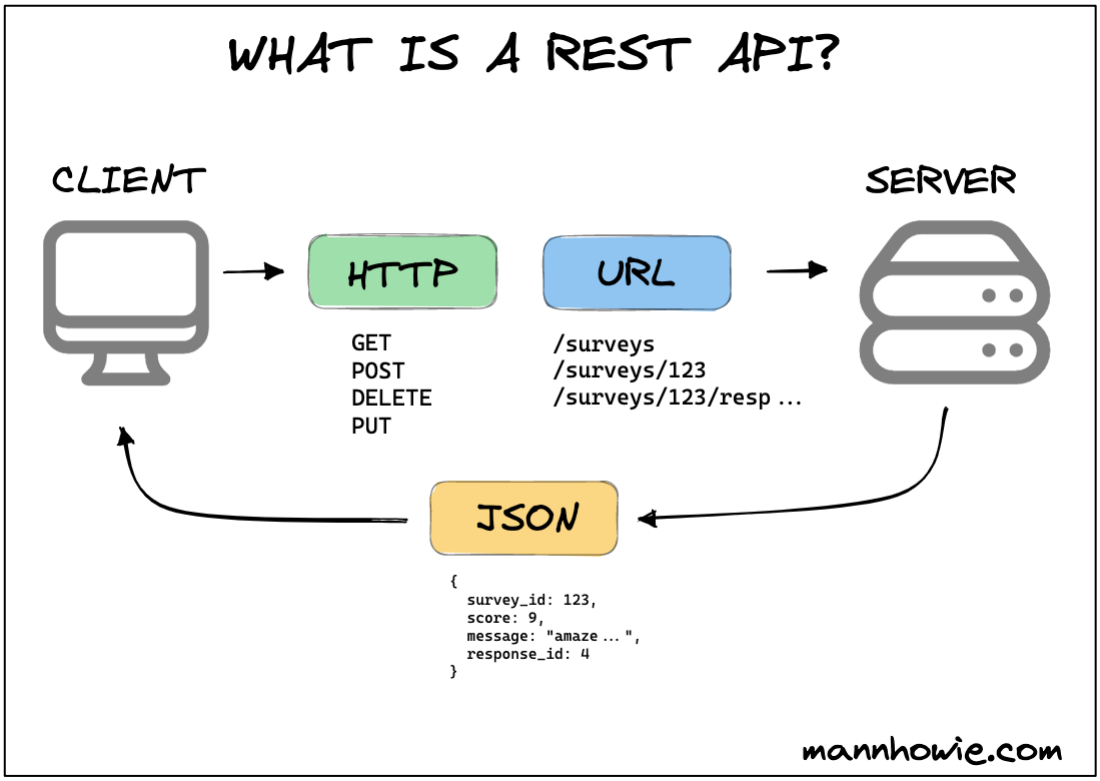
REST utilise les méthodes HTTP standards pour effectuer différentes actions :

- **GET** : Récupérer des données (lecture)
- **POST** : Créer une nouvelle ressource
- **PUT/PATCH** : Modifier une ressource existante
- **DELETE** : Supprimer une ressource

**Exemple concret :**

Imaginons une API REST pour gérer des livres :

- GET /books → Liste tous les livres
- GET /books/5 → Récupère le livre n°5
- POST /books → Ajoute un nouveau livre
- PUT /books/5 → Modifie le livre n°5
- DELETE /books/5 → Supprime le livre n°5




*https://mannhowie.com/rest-api*

**Un ou plusieurs contrôleurs d'une application Symfony peuvent renvoyer des réponses JSON (après requêtes *GET*) ou accepter des données (par ex. après requêtes *POST*), ce qui permet de proposer une API.**

## 1.2 Développer l'API de son appli.

Le but de la première partie de ce TP est de développer l'API (un ensemble de webservices) de votre application.

### Controller

Commencez par créer un nouveau contrôleur, **`ApiController`**, qui contiendra toutes les méthodes liées à votre API.

### Vues

Il n'y a pas de vues à développer. Vos méthodes doivent renvoyer du JSON valide et uniquement cela. Tout élément de vue (templates ou morceaux de templates) est donc à

proscrire.

## Routes

Chaque webservice devra être lié à une route (url) de la forme :

**…./api/<route_du_ webservice >**/<parametres_eventuels>

Le « morceau de route » **`api`** est donc à placer « au niveau du contrôleur » :

```php
/**
 * ApiController
 */
#[Route('/api')]

final class ApiController extends AbstractController
{
```

Ensuite chaque méthode aura un ou plusieurs autres « morceaux de routes » (→ comme réalisé dans les autres contrôleurs de notre application).

Les routes de l'API devront ressembler à celles de l'application.
Par exemple, dans votre application, la route **`/mon_projet/jeu_video`** liste les jeux vidéo, et la route **`/mon_projet/api/jeu_video`** listera également les jeux vidéo en JSON

## Méthodes de contrôleur

Au sein de votre contrôleur `ApiController`, développer les méthodes suivantes, renvoyant les données **en JSON** (soit une ***`JsonResponse`***).

| | **Fonctionnalité** | **Route associée** |
|---|---|---|
| 1 | Retourne les données liées aux jeux vidéo<br>*(Cf. image ci-après)* | **…/mon_projet/api/jeu_video** |
| 2 | Retourne les données liées à un jeu vidéo, identifié via son Id | **…/mon_projet/api/jeu_video/<id>** |
| 3 | Retourne, les données liées aux genres | **…/mon_projet/api/genre** |
| 4 | Retourne les données liées à genre, identifié via son id | **…/mon_projet/api/genre/<id>** |
| 5 | Retourne la collection d'un utilisateur, identifié via son id | **… …** |
| 6 | Supprime un genre via son id | **… …** |

→ Il s'agit de services de votre API.

*Conseils* :

- Avant de tester certains actions, telles que les suppressions (que ça soit via l'IHM ou via l'API), sauvegarder vos données, afin de les restaurées plus facilement.
- Vous pouvez utiliser un validateur de code JSON, tel que *JSONLint* : https://jsonlint.com


1 - Retourne les données liées aux jeux vidéo en JSON

## Gestion des erreurs et codes retours

L'appel à une url (route) via votre navigateur est une **requête HTTP**. En retour vous recevez une **Réponse HTTP** (***Cf. Cours Requêtes et réponses http + Schéma Symfony***)

Les **codes retour « 2XX »** sont les résultats des requêtes exécutées avec **succès**.
Le code le plus courant est le code 200. Il en existe cependant d'autres qui répondent à des cas précis.
Lorsque l'appel au webservice de votre API est OK, vous recevez donc une réponse HTTP **200**.

Cependant, **vous devez gérer les autres cas**, tel que les appels à une url n'existant pas ou bien étant erroné (par exemple : passage de paramètres erronés).
Dans ces cas-là, le code HTTP approprié doit être renvoyé, accompagné, si nécessaire d'un message d'erreur.

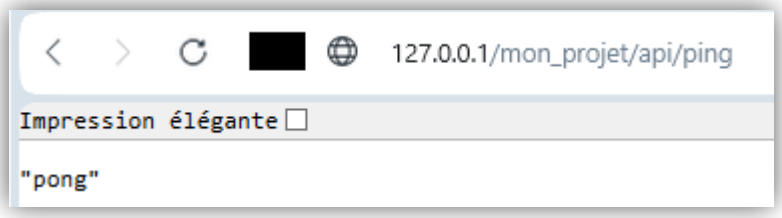
📖 **Wikipédia > Liste des codes http :** https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

## Routes d'évaluation de la santé d'une API

Afin de pouvoir s'assurer qu'une API est disponible et opérationnelle, il est recommandé que celle-ci expose des routes dédiées, telles que les routes **`/ping`** et **`/healthcheck`**

Ces webservices permettent au consommateur et au producteur de l'API de connaître l'état de santé de de celle-ci.

| | Fonctionnalité | Route associée | |
|---|---|---|---|
| 7 | Ping : Renvoi une **Réponse HTTP 200** + le texte *pong* (si tout va bien) | **…/mon_projet/api/ping** | GET |
| 8 | Healthcheck : Fourni (en JSON) des informations donnant **l'état de santé de l'api et de ses dépendances (BDD, …)**<br>Renvoie les résultats des vérifications.<br><br>Ce endpoint peut donner des informations diverses, telles que :<br>• [Si l'API utilise une/des BDD] L'API est capable d'accéder à la base de données<br>• [Si l'API est consommatrice d'autres API] L'API peut interagir avec toutes les APIs tierces utilisée (via un appel à leurs routes *ping* respectives)<br>• L'API peut interagir avec les services d'authentification sur lesquels elle se base<br>• L'API arrive bien à accéder aux systèmes d'édition dont elle a besoin.<br>• L'API arrive bien à accéder aux systèmes de fichiers du serveur dont elle a besoin<br>• …<br>Il est donc utile pour surveiller la santé d'une API à l'aide d'un outil de surveillance et/ou manuellement. | **…/mon_projet/api/healthcheck** | GET |




7 – Ping Pong

# 2. Logs

## 2.1 📖 La gestion des Logs dans Symfony

Le ***logging*** (enregistrement, gestion des logs) est le process de collecte et de gestion des **événements** relatifs à l'état d'un système. Il existe une multitude de logs et de méthodes de gestion des logs, variant en fonction des différents systèmes.

Les *logs* (« journaux ») fournissent de la visibilité sur le comportement d'une application. Dans le cas des applications basées sur un serveur, ils sont **généralement écrits dans un ou des fichiers sur le disque** ( « fichier journal ») ; mais d'autres formats de sortie sont possibles.

Symfony dispose de deux loggers **PSR-3** minimalistes :

- **`Logger`** pour le contexte HTTP (le web)
- **`ConsoleLogger`** pour le contexte CLI (l'environnement d'exécution en ligne de commande, par opposition au contexte web / HTTP).

📖 **PSR-3: Logger Interface :** https://www.php-fig.org/psr/psr-3/

Les loggers Symfony envoient des messages, à partir du niveau WARNING, vers **stderr (flux de sortie standard des erreurs)**.

Les *log levels* (niveaux de log) courants et inclus dans la PSR-3 sont :

- DEBUG
- INFO
- NOTICE
- WARNING
- ERROR
- CRITICAL
- ALERT
- EMERGENCY

Donc les messages de type DEBUG, INFO et NOTICE ne seront pas enregistrés par défaut par Symfony, alors que ceux de type WARNING, ERROR, CRITICAL.... le seront.

Le niveau de logs peut être modifié en définissant la **variable d'environnement SHELL_VERBOSITY** :

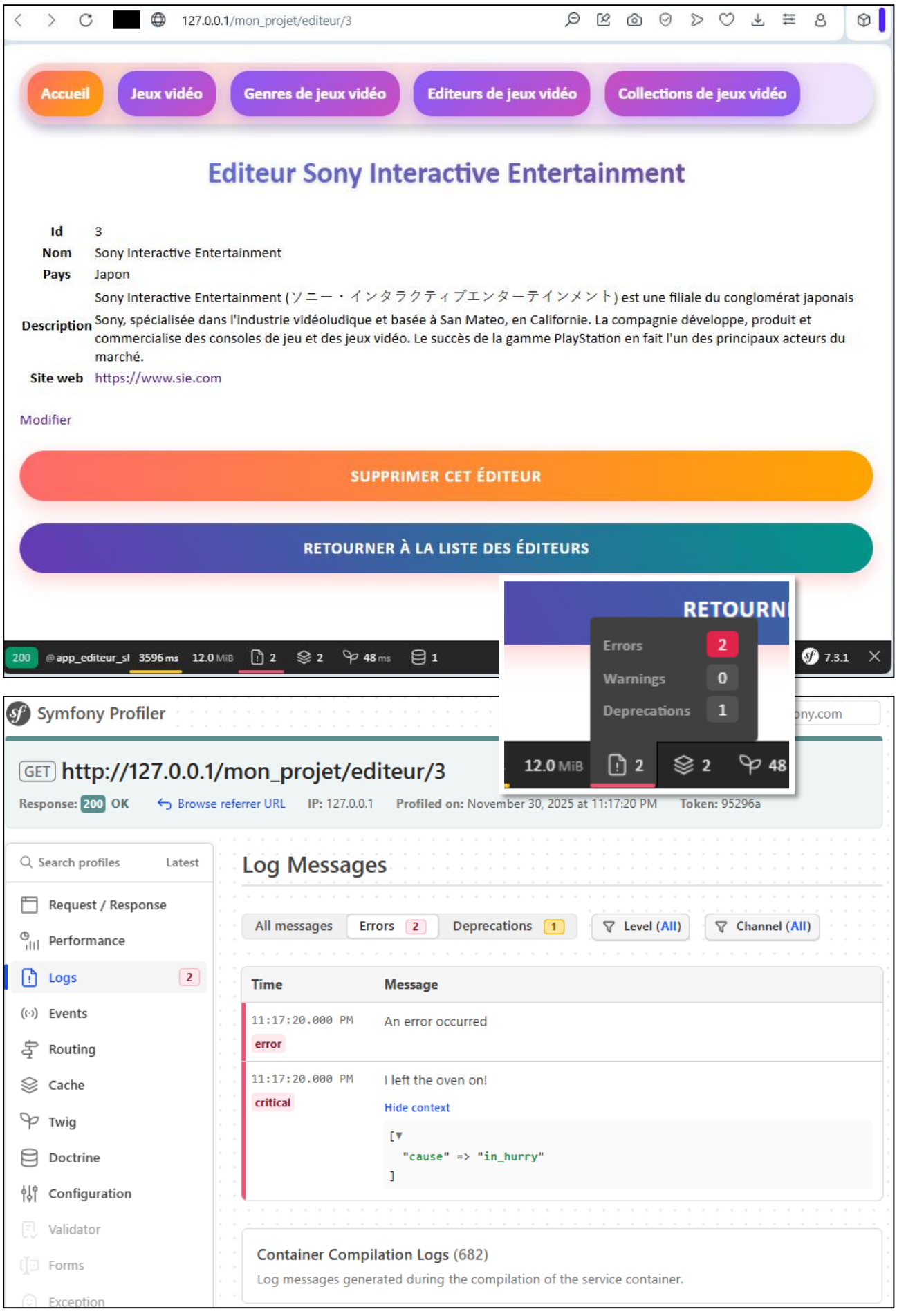
| Valeur de SHELL_VERBOSITÉ | Niveau minimum de logs |
|---|---|
| -1 | ERROR |
| 1 | NOTICE |
| 2 | INFO |
| 3 | DEBUG |

Le niveau de log, la sortie par défaut (*stderr*, …) et le format des logs peuvent également être modifiés en passant les arguments appropriés au constructeur de `Logger` ou `ConsoleLogger`.

## Logger (journaliser) un message

Pour enregistrer un message, **injectez le `Logger` par défaut** dans votre contrôleur ou service, soit, par exemple :

```php
use Symfony\Component\Routing\Attribute\Route;

use Psr\Log\LoggerInterface;

#[Route('/editeur')]
final class EditeurController extends AbstractController
{
    #[Route(name: 'app_editeur_index', methods: ['GET'])]
    public function index(EditeurRepository $editeurRepository): Response
    { ...
    }

    #[Route('/new', name: 'app_editeur_new', methods: ['GET', 'POST'])]
    public function new(Request $request, EntityManagerInterface $entityManager): Response
    { ...
    }

    #[Route('/{id}', name: 'app_editeur_show', methods: ['GET'])]
    public function show(Editeur $editeur, LoggerInterface $logger): Response
    {
        // Exemple de log INFO
        $logger->info('I just got the logger');
        // Exemple de log ERROR
        $logger->error('An error occurred');

        // les logs  peuvent contenir des noms de variables encadrées par des accolades
        // dont les valeurs sont passées comme second argument
        $logger->debug('Consultation de l\'éditeur n°{editeurId}', [
            'editeurId' => $editeur->getId(),
        ]);

        $logger->critical('I left the oven on!', [
            // Il est possible d'inclure des elements de contexte dans les logs
            'cause' => 'in_hurry',
        ]);

        return $this->render('editeur/show.html.twig', [
            'editeur' => $editeur,
        ]);
    }
```




Le *Profiler* permet de consulter les messages de logs

Le `Logger` possède différentes méthodes pour les différents niveaux/priorités de logs.
📖 **Voir *LoggerInterface* pour la liste de toutes ces méthodes :** https://github.com/php-fig/log/blob/master/src/LoggerInterface.php

### Monolog

Symfony s'intègre parfaitement avec **`Monolog`**, bibliothèque de gestions des logs PHP, permettant de créer et stocker des messages de logs dans une variété d'endroits différents.

En utilisant *Monolog*, vous pouvez configurer le logger pour faire différentes choses / des choses différentes en fonction du niveau d'un message.
Par exemple, vous pouvez décider que lorsque l'on a un message de log de niveau ERROR, il faut envoyer celui-ci par mail à tel destinataire.

Cmde d'installation de *Monolog* : **`composer require symfony/monolog-bundle`**

### Place des logs

Par défaut, les entrées loguées sont écrites dans le fichier `var/log/dev.log` lorsque l'on est en environnement de développement. Et les logs sont écrits sur le flux **stderr** lorsque l'on est en environnement de production,

Cela dit, il est possible de modifier ces comportements par défaut via de la **configuration**.
Par exemple, si vous préférez stocker les logs de production dans un fichier, il conviendra de définir ce ***path*** (exemple : à la valeur : `var/log/prod.log`).

## 2.1 Gérer les logs de son appli.

Vous devez mettre en place un système de logs au sein de votre application, respectant les préconisations suivantes :

- Utilisation de **`Monolog`**.
- Chaque **action** (chaque appel à la méthode d'un contrôleur) doit être loguées.
- Une action donne lieu à une entrée, une ligne dans le fichier des logs
- Les entrées doivent être **horodatées** (date et heure enregistrées)
- Les entrées doivent préciser le niveau de log (DEBUG, WARNING, ERROR …)
- Les appels aux *endpoints* (routes) de votre **API** doivent également être logués.

- **Rotation des logs** mise en place avec **`Monolog`** (*`rotating_file handler`*)
  - Création d'un nouveau fichier de logs chaque jour
  - Nommage des fichiers de logs avec la date (par exemple `dev-2024-01-15.log`
  - Conservation des 30 derniers fichiers uniquement, et
  - Suppression automatique des anciens fichiers

📖 **Ressources documentaires :**

- Symfony > Logging : https://symfony.com/doc/current/logging.html
- How to Get Started with Monolog Logging in PHP - Eric Hu - Updated on November 23, 2023 : https://betterstack.com/community/guides/logging/how-to-start-logging-with-monolog/
- Mon-code.net > Pense bête: activer la rotation de logs de Symfony dans la configuration de Monolog :https://www.mon-code.net/article/68/pense-bete-activer-la-rotation-de-logs-de-symfony-dans-la-configuration-de-monolog