Création d'un un kit de développement logiciel (*Software development kit* ou SDK) pour l'API

Qu'est-ce qu'un SDK et pourquoi est-il important ?

Un **Software Development Kit** (SDK) est un ensemble d'outils, de bibliothèques, de documentation et d'exemples de code qui permettent aux développeurs de facilement intégrer, étendre ou interagir avec une application, un service ou une API. Dans le contexte de notre projet, le SDK sera un package Python qui fournira une interface simple et intuitive pour interagir avec notre API MovieLens.

Les bénéfices de la création d'un SDK pour l'API sont nombreux :

- **Faciliter l'intégration** : Les utilisateurs n'ont pas besoin de comprendre les détails techniques de l'API, comme l'envoi de requêtes HTTP ou la gestion des réponses. Le SDK simplifie ces étapes.
- **Accélérer le développement** : En fournissant des fonctions prédéfinies pour effectuer des actions courantes, le SDK permet aux utilisateurs de gagner du temps.
- **Assurer la cohérence** : Un SDK bien conçu garantit que tous les utilisateurs interagiront avec l'API de manière uniforme et cohérente.
- **Support de la communauté** : En partageant un SDK via PyPI, il devient accessible à d'autres développeurs et analystes de données qui pourraient l'utiliser dans leurs projets.

Pourquoi Python pour le SDK ?

Le choix de **Python** pour développer le SDK est particulièrement adapté au contexte de ce projet, car la plupart des utilisateurs finaux de l'API seront des **data analysts** et **data scientists**, des professionnels qui utilisent principalement Python dans leur travail quotidien. Python est largement adopté dans le domaine de l'analyse de données, grâce à sa simplicité et à l'écosystème riche de bibliothèques pour le traitement des données, telles que **pandas**, **NumPy**, **scikit-learn**, etc.

En publiant ce SDK sous forme de package Python sur **PyPI** (Python Package Index), nous permettons aux utilisateurs de facilement l'installer avec une simple commande `pip install`. Cela rend l'intégration de l'API dans leurs projets Python rapide et transparente, tout en maximisant la compatibilité avec leurs outils existants.

En résumé, un SDK bien conçu permet de rendre notre API accessible et facile à utiliser pour les professionnels de l'analyse de données, tout en utilisant un langage et des outils avec lesquels ils sont déjà familiers.

Etapes de création du Package

Structure du dossier du package

Créez les différents fichiers et sous-dossiers du répertoire sdk/

En exécutant la commande ci-dessous pour visualiser la structure du dossier sdk/ :

```
tree --prune -I 'build|*.egg-info|__pycache__'
```

Vous devez avoir une structure pareille pour le dossier sdk/ :

```
.
├── pyproject.toml
├── src
│   └── moviesdk
│       ├── __init__.py
│       ├── movie_client.py
│       ├── movie_config.py
│       └── schemas
│           ├── __init__.py
│           └── schemas.py
└── test_sdk.py
```

Fichier `sdk/pyproject.toml`

Le fichier `pyproject.toml` est un fichier de configuration pour les projets Python modernes. Il est utilisé pour décrire les métadonnées du projet, ses dépendances, et les outils nécessaires à sa construction et à son packaging. Ce fichier remplace progressivement des fichiers comme `setup.py` et `requirements.txt` pour la configuration des projets Python.

```toml
[build-system]
requires = ["setuptools>=78.1.0"]
build-backend = "setuptools.build_meta"

[project]
name = "moviesdk"
version = "0.0.1"
authors = [
  { name="[Josh]" },
]
description = "Software Development Kit (SDK) for MovieLens API"
requires-python = ">=3.12"
classifiers = [
    "Development Status :: 3 - Alpha",
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]
dependencies = [
        'httpx>=0.28.1',
        'pydantic>=2.11.2',
        'backoff>=2.2.1',
        'python-dotenv',
]
```

Explication de ce fichier `pyproject.toml`

- **[build-system]** : Cette section décrit comment le projet doit être construit. Il est nécessaire d'indiquer `setuptools` comme backend de construction et la version minimale requise de `setuptools`.
- **[project]** : C'est ici que se trouvent les métadonnées du projet :
  - `name` : Le nom du projet, dans ce cas "moviesdk".
  - `version` : La version actuelle du SDK.
  - `authors` : Les informations sur les auteurs du projet, incluant potentiellement un email.
  - `description` : Une brève description du SDK.
  - `requires-python` : La version minimale de Python requise pour utiliser le SDK.
  - `classifiers` : Des informations sur le statut du projet (par exemple, si c'est un projet alpha) et des détails techniques sur la compatibilité.
  - `dependencies` : La liste des dépendances Python nécessaires à l'exécution du SDK.

Le fichier `pyproject.toml` est essentiel pour gérer ton projet Python de manière moderne et cohérente, et il est requis si tu souhaites publier ton SDK sur **PyPI**.

Fichier `sdk/src/moviesdk/movie_config.py`

```
import os
from dotenv import load_dotenv

load_dotenv()

class MovieConfig:
    """Classe de configuration contenant des arguments pour le client SDK.

    Contient la configuration de l'URL de base et du backoff progressif.
    """

    movie_base_url: str
    movie_backoff: bool
    movie_backoff_max_time: int

    def __init__(
        self,
        movie_base_url: str = None,
        backoff: bool = True,
        backoff_max_time: int = 30,
    ):
        """Constructeur pour la classe de configuration.

        Contient des valeurs d'initialisation pour écraser les valeurs par
défaut.

        Args:
        movie_base_url (optional):
            L'URL de base à utiliser pour tous les appels d'API.
Transmettez-la ou définissez-la dans une variable d'environnement.
        movie_backoff:
            Un booléen qui détermine si le SDK doit réessayer l'appel en
utilisant un backoff lorsque des erreurs se produisent.
```

```
        movie_backoff_max_time:
            Le nombre maximal de secondes pendant lesquelles le SDK doit
continuer à essayer un appel API avant de s'arrêter.
        """

        self.movie_base_url = movie_base_url or
os.getenv("MOVIE_API_BASE_URL")
        print(f"MOVIE_API_BASE_URL in MovieConfig init:
{self.movie_base_url}")

        if not self.movie_base_url:
            raise ValueError("L'URL de base est requise. Définissez la
variable d'environnement MOVIE_API_BASE_URL.")

        self.movie_backoff = backoff
        self.movie_backoff_max_time = backoff_max_time

    def __str__(self):
        """Fonction Stringify pour renvoyer le contenu de l'objet de
configuration pour la journalisation"""
        return f"{self.movie_base_url} {self.movie_backoff}
{self.movie_backoff_max_time}"
```

Le fichier `movie_config.py` joue un rôle fondamental dans le SDK en centralisant la **configuration de base** nécessaire pour que le client puisse fonctionner correctement. Voici une **revue complète et une explication simplifiée** de ce fichier :

Objectif du fichier `movie_config.py`

Ce fichier définit une classe `MovieConfig` utilisée pour **configurer le SDK** avant de faire des appels à l'API. Il permet de définir :

- l'URL de base de l'API,
- si on souhaite appliquer un mécanisme de **backoff** (réessais automatiques en cas d'échec),
- et combien de temps maximum on doit réessayer.

Il permet aussi de charger la configuration à partir d'un **fichier** `.env`, ce qui est une très bonne pratique pour éviter de stocker des données sensibles ou spécifiques à un environnement directement dans le code.

Détail du contenu

```
from dotenv import load_dotenv
import os

load_dotenv()
```

Charge les variables d'environnement depuis un fichier `.env` (utile pour ne pas écrire les secrets ou URL directement dans le code).

Classe `MovieConfig`

```
class MovieConfig:
    ...
```

Cette classe sert à **centraliser la configuration du SDK**.

Le constructeur `__init__`

Il permet d'instancier l'objet de configuration avec :

- une `movie_base_url` : si elle n'est pas passée, on la récupère depuis une variable d'environnement.
- un `backoff` : booléen qui permet de dire si on veut réessayer automatiquement en cas d'erreur.
- `backoff_max_time` : durée maximale (en secondes) pendant laquelle on veut réessayer.

```
self.movie_base_url = movie_base_url or os.getenv("MOVIE_API_BASE_URL")
```

Cette ligne dit : « Si `movie_base_url` est fourni, je l'utilise. Sinon, je vais chercher dans les variables d'environnement. »

Et si `movie_base_url` est toujours vide, on déclenche une erreur claire :

```
if not self.movie_base_url:
    raise ValueError("L'URL de base est requise...")
```

Le `print()` temporaire peut t'aider au débogage (mais tu pourras le supprimer ou le logger proprement plus tard).

Fonction `__str__`

```
def __str__(self):
    return f"{self.movie_base_url} {self.movie_backoff}
{self.movie_backoff_max_time}"
```

Permet d'afficher la config facilement, par exemple dans un `print(MovieConfig())`.

En résumé

Élément	Rôle
`load_dotenv()`	Charge les variables d'environnement depuis un fichier `.env`
`movie_base_url`	URL de base de l'API (obligatoire)
`movie_backoff`	Active/désactive les tentatives automatiques en cas d'erreur réseau
`movie_backoff_max_time`	Définit la durée maximale du backoff en secondes
`__str__`	Sert à afficher le contenu de la config de manière lisible

Ce fichier est donc la **brique de configuration principale du SDK**.

Fichier `sdk/src/moviesdk/schemas/schemas.py`

Code : C'est le mème code que celui du fichier `api/schemas.py`.

Ce fichier contient les **schémas de données Pydantic** utilisés dans l'API FastAPI.

Ce sont **des classes Python qui définissent la forme et le type des objets JSON échangés avec l'API** : films, notes, tags, etc.

Le fait de les utilise encore dans le SDK ?** **garantit la cohérence entre ce que l'API renvoie et ce que le SDK attend**.

Dans le SDK, ces classes Pydantic servent à :

1. **Valider les réponses JSON de l'API**

Exemple :

```
response = httpx.get(url)
return MovieDetailed(**response.json())
```

➡ Cette ligne transforme le JSON de l'API en un **objet Python fort typé** (`MovieDetailed`)
➡ Si les données ne correspondent pas, Pydantic lève une erreur immédiatement : sécurité.

---

2. **Offrir une expérience Pythonic**

Tu peux faire :

```
movie = client.get_movie(123)
print(movie.title)
print(movie.genres.split("|"))
```

Plutôt que :

```
data = response.json()
print(data["title"])
```

C'est plus sûr, plus lisible, plus robuste.

---

3. **Mutualiser le code (DRY)**

Tu **réutilises les mêmes classes Pydantic que pour l'API**, donc :

- Pas besoin de redéfinir les modèles deux fois.
- Moins de risques d'incohérences.
- Maintenance simplifiée.

Fichier `sdk/src/moviesdk/__init__.py`

Dans un package Python (comme `moviesdk` ici), le fichier `__init__.py` permet de **définir ce qui est exposé lorsque quelqu'un importe le package**.

Concrètement, avec ce contenu :

```
from .movie_client import MovieClient
from .movie_config import MovieConfig
```

Tu permets à l'utilisateur final de faire :

```
from moviesdk import MovieClient, MovieConfig
```

Au lieu de :

```
from moviesdk.movie_client import MovieClient
from moviesdk.movie_config import MovieConfig
```

Pourquoi c'est utile ?

1. **Simplifie l'import pour l'utilisateur final**

Un SDK bien conçu doit être **facile à importer et à utiliser**.

Tu veux que les gens fassent :

```
from moviesdk import MovieClient
```

Pas :

```
from moviesdk.movie_client import MovieClient
```

C'est plus lisible, plus direct.

---

2. **Masque la structure interne**

Avec l'import centralisé dans `__init__.py`, tu **masques les sous-modules internes** à l'utilisateur.
Cela permet de :

- Garder une API publique propre.
- Pouvoir modifier la structure interne du package **sans casser les imports** pour l'utilisateur.

---

3. **Prépare le terrain pour une belle doc**

Quand tu rédigeras la documentation de ton SDK (README, docstring, etc.), tu n'auras qu'à montrer :

```
from moviesdk import MovieClient
client = MovieClient()
```

C'est pro, clair et élégant.

En résumé

Élément	Rôle
`__init__.py`	Déclare les composants "publics" du package
`from .movie_client import ...`	Rends directement accessibles ces classes depuis `moviesdk`
Avantage pour l'utilisateur	Importation simple et intuitive (`from moviesdk import MovieClient`)

Fichier `sdk/src/moviesdk/schemas/__init__.py`

Le fichier `schemas/__init__.py` avec :

```
from .schemas import *
```

a un objectif **simple mais important** : rendre toutes les classes définies dans `schemas.py` **accessibles directement** depuis le sous-module `schemas`.

Fichier `movie_client.py`

```
import httpx
from typing import Optional, List

from .schemas import MovieSimple, MovieDetailed, RatingSimple, TagSimple,
LinkSimple, AnalyticsResponse
from .movie_config import MovieConfig


class MovieClient:
    def __init__(self, config: Optional[MovieConfig] = None):
        self.config = config or MovieConfig()
        self.movie_base_url = self.config.movie_base_url

    def health_check(self) -> dict:
        url = f"{self.movie_base_url}/"
        response = httpx.get(url)
        response.raise_for_status()
        return response.json()

    def get_movie(self, movie_id: int) -> MovieDetailed:
        url = f"{self.movie_base_url}/movies/{movie_id}"
        response = httpx.get(url)
        response.raise_for_status()
        return MovieDetailed(**response.json())

    def list_movies(self, skip: int = 0, limit: int = 100, title:
Optional[str] = None, genre: Optional[str] = None) -> List[MovieSimple]:
        url = f"{self.movie_base_url}/movies"
        params = {"skip": skip, "limit": limit}
        if title:
            params["title"] = title
        if genre:
            params["genre"] = genre
        response = httpx.get(url, params=params)
        response.raise_for_status()
        return [MovieSimple(**movie) for movie in response.json()]

    def get_rating(self, user_id: int, movie_id: int) -> RatingSimple:
        url = f"{self.movie_base_url}/ratings/{user_id}/{movie_id}"
        response = httpx.get(url)
        response.raise_for_status()
        return RatingSimple(**response.json())

    def list_ratings(self, skip: int = 0, limit: int = 100, movie_id:
Optional[int] = None, user_id: Optional[int] = None, min_rating:
Optional[float] = None) -> List[RatingSimple]:
```

```
        url = f"{self.movie_base_url}/ratings"
        params = {"skip": skip, "limit": limit}
        if movie_id:
            params["movie_id"] = movie_id
        if user_id:
            params["user_id"] = user_id
        if min_rating:
            params["min_rating"] = min_rating
        response = httpx.get(url, params=params)
        response.raise_for_status()
        return [RatingSimple(**rating) for rating in response.json()]

    def get_tag(self, user_id: int, movie_id: int, tag_text: str) ->
TagSimple:
        url = f"{self.movie_base_url}/tags/{user_id}/{movie_id}/{tag_text}"
        response = httpx.get(url)
        response.raise_for_status()
        return TagSimple(**response.json())

    def list_tags(self, skip: int = 0, limit: int = 100, movie_id:
Optional[int] = None, user_id: Optional[int] = None) -> List[TagSimple]:
        url = f"{self.movie_base_url}/tags"
        params = {"skip": skip, "limit": limit}
        if movie_id:
            params["movie_id"] = movie_id
        if user_id:
            params["user_id"] = user_id
        response = httpx.get(url, params=params)
        response.raise_for_status()
        return [TagSimple(**tag) for tag in response.json()]

    def get_link(self, movie_id: int) -> LinkSimple:
        url = f"{self.movie_base_url}/links/{movie_id}"
        response = httpx.get(url)
        response.raise_for_status()
        return LinkSimple(**response.json())

    def list_links(self, skip: int = 0, limit: int = 100) ->
List[LinkSimple]:
        url = f"{self.movie_base_url}/links"
        params = {"skip": skip, "limit": limit}
        response = httpx.get(url, params=params)
        response.raise_for_status()
        return [LinkSimple(**link) for link in response.json()]

    def get_analytics(self) -> AnalyticsResponse:
        url = f"{self.movie_base_url}/analytics"
        response = httpx.get(url)
        response.raise_for_status()
        return AnalyticsResponse(**response.json())
```

TLe fichier `movie_client.py` est **le cœur du SDK**. Il encapsule toutes les **fonctions de communication avec l'API MovieLens** et fournit une interface Python simple, propre et typée pour les utilisateurs (data analysts, data scientists, développeurs, etc.).

Ce fichier définit la **classe `MovieClient`**, qui agit comme **le client principal pour interagir avec l'API MovieLens**. Elle permet à l'utilisateur final de :

- récupérer des films, des notes, des tags, des liens,
- faire des recherches (filtrées par titre, genre, etc.),
- accéder à des données analytiques,
- vérifier que l'API est opérationnelle (`health_check`).

Le but est d'avoir une **expérience Python "clé en main"** pour appeler l'API sans écrire de requêtes HTTP brutes.

Voici la Structure et fonctionnement de `movie_client.py` :

1. **Initialisation du client**

```
def __init__(self, config: Optional[MovieConfig] = None):
    self.config = config or MovieConfig()
    self.movie_base_url = self.config.movie_base_url
```

Si l'utilisateur fournit une configuration (`MovieConfig`), on l'utilise.
Sinon, on en crée une par défaut (avec lecture `.env` automatique).
Puis on récupère l'URL de base de l'API.

---

2. **Méthode `health_check()`**

Vérifie que l'API est accessible (utile avant d'enchaîner les requêtes).

---

3. **Méthodes pour les films**

- `get_movie(movie_id)` : récupère un film avec tous ses détails (`MovieDetailed`)
- `list_movies(...)` : permet de récupérer une **liste paginée** de films, avec filtres `title` et `genre`

Très utile dans un notebook pour faire une recherche rapide :

```
client.list_movies(title="Matrix", genre="Action")
```

---

4. **Méthodes pour les notes (ratings)**

- `get_rating(user_id, movie_id)` : récupère la note d'un utilisateur pour un film.
- `list_ratings(...)` : liste paginée des notes, avec filtres `movie_id`, `user_id`, ou `min_rating`.

---

5. **Méthodes pour les tags**

- `get_tag(...)` : un tag spécifique pour un film par un utilisateur.
- `list_tags(...)` : liste paginée des tags avec possibilité de filtrer.

---

6. **Méthodes pour les liens (vers IMDb, TMDb)**

- `get_link(movie_id)` : récupère les IDs externes du film.
- `list_links()` : récupère tous les liens disponibles.

---

7. **Méthode d'analytics**

```
def get_analytics(self) -> AnalyticsResponse:
```

Récupère un rapport agrégé (statistiques globales comme nombre de films, de notes, etc.).

Installation du package dans l'environnement de développement

Au niveau du terminal, assurez-vous d'être dans sdk/ et tapez cette commande :

```
pip install -e .
```

Cette commande est très utilisée pendant le **développement d'un package Python** (comme ton SDK `moviesdk`). Voici une explication simple et claire pour bien comprendre :

Que fait cette commande ?

`pip install`

Installe un package Python dans ton environnement (comme tu le fais avec `pip install requests`).

`-e` = `--editable`

Le flag `-e` signifie **"editable mode"** (mode modifiable). Cela veut dire que **le package n'est pas copié** dans le dossier `site-packages`, mais un **lien symbolique** est créé vers ton dossier local (`.`).

`.`

C'est le chemin vers le **répertoire courant**, donc ici le dossier `sdk/`, qui contient ton `pyproject.toml`.

Pourquoi c'est utile pour ton SDK ?

Quand tu développes ton SDK, il est encore en évolution. Grâce à l'installation en mode *editable* :

- Tu peux modifier tes fichiers source (ex : `movie_client.py`, `schemas.py`, etc.)
- Et **voir les changements immédiatement** sans devoir réinstaller le package à chaque fois.

Pour résumer, en exécutant :

```
pip install -e .
```

Tu rends le package `moviesdk` utilisable **partout dans ton projet** (ou dans des notebooks, scripts, etc.) comme s'il était installé normalement, **mais tu peux continuer à modifier le code source** en temps réel.

Élément	Description
`pip install`	Installe un package
`-e` ou `--editable`	Crée un lien vers le code source, permet de le modifier en direct
`.`	Répertoire courant contenant `pyproject.toml`
Avantage	Pas besoin de réinstaller après chaque changement

Fichier `sdk/test_sdk.py`

```
from moviesdk import MovieClient, MovieConfig

# Initialisation du client avec l'URL de l'API
config = MovieConfig(movie_base_url="https://movielens-api-
rmr7.onrender.com")
client = MovieClient(config=config)

# 1. Health check
print("Health check:")
print(client.health_check())

# 2. Récupérer un film par ID
print("\n Movie ID 1:")
movie = client.get_movie(1)
print(movie)
print(type(movie))

# 3. Lister les films (premiers 5)
print("\n List of movies:")
movies = client.list_movies(limit=5)
print(type(movies))
for m in movies:
    print(m)
    print(type(m))

# 4. Récupérer un rating spécifique
print("\n Rating for user 1 and movie 1:")
```

```
rating = client.get_rating(user_id=1, movie_id=1)
print(rating)
print(type(rating))

# 5. Lien du film
print("\n Link for movie ID 1:")
link = client.get_link(movie_id=1)
print(link)
print(type(link))


# 6. Analytics
print("\n Analytics:")
analytics = client.get_analytics()
print(analytics)
print(type(analytics))
```

Ce fichier joue le rôle de **script de test manuel** pour le SDK. Il sert à **valider que le SDK fonctionne correctement** en appelant les différentes méthodes fournies par la classe `MovieClient`.

Mise à jour du fichier `sdk/src/moviesdk/movie_client.py`

```
import httpx
from typing import Optional, List, Literal, Union

from .schemas import MovieSimple, MovieDetailed, RatingSimple, TagSimple,
LinkSimple, AnalyticsResponse
from .movie_config import MovieConfig

import pandas as pd


class MovieClient:
    def __init__(self, config: Optional[MovieConfig] = None):
        self.config = config or MovieConfig()
        self.movie_base_url = self.config.movie_base_url

    def _format_output(self, data, model, output_format:
Literal["pydantic", "dict", "pandas"]):
        if output_format == "pydantic":
            return [model(**item) for item in data]
        elif output_format == "dict":
            return data
        elif output_format == "pandas":
            import pandas as pd
            return pd.DataFrame(data)
        else:
            raise ValueError("Invalid output_format. Choose from
'pydantic', 'dict', or 'pandas'.")

    def health_check(self) -> dict:
```

```
        url = f"{self.movie_base_url}/"
        response = httpx.get(url)
        response.raise_for_status()
        return response.json()

    def get_movie(self, movie_id: int) -> MovieDetailed:
        url = f"{self.movie_base_url}/movies/{movie_id}"
        response = httpx.get(url)
        response.raise_for_status()
        return MovieDetailed(**response.json())

    def list_movies(
        self,
        skip: int = 0,
        limit: int = 100,
        title: Optional[str] = None,
        genre: Optional[str] = None,
        output_format: Literal["pydantic", "dict", "pandas"] = "pydantic"
    ) -> Union[List[MovieSimple], List[dict], "pd.DataFrame"]:
        url = f"{self.movie_base_url}/movies"
        params = {"skip": skip, "limit": limit}
        if title:
            params["title"] = title
        if genre:
            params["genre"] = genre
        response = httpx.get(url, params=params)
        response.raise_for_status()
        return self._format_output(response.json(), MovieSimple,
output_format)

    def get_rating(self, user_id: int, movie_id: int) -> RatingSimple:
        url = f"{self.movie_base_url}/ratings/{user_id}/{movie_id}"
        response = httpx.get(url)
        response.raise_for_status()
        return RatingSimple(**response.json())

    def list_ratings(
        self,
        skip: int = 0,
        limit: int = 100,
        movie_id: Optional[int] = None,
        user_id: Optional[int] = None,
        min_rating: Optional[float] = None,
        output_format: Literal["pydantic", "dict", "pandas"] = "pydantic"
    ) -> Union[List[RatingSimple], List[dict], "pd.DataFrame"]:
        url = f"{self.movie_base_url}/ratings"
        params = {"skip": skip, "limit": limit}
        if movie_id:
            params["movie_id"] = movie_id
        if user_id:
            params["user_id"] = user_id
        if min_rating:
            params["min_rating"] = min_rating
        response = httpx.get(url, params=params)
```

```
        response.raise_for_status()
        return self._format_output(response.json(), RatingSimple,
output_format)

    def get_tag(self, user_id: int, movie_id: int, tag_text: str) ->
TagSimple:
        url = f"{self.movie_base_url}/tags/{user_id}/{movie_id}/{tag_text}"
        response = httpx.get(url)
        response.raise_for_status()
        return TagSimple(**response.json())

    def list_tags(
        self,
        skip: int = 0,
        limit: int = 100,
        movie_id: Optional[int] = None,
        user_id: Optional[int] = None,
        output_format: Literal["pydantic", "dict", "pandas"] = "pydantic"
    ) -> Union[List[TagSimple], List[dict], "pd.DataFrame"]:
        url = f"{self.movie_base_url}/tags"
        params = {"skip": skip, "limit": limit}
        if movie_id:
            params["movie_id"] = movie_id
        if user_id:
            params["user_id"] = user_id
        response = httpx.get(url, params=params)
        response.raise_for_status()
        return self._format_output(response.json(), TagSimple,
output_format)

    def get_link(self, movie_id: int) -> LinkSimple:
        url = f"{self.movie_base_url}/links/{movie_id}"
        response = httpx.get(url)
        response.raise_for_status()
        return LinkSimple(**response.json())

    def list_links(
        self,
        skip: int = 0,
        limit: int = 100,
        output_format: Literal["pydantic", "dict", "pandas"] = "pydantic"
    ) -> Union[List[LinkSimple], List[dict], "pd.DataFrame"]:
        url = f"{self.movie_base_url}/links"
        params = {"skip": skip, "limit": limit}
        response = httpx.get(url, params=params)
        response.raise_for_status()
        return self._format_output(response.json(), LinkSimple,
output_format)

    def get_analytics(self) -> AnalyticsResponse:
        url = f"{self.movie_base_url}/analytics"
        response = httpx.get(url)
        response.raise_for_status()
```

```
            return AnalyticsResponse(**response.json())
```

Voici une explication détaillée des changements apportés au fichier `movie_client.py` pour offrir deux modes de sortie :

L'objectif principal de ces changements est de rendre ton API plus flexible pour les différents types d'utilisateurs. En particulier, tu souhaites faciliter l'interaction avec l'API pour les **Data Analysts** et **Data Scientists**, qui sont habitués à utiliser des structures comme des **pandas DataFrame** ou des **dictionnaires** pour leurs analyses.

1. Retour par défaut des objets Pydantic

Dans le code, nous continuons de retourner des objets basés sur **Pydantic**, comme `MovieDetailed`, `RatingSimple`, etc., dans la plupart des cas. C'est un choix courant pour plusieurs raisons :

- **Clarté** et **Typage** : Pydantic garantit que les données respectent bien les modèles, ce qui améliore la lisibilité et l'autocomplétion dans les IDE.
- **Validation** : Les objets Pydantic offrent une validation automatique des données en fonction du modèle, ce qui aide à garantir que les données reçues sont bien structurées.
- **Pratique pour le backend** : Les objets Pydantic sont particulièrement utiles pour les systèmes qui nécessitent des objets structurés, comme pour les intégrations avec d'autres APIs ou services backend.

2. Retour de dictionnaires ou pandas DataFrame selon un format spécifié

Pour rendre l'API plus accessible aux utilisateurs de **pandas** et des **Data Analysts/Data Scientists**, nous avons ajouté un paramètre `output_format` optionnel aux méthodes de notre client API. Ce paramètre permet de choisir le format de sortie des données en fonction des préférences de l'utilisateur. Les options sont :

- **"pydantic"** (par défaut) : Retourne les objets Pydantic (modèles structurés).
- **"dict"** : Retourne les données sous forme de dictionnaires Python natifs. Cela peut être plus pratique pour ceux qui souhaitent manipuler des données sans avoir à traiter les objets Pydantic.
- **"pandas"** : Retourne un **DataFrame pandas**, ce qui est très pratique pour les Data Scientists et Analystes qui travaillent directement avec des tableaux de données.

3. La méthode `_format_output`

Cette méthode est utilisée pour formater la sortie en fonction de l'option `output_format` choisie. Elle prend en entrée les données brutes reçues de l'API (`data`), le modèle Pydantic (`model`) et le format de sortie demandé (`output_format`). Elle décide ensuite quel type d'objet retourner en fonction de la valeur de `output_format` :

- Si le format est `"pydantic"`, il retourne des objets basés sur le modèle Pydantic.
- Si le format est `"dict"`, il retourne directement les données sous forme de dictionnaires.
- Si le format est `"pandas"`, il utilise pandas pour convertir les données en un DataFrame.

Exemple d'utilisation de `output_format` dans une méthode comme `list_movies`

Voici un rajout de code dans le fichier `sdk/test_sdk.py` :

```
# 7. Tests avec option du choix de format de sortie
    # Par défaut : objets Pydantic
new_movies = client.list_movies(limit=5)
print(new_movies)
print(type(new_movies))

    # Format dictionnaire
new_movies_dicts = client.list_movies(limit=5, output_format="dict")
print(new_movies_dicts)
print(type(new_movies_dicts))

    # Format DataFrame
new_movies_df = client.list_movies(limit=5, output_format="pandas")
print(new_movies_df)
print(type(new_movies_df))
```

En conclusion, les principaux changements apportés au fichier `movie_client.py` permettent de rendre l'API plus flexible et mieux adaptée aux **Data Analysts** et **Data Scientists**, tout en maintenant les avantages de Pydantic pour la validation et le typage. En offrant les trois options (`pydantic`, `dict`, `pandas`), nous pouvons facilement répondre aux besoins de différents types d'utilisateurs. Ces améliorations rendent l'API plus conviviale et plus facile à intégrer dans des pipelines de données Python typiques.