TD OpenAPI Specification - AA 2024-25

Lire attentivement avant de commencer les exercices.

1. Tout d'abord, il faut créer un nouveau référentiel (repository) **publique** sur GitHub avec un projet vide (un README suffit)

2. Ensuite, répondre au formulaire suivant : https://forms.gle/Fh8bNyhwTXgw23pY8. Veillez à ce que le format de la référence étudiante fournie soit bien comme celui-ci : STD24001. Pas 24001, ou 001, mais uniquement STD24001.

3. Pour chaque exercice, créez un fichier YML qui va être nommé entre `exo1.yml` jusqu'à `exo6.yml`, et qui sera directement mis dans la racine de votre projet, c'est-à-dire, il ne faut pas les mettre dans des dossiers.

4. Chaque fois qu'un exercice est effectué, n'oubliez pas de directement push sur le référentiel les changements. Votre activité sur GitHub ***pourrait*** contribuer/influencer votre note finale sur le cours. En particulier, si aucune activité n'est visible entre la période du 21 août jusqu'au 26 août (période entre les séances), vous pourriez obtenir un malus sur votre prochaine évaluation pratique (OAS).

Exercice 1 – Décrire une API simple de "Ping"

Objectif : Découvrir la structure minimale d'une spécification OAS3.

Écris une spécification OAS3 qui décrit une API avec une seule route :

- `GET /ping` qui retourne un 200 OK avec text/plain contenant la chaîne "pong".

Pistes :

- Définir openapi, info, paths.
- Définir une réponse avec content: text/plain.

Exercice 2 – Manipuler les paramètres et les réponses

Objectif : Introduire les paramètres et les schémas JSON.

Définis une spécification OAS3 pour une API de gestion d'utilisateurs qui contient la route ainsi que les paramètres suivants :

`GET /users?page=1&size=20` :

Où page et size sont des paramètres de requêtes de type nombre.

Retourne un utilisateur sous forme JSON, par exemple :

```
{
  "id": 1,
  "name": "Alice",
  "email": "alice@example.com"
}
```

Réponses attendues :

- 200 : liste des utilisateurs retrouvés.
- 400 : mauvais types de paramètres fournis { "error": "Bad types for provided query parameters" }.

Piste : Parameters Object doit être utilisé pour pouvoir définir un paramètre dans la requête. Il peut être mis dans la requête (query), le chemin (path) ou encore l'entête (headers)

Exercice 3 – Définir un modèle et plusieurs opérations

Objectif : Structurer l'API avec des composants réutilisables (components/schemas).

Écris une spécification pour une API de gestion de tâches :

Modèle Task :

```
{
  "id": 1,
  "title": "Faire les courses",
  "completed": false
}
```

- **Routes à spécifier** :
  - `GET /tasks` → Retourne une liste de `Task[]`.
  - `POST /tasks` → Crée une liste de nouvelles tâches fournies à travers le corps de la requête, et retourne la liste nouvelle créée avec un code de statut 201.
  - `GET /tasks/{id}` → Retourne la tâche portant l'ID fourni, sinon retourne une réponse 404 comportant le message "Task not found".
  - `DELETE /tasks/{id}` → Supprime une tâche portant l'ID fourni, et retourne la tâche supprimée avec un code 200.
  - `DELETE /tasks` → Supprime une liste de tâches fournies à travers une liste d'identifiants à supprimer, et retourne la liste de tâches ayant été supprimée avec un code 200.

Pistes :

- Définir `Task` dans `components/schemas` et invoquer avec $ref pour éviter les redondances.
- Référencer ce schéma dans les différentes réponses.
- Gérer les différents codes (`200`, `201`, `404`).

Exercice 4 – Recherche de produits

Objectifs :

- Utiliser `components/parameters` pour centraliser les définitions des paramètres.
- Ajouter des `examples` aux paramètres et aux réponses.

1. Définir un modèle `Product` dans `components/schemas`, où un `Product` possède les attributs suivants :
    - name de type chaîne de caractère
    - expiration_datetime de type date et heure
    - price de type nombre décimal
2. Créer une route `GET /products` avec des **paramètres de query réutilisables** définis dans `components/parameters` :
    - `limit` (integer, optionnel) – nombre maximum d'éléments.
    - q (string, optionnel) – terme de recherche.
3. Retourner une liste de `Product`.
4. Ajouter au moins **deux examples** : un résultat avec plusieurs produits et un avec une liste vide.

Exercice 5 – API avec authentification + tags (niveau expert)

Objectifs :

- Organiser l'API avec des **tags** pour grouper les endpoints.
- Définir un **security scheme** de type Basic Authentication.
- Protéger un endpoint avec authentification.
- Réutiliser des `components/parameters` et fournir des `examples`.

Pré-requis :

Reprendre le contexte de l'exercice 4, avant d'enchaîner sur l'exercice 5, en particulier le modèle `Product`. Ensuite, associer la route `GET /products` au tag Products, qui retourne toujours une liste de `Product`.

1. Définir un modèle `Order` dans `components/schemas`, où un `Order` possède les attributs suivants :
   - identifier de type nombre entier
   - customer_name de type chaîne de caractères
   - creation_datetime de type date et heure
   - total_amount de type nombre decimal
2. Ajouter deux endpoints avec des tags différents :
   - `GET /orders` (tag `Orders`) – Liste paginée des commandes.
   - `POST /orders` (tag `Orders`) – Créer une commande (protégé par Basic Authentication).
3. Définir les paramètres `page` et `size` dans `components/parameters`.
4. Ajouter un `securitySchemes` de type Basic Authentication pour POST.
5. Fournir des `examples` dans la réponse et dans la requête de création.

Exercice 6 – Bonnes pratiques sur OAS

Objectifs pédagogiques :

1. Travailler avec un modèle **complexe** décomposé en **sous-schémas**.
2. Éviter l'usage de `PATCH` en préférant des **PUT ciblés** sur des sous-ressources.
3. Pratiquer la **décomposition des routes** pour la mise à jour d'attributs spécifiques.
4. Réutiliser `components/parameters` pour les paramètres communs (`id`).
5. Fournir des **examples** clairs pour illustrer les requêtes et réponses.
6. Prendre en main la composition et l'héritage avec `allOf`.

Contexte et guide :

Un profil d'utilisateur (UserProfile) est décrit par les informations suivantes :

- identifier de type chaîne de caractère;
- first_name de type chaîne de caractère,
- last_name de type chaîne de caractère,
- birthdate de type chaîne de date,
- email de type email,
- address_street de type chaîne de caractère,
- address_city de type chaîne de caractère,
- address_country de type chaîne de caractère,
- address_postal_code de type chaîne de nombre entier,
- needs_newsletter de type booléen ,
- language de type enum avec les valeurs suivantes possibles : mg, fr ou eng.

L'objectif est de pouvoir créer, lister et modifier les utilisateurs et leurs informations. Si nous ne décomposons pas la ressource UserProfile ou plusieurs sous ressources, alors pour mettre à jour un attribut de UserProfile, il faudra tout envoyer en utilisant PUT /users ou PUT /users/{id}.

PATCH pourrait être une alternative, mais le soucis c'est qu'il faut spécifier un par un les attributs qui peuvent être modifiés avec PATCH, en particulier l'identifiant ne doit pas être modifiables par exemple, et d'autres attributs peuvent également ne pas être accessibles à la modification.

Enfin, veillez à ce que lors de la création, on ne fournisse pas d'identifiant car il sera généré automatiquement par le serveur, et ainsi lors de la création, il faut plutôt utiliser un sous-schéma (CreateUserProfle par exemple) qui ne prend pas un identifiant et qui sera utilisé par le schéma principal UserProfile, qui lui contient bien l'identifiant ainsi que les autres éléments.

Consignes :

- Créez le modèle `UserProfile` avec ses sous-schemas (`CreateUserProfile`, `PersonalInfo`, `Address`, `Preferences`), qui sont décrits comme suit :

- Implémentez :

    - `GET /users/{id}` → retourne l'utilisateur avec toutes ses informations.

    - `POST /users` → crée une liste d'utilisateurs sans identifiants, et retourne la liste des utilisateurs créés avec leurs identifiants.

    - `PUT /users/{id}/personalInfo` → met à jour uniquement la partie **informations personnelles**.

    - `PUT /users/{id}/address` → met à jour uniquement l'**adresse**.

    - `PUT /users/{id}/preferences` → met à jour uniquement les **préférences**.