TD 3 *Flask-Restx*

Nous allons voir comment construire une API REST avec l'aide de Flask-RESTX, `https://flask-restx.readthedocs.io/`
Flask-RESTX est une extension de Flask qui permet de créer plus facilement des API REST, de générer la documentation OpenAPI `https://www.openapis.org/`

Exercice 1 *Installation*
Activer votre environnement virtuel, et installer Flask-RESTX

```
pip install flask-restx
```

Exercice 2 *Initialisation*
Nous allons tout d'abord initialiser notre application Flask

2.1 Faire un dossier `blog`, et créer deux fichiers `__init__.py` et `myapp.py`

`myapp.py`

```
from flask import Flask

app = Flask(__name__)
```

`__init__.py`

```
from .myapp import app
```

Vérifier avec un `flask run --debug` que votre application fonctionne

2.2 Nous allons maintenant initialiser flask-RESTX, et notre BD. Faire un fichier `extensions.py` qui créer l'api flask-RESTX et la BD avec SQLAlchemy.

`extensions.py`

```
from flask_sqlalchemy import SQLAlchemy
from flask_restx import Api

api = Api()
db = SQLAlchemy()
```

2.3 Modifier le fichier `myapp.py` pour intégrer l' api et la bd

`myapp.py`

```
from flask import Flask
from .extensions import api, db


app = Flask(__name__)

# initialisation de la BD
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///db.sqlite3"

# initialisation de restx
api.init_app(app)
db.init_app(app)
```

Vous avez désormais une première application (vide) construite avec flask-RESTX.

2.4 Rendez-vous sur la `http://127.0.0.1:5000`, pour consulter la page de Open-API. Celle-ci est vide pour le moment, car nous n'avons défini aucune route.

Exercice 3 *Création de la première route*
Nous allons faire un exemple de test

3.1 Créer un fichier `views.py` qui regroupera toutes les routes de notre application. Nous allons créer notre ***namespace***, qui regroupera toutes nos routes. En définissant notre *namespace* nous définissons également la racine de toutes nos routes (ici `api`)

`views.py`

```
from flask_restx import Resource, Namespace
# creation du namespace, racine de tous les endpoints
ns = Namespace("api")
```

3.2 Nous définissons notre première route grâce aux décorateurs de flask-RESTX. Nous construisons une classe qui hérite de *Resource*. On définit ensuite une méthode (objet, donc avec self) correspond à une des méthodes http (ici get). On retourne ensuite une valeur JSON serializable, ici un dictionnaire Le décorateur `@ns.route` permet de spécifier notre route

`views.py`

```
...
# definition d'une route
@ns.route("/hello")
class Hello(Resource):
    def get(self):
        return  {"hello": "restx"}
```

3.3 Il faut maintenant enregistrer notre namespace dans l'api. Pour cela, nous allons modifier le fichier `myapp.py` : Nous importons donc notre namespace, et on l'enregistre sur notre api avec la méthode `add_namespace`

`myapp.py`

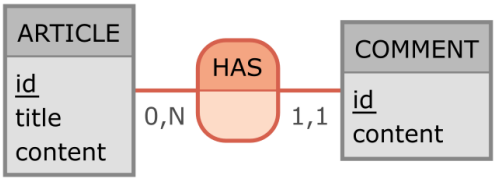
```
...
from .views import ns

...

 # ajout du namespace defini dans views
api.add_namespace(ns)
```

3.4 Retourner sur la page Open-API. Vous pouvez désormais voir votre namespace api, et en l'ouvrant, vous pouvez trouver la route que nous venons de définir, associée à la méthode get.
Vous pouvez tester la méthode en cliquant sur *Try it out*, puis sur *Execute*. Vous pouvez voir la réponse (notre dictionnaire).

Exercice 4 *Modèle plus complet*
Nous allons maintenant utiliser une modèle plus complet, relié à la base de données. Nous souhaitons une API REST qui permet de gérer des articles de blog, qui peuvent être accompagnés de commentaires
Le MCD est le suivant :



4.1 Nous créons nos modèles. Pour cela créer un fichier `models.py` avec deux classes correspondant aux articles et aux commentaires

`models.py`

```
from .extensions import db

class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(50))
    content = db.Column(db.String(500))
    comments = db.relationship("Comment", back_populates="article")


class Comment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    content = db.Column(db.String(100))
    article_id = db.Column(db.ForeignKey("article.id"))
    article = db.relationship("Article",back_populates="comments")
```

4.2 Nous définissons une méthode pour créer et remplir notre base de données. Créer un fichier `commands.py`

`commands.py`

```
from .extensions import db
from .models import Article, Comment
from .myapp import app

@app.cli.command()
def syncdb():
    db.create_all()

    db.session.query(Article).delete()
    db.session.query(Comment).delete()

    article1 = Article(title="Premier Article", content="Ceci est mon premier
       article")
    article2 = Article(title="Second Article", content="Ceci est mon second
       article")

    comment1 = Comment(content="Super article",article_id=1)
    db.session.add(article1)
    db.session.add(article2)
    db.session.add(comment1)
    db.session.commit()
```

4.3 Modifier `__init__.py` pour ajouter cette commande

`__init__.py`

```
from .myapp import app
from .commands import syncdb
```

4.4 Exécuter la commande `flask syncdb` pour créer et remplir votre BD

Exercice 5 *Route avec notre modèle*
Nous allons tout d'abord créer une route pour récupérer tous les articles. Pour cela, procédons comme pour la route précédente

5.1 Dans le fichier `views.py`, nous créons une route, avec le décorateur, la classe associée avec la méthode *get*. Nous donnons à la classe un nom différent de celle du modèle pour ne pas les confondre

`views.py`

```
...
from .models import Article, Comment, get_all_articles
...

@ns.route("/articles")
class ArticleCollection(Resource):
     def get(self):
         return get_all_articles()
```

`models.py`

```
...

def get_all_articles():
    return Course.query.all()
```

5.2 Retourner sur la page OpenApi, et tester cette nouvelle route. Pourquoi cela ne fonctionne-t-il pas ?

5.3 Il est donc nécessaire que le retour de notre fonction soit *JSON serialisable*. Flask-RESTX nous permet de faciliter cette conversion, en nous permettant de définir le format du retour. Pour cela créer un fichier `api_models.py`, qui contiendra les modèles à exposer(attention à ne pas confondre les modèles métiers (nos classes) et ces modèles)

`api_models.py`

```
from flask_restx import fields
from .extensions import api

article_model = api.model("Article",{
    "id": fields.Integer,
    "title":fields.String,
    "content":fields.String
})
```

5.4 Il est maintenant nécessaire de préciser que notre méthode *get* utilise ce modèle pour fournir la réponse. Pour cela on passe par un nouveau décorateur `@ns.marshal_list_with`

`views.py`

```
@ns.route("/articles")
class ArticleCollection(Resource):
    @ns.marshal_list_with(article_model)
    def get(self):
        return get_all_articles()
```

5.5 Tester cette modification, vous devriez obtenir désormais tous vos articles, affichés avec les champs définis dans votre modèle

5.6 Faire la même chose avec les commentaires

Exercice 6 *Création d'un nouvel article*
On souhaite désormais avoir la possibilité de créer un nouvel article. On définit donc une nouvelle méthode, `post`, dans notre classe `ArticleCollection`. Il est nécessaire de préciser les données attendues. Cela est possible grâce au décorateur `@ns.expect`. Pour l'instant notre méthode ne fait rien

views.py

```
@ns.route("/articles")
class ArticleCollection(Resource):
    ...
    @ns.expect(article_model)
    def post(self):
        return {}
```

6.1 Tester cette nouvelle méthode. Vous pouvez constater que les données attendus sont l'id, le titre et le contenu. Nous ne voulons pas spécifier l'id d'un article lors de sa création. Nous laissons l'application le gérer. Il est donc nécessaire de modifier le modèle de données que nous attendons. Pour cela nous créons un nouveau modèle, spécifique à la création d'article.

api_models.py

```
...
article_input_model = api.model("ArticleInput",{
    "title": fields.String,
    "content":fields.String
})
```

6.2 Modifier votre méthode *post* pour intégrer ce nouveau modèle. Pour récupérer les données envoyées, nous allons utiliser `ns.payload`. Nous créons le nouvel article, et nous l'ajoutons à la bd

views.py

```
@ns.route("/articles")
class ArticleCollection(Resource):
    ...
    @ns.expect(article_input_model)
    def post(self):
        create_article(title=ns.payload["title"], content=ns.payload["content"
            ])
        return {},201
```

models.py

```
...

def create_article(title,content):
    article = Article(title=title, content=content)
    db.session.add(article)
    db.session.commit()
    return article
```

6.3 Si on souhaite retourner en réponse l'article qui vient d'être crée, il faut ajouter un nouveau décorateur `@ns.marshal_with`, en spécifiant cette fois le modèle de sortie

views.py

```
@ns.route("/articles")
class ArticleCollection(Resource):
    ...
    @ns.expect(article_input_model)
    @ns.marshal_with(article_model)
    def post(self):
        article = create_article(title,content)
        return article,201
```

6.4 Tester cette nouvelle méthode en créant un nouvel article

Exercice 7 *Consulter un article*
Nous souhaitons maintenant récupérer une seul article, à partir de son id. Nous créons donc une nouvelle route `articles/<int:id>` et une nouvelle classe associée `ArticleItem`. nous allons faire une méthode get, avec l'id de l'article à récupérer.
Il est possible qu'il n'existe pas d'article associé à l'id. Il faut donc utiliser la fonction `abort`. Pour la documentation on ajoute la possibilité d'un autre type de réponse avec le décorateur `ns.response`

`views.py`

```
from flask_restx import Resource, Namespace, abort
...
@ns.route("/articles/<int:id>")
@ns.response(404, 'Article not found')
class ArticleItem(Resource):
    @ns.marshal_with(article_model)
    def get(self,id):
        article = get_article(id)
        if article is None:
            abort(404,"Article not found")
        return article
```

`models.py`

```
...

def get_article(id)
    Article.query.get(id)
```

7.1 Tester votre méthode *get*

7.2 Faire la même chose avec les commentaires

7.3 Ajouter une méthode *post* pour les commentaires. Attention, l'article associé au commentaire doit exister

Exercice 8 *Modifier*
On souhaite avoir la possibilité de modifier un article. Pour cela nous allons ajouter une méthode *put*, avec les décorateurs adaptés

```
@ns.route("/articles/<int:id>")
class ArticleItem(Resource):
...
    @ns.expect(article_input_model)
    @ns.marshal_with(article_model)
    def put(self,id):
        article = modify_article(id,title,content)
        if article is None:
            abort(404,"Article not found")
        return article, 200
```

`models.py`

```
...

def modify_article(id,title,content)
    article = Article.query.get(id)
    if article is None:
        return None
    article.title = ns.payload["title"]
    article.content = ns.payload["content"]
    db.session.commit()
```

8.1 Tester la modification d'un article

8.2 Ajouter cette méthode aux commentaires

Exercice 9 *Suppression*

On souhaite supprimer un article, il suffit donc de rajouter une méthode delete. Ici pas besoin de décorateur puisque l'on n'attend aucune donnée, et nous ne retournons pas d'article `views.py`

```
@ns.route("/articles/<int:id>")
class ArticleItem(Resource):
...
    def delete(self,id):
        delete_article(id)
        return {}, 204
```

`models.py`

```
...

def delete_article(id)
    article = Article.query.get(id)
    db.session.delete(article)
    db.session.commit()
```

9.1 Tester la suppression d'article

9.2 Ajouter la suppression de commentaire

Exercice 10 *Ajout des url dans le retour*

Les données retournées ne contiennent pas d'url, pointant vers la ressource. Pour l'ajouter, nous allons donc modifier les modèles retournées. Nous allons utiliser `fields.Url`, qui prend en paramètre le nom de la classe (voir exemple). Par défaut, la fonction retourne une uri relative. Pour ajouter le nom de l'hôte et du port, on ajoute le paramètre `absolute` à vrai.

`api_models.py`

```
article_model = api.model("Article",{
    "id": fields.Integer,
    "title":fields.String,
    "content":fields.String,
    "uri":fields.Url('api_article_item',absolute=True),
})
```

10.1 Ajouter cette uri à l'article et au commentaire. Vérifier que cela fonctionne

Exercice 11 *Commentaires d'un article donné*

Actuellement nous pouvons récupérer tous les commentaires, ou le commentaire d'un id donné. On souhaite avoir la possibilité d'avoir tous les commentaires d'un article donné.

11.1 Ajouter à `views.py` la nouvelle route et la nouvelle classe nécessaire pour avoir ces commentaires

11.2 Ajouter une nouvelle url dans votre article pour avoir cette liste de commentaires

Exercice 12 *Validation des données envoyées*
On souhaite que les données que le serveur recoit soient vérifiées avant de créer un article, autrement dit qu'elle correspondent au modèle. Actuellement, notre décorateur `ns.expect` sert uniquement à générer la documentation.

12.1 Tester la création d'article en supprimant le champs `title` du json envoyé. Que se passe-t-il ?

12.2 Nous modifions le modèle pour préciser que les champs sont obligatoires. On active la vérification du modèle dans le décorateur `ns.expect`

`api_models.py`

```
...
article_input_model = api.model("ArticleInput",{
    "title": fields.String(required=True),
    "content":fields.String(required=True)
})
```

`views.py`

```
    @ns.expect(article_input_model,validate=True)
    @ns.marshal_with(article_model)
    def post(self):
        # ns.payload contient les donnees recues
        article = create_article(title=ns.payload["title"], content=ns.payload[
            "content"])
        return article,201
```

12.3 Tester cette vérification, en envoyant un json incomplet (sans le titre par exemple), ou en n'utilisant pas le bon type (un entier pour le titre par exemple)