Documentation API

Projet simulateur de mise en orbite

ARMENOULT Florian
LESUEUR Cédric

Table des matières

Mise en place de l'environnement local de l'API

Éléments requis

Python 3.6 ou supérieur
PostgreSQL 10

L'utilisation d'un environnement virtuel est recommandée pour ce projet.

Installation des dépendances

Les paquets python nécessaires sont présents dans le fichier **requirements.txt** et peuvent être installés avec la commande **pip install -r requirements.txt**

Mise en place de la base de données

Les informations relatives à la base de données sont à précisées dans le fichier **params.py** du dossier **Orbit_api**

Après création de la base de données, migration des tables :
python manage.py makemigrations
python manage.py migrate && python manage.py migrate api

On peut ensuite lancer le serveur :

python manage.py runserver

Présentation de l'API

Présentation et avantages du framework

L'API repose sur le framework Django, qui repose sur le langage python. Bien que ressemblant à d'autres framework MVC comme Symfony, Django lui repose sur une architecture MVT (Modèle – Vue – Template).

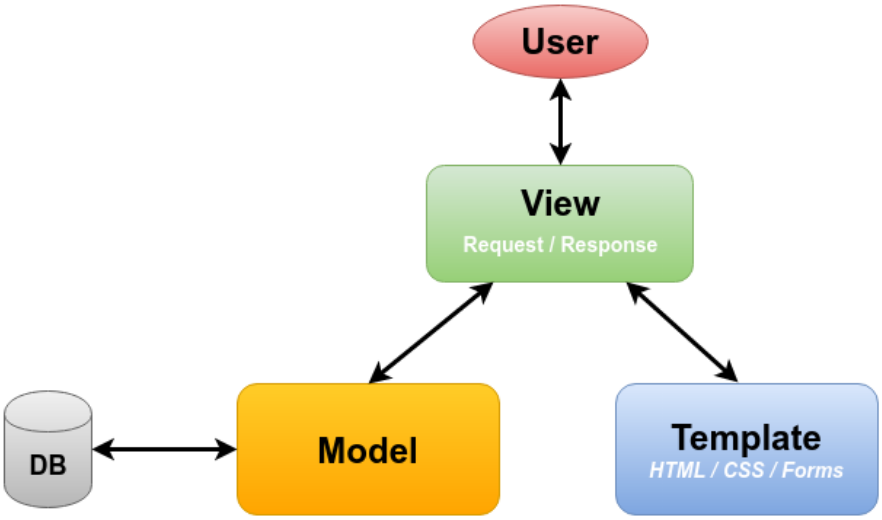

Figure 1: Représentation du modèle MVT

Django est également OpenSource, et possède une communauté très active, il est ainsi régulièrement mis à jour et possède une documentation très complète.

Toutes ces informations sont disponibles sur le site du projet : https://www.djangoproject.com/

Ce framework possède de nombreux avantages, tant sur la sécurité que sur ses fonctionnalités implémentées par défaut.

Entre autres :

- Protection XSS
- Protection CSRF, avec génération de token obligatoire pour chaque formulaire
- Protection contre les injections SQL

Le framework contient par défaut une interface d'administration, que l'on peut configurer pour chaque modèle, et permet de modifier chaque champ en fonction de son type (texte, nombre, liste de choix …)

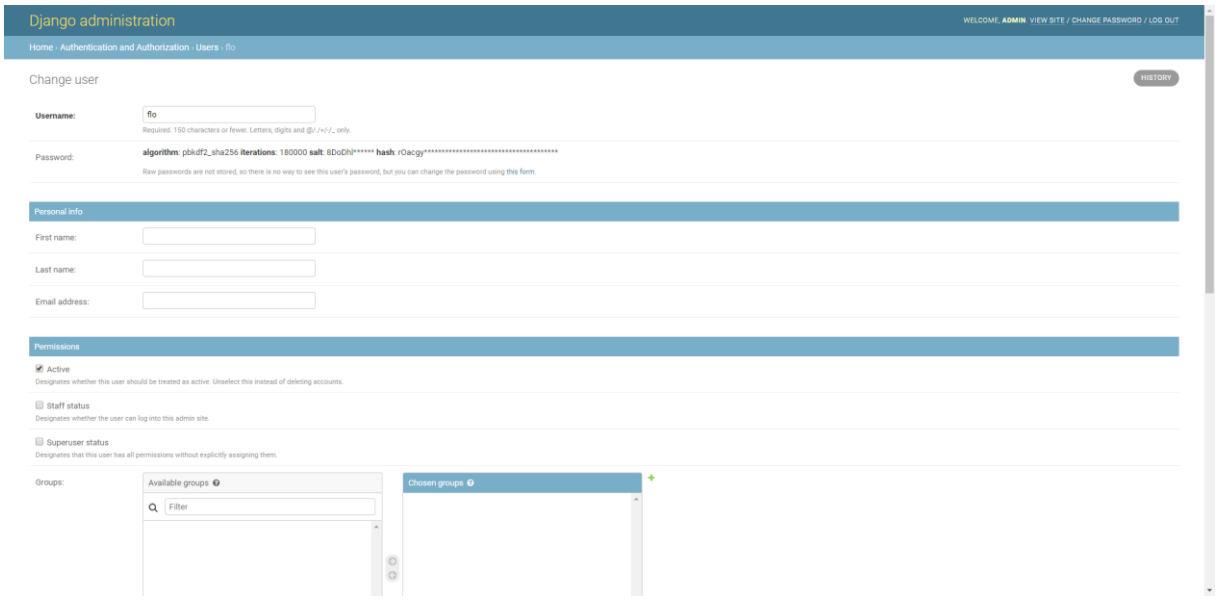

Figure 2: Exemple d'interface d'administration sur un objet utilisateur

Pour la sécurité, les mots de passe sont par défaut soumis à ces 3 conditions :

- Pas d'informations de l'utilisateur dans le mot de passe (username, prénom, nom…)
- Au moins 9 caractères
- Pas de mot de passe trop communs (évite les 20 000 mots de passe les plus communs)
- Le mot de passe n'est pas uniquement composé de chiffres

De plus, le mot de passe est par défaut encodé, comme on peut le voir dans l'interface d'administration :

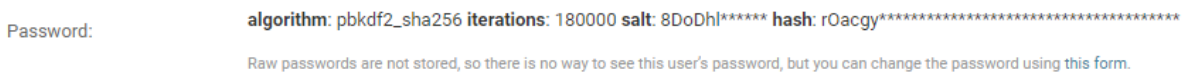

Figure 3: Exemple d'encodage des mots de passe

Configuration de DRF

L'utilisation de Django Rest Framework permet de facilement mettre en place notre API, qui peut être résumé en deux points : les serializers et les viewsets

Serializers

Les serializers vont nous permettre de transformer une instance d'un objet python en données sérialisées de type JSON par exemple

Ce processus est également exécutable dans l'autre sens, et permet de récupérer une instance d'un objet à partir des données sérialisées.

```
class CelestialBodySerializer(serializers.ModelSerializer):
    class Meta:
        model = CelestialBody
        fields = '__all__'
```

Figure 4: Exemple de serializer

Ce serializer se compose ainsi à minima d'une classe Meta, contenant le modèle que l'on souhaite intégrer à notre API ainsi que les champs souhaités

On peut également surcharger les méthodes présentes par défaut, qui correspondent au fonctionnement REST, par exemple :

En surchargeant la méthode **to_representation** on peut modifier à la volée les éléments tels qu'ils apparaitront après une requête GET.

Viewsets

Les viewsets vont finalement fonctionner comme une vue classique de Django, et vont permettre de faire le lien entre nos instances et les serializers correspondants

```
class CelestialBodyViewSet(viewsets.ModelViewSet):
    serializer_class = CelestialBodySerializer
    queryset = CelestialBody.objects.all()
```

Figure 5: Exemple de viewset

On précise ainsi pour chaque serializer créé son viewset correspondant, avec la liste des instances qui seront gérées, ici la requête **CelestialBody.objects.all()** permet de récupérer tous les éléments présents au sein de la base de données.

Router

La dernière étape consiste à configurer les URLs qui redirigeront vers nos viewsets.
On instancie notre router puis on définit pour chaque viewset l'URL qui lui correspondra.

```
router = routers.DefaultRouter()
router.register(r'celestialbody', CelestialBodyViewSet)
router.register(r'launchparams', LaunchParamsViewSet)
router.register(r'startparams', StartParamsViewSet)
router.register(r'results', ResultsViewSet)
```

Figure 6: Exemple de router sur notre API

A la suite de ce bloc on gère nos routes « classiques » de django, puis on vient ajouter à celles-ci les routes de notre API

```
urlpatterns += router.urls
```

Figure 7 : Ajout des URLs de l'API

Après cette étape, l'API est désormais totalement configurée et accessible depuis les URLs configurées

Déploiement de l'API

L'utilisation du django rest framework (DRF) permet de visualiser facilement les éléments de notre API depuis un navigateur classique

Elle est ainsi disponible à cette adresse : http://89.234.182.111/

Vue navigateur de DRF

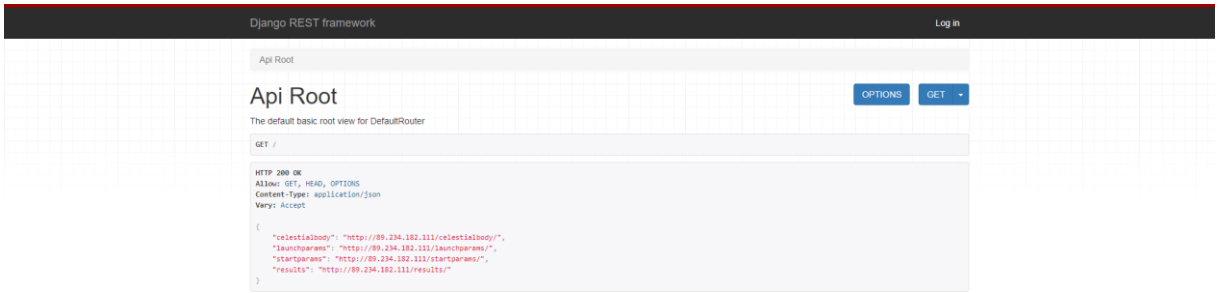

Figure 8 : Page d'accueil de l'API

Chaque type d'objet est ainsi disponible au sein de l'API REST : on pourra ainsi lire, ajouter, modifier ou supprimer des éléments

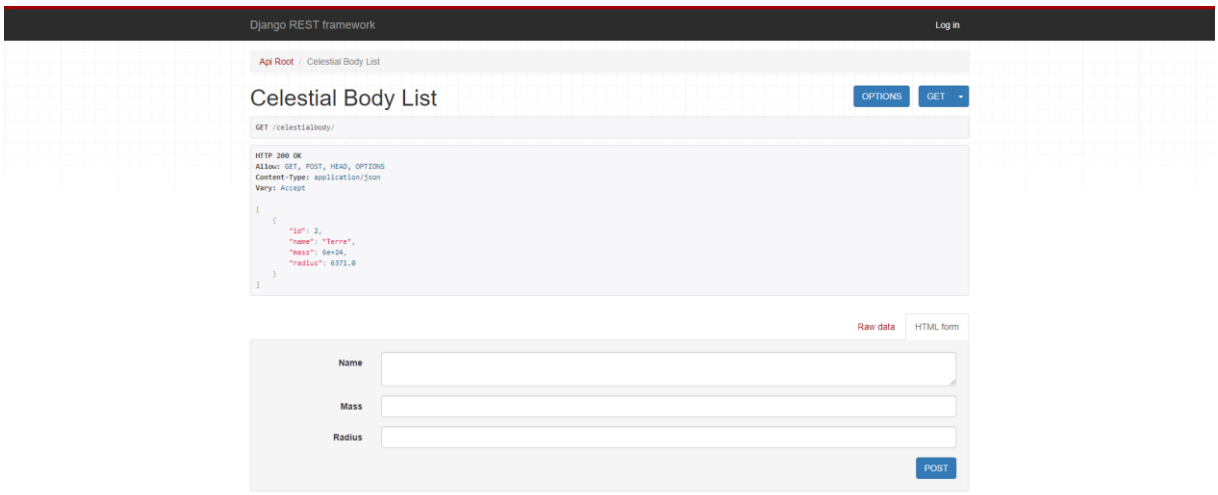

Figure 9: Page d'accueil des objets célestes, listant chaque élément et permettant d'en ajouter (POST)

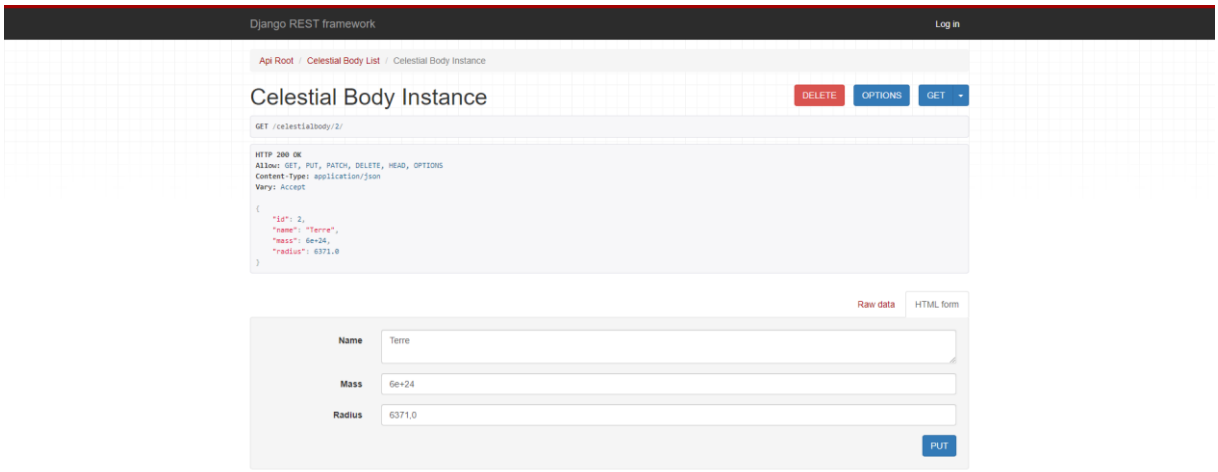

Figure 10: Page d'édition et de suppression d'un élément de la base

Requête sur l'API

En effectuant une requête GET depuis du code on récupère les données au format JSON :

```
>>> requests.get("http://89.234.182.111/celestialbody/2/").content.decode("utf-8")
'{"id":2,"name":"Terre","mass":6e+24,"radius":6371.0}'
```

Figure 11: Exemple de requête GET, ici via Python